UNIVERSIDAD DE SANTIAGO DE CHILE

FACULTAD DE INGENIERÍA

Departamento de Ingeniería Informática

PROYECTO DE LABORATORIO 3 – PARADIGMA POO

Paradigmas de Programación

Isaac Alejandro Espinoza Barría 21.278.828-7

Sección:

13204-0-A-1 E.L

Profesores:

Edmundo Leiva

Daniel Gacitúa

Fecha:

24 de junio de 2024

TABLA DE CONTENIDO

1. INTRODUCCIÓN

Conocer diferentes paradigmas de programación es esencial para que los programadores desarrollen soluciones efectivas ante problemáticas considerando todas las perspectivas posibles y así, desde una visión holista, elegir los mejores métodos al escribir sus programas.

En ciudades altamente densas de población, la necesidad de transporte es indispensable; las personas requieren moverse por diferentes puntos críticos, donde muchas veces la locomoción terrestre no da abasto con la demanda de transporte. Por este motivo, una red de metro se vuelve un satisfactor primordial a la hora de abordar esta problemática de forma eficiente.

El presente laboratorio plantea el problema de la demanda de movilidad urbana y propone emplear la solución de un sistema de administración de una red de metro, mediante un programa en el lenguaje Java bajo el paradigma de programación orientada a objetos.

El documento consta de introducción, descripción del problema, análisis del problema, diseño de la solución, consideraciones de implementación, instrucciones de uso, resultados y autoevaluación, y conclusiones.

2. DESCRIPCIÓN DE LA PROBLEMÁTICA

2.1. PROBLEMA

En el mundo actual, y con miras al futuro, la humanidad enfrenta grandes desafíos en torno a las condiciones de vida; la contaminación y mala calidad del aire son dificultades que se han vuelto importantes de combatir.

Asimismo, en las grandes urbes donde millones de personas confluyen en el diario vivir, los problemas de contaminación son evidentes; particularmente la contaminación que produce la locomoción de vehículos particulares produciendo gases de invernadero. Además, las grandes ciudades enfrentan conflictos de sobrepoblación y crisis migratoria, lo que produce un crecimiento exponencial de la cantidad de habitantes y, por ende, una gran cantidad de autos en las calles.

La locomoción tradicional no abarca la demanda de transporte existente. El aumento de habitantes produce que sea muy difícil el transportarse en la ciudad, por lo que las personas optan por usar sus vehículos particulares a pesar del costo monetario que esto implica y lo contaminantes que son.

Ante esto, es indispensable que las ciudades se expandan para recibir nuevos habitantes. No basta con la construcción de viviendas y edificios de servicios públicos, es necesario la creación de un sistema de locomoción que sea rápido, sustentable y amigable con el medio ambiente; dado que ayude a todos los habitantes a desplazarse a sus lugares de trabajo, estudio u hogares.

La creación de una red de metro es una beneficiosa solución al problema de contaminación y la sobrepoblación, ya que, es un sistema de locomoción sustentable y poco contaminante, alivia la congestión de tráfico automotriz en la superficie, además de lograr conectar diversos puntos de la ciudad en poco tiempo.

2.2. DESCRIPCIÓN DEL PARADGIMA

Para entender los procedimientos a emplear en la solución del problema, es necesario conocer algunos conceptos previos relevantes que logran facilitar su comprensión. A continuación, se enumeran y definen:

1.- Paradigma: Forma o manera de percibir el mundo; modo de hacer las cosas para abordar situaciones. “Realizaciones científicas universalmente reconocidas que, durante cierto tiempo, proporcionan modelos de problemas y soluciones a una comunidad científica” (Kuhn, 1962).

2.- Paradigma de programación: Modelo básico de diseño e implementación de programas en un computador. Donde se aceptan válidos ciertos criterios y principios que modelan el proceso de diseño de programa, orientan la forma de pensar y solucionar problemas (Spigariol, 2005).

3.- Tipo de dato abstracto (TDA): Estructura de datos conceptual creada por un programador en un computador; busca representar entidades de la realidad rescatando los elementos esenciales y omitiendo lo superfluo o poco necesario para la representación de estos. La especificación de un tipo de dato abstracto se basa en capas que facilitan la implementación propia del TDA y sus operaciones, las que son: constructores, funciones de pertenencia, selectores, modificadores y otras operaciones (Departamento de Ingeniería Informática [DIINF], 2024a).

El paradigma de programación orientada a objetos (POO) “es un método de implementación en el que los programas se organizan como colecciones cooperativas de objetos, cada uno de los cuales representa una instancia de alguna clase, y cuyas clases son, todas ellas miembros de una jerarquía de clases unidas” (Booch, 2010). Es de modo imperativo, esto quiere decir que describe cómo se realizan los pasos para llegar a la solución de un problema, el cómo llegamos al qué (DIINF, 2024b). Se basa en la concepción de “objetos” como abstracciones de entidades de la realidad, dado que “los objetos pueden contener características (atributos) y comportamientos (métodos)” (DIINF, 2024c). Posee varios conceptos claves pero los más destacables son: clases, especificación para cierto tipo de objeto; objetos, instancias de la clase, es decir, representaciones activas en memoria de una clase durante la ejecución de un programa; herencia, una subclase obtiene todos los métodos y atributos de una superclase; polimorfismo, se refiere a realizar la misma acción de diferentes maneras y/o a que un objeto puede tener diferentes formas; encapsulación, proceso de envolver código y datos en una sola unidad, idealmente, cada clase debe estar encapsulada; clases abstractas, no son instánciales y sólo pueden heredar a otras clases; interfaces, son clases abstractas pero que no poseen atributos y solo la definición de métodos, no la implementación de estos (DIINF, 2024c).

Este paradigma promueve la modularidad, reutilización de código y mantención del software. Por lo que facilita la creación de sistemas complejos (González, 2013).

3. ANÁLISIS DEL PROBLEMA

Para solucionar el problema, se desea crear un sistema de administración de una red de metro en el lenguaje de programación Java, bajo el paradigma imperativo de programación orientada a objetos. Esto quiere decir que se debe crear un programa que, mediante la conceptualización de objetos, permita realizar todas las operaciones básicas para operar un sistema de una red de metro.

Por consiguiente, se deben considerar diferentes TDAs, cada uno definido como una clase, que logran abstraer los elementos importantes de la red metro para poder trabajarlos dentro del programa. Donde mediante archivos .java, se definan todos los atributos y métodos asociados a un tipo de objeto; para que después, en una terminal y con la

ayuda de la herramienta Gradle, se compile e interprete el programa por la Java Virtual Machine. De esta forma, será posible operar el sistema de metro.

La red de metro está constituida por un conjunto de líneas, estaciones, secciones, trenes, carros, sistemas de control y mantenimiento, etc. Como requisitos, el programa de debe cumplir con las siguientes funcionalidades: creación y gestión de líneas de metro, añadir estaciones, planificación de rutas y conexiones, gestión de trenes, gestión de personal trabajador, mantenimiento de infraestructura, monitoreo y control.

A diferencia de los laboratorios anteriores, el presente paradigma define a el objeto como una unidad, lo que facilidad la definición de TDAs ya que no es necesario establecer estructuras de datos como listas para relacionar los elementos de un tipo de objeto.

Para la construcción del programa, se hizo un análisis previo de un diagrama de clases UML (ver Anexo 1), donde se establece la implementación del código en POO, los atributos y métodos para las clases, y los tipos de relaciones entre estos. Por ejemplo, en primera instancia se determina que el objeto Station tiene como atributos una ID (int), un nombre (String), un tipo (String m,r,c,t), y un tipo de parada en segundos (int); además del método constructor, los getters y los setters.

La creación de estos tipos de datos abstractos no conlleva grandes dificultades si se siguen los pasos de implementación. No obstante, la definición de los métodos de los TDAs requiere mayor trabajo. En el Anexo 4 se presenta una tabla con estas funcionalidades y la explicación de sus procedimientos.

4. DISEÑO DE LA SOLUCIÓN

Para implementar la red de metro, surge la necesidad de crear diferentes abstracciones de elementos mediante TDAs. De modo de ir conteniendo tipos de datos abstractos dentro de un gran TDA Subway que represente el metro y controle todos sus elementos.

Debido a la naturaleza de Java y de la programación orientada a objetos, cada TDA fue definido en una clase que represente al objeto con la abstracción de sus características y funcionalidades. A continuación, se mencionan los tipos de datos abstractos usados, y en el Anexo 3 se expone la estructura sobre cómo están constituidos: Station, Section, Line, PassengerCar, Train, Driver y Subway. Además de estas entidades mínimas, fue necesario definir estructuras como RecorridoDriverTrain, TerminalPCar y CentralPCar. Como librerías nativas de Java para trabajar estructuras se utilizaron List, ArrayList, HashMap, Map y GregorianCalendar. Sobre esta última, en el enunciado del laboratorio se indica el uso de la librería Date para trabajar con fechas, sin embargo, al usarla Gradle indicaba que estaba obsoleta (Ver Anexo 5), por lo que se decidió utilizar la librería GregorianCalendar, la que es similar a Date y también pertenece a “java.util”.

Algunos elementos importantes a destacar son que, para que un tren sea válido, debe cumplir con que sus carros extremos sean del tipo terminal y que sus carros internos sean del tipo central, dado que la estructura mínima de un tren válido consiste en dos carros del tipo terminal unidos. Para que una línea sea válida, puede ser una línea normal donde sus estaciones extremas sean del tipo terminal u combinación, pudiendo también ser del tipo mantención sólo si la estación anterior es terminal u combinación, y que desde una estación se pueda recorrer todas las otras estaciones de la línea; o puede ser una línea circular donde no tiene estaciones terminales, pero se puede regresar

a la misma estación inicial al recorrer todas las estaciones sin cambiar de sentido. Y, por último, cualquier dato de entrada incorrecto al realizar consultas, puede producir un error de ejecución.

A medida en que se iba creando el código, surgieron ciertos inconvenientes que forzaron la modificación del diagrama de clases UML de análisis (ver Anexo 2). Uno de los más notorios es el cambio casi por completo de los atributos del TDA Subway. Debido a la implementación de un menú interactivo que cargue todos los datos del metro desde archivos .txt externos, fue necesaria una estructura para almacenar varios objetos de un tipo y que sea de fácil acceso mediante una ID. Es ahí donde el uso de HashMap del paquete "java.util" fue esencial porque resolvió el problema de que el usuario no tuviera que crear variables con objetos dentro del código de la clase Main; en su lugar, sólo se crea un objeto Subway, del cual únicamente mediante métodos es operable.

Respecto a los archivos .txt, fue necesario definir una estructura sobre cómo ordenar la información, la cual se muestra en el Anexo 6. Estos archivos cargados poseen datos mayormente reales sobre el Metro de Santiago; todas las líneas, las IDs, las estaciones, las distancias entre estaciones, los modelos de trenes, las asignaciones de tren línea, fueron obtenidas de https://es.wikipedia.org/wiki/Anexo:Estaciones_del_Metro_de_Santiago y https://es.wikipedia.org/wiki/Anexo:Material_rodante_del_Metro_de_Santiago; los tipos de rieles de las líneas se obtuvieron de https://www.metro.cl/licitaciones/descarga/98b17f068d5d9b7668e19fb8ae470841; los tiempos de paradas, los conductores, la cantidad de trenes y carros son creación ficticia.

Por consiguiente, el programa consta de: 143 estaciones de metro más otras estaciones de mantención, 7 líneas de metro (considera la línea 4A), 89 trenes, 545 carros de pasajeros, 21 conductores y 60 recorridos asignados.

Todas las operaciones del sistema de metro son indicadas por consola en color rojo. Esto funciona como retroalimentación al usuario para ver el estado de las funcionalidades.

5. CONSIDERACIONES DE IMPLEMENTACIÓN

5.1. ESTRUCTURA DEL PROYECTO

El proyecto se basa en la definición de diferentes TDAs como clases, todos contenidos de alguna u otra forma en el TDA principal llamado "Subway". Donde cada sub TDA tiene sus funciones propias que facilitan la operación de estos en TDAs externos, sin preocuparnos del cómo funcionan.

5.2. BIBLIOTECAS EMPLEADAS

Fueron utilizadas sólo bibliotecas nativas de Java. Se usó FileReader y BufferdReader pertenecientes al paquete "java.io"; luego List, ArrayList, HashMap, Map, Scanner y GregorianCalendar del paquete "java.util".

5.3. INTERPRETE USADO

El programa está desarrollado totalmente en Java con el OpenJDK Eclipse Temurin 11.0.23+9 usando Gradle 8.5 y escrito en el software IntelliJ IDEA 2024.1.2 (Ultimate Edition) Build #IU-241.17011.79. Además, mediante la herramienta Gradle, se compila y ejecuta desde la terminal con comandos determinados.

6. INSTRUCCIONES DE USO

6.1. PASOS GENERALES

- Instalar el OpenJDK Eclipse Temurin 11 en el sistema computacional.
- En caso de no poseerla, descargar la carpeta “Proyecto_Espinoza_Barria_212788287” o clonarla desde el repositorio en GitHub.
- Abrir la carpeta y verificar la existencia todos archivos necesarios para la ejecución, ya sean los archivos de texto y los de código (ver Anexo 7).
- Abrir una terminal desde la carpeta fuente “Proyecto_Espinoza_Barria_212788287” (ver Anexo 8).
- Escribir y ejecutar el código “gradle.bat build” en Windows, o el código “./gradlew build” en Linux/Unix, para compilar el programa (ver Anexo 9).
- Desde la misma terminal, escribir y ejecutar el código “gradle.bat run”, o el código “./gradlew run” en Linux/Unix, para ejecutar el programa (ver Anexo 10).
- Administrar el sistema de red de metro.

En caso de no compilar mediante Gradle, se debe modificar el archivo build.gradle y copiar toda la configuración del Anexo 11.

6.2. EJEMPLOS

Se definen 3 ejemplos de ejecución de funcionalidades: cargar todos los datos del metro (ver Anexo 12), obtener la distancia de una línea (ver Anexo 13) y obtener la distancia entre estaciones de una línea (ver Anexo 14). Esos dos últimos ejemplos, consideran que los datos del metro ya fueron cargados en el programa.

6.3. RESULTADOS ESPERADOS

Se espera que los todos los 25 requerimientos funcionales se ejecuten correctamente, sin errores de compatibilidad y de análisis en las estructuras.

6.4. POSIBLES ERRORES

Considerando que siempre se ingresen parámetros validos correspondiente a los dominios solicitados en el menú. El programa no presenta errores, incluso posee varios condicionales que, al detectar parámetros inválidos, los vuelve a solicitar. Sin embargo, esto no está implementado para todas funcionalidades.

7. RESULTADOS Y AUTOEVALUACIÓN

En el Anexo 15 se presenta una tabla detallada donde se listan todas los requerimientos funcionales y su respectivo nivel de implementación. A modo general, las 25 funcionalidades se ejecutan correctamente considerando que los elementos de entrada válidos. Además, en el Anexo 16 se presenta la tabla autoevaluación de los requerimientos no funcionales.

8. CONCLUSIONES

En este último laboratorio del curso se logró implementar el 100% de los requerimientos funcionales solicitados, con lo que estoy bastante satisfecho. En comparación con el laboratorio 1, no me fue posible completarlo; y con el

laboratorio anterior que, si bien completé todos los requerimientos, había algunos errores cuando se solicitaba la estación en donde se encontraba un tren en una hora determinada, específicamente cuando este tenía más recorridos asignados a diferentes horas del día.

Para este informe no fue posible corregir elementos del anterior, debido a qué no se ha recibido la retroalimentación de este, sin embargo, en el informe 2 se corrigieron las deficiencias del informe 1.

Elementos por destacar de la POO es su concepción del objeto como la unidad, esto facilita mucho la construcción de TDAs que representen únicamente a un objeto y sus propiedades. También, los tipos de relaciones entre clases; por su naturaleza, un tipo de objeto puede contener o estar relacionado con otros, que es lo que efectivamente pasa en la realidad. Por último, el polimorfismo que implica que un objeto tiene diferentes formas y comportamientos dependiendo del contexto en que se use.

A modo personal, me gustó bastante trabajar con el paradigma orientado a objetos, quizás porque tiene elementos similares al lenguaje C y al paradigma imperativo procedural, del cual tuve mi primer acercamiento a la programación. Luego de leer varias veces el material de clases, hacer ejercicios y ver videos divulgativos en internet, logré comprender la mayoría de los diversos conceptos que el paradigma contiene y pude realizar el programa.

Siento que este tercer laboratorio fue en el que mejor la pasé programando, en comparación con los anteriores donde la documentación existente en la web para los lenguajes es menor y son paradigmas declarativos que, su misma naturaleza me obligó a olvidar todo lo que sabía de programar y empezar aprender desde cero. También, el que hubiera una paralización estudiantil ayudó a que me dedicara plenamente a desarrollar el proyecto, pero me quitó clases de materia.

Como informáticos y programadores, debemos conocer los diferentes paradigmas y modos de abordar problemas. Trabajar con variados paradigmas no sólo es importante para solucionar problemáticas con los mejores métodos posibles, sino también para abrir nuestro entendimiento y comprender que lo que es difícil de abordar desde una perspectiva puede ser fácil desde otro punto de vista.

REFERENCIAS

Booch, G. (2010). *Análisis y diseño orientado a objetos con UML.* Pearson Educación.

Departamento de Ingeniería Informática [DIINF]. (2024a). Tipo de Dato Abstracto (TDA) - Material de Clases. *Paradigmas de Programación*. Santiago, Chile: USACH.

Departamento de Ingeniería Informática [DIINF]. (2024b). Programación Funcional - Material de Clases. *Paradigmas de Programación*. Santiago, Chile: USACH.

Departamento de Ingeniería Informática [DIINF]. (2024c). Programación Orientada a Objetos - Material de Clases. *Paradigmas de Programación*. Santiago, Chile: USACH.

González, A. H. (2013). *Introducción a la Programación Orientada a Objetos (POO).*

Kuhn, T. S. (1962). *La estructura de las revoluciones científicas.*

Spigariol, L. (2005). *Fundamentos teóricos de los Paradigmas de Programación.* Buenos Aires: Facultad Regional Buenos Aires Universidad Tecnológica Nacional.

ANEXO

1.

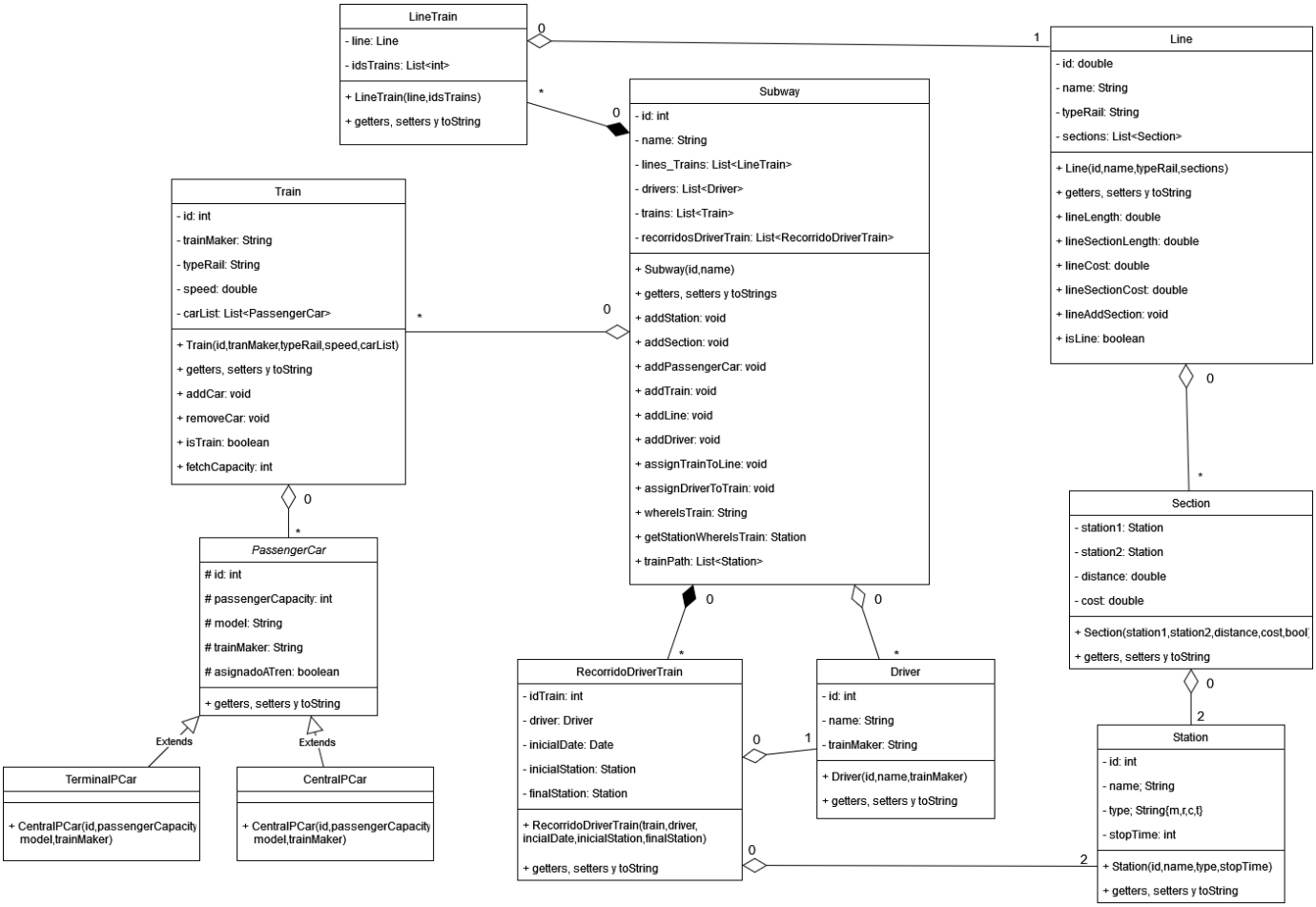

Imagen: Diagrama de clases UML de análisis.

2.

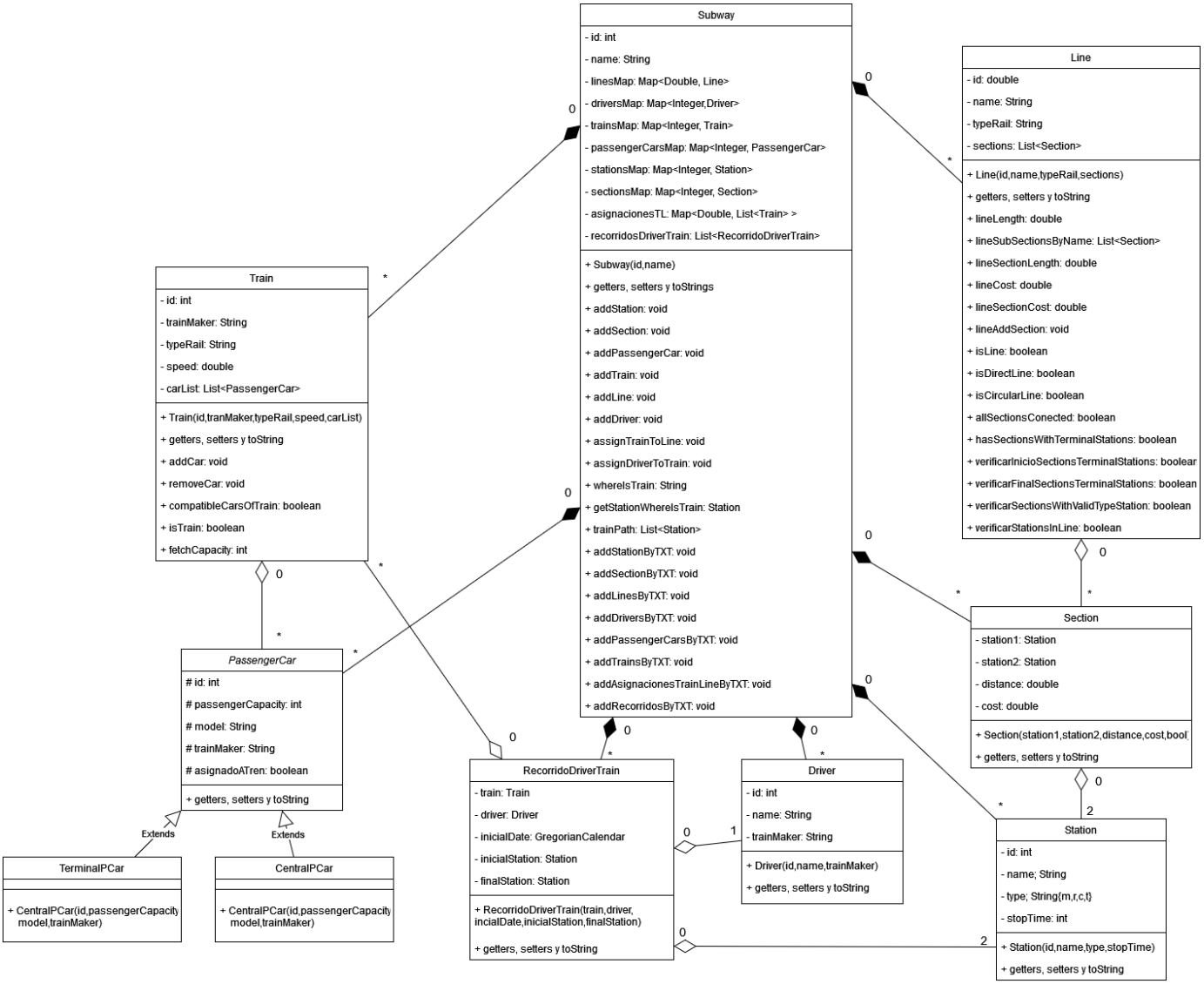

Imagen: Diagrama de clases UML posterior al desarrollo.

3.

Nombre TDA	Dato Abstracto	Atributos	Operaciones Relevantes
Station	Estación de metro.	ID, nombre, tipo (regular, mantención, combinación y terminal), y tiempo de parada en segundos.	Crear station.
Section	Sección entre estaciones.	Estacion1, Estacion2, distancia entre estaciones en Km y costo.	Crear section.
Line	Línea de metro.	ID, nombre, tipo de riel y secciones.	Crear line, agregar sections a un line, obtener datos de un line, verificar validez.
Passengercar	Carro de metro.	ID, capacidad de pasajeros, modelo, fabricante del carro, y un valor booleano que indica si está asignado a algún tren.	Crear pcar.
TerminalPCar	Carro terminal.	Sin atributos.	Crear carro terminal.
CentralPCar	Carro central	Sin atributos.	Crear carro central.
Train	Tren de metro. Conjunto de carros.	ID, fabricante, tipo de riel, rapidez máxima, y carros de pasajeros.	Crear train, verificar validez, añadir o eliminar carros, obtener datos de train.
Driver	Conductor de tren.	ID, nombre, fabricante de carros que controla.	Crear driver.
Subway	Metro.	ID, nombre, mapa de líneas, mapa de conductores, mapa de carros, mapa de trenes, mapa de estaciones, mapa de secciones, mapa de asignaciones tren línea, y recorridos para un conductor en un tren.	Crear subway, añadir y asignar elementos, obtener datos, leer documentos para cargar elementos de un subway.
RecorridoDriverTrain	Recorrido de un conductor en un tren para una fecha específica.	Tren, conductor, fecha de inicio del recorrido, estación inicial, y estación final.	Crear recorrido.

Tabla: Especificación TDAs usados.

4.

Nombre Funcionalidad	Procedimientos
lineLength - lineCost	Recorre una lista de Sections de una Line, acumulando datos de la distancia o el costo, según corresponda, entre las estaciones.
lineSectionLength– lineSectionCost	Similar a la anterior, pero primero se reduce la lista de Sections entre dos Stations específicas buscadas por su nombre.
lineAddSection	Antes de agregar una nueva Section a la Line, se verifica que no esté repetida y que sea consecutiva con la anterior.
isLine	Analiza la validez de una Line, ya sea línea circular o directa, considerando la compatibilidad de las estaciones extremas.
trainAddCar	Recorre la lista de carros para insertar uno nuevo en una posición determinada.
trainRemoveCar	Recorre la lista de carros para eliminar uno en una posición determinada.
isTrain	Verifica la validez de un Train, considerando los tipos de carros y su compatibilidad de modelos.
trainfetchCapacity	Recorre la lista de carros acumulando la capacidad de pasajeros de cada uno.
subwayAddTrain, subwayAddLine, subwayAddDriver	Verifican que los elementos a agregar no existan previamente en el Subway y luego los ingresan.
subwayToString	Recorre todos los elementos del Subway y los añade a un String en un formato legible.
subwayAssignTrainToLine	Busca un Train y una Line dentro del Subway, verifica su compatibilidad de ambos y asigna el Train a la Line en el atributo mapa de asignaciones tren línea.
subwayAssignDriverToTrain	Similar al anterior, pero se considera la compatibilidad del Driver con el Train y la Line asignada con las Stations de inicio y fin. Además crea objetos de la clase RecorridoDriverTrain y los agrega al atributo correspondiente del Subway.
whereIsTrain	Encuentra el recorrido de un Train, suma los tiempos de parada por Station y calcula el tiempo de viaje entre Sections dada la rapidez media del Tren; para determinar la última Station donde estuvo el Train en un horario específico, considerando ambos sentidos de recorridos en la Line. Admás, implementa un proceso para considerar el recorrido más cercano a la hora determina, es decir, si el tren tiene varios recorridos, el método considera el más cercano.
subwayTrainPath	Similar al anterior, pero recorre las Sections considerando las Stations que faltan por recorrer en un horario determinado.

Tabla: Explicación de procedimientos de requerimientos funcionales.

5.

```
Note: C:\Users\isaac\Desktop\Lab3_EspinozaBarria_212788287\Proyecto_212788287_EspinozaBarria\src\main\java\Main.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.
```

Imagen: Mensaje al utilizar la librería Date.

6.

Archivo	Estructura por línea de texto
Stations	ID, nombre estación, tipo, tiempo de parada
Sections	ID, ID estación 1, ID estaciones 2, distancia en Km, costo
Lines	ID, nombre, tipo de riel, IDs de secciones separadas por guion medio
PassengerCars	ID, capacidad pasajeros, modelo, fabricante, tipo {c: central, t: terminal}
Trains	ID, fabricante, tipo de riel, rapidez, IDs de carros en orden y separados por guion medio
Drivers	ID, nombre, fabricante de carros que controla
AsigancionesTrainLine	ID línea, IDs Trenes separados por guion medio
Recorridos	ID tren, ID conductor, año de fecha, mes de fecha, día de fecha, hora, minutos, ID estación inicial, ID estación final

Tabla: Estructuras de los archivos de texto del programa.

7.

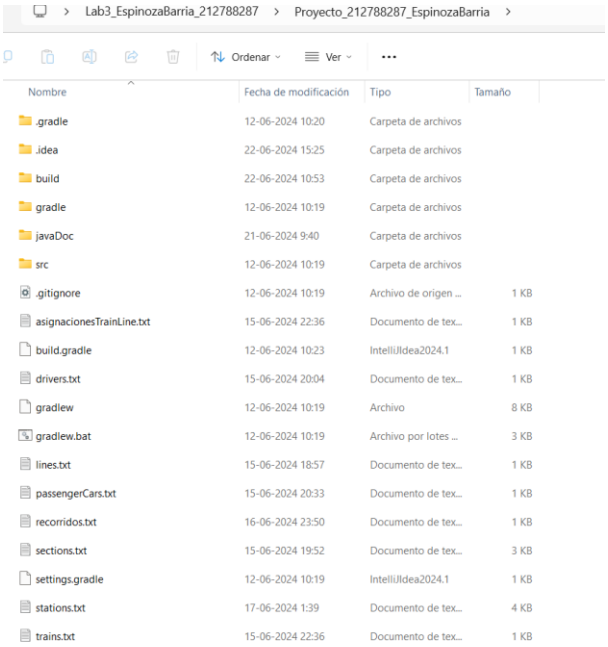

Imagen: Carpeta fuente del programa.

8.

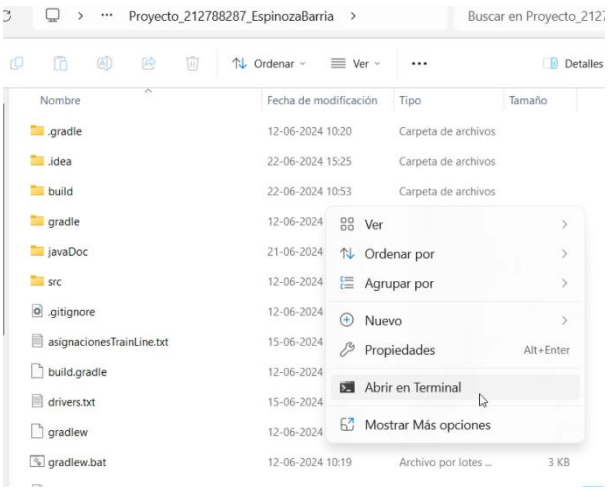

Imagen: Abrir carpeta fuente en Terminal.

9.

```
PowerShell 7.4.3
Loading personal and system profiles took 2927ms.
❯ gradle.bat build
Starting a Gradle Daemon (subsequent builds will be faster)

BUILD SUCCESSFUL in 19s
5 actionable tasks: 1 executed, 4 up-to-date
isaac ~\Desktop\Lab3_EspinozaBarria_212788287\Proyecto_212788287_EspinozaBarria main ≡*~1  20.
367s
❯
```

Imagen: Código “gradle.bat build” en la Terminal.

10.

```
PowerShell 7.4.3
Loading personal and system profiles took 1156ms.
❯ gradle.bat run

> Task :run
 ── Se creo exitosamente el objeto Subway ──




### Sistema Metro - Inicio ###
Opciones de creacion de la red de metro y simulacion de ejecucion


        1.  Cargar informacion del sistema de metro
        2.  Visualizacion del sistema de metro
        3.  Interactuar con el sistema de metro
        4.  Salir del programa

Ingresa una opcion y presiona Enter:
<=========----> 75% EXECUTING [5s]
> :run
```

Imagen: Código “gradle.bat run” en la Terminal.

11.

build.gradle en Groovy (Si realizaron el archivo de configuración build.gradle en otro lenguaje, tal como Kotlin, debe buscar el equivalente de la siguiente configuración).

```
# IMPORTANTE: SI SU PROGRAMA FALLA EN TERMINAL/CONSOLA NECESITA AGREGAR ESTO
plugins {
   id 'application'
}

repositories {
   mavenCentral()
}

# IMPORTANTE: SI SU PROGRAMA FALLA EN TERMINAL/CONSOLA NECESITA AGREGAR ESTO
run {
   standardInput = System.in
}

# Acá debe ir la ruta en donde se encuentra el main, org es la carpeta de mayor jerarquia, luego la carpeta example y luego la clase Main. OJO: NO ES QUE SU PROGRAMA TENGA QUE TENER LA CARPETA org, O LA CARPETA example, ES SOLO UN EJEMPLO.
application {
   mainClass = 'org.example.Main'
}
```

Al momento de usar de referencia la configuración anterior, eliminen los comentarios (#)

Imagen: Código a modificar dentro de build.gradle.

Fuente: Enunciado Laboratorio 3 – Paradigmas de Programación 2024/1

12.

```
 --- Se creo exitosamente el objeto Subway ---




### Sistema Metro - Inicio ###
Opciones de creacion de la red de metro y simulacion de ejecucion


    1.  Cargar informacion del sistema de metro
    2.  Visualizacion del sistema de metro
    3.  Interactuar con el sistema de metro
    4.  Salir del programa

Ingresa una opcion y presiona Enter:
1
```

Imagen: Cargar datos del Metro. Parte 1.

```
### Sistema Metro - Cargar informacion del sistema de metro ###
Definiciones estructurales de su sistema subido desde archivos


    1.  Creacion automatica del sistema de metro. Carga todos los archivos txt.
           En caso de no seleccionar esta opcion, se debera carga manualmente las estaciones, las secciones, las lineas,
           los carros, los trenes, los conductores, las asignaciones y los recorridos, EN ESTE MISMO ORDEN
    2.  Creacion de las estaciones del sistema de metro (cargar archivo stations.txt)
    3.  Creacion de las secciones del sistema de metro (cargar archivo sections.txt)
    4.  Creacion de las lineas del sistema de metro (cargar archivo lines.txt)
    5.  Creacion de los carros de pasajeros del sistema de metro  (cargar archivo passengerCars.txt)
    6.  Creacion de los trenes del sistema de metro (cargar archivo trains.txt)
    7.  Creacion de los conductores del sistema de metro (cargar archivo drivers.txt)
    8.  Creacion de las asignaciones Tren-Linea del sistema de metro (cargar archivo asignacionesTrainLine.txt)
    9.  Creacion de recorridos del sistema de metro (cargar archivo recorridos.txt)
    10. Volver al Menu Principal


Ingresa una opcion y presiona Enter:
1
```

Imagen: Cargar datos del Metro. Parte 2.

```
 --- Se agrego exitosamente el conductor al metro en addDriver ---
 --- Se creo exitosamente el objeto Driver ---
 --- Se agrego exitosamente el conductor al metro en addDriver ---
 --- Se creo exitosamente el objeto Driver ---
 --- Se agrego exitosamente el conductor al metro en addDriver ---
 --- Se creo exitosamente el objeto Driver ---
 --- Se agrego exitosamente el conductor al metro en addDriver ---
 --- Se agregaron exitosamente los conductores a la red de metro al leer el archivo ---
 --- Se verifico exitosamente si el tren es valido en isTrain ---
 --- Se verifico exitosamente si la line es valida en isLine ---
 --- Se realizo la asignacion Tren-Linea correctamente al metro en assignTrainToLine---
 --- Se agregaron exitosamente las asignaciones tren-linea a la red de metro al leer el archivo ---
 --- Se verifico exitosamente si el tren es valido en isTrain ---
 --- Se realizo la asignacion de recorrido Conductor-Tren correctamente al metro en assignDriverToTrain---
 --- Se verifico exitosamente si el tren es valido en isTrain ---
 --- Se realizo la asignacion de recorrido Conductor-Tren correctamente al metro en assignDriverToTrain---
 --- Se verifico exitosamente si el tren es valido en isTrain ---
 --- Se realizo la asignacion de recorrido Conductor-Tren correctamente al metro en assignDriverToTrain---
 --- Se verifico exitosamente si el tren es valido en isTrain ---
 --- Se realizo la asignacion de recorrido Conductor-Tren correctamente al metro en assignDriverToTrain---
 --- Se agregaron exitosamente los recorridos a la red de metro al leer el archivo ---
```

Imagen: Cargar datos del Metro. Parte 3.

13.

```
### Sistema Metro - Inicio ###
Opciones de creacion de la red de metro y simulacion de ejecucion


    1.  Cargar informacion del sistema de metro
    2.  Visualizacion del sistema de metro
    3.  Interactuar con el sistema de metro
    4.  Salir del programa

Ingresa una opcion y presiona Enter:
3
```

Imagen: Obtener distancia de una línea. Parte 1.

```
### Sistema Metro - Interactuar con el sistema de metros ###
A traves de las siguientes opciones usted puede interactuar con la red de metros cargada previamente por todos los archivos de texto.


    1.  lineLength: obtener el largo total de una linea.
    2.  lineSectionLength: determinar el largo entre una estacion origen y final.
    3.  lineCost: determinar el costo total de recorrer una linea.
    4.  lineSectionCost: determinar el costo de un trayecto entre estación origen y final.
    5.  isLine: verificar si una linea cumple con las restricciones especificadas.
    6.  Train - addCar: añade un carro de pasajeros a un tren en la posicion establecida.
    7.  Train - removeCar: remueve un carro de pasajeros de un tren en la posicion establecida.
    8.  Train - isTrain: verifica si un tren cumple con las especificaciones de los carros de pasajeros.
    9.  Train - fetchCapacity: entrega la capacidad maxima de pasajeros de un tren.
    10. Subway - whereIsTrain: determina la ubicacion de un tren a partir de una hora indicada del dia.
    11. Subway - trainPath: armar el recorrido del tren a partir de una hora especificada y que retorna la lista de estaciones futuras por recorrer.
    12. Volver al Menu Principal


Ingresa una opcion y presiona Enter:
1
```

Imagen: Obtener distancia de una línea. Parte 2.

```
Por favor, ingrese la id de la linea requerida:
4,1
 --- Se calculo exitosamente la distancia total de la linea en lineLength ---



La linea: 4.1 tiene un largo de: 7.720000000000001 Km.
```

Imagen: Obtener distancia de una línea. Parte 3.

14.

```
### Sistema Metro - Inicio ###
Opciones de creacion de la red de metro y simulacion de ejecucion


    1.  Cargar informacion del sistema de metro
    2.  Visualizacion del sistema de metro
    3.  Interactuar con el sistema de metro
    4.  Salir del programa

Ingresa una opcion y presiona Enter:
3
```

Imagen: Obtener distancia entre estaciones de una línea. Parte 1.

```
### Sistema Metro - Interactuar con el sistema de metros ###
A traves de las siguientes opciones usted puede interactuar con la red de metros cargada previamente por todos los archivos de texto.


    1.  lineLength: obtener el largo total de una linea.
    2.  lineSectionLength: determinar el largo entre una estacion origen y final.
    3.  lineCost: determinar el costo total de recorrer una linea.
    4.  lineSectionCost: determinar el costo de un trayecto entre estación origen y final.
    5.  isLine: verificar si una linea cumple con las restricciones especificadas.
    6.  Train - addCar: añade un carro de pasajeros a un tren en la posicion establecida.
    7.  Train - removeCar: remueve un carro de pasajeros de un tren en la posicion establecida.
    8.  Train - isTrain: verifica si un tren cumple con las especificaciones de los carros de pasajeros.
    9.  Train - fetchCapacity: entrega la capacidad maxima de pasajeros de un tren.
    10. Subway - whereIsTrain: determina la ubicacion de un tren a partir de una hora indicada del dia.
    11. Subway - trainPath: armar el recorrido del tren a partir de una hora especificada y que retorna la lista de estaciones futuras por recorrer.
    12. Volver al Menu Principal


Ingresa una opcion y presiona Enter:
2
```

Imagen: Obtener distancia entre estaciones de una línea. Parte 2.

```
Por favor, ingrese la id de la linea requerida:
1

Ingrese el nombre de una estacion:
Escuela Militar

Ingrese el nombre de la otra estacion:
USACH
 --- Se calculo exitosamente la distancia entre estaciones de la linea en lineSectionLength ---


En la linea: 1.0, entre la estacion: Escuela Militar y la estacion: USACH hay un largo de: 10.910000000000002 Km.
```

Imagen: Obtener distancia entre estaciones de una línea. Parte 3.

15.

Requerimientos	Grado de Implementación	Pruebas realizadas	% Pruebas exitosas	% Pruebas fracasadas	Anotación
TDAs	1	10 TDAs creados exitosamente.	100	0	
Station	1	(+10) casos.	100	0	
Section	1	(+10) casos.	100	0	
Line	1	(+10) casos exitosos agregando secciones al definir, y (+3) casos exitosos agregando secciones después.	100	0	Crea estructuras Line pero no necesariamente válidas, esto se comprueba en isLine.
lineLength	1	(+10) casos.	100	0	
lineSectionLength	1	(+10) casos.	100	0	
lineAddSection	1	(+15) casos exitosos, (+10) casos exitosos de secciones repetidas	100	0	
isLine	1	(+10) casos exitosos con líneas válidas e inválidas.	100	0	Comprueba Lines del tipo regular y circular, donde se recorran todas las estaciones, y verifica estaciones extremas.
PassengerCar	1	(+15) casos exitosos.	100	0	
Train	1	(+3) casos exitosos.	100	0	Crea estructuras Train pero no necesariamente válidas, esto se analiza en isTrain.
trainAddCar	1	(+10) casos exitosos, (3) casos exitosos de carros incompatibles.	100	0	Es false si se intenta agregar carros incompatibles de modelo.
trainRemoveCar	1	(+10) casos exitosos.	100	0	Es false si se ingresa una posición inválida.
isTrain	1	(+10) casos exitosos con trenes válidos e inválidos considerando la estructura mínima tr-tr.	100	0	Comprueba compatibilidad carros y de tipo de carro (terminales y centrales).
trainCapacity	1	(+10) casos exitosos.	100	0	
driver	1	(+5) casos exitosos.	100	0	
subway	1	(+2) casos exitosos.	100	0	
subwayAddTrain	1	(+10) casos exitosos.	100	0	No agrega Trains repetidos.
subwayAddLine	1	(+10) casos exitosos.	100	0	No agrega Lines repetidas.
subwayAddDriver	1	(+10) casos exitosos.	100	0	No agrega Drivers repetidos.
subwayToString	1	(+ 3) casos exitosos.	100	0	

subwayAssignTrainToLine	1	(+10) casos, (5) casos exitosos con trenes repetidos y/o incompatibles con el tipo de riel.	100	0	
subwayAssignDriverToTrain	1	(+10) casos exitosos, considerando compatibilidad de conductor y tren.	100	0	No hay condiciones que lo impidan asignar un tren en un horario que ya tiene de recorrido
whereIsTrain	1	(+10) casos exitosos, considerando recorridos en ambos sentidos de dirección.	100	0	Se considera el recorrido con un tiempo de inicio más cercano anterior al tiempo ingresado.
subwayTrainPath	1	(+10) casos exitosos, considerando recorridos en ambos sentidos de dirección.	100	0	Se considera el recorrido con un tiempo de inicio más cercano anterior al tiempo ingresado. En caso de que el tren ya haya terminado su recorrido entrega null.

Tabla: Autoevaluación requerimientos funcionales.

16.

Requerimientos No Funcionales	Grado de Implementación	Anotaciones
Autoevaluación	1	Está presente en este documento como también en un archivo txt presente el zip contendor del laboratorio.
Lenguaje	1	Java.
Versión	1	Se utilizó la versión 11 con el OpenJDK Eclipse Temurin 11.0.23+9.
IDE	1	Para crear el programa se usó IntelliJ IDEA 2024.
Gradle	1	Se utilizó Gradle 8.5.
Interacciones con el programa	1	Todas las interacciones con el programa son mediante teclado en la consola, a través de un menú.
Uso del paradigma	1	Se utilizó la POO, definiendo clases, relaciones, herencia, etc.
Prerrequisitos	1	Se cumplen los prerrequisitos de todas las funcionalidades.
Documentación	1	Todos los predicados están correctamente documentados en JavaDoc.
Organización	1	El laboratorio consta con cada TDA como una clase en un archivo java, donde se incluyen los requerimientos funcionales correspondientes a cada TDA.
Diagrama de análisis	1	Se incluye en el Anexo 1.
Diagrama de diseño	1	Se incluye en el Anexo 2.
Historial	1	Se utiliza GitHub para guardar versiones del avance del proyecto, donde se hicieron más de 20 commits desde hace dos semanas.

Tabla: Autoevaluación requerimientos no funcionales.