Universidad de Santiago de Chile

Facultad de ingeniería

Departamento de Ingeniería en informática

Informe Laboratorio N°3

Paradigma de programación orientada a objetos

Nombre: Jordán Arias Hurtado

Profesor: Víctor Flores Sánchez

Sección: B-2

INDICE

Introducción:

Este informe esta seccionado, comenzando por una descripción de la problemática a resolver, una descripción del paradigma empleado, y la razón de porque se eligió este, posteriormente un análisis del problema ya planteado, seguido de un diseño de solución para tratar el dilema, aspectos en la implementación de la solución, instrucciones necesarias para el uso del programa, resultados obtenidos y finalmente conclusiones pertinentes al tema. El fin de este proyecto es el de poder crear una red de metro funcional y operativa, problema el cual año a año se complejiza más, con cada nueva estación, línea, trenes que ingresan, entre otros. Además de la gestión, y control de horario de esta. Para abarcar este dilema se empleó un nuevo paradigma, el de programación orientada, implementado en el lenguaje Java.

Descripción del problema:

Este proyecto se origina desde la necesidad de poder diseñar, organizar, gestionar, y modificar una red de metro de manera eficiente. El programa en su fase final, debe poder conseguir modelar por completo una red de metro operativa, teniendo siempre en consideración, el uso del paradigma de programación orientada a objetos, para lograr una adecuada abstracción para el tratamiento del problema.

Descripción del paradigma:

El paradigma de programación orientada a objetos consiste en un paradigma de tipo imperativo que basa su abstracción en el manejo de clases y objetos, siendo este último, una instancia de la clase en si. Cada objeto tiene una identidad, ósea ser distinguido de otro. Estados, que corresponden a los atributos. Y comportamiento, refiérase a las funciones que en este paradigma se le conocen como “Métodos”, y son aplicadas directamente sobre el objeto que está trabajando. Además, las clases permiten la herencia, ósea, una clase “hija”, copia (ósea hereda), los atributos y métodos de la clase “madre”. Y las clases además tienen relaciones entre sí, la composición, siendo una relación fuerte, en la que una clase depende estrictamente de otra, mientras que la agregación es una relación débil, una clase no depende de la existencia de otra.

Todas estas cualidades permiten una mejor flexibilidad, organización, y comodidad para la manejo del código.

Análisis del problema:

Los requisitos específicos que se deben de cubrir son el de crear estaciones, que en pares componen secciones, y un conjunto de estas secciones junto a cierta información adicional crean una línea de metro. Crear vagones o carros, que la unión de estos crea un tren. Un conductor habilitado para la conducción de los trenes de cierta manufactura en específico. La creación de un subway, que reúne las líneas, trenes y conductores creados, y permite asignar un tren a una línea en particular, un conductor a un tren, si es que este esté habilitado para ese tren, y trenes a las líneas. Y un menú interactivo con el cual el usuario pueda acceder al subway para consultar información, modificar o crear líneas, estaciones, trenes, vagones y conductores. Y poder asignar conductores a trenes, y trenes a líneas, con sus respectivos horarios y recorridos. Cada una de estas entidades, corresponderá a una clase propia, con sus métodos respectivos para el correcto funcionamiento.

Diseño de la solución:

Se opto por diseñar cada TDA como una clase propia, comenzando con, Station, la unidad más básica de una línea de metro. Una vez completado, siguió la clase Section, siendo esta de composición de 2 Station, ósea una sección no puede existir sin estaciones. Despues se creo la clase Line, conteniendo en sus datos, una lista con las secciones que la componen. En esta clase se implementaron los métodos solicitados para calcular el largo y costo, tanto total como parcial de la línea, además del modificador para añadir una sección al final de la línea. Luego siguió la clase PassangerCar, que una vez finalizada, continuó con la clase Train, que con un conjunto de vagones, permite la creación de un tren. Despues se desarrollo la clase Driver, sin ninguna relación de composición o agregación inmediata. Para finalizar la construcción del metro, el modelamiento de la clase Subway, que relaciona todas las clases anteriores como agregación, sin embargo, el Subway solo funciona correctamente cuando hay al menos una línea, con un tren, y un conductor

Cabe mencionar que, se crearon algunas clases auxiliares para poder precargar una red de metro ya operativa, siendo estas LineReader, TrainReader, y DriverReader.

Y para concluir, se reunieron todas las clases anteriores en la clase Menu, que es con la que el usuario puede operar la red de metro, y esta es ejecutada a través de la clase Main.

Consideraciones de la implementación

Solo se utilizaron las librerías nativas de Java (Anexo 1), en el IDE IntelliJ IDEA 2024.1.2 (Ultimate Edition), en el sistema operativo Windows 11

Para la implementacion, este laboratorio, se estructuro en 7 TDA'S , siendo estos Station, Section, Line, PassangerCar, Train, Driver y Subway, además del Menu para la interacción con el usuario y las clases Reader auxiliares. Estos TDA's fueron necesarios para la implementación funcional. Y cada uno con sus respectivos constructores, modificadores, selectores, y métodos propios de cada uno

Instrucciones de uso

En la carpeta CodigoLab3_213170554_JordanAriasHurtado, hay que copiar la ruta de acceso, luego en la cmd utilizar el comando cd, y copiar la dirección para acceder. Posteriormente se debe utilizar el comando gradlew.bat build. Una vez se termine de construir, utilizar el comando gradlew.bat run, para ejecutarlo.

También es posible utilizar el código a través de powershell en lugar de la cmd, en este caso, repetir el mismo proceso de cd mas la ruta de acceso a la carpeta CodigoLab3_213170554_JordanAriasHurtado, luego utilizar el comando .\gradle build, y una vez se termine de construir, utilizar .\gradle run para ejecutar. No se verifico su funcionamiento en Linux.

Una vez interactuando con el menú, puede operar con las opciones que presenta, se tiene presente el caso de que el usuario introduzca opciones fuera de rango, caracteres no válidos, y vacío, tanto para seleccionar opciones, como para la creación y manipulación de la red de metro.

Resultados

Los requerimientos fueron implementados y verificado su funcionamiento en 100% de los casos.

A excepción de proyectos anteriores, este cuenta con una interacción directa con el usuario, a través del menú, y no a través de comandos

Es posible la creación de cada componente del metro por parte del usuario, y la operación con estos, siempre y cuando cumplan los requerimientos necesarios.

El menú dentro de la cmd presenta ligeros errores visuales menores, como error al interpretar la letra ñ, o alguna vocal con tilde, pero que no entorpecen el funcionamiento del codigo

El menú carga correctamente (Anexo 2), al elegir la opción 4, y acceder a las opciones de visualizar o modificar, también carga correctamente (Anexo 3). Al elegir la opción de visualizar, carga correctamente (Anexo 4), y al volver a la opción de visualizar, y elegir la opción de modificar (Anexo 5), esta también carga correctamente, solo que presenta un error como se menciono, con la letra Ñ en la palabra añadir.

Conclusiones:

Diagramas de analisis (Anexo 6) y diseño (Anexo 7).

Este paradigma, permite una comoda representación y manejo sobre los TDA, además un cómodo manejo de estos mediante las clases, y lo sencillo que es operar y trabajar sobre los mismo. Sumado a lo intuitivo que llega a ser con el tiempo, Java ha sido un lenguaje, por sobre todo cómodo.

Tomando en cuenta el trabajo realizado, se logro implementar en su totalidad lo solicitado, y asi logrando un gran entendimiento de este atractivo paradigma

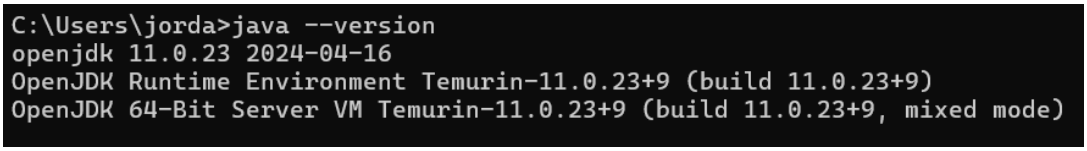
Referencias

-Programación orientada a objetos. (2021, agosto 17). IBM. https://www.ibm.com/es-es/technology/ai

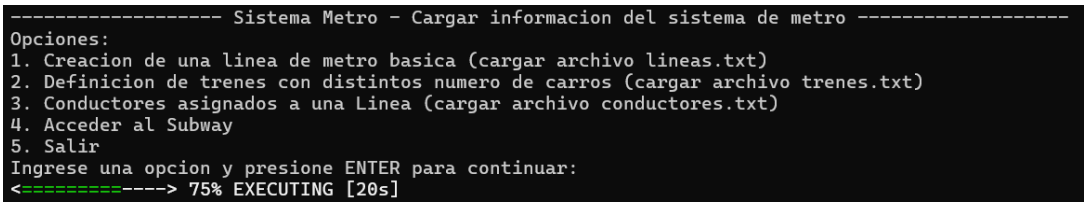
- Lopez, Y. (2022). ¿Qué es la Programación Orientada a Objetos (POO)? EDteam. https://ed.team/blog/que-es-la-programacion-orientada-a-objetos-poo

Anexos

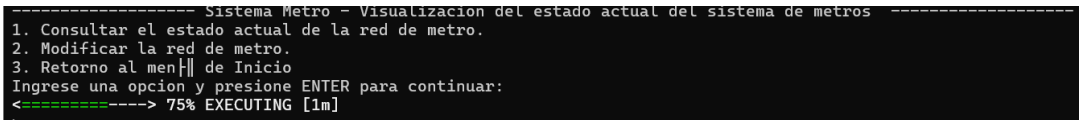
Anexo 1:



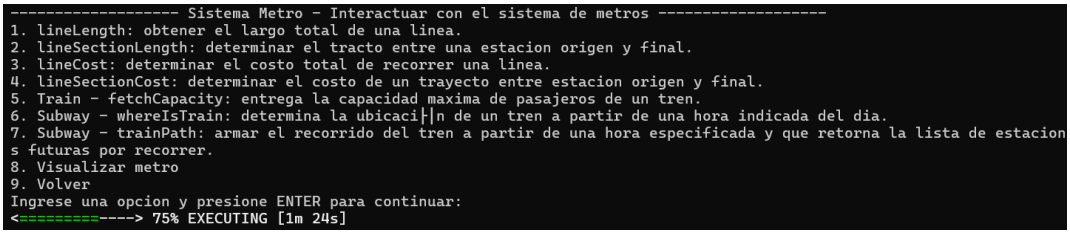
Anexo 2:



Anexo 3:



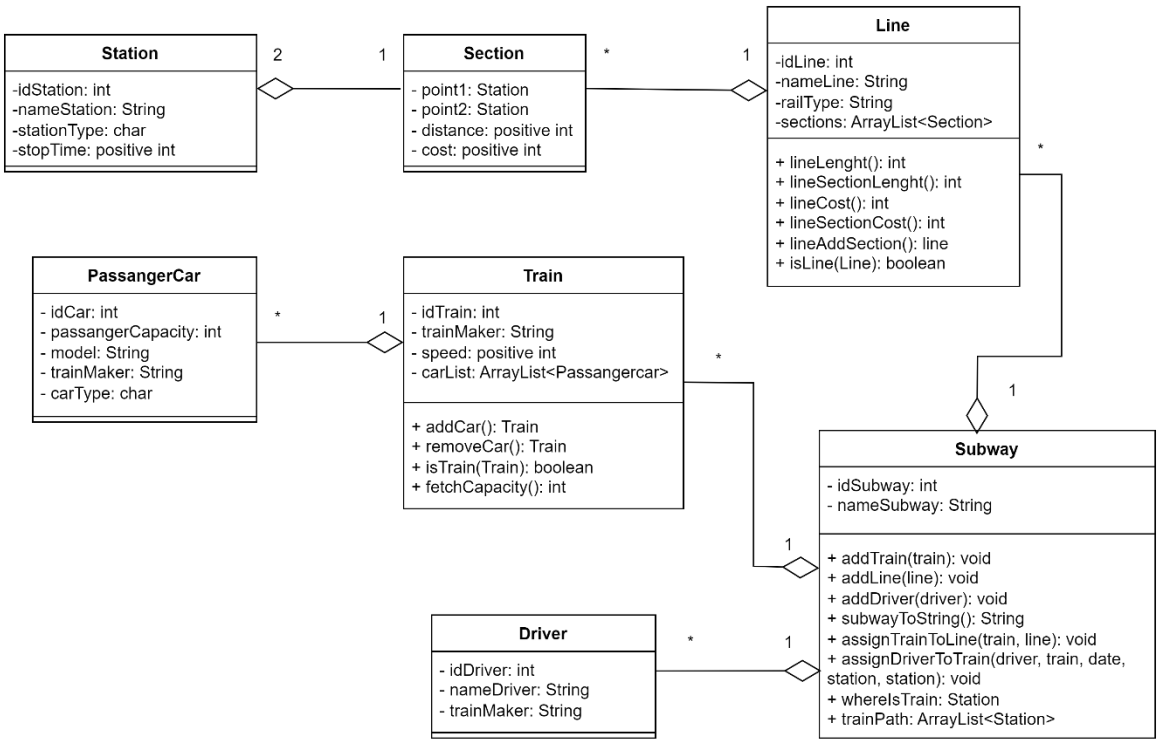
Anexo 4:



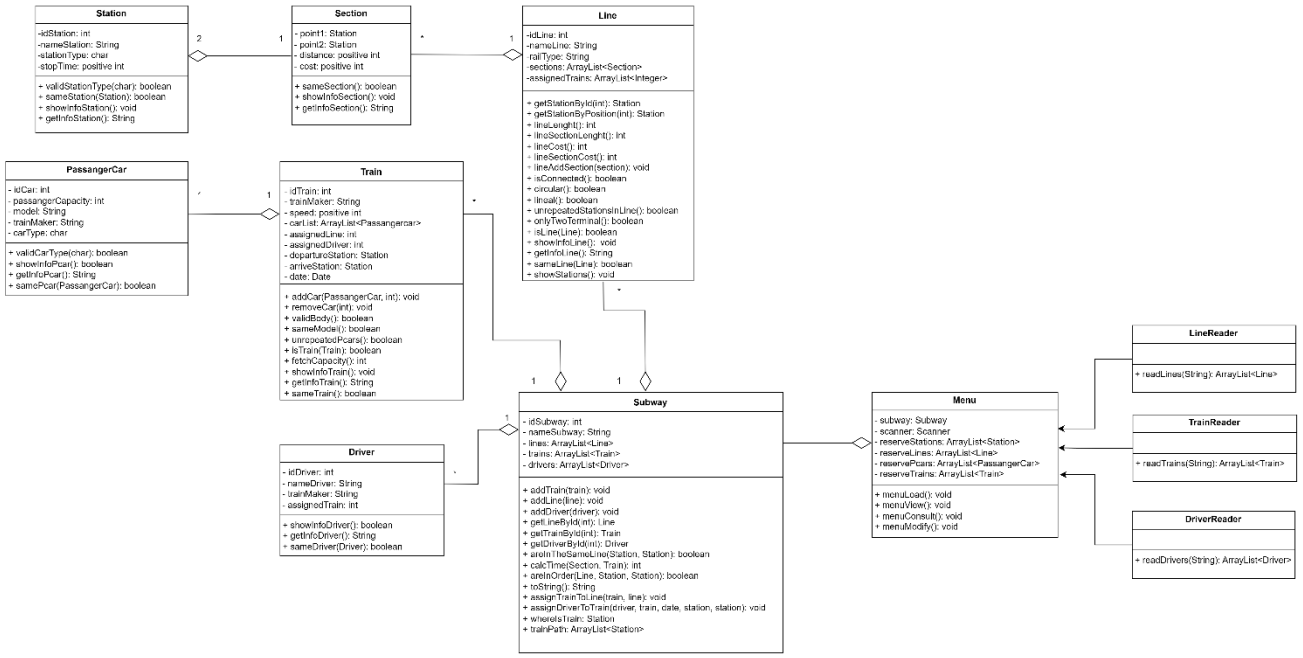
Anexo 5:

```
<=========----> 75% EXECUTING [1m 44s]
1. Crear una linea nueva
2. Crear una estacion nueva
3. A├▒adir estacion a una linea
4. Consultar si una linea es valida para ingresar a la red de metro
5. A├▒adir lineas a la red de metro
6. Quitar lineas de la red de metro
7. Crear tren nuevo
8. Crear vagon nuevo
9. A├▒adir vagones a un tren
10. Quitar vagones a un tren
11. Consultar si un tren es valido para ingresar a la red de metro
12. A├▒adir trenes a la red de metro
13. Quitar trenes de la red de metro
14. Ingresar nuevo conductor
15. Quitar conductor del metro
16. Asignar tren a linea
17. Asignar conductor a tren
18. Volver
<=========----> 75% EXECUTING [1m 53s]]
```

Anexo 6:



Anexo 7:


Station
-idStation: int
-nameStation: String
-stationType: char
-stopTime: positive int
+ validStationType(char): boolean
+ sameStation(Station): boolean
+ showInfoStation(): void
+ getInfoStation(): String
Section
- point1: Station
- point2: Station
- distance: positive int
- cost: positive int
+ sameSection(): boolean
+ showInfoSection(): void
+ getInfoSection(): String
Line
-idLine: int
-nameLine: String
railType: String
-sections: ArrayList<Section>
-assignedTrains: ArrayList<Integer>
+ getStationById(int): Station
+ getStationByPosition(int): Station
+ lineLenght(): int
- lineSectionLenght(): int
+ lineCost(): int
- lineSectionCost(): int
+ lineAddSection(section): void
+ isConnected(): boolean
+ circular(): boolean
+ lineal(): boolean
+ unrepeatedStationsInLine(): boolean
+ onlyTwoTerminal(): boolean
+ isLine(Line): boolean
+ showInfoLine(): void
+ getInfoLine(): String
+ sameLine(Line): boolean
+ showStations(): void
PassangerCar
- idCar: int
- passangerCapacity: int
- model: String
- trainMaker: String
- carType: char
+ validCarType(char): boolean
+ showInfoPcar(): boolean
+ getInfoPcar(): String
+ samePcar(PassangerCar): boolean
Train
- idTrain: int
- trainMaker: String
- speed: positive int
- carList: ArrayList<Passangercar>
- assignedLine: int
- assignedDriver: int
- departureStation: Station
- arriveStation: Station
- date: Date
+ addCar(PassangerCar, int): void
+ removeCar(int): void
+ validBody(): boolean
+ sameModel(): boolean
+ unrepeatedPcars(): boolean
+ isTrain(Train): boolean
+ fetchCapacity(): int
+ showInfoTrain(): void
+ getInfoTrain(): String
+ sameTrain(): boolean
Driver
- idDriver: int
- nameDriver: String
- trainMaker: String
- assignedTrain: int
+ showInfoDriver(): boolean
+ getInfoDriver(): String
+ sameDriver(Driver): boolean
Subway
- idSubway: int
- nameSubway: String
- lines: ArrayList<Line>
- trains: ArrayList<Train>
- drivers: ArrayList<Driver>
+ addTrain(train): void
+ addLine(line): void
+ addDriver(driver): void
+ getLineById(int): Line
+ getTrainById(int): Train
+ getDriverById(int): Driver
+ areInTheSameLine(Station, Station): boolean
+ calcTime(Section, Train): int
+ areInOrder(Line, Station, Station): boolean
+ toString(): String
+ assignTrainToLine(train, line): void
+ assignDriverToTrain(driver, train, date, station, station): void
+ whereIsTrain: Station
+ trainPath: ArrayList<Station>
Menu
- subway: Subway
- scanner: Scanner
- reserveStations: ArrayList<Station>
- reserveLines: ArrayList<Line>
- reservePcars: ArrayList<PassangerCar>
- reserveTrains: ArrayList<Train>
+ menuLoad(): void
+ menuView(): void
+ menuConsult(): void
+ menuModify(): void
LineReader
+ readLines(String): ArrayList<Line>
TrainReader
+ readTrains(String): ArrayList<Train>
DriverReader
+ readDrivers(String): ArrayList<Driver>