

Simulación de líneas de colectivos urbanas



Integrantes:

- Agustín Gigliotti Farias
- Moisés Kutnich
- Martina Liberatori

Profesores:

- Renato Mazzanti
- Gustavo Samec
- Marcos Zarate

Materia:

- Algorítmica y Programación II

Índice

Índice.....	2
Introducción.....	5
Planteo del problema.....	5
Análisis de las estructuras seleccionadas.....	5
Datos.....	5
Configuración.....	5
LectorLineas.....	6
LectorParadas.....	6
Interfaz.....	7
Simulador.....	7
Lógica.....	8
GestorSimulador.....	8
Modelo.....	8
Colectivo.....	8
Línea.....	9
Parada.....	9
Pasajero.....	9
Diagrama de clases.....	10
Implementación de la solución.....	11
Modelo.....	11
Datos.....	11
Lógica.....	11
Interfaz.....	12
Manual de funcionamiento.....	13
Errores detectados.....	16
Lotes de prueba.....	18
TestPasajeroReal.....	18
TestIndiceSatisfaccion.....	18
TestConfiguracion.....	18
Test 1: Lectura exitosa de archivo.....	18
Test 2: Archivo sin la clave esperada.....	19
Posibles mejoras.....	20
Conclusión.....	21

Introducción

Este informe describe el desarrollo de un sistema de simulación de líneas de colectivos urbanos. El sistema simula el funcionamiento de colectivos, la interacción con pasajeros, y la posterior evaluación de rendimiento y satisfacción.

Inicialmente, el sistema carga información de líneas y paradas desde archivos de texto, generando pasajeros y un colectivo por línea para un solo recorrido. Se visualizan las paradas y el ascenso/descenso de pasajeros en cada una

Planteo del problema

La gestión eficiente del transporte público urbano presenta desafíos significativos en términos de optimización de rutas, control de capacidad y satisfacción del usuario. Actualmente, la falta de herramientas adecuadas para simular y analizar el comportamiento de las líneas de colectivos dificulta la toma de decisiones informadas para mejorar el servicio. Este proyecto busca desarrollar un sistema de simulación que modele el flujo de pasajeros y colectivos en una red urbana, permitiendo evaluar la eficiencia operativa mediante métricas como la ocupación de las unidades y la satisfacción del cliente, con el fin de contribuir a la optimización del servicio de transporte.

Análisis de las estructuras seleccionadas

En el desarrollo del sistema de simulación de líneas de colectivos urbanas, se han empleado diversas estructuras de datos y Tipos Abstractos de Datos (TADs) para modelar las entidades del problema y sus interacciones. A continuación, se detalla el análisis de las estructuras seleccionadas:

Datos

Configuración

En esta clase, no consideramos necesario utilizar ningún TAD ya que su función principal es leer un valor específico, la (`cantidadPasajeros`), de un archivo de configuración. Dado que este valor es un simple número entero (`int`), una variable primitiva es la estructura de datos más directa y eficiente para almacenarlo

LectorLineas

En este módulo se utilizaron tanto estructuras del paquete `java.util` como estructuras provistas por el paquete `net.datastructures` del libro de Goodrich. Esta decisión fue para responder a las necesidades particulares de cada parte del código y a la integración con otras clases del sistema.

Por un lado, se empleó `java.util.List` y `java.util.ArrayList` para almacenar las líneas de colectivo ya que en este caso se necesitaba simplemente una colección ordenada de elementos, donde:

- No se requerían operaciones posicionales avanzada

- El uso se limita a agregar elementos al final y recorrer la lista secuencialmente.

Dado esto, utilizar la clase estándar `ArrayList` de Java resulta más simple. Además, como este listado no necesita estar estrechamente integrado con las estructuras del libro ni se le aplican algoritmos basados en `Position`, para evitar complejidad innecesaria.

Por otro lado, para representar el recorrido de paradas de cada línea (`List<Parada>`), se optó por la lista del paquete `net.datastructures`. Esta decisión se justifica porque:

- Como en otras clases los constructores utilizan las estructuras y las interfaces del libro
- Como utilizan a partir de las posiciones, a la hora de manipular los objetos como recorrer o modificar se facilita

LectorParadas

En la clase `LectorParadas` se utilizó un `HashMap<Integer, Parada>` para asociar cada identificador único con su objeto `Parada`. Esta elección se basó en la necesidad de realizar búsquedas rápidas por clave, su simplicidad y la no necesidad de compatibilidad con las estructuras del libro. Esta estructura resultó para la carga inicial del sistema, asegurando un acceso rápido a las paradas.

Interfaz

Simulador

En la clase `Simulador` se utilizaron estructuras de datos tanto del paquete `java.util` como del paquete `net.datastructures` del libro de Goodrich. Esta decisión se debió a la necesidad concreta de dividir el sistema en dos fases bien diferenciadas:

Fase de carga de datos (entrada)

Durante la lectura de archivos (`parada.txt` y `linea.txt`), se usaron estructuras de `java.util` como:

- `java.util.Map<Integer, Parada>` (implementado como `HashMap`)
- `java.util.List<Linea>` (implementado como `ArrayList`)

Fueron elegidas por su simplicidad, eficiencia y facilidad de uso al trabajar con archivos:

- Están optimizadas para inserciones rápidas y acceso directo.

- Son suficientes para representar temporalmente los datos en memoria mientras se los construye

Usar `java.util` en esta etapa permite que el código de lectura sea más corto, directo y fácil de mantener.

Fase de simulación (núcleo del sistema)

Luego de cargar los datos, se realiza una conversión explícita a los TADs del libro:

- `net.datastructures.Map<Integer, Parada>` (implementado como `UnsortedTableMap`)
- `net.datastructures.List<Linea>` (implementado como `ArrayList`)

Estas estructuras son las utilizadas por todo el núcleo del sistema, incluyendo las clases del paquete `logica`, como `GestorSimulacion`. El motivo es que el diseño general del sistema se construyó siguiendo la interfaz de TADs del libro, por varias razones:

- Permiten utilizar operaciones posicionales (`add`, `positions`, `remove`, entre otras.) y entradas clave-valor (`Entry<K, V>`).
- Ofrecen tipos de datos genéricos adaptados a estructuras académicas, ideales para practicar diseño de software con abstracciones limpias.
- Garantizan coherencia y compatibilidad con todas las demás clases del sistema que también usan `net.datastructures`.

Además, la conversión explícita entre estructuras (`java.util → net.datastructures`) deja claro en el código cuándo se está pasando de una etapa a otra, ayudando a mantener una arquitectura modular y entendible.

Lógica

GestorSimulador

La clase `GestorSimulacion` es el núcleo lógico del sistema de colectivos. Para manejar todos los elementos de la simulación (colectivos, líneas, paradas, pasajeros y estadísticas), se utilizaron exclusivamente estructuras de datos del paquete `net.datastructures` del libro de Goodrich. Esta decisión se basó en la necesidad de abstracción, claridad y eficiencia del sistema.

Durante la simulación se utilizaron las siguientes estructuras:

- `net.datastructures.List<Linea>` (implementado como `ArrayList`)
- `net.datastructures.Map<Integer, Parada>` (implementado como `UnsortedTableMap`)
- `net.datastructures.List<Pasajero>` para registrar los pasajeros procesados
- `net.datastructures.Queue<Pasajero>` (implementado como `LinkedQueue`) para administrar las colas de espera en cada parada

Estas estructuras fueron elegidas por varias razones:

- Están integradas con las interfaces del libro, lo que permite un diseño coherente con el resto del sistema.
- Permiten el uso de operaciones clave como `add`, `positions`, `remove`, `enqueue` y `dequeue`, fundamentales para simular de manera precisa el comportamiento del sistema.
- Son ideales para representar colecciones dinámicas como listas de colectivos, paradas o pasajeros en espera.

El uso exclusivo de estructuras del paquete `net.datastructures` refuerza la modularidad del sistema y permite una integración directa con las demás clases del modelo. A su vez, evita inconsistencias que podrían surgir al mezclar implementaciones de diferentes bibliotecas, y permite mantener una arquitectura uniforme orientada al diseño con TADs académicos.

La lógica implementada en esta clase incluye:

- Generación aleatoria de pasajeros con origen y destino válidos
- Control de subida y bajada de pasajeros en cada parada
- Registro de la ocupación del colectivo por tramo
- Cálculo de métricas como la ocupación promedio y el índice de satisfacción del usuario

Estas funcionalidades están directamente ligadas a las estructuras utilizadas, las cuales permiten una implementación eficiente y clara del sistema de simulación.

Modelo

Colectivo

En la clase `Colectivo`, se utilizó el TAD Lista (`List<Pasajero>`) implementado mediante `net.datastructures.ArrayList`. Esta elección permite mantener el orden de los

pasajeros, acceder por posición, recorrerlos y eliminarlos en base a condiciones específicas.

Además, se eligió la implementación del libro por su compatibilidad con otras estructuras del sistema y su integración con los algoritmos posicionales para facilitar los recorridos seguros

Esta consistencia estructural favorece el mantenimiento del código y cumple con los lineamientos académicos de la materia.

Línea

En la clase Línea se utilizó el TAD Lista (List<Parada>) con la implementación net.datastructures.ArrayList, del libro de Goodrich. Esta estructura permite mantener el recorrido ordenado de paradas de una línea de colectivo.

Se eligió la versión del libro para mantener la coherencia con el resto del sistema, aprovechar la compatibilidad con algoritmos posicionales

El uso de listas del libro permite una representación fiel del dominio del problema y una estructura flexible que puede adaptarse fácilmente a futuras modificaciones del recorrido.

Parada

En la clase Parada, se utilizó el TAD Cola (Queue<Pasajero>) implementado mediante net.datastructures.LinkedList, del libro de Goodrich

Esta estructura modela la fila de espera de pasajeros en una parada, aplicando correctamente el principio FIFO.

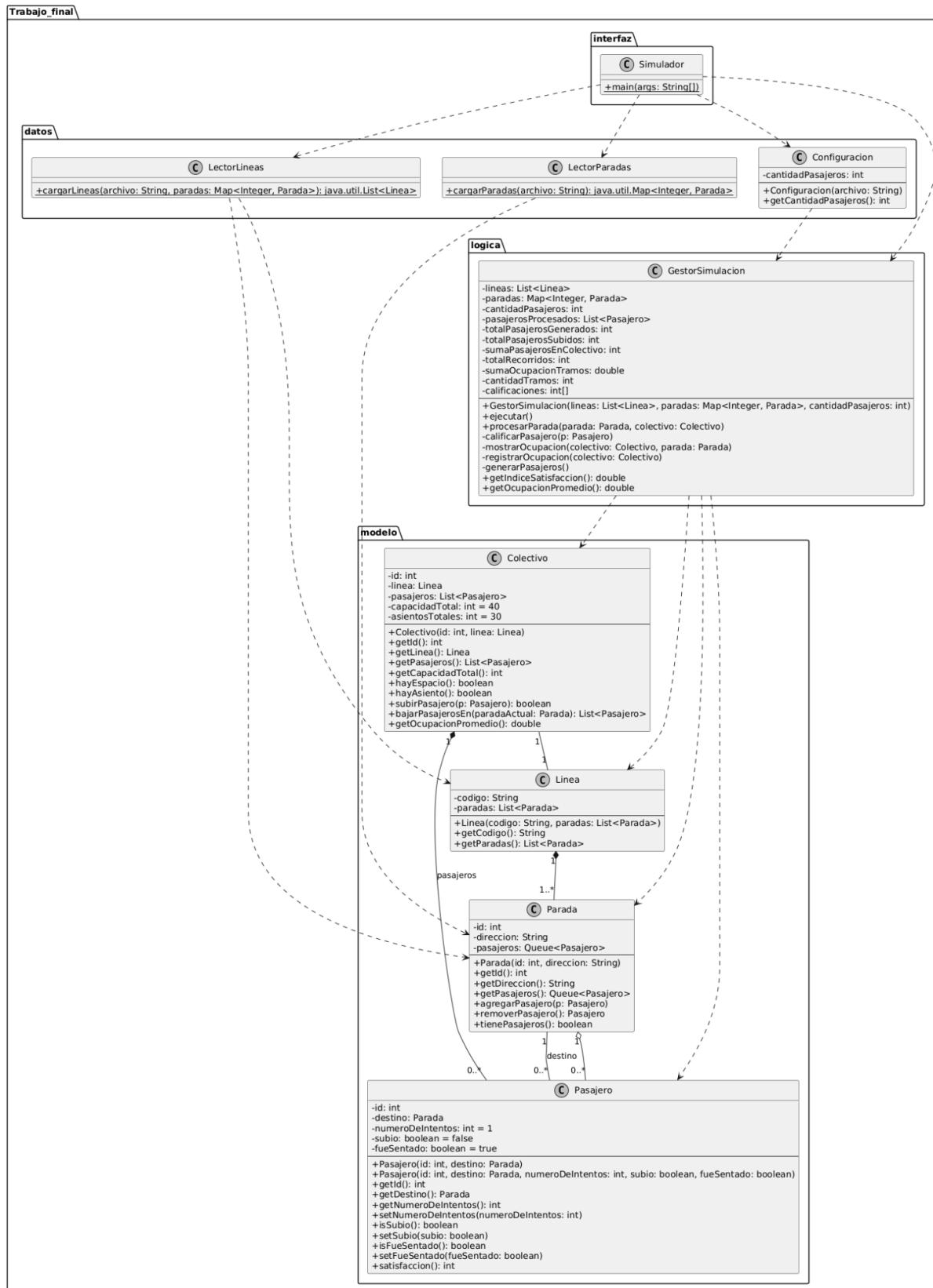
La implementación elegida se integra con el resto del sistema, ofrece una funcionalidad clara y sencilla, y cumple con los requisitos académicos del proyecto.

Pasajero

La clase Pasajero no emplea directamente estructuras de datos complejas

Su diseño está enfocado en encapsular los datos y comportamientos de cada pasajero, como su destino, si logró subir, si estaba sentado y cuántos intentos necesitó.

Diagrama de clases



Implementación de la solución

El proyecto se desarrolló respetando una arquitectura en capas, separando la lógica de negocio, el modelo de datos, la carga desde archivos, y la interfaz con el usuario. La implementación se dividió en cuatro paquetes principales:

Modelo

Contiene las clases que modelan los objetos del sistema:

- **Colectivo**: representa un colectivo, incluyendo su capacidad, los pasajeros a bordo y la ocupación por tramo.
- **Línea**: representa una línea de colectivo con una lista ordenada de paradas.
- **Parada**: representa una parada de colectivo, incluyendo una cola de pasajeros esperando.
- **Pasajero**: representa un pasajero con su parada de origen, destino y calificación del servicio.

Datos

Clases que leen los archivos externos:

- **LectorParadas**: lee el archivo de paradas y crea instancias de Parada.
- **LectorLineas**: lee el archivo de líneas y arma las relaciones con las paradas.
- **Configuración**: carga parámetros de configuración, como cantidad de pasajeros, repeticiones, etc.

Estas clases permiten modificar el formato de entrada (ej. usar binarios o base de datos en lugar de texto) sin cambiar el resto del sistema.

Lógica

Contiene la clase central del sistema:

- **GestorSimulacion:** coordina toda la simulación. Genera pasajeros, gestiona los recorridos de los colectivos, calcula el índice de satisfacción del cliente y la ocupación promedio. Utiliza listas, colas y mapas para organizar la simulación.
-

Interfaz

- **Simulador:** clase que inicia el sistema, presenta un menú de texto y permite al usuario lanzar la simulación, repetirla o salir. Muestra por consola los resultados de cada recorrido.

Manual de funcionamiento

1. Requerimientos del sistema

- **Sistema Operativo:** Compatible con cualquier sistema operativo que soporte la Máquina Virtual de Java (JVM).
- **Java Development Kit (JDK):** Versión 8 o superior.
- **Bibliotecas Externas:** El proyecto utiliza la biblioteca net.datastructures para sus estructuras de datos (List, Map, Queue). Asegurarse de que las JARs de esta librería estén incluidas en el classpath del proyecto.
- **Archivos de Configuración:** Los archivos config.properties, linea.txt y parada.txt deben estar presentes en el directorio raíz donde se ejecuta el programa o en la ruta especificada.

2. Estructura de Archivos de Entrada

El sistema se alimenta de tres archivos de texto plano para su configuración inicial:

2.1. config.properties

Este archivo define parámetros globales de la simulación. Actualmente, solo contiene la cantidad total de pasajeros a generar.

Formato:

cantidadPasajeros=<cantidad_total_de_pasajeros>

Ejemplo:

cantidadPasajeros=1000

Descripción de parámetros:

- cantidadPasajeros: Un número entero que indica la cantidad total de pasajeros que se generarán y distribuirán aleatoriamente entre las líneas al inicio de la simulación.

4.3. parada.txt

Define las paradas disponibles en el sistema de transporte, con un identificador único y su dirección.

Formato:

Cada línea representa una parada y debe seguir el formato:

<ID_PARADA>;<DIRECCION_PARADA>

Ejemplo:

1;1 De Marzo, 405;

2;1 De Marzo, 499;

...

104;Vito Roca 705;

Descripción de parámetros:

- <ID_PARADA>: Un número entero único que identifica la parada.
- <DIRECCION_PARADA>: Una cadena de texto que describe la ubicación de la parada.

Consideraciones:

- Las líneas que comienzan con # o están en blanco serán ignoradas.

2.3. linea.txt

Define las diferentes líneas de colectivo y las paradas que componen su recorrido en un orden específico.

Formato:

Cada línea representa una línea de colectivo y debe seguir el formato:

<CODIGO_LINEA>;<ID_PARADA_1>;<ID_PARADA_2>;...;<ID_PARADA_N>

Ejemplo:

L1I;88;97;44;43;47;58;37;74;77;25;24;5;52;14;61;35;34;89;

L1R;89;84;83;90;16;17;53;26;3;4;36;73;96;95;46;102;42;62;45;100;

...

L6R;66;32;39;12;22;33;20;23;25;24;5;1;

Descripción de parámetros:

- <CODIGO_LINEA>: Una cadena de texto (ej. "L1I", "L2R") que identifica de forma única la línea y su sentido (Ida/Regreso).
- <ID_PARADA_X>: Los identificadores numéricos de las paradas que conforman el recorrido de la línea, separados por ;. El orden es crucial para definir la ruta.

Consideraciones:

- Las líneas que comienzan con # o están en blanco serán ignoradas.
- Todos los <ID_PARADA_X> deben corresponder a un ID de parada válido definido en parada.txt. Si una parada no existe, será ignorada para esa línea.

3. Ejecución del Programa

Para ejecutar la simulación, se debe compilar y ejecutar la clase Simulador ubicada en el paquete Trabajo_final.interfaz.

Desde un IDE (Eclipse, IntelliJ IDEA, NetBeans, etc.):

1. Abrir el proyecto en el IDE.
2. Asegurarse de que la librería net.datastructures esté correctamente configurada como una dependencia del proyecto.
3. Ejecutar la clase Simulador (ej. clic derecho sobre Simulador.java -> Run/Ejecutar).

Una vez iniciado, el programa mostrará mensajes en la consola sobre el progreso de la simulación (procesamiento de paradas, subida y bajada de pasajeros). Al finalizar, un cuadro de diálogo (JOptionPane) mostrará un resumen de los resultados.

4. Pruebas y Escenarios de Simulación

El sistema está diseñado para ser flexible en la configuración de la simulación a través de los archivos de entrada. Para realizar diferentes pruebas o escenarios, se pueden modificar los siguientes aspectos:

- **Cantidad de Pasajeros:**
 - **Objetivo:** Evaluar el impacto de la demanda de pasajeros en la ocupación y satisfacción.
 - **Modificación:** Ajustar el valor de cantidadPasajeros en el archivo config.properties.
 - **Ejemplo de prueba:** Probar con cantidadPasajeros=500 (demanda baja), cantidadPasajeros=1000 (demanda media, como en el ejemplo), cantidadPasajeros=2000 (demanda alta).
- **Definición de Líneas y Paradas:**
 - **Objetivo:** Simular diferentes rutas, añadir o quitar paradas, o incluso crear nuevas líneas.
 - **Modificación:** Editar los archivos linea.txt y parada.txt.
 - **Ejemplo de prueba:**
 - Eliminar una parada clave para ver cómo afecta el flujo de pasajeros.
 - Añadir una nueva línea con un recorrido corto o largo.
 - Modificar el orden de las paradas en una línea para simular un cambio de ruta.
- **Capacidad de los Colectivos (requiere modificación de código):**
 - **Objetivo:** Analizar el efecto de la capacidad de los vehículos en la satisfacción y la ocupación.
 - **Modificación:** Aunque no configurable por archivo, la capacidad (capacidadTotal y asientosTotales) de los colectivos puede ajustarse directamente en la clase Colectivo.java.

- **Ejemplo de prueba:** Cambiar capacidadTotal a 30 y asientosTotales a 20 para simular colectivos más pequeños, o aumentar para simular colectivos de mayor capacidad.
- **Número de Recorridos (requiere modificación de código):**
Objetivo: Extender la duración de la simulación para observar el comportamiento a largo plazo.
 - **Modificación:** La variable cantidadRecorridos en la clase GestorSimulacion.java (ejecutar método) puede ser modificada.
 - **Ejemplo de prueba:** Aumentar cantidadRecorridos a 5 o 10 para simular más ciclos de ida y vuelta por línea.

5. Resultados de Salida

Al finalizar la simulación, el programa mostrará una ventana emergente (JOptionPane) con los resultados consolidados, y también imprimirá detalles en la consola durante la ejecución.

5.1. Salida en Consola

Durante la ejecución, la consola mostrará información detallada del proceso:

- == Recorriendo línea: <CODIGO_LINEA> ==: Indica la línea que el colectivo está simulando.
- == Recorrido número <N> ==: Indica el número de recorrido (ida/vuelta) actual de la línea.
- Parada actual: Parada{id=<ID>, direccion='<DIRECCION>'}: Muestra la parada donde se encuentra el colectivo.
- Pasajeros que bajan: [<Pasajero>, ...]: Lista de pasajeros que descienden en la parada actual.
- Pasajero que sube: Pasajero{id=<ID>, destino=<ID_DESTINO>}: Indica cada pasajero que logra abordar el colectivo.
- Ocupación del colectivo <ID_COLECTIVO> en tramo después de parada <Parada{id=ID, direccion='DIRECCION'}>: <PORCENTAJE>%: Muestra la ocupación del colectivo después de procesar la parada, antes de avanzar al siguiente tramo.

5.2. Ventana de Resultados Finales

Una vez completada la simulación, aparecerá un cuadro de diálogo con las métricas principales:

```
===== RESULTADOS DE LA SIMULACIÓN =====
Tiempo de Ejecución: X.XXXX segundos
Índice de Satisfacción: XX.XX%
Ocupación Promedio: XX.XX%
=====
```

Descripción de las métricas:

- **Tiempo de Ejecución:** Indica el tiempo total que tomó la simulación en ejecutarse, medido en segundos. Es útil para evaluar el rendimiento del programa.
- **Índice de Satisfacción:** Un porcentaje que representa la satisfacción general de los pasajeros. Se calcula en base a la cantidad de intentos que un pasajero necesitó para subir y si logró viajar sentado.
 - *Calificación 5 (Muy satisfecho):* Subió en el primer intento y viajó sentado.
 - *Calificación 4 (Satisfecho):* Subió en el primer intento pero viajó parado.
 - *Calificación 3 (Neutro):* Subió en el segundo intento.
 - *Calificación 2 (Insatisfecho):* Subió en el tercer intento o más.
 - *Calificación 1 (Muy insatisfecho):* No logró subir. (Nota: la implementación actual asigna calificación 5 si intentos == 0, lo cual puede ser un detalle a revisar si numeroDeIntentos comienza en 1).
- **Ocupación Promedio:** Un porcentaje que indica el promedio de ocupación de los colectivos a lo largo de todos los tramos recorridos durante la simulación. Refleja la eficiencia en el uso de la capacidad de los vehículos.

Errores detectados

Durante el desarrollo del sistema se identificaron posibles situaciones de error que podrían ocurrir en la ejecución. Algunas de ellas se manejan con validaciones y mensajes informativos:

- Archivos no encontrados
 - Si no se encuentra alguno de los archivos de entrada (paradas.txt, lineas.txt), el sistema lanza una excepción o muestra un mensaje de error indicando el problema.
- Formato inválido de archivos
 - Si los archivos tienen líneas mal formateadas (por ejemplo, paradas inexistentes o datos faltantes), el sistema puede ignorar esas líneas o lanzar una excepción.
 - En algunos casos, se imprime un mensaje para advertir que los datos no pudieron ser leídos correctamente.
- Paradas sin líneas o líneas sin paradas
 - Si una línea no tiene paradas asociadas, no se simula ese recorrido.
 - Si hay paradas que no pertenecen a ninguna línea, quedan aisladas y no se usan.

Lotes de prueba

Para validar el correcto funcionamiento del sistema de simulación de transporte urbano desarrollado, se implementaron pruebas unitarias utilizando JUnit. Estas pruebas permiten verificar tanto el comportamiento individual de ciertos componentes clave (como pasajeros, colectivos y configuración) como también el funcionamiento integral de la simulación. A continuación, se describen los casos de prueba implementados.

TestPasajeroReal

En este test se verifica que un pasajero real suba y baje correctamente del colectivo durante el recorrido, en el cual:

- Se comprueba que el pasajero efectivamente haya subido al colectivo.
 - Se verifica que el colectivo quede vacío una vez que el pasajero baja en su destino.
-

TestIndiceSatisfaccion

En este test comprobar que el índice de satisfacción calculado por el sistema sea válido tras ejecutar una simulación con múltiples pasajeros, en el cual

- Se asegura que el índice esté dentro del rango lógico de [0.0, 100.0].
-

TestConfiguracion

En este test se verifica que la clase Configuracion lea correctamente los parámetros desde un archivo externo .txt.

Test 1: Lectura exitosa de archivo

Se genera dinámicamente un archivo de configuración con la línea cantidadPasajeros=1234. Luego se instancia la clase Configuracion con ese archivo, en el cual:

- Se comprueba que el valor leído coincida con el esperado (1234).

Test 2: Archivo sin la clave esperada

Se crea un archivo de prueba que no contiene la clave cantidadPasajeros, sino otra clave irrelevante. Se intenta leer dicho archivo con la clase Configuracion, en el cual:

- Se verifica que la operación lanza una excepción (NumberFormatException), ya que intenta convertir null a int.

Posibles mejoras

Algunas posibles mejoras son:

- Incorporar una interfaz gráfica
 - Actualmente el sistema utiliza una interfaz por consola. Podría mejorarse utilizando bibliotecas gráficas como JavaFX o Swing para ofrecer una experiencia más visual e intuitiva para el usuario.
- Simular con horarios reales
 - Incorporar el manejo de horarios de llegada de colectivos y tiempos entre recorridos permitiría simular con mayor precisión el comportamiento real del transporte urbano. Esto también permitiría modelar retrasos y horarios pico.
- Mejora comportamiento de pasajeros
 - Los pasajeros sólo tienen origen y destino. Se podría añadir un perfil de usuario, tiempo de espera tolerado, o reacciones según experiencia previa, mejorando la simulación del índice de satisfacción.

Conclusión

El sistema simula el funcionamiento de líneas de colectivos urbanas, incluyendo el recorrido, el ascenso y descenso de pasajeros, y el análisis de indicadores como la ocupación promedio y la satisfacción.

Durante el desarrollo utilizamos Tipos Abstractos de Datos (TADs) del paquete `net.datastructures`:

- List (`ArrayList`) para manejar pasajeros, paradas y líneas.
- Queue (`LinkedList`) para representar la espera de pasajeros en cada parada.
- Map (`UnsortedTableMap`) para acceder a las paradas por su ID.

El proyecto nos ayudó a poner en práctica lo aprendido, comprender mejor el rol de los TADs y ver cómo una buena modelación puede aportar al análisis y mejora de sistemas reales como el transporte público.