

# PRÁCTICA

## Diseño y codificación

Uoc

### Información relevante:

- Fecha límite de entrega: 24 de junio.
- Peso en la nota final de Prácticas: 100%.

# Contenido

<b>Información docente</b>	<b>3</b>
Prerrequisitos	3
Objetivos	3
Resultados de aprendizaje	3
<b>Introducción</b>	<b>4</b>
Metodología en cascada o waterfall	4
Patrón Modelo-Vista-Controlador (MVC)	6
<b>Enunciado</b>	<b>7</b>
Aspectos generales a tener en cuenta	7
Ejercicio 1 - Diseño (5 puntos)	8
Ejercicio 2 - Codificación (4.5 puntos)	15
Ejercicio 3 - Vista (0.5 puntos)	20
<b>Corolario</b>	<b>22</b>
<b>Evaluación</b>	<b>24</b>
Ejercicio 1 - Diseño (5 puntos)	24
Ejercicio 2 - Codificación (4.5 puntos)	24
Ejercicio 3 - Vista (0.5 puntos)	25
<b>Formato y fecha de entrega</b>	<b>26</b>
<b>Anexo: Guía Dia</b>	<b>27</b>
Instalación	27
Guía rápida de uso	27
Escoger elementos UML para dibujar	27
Elementos básicos de dibujo	28
Definir clases y métodos abstractos	29
Definir atributos y métodos estáticos	29
Definir clases, atributos y métodos final	29
Definir una interfaz	30
Definir una enumeración	30

## Información docente

Esta actividad pretende que pongas en práctica todos los conceptos relacionados con el paradigma de la programación orientada a objetos vistos en la asignatura. La aplicación de estos conceptos se llevará a cabo con el diseño de un programa que solucione un problema dado (i.e. diagrama de clases UML) y su codificación en Java.

### Prerrequisitos

Para hacer esta Práctica necesitas:

- Tener asimilados los conceptos de los apuntes teóricos (i.e. los 4 módulos tratados en las PEC), incluyendo aquellos relacionados con los diagramas de clases UML.
- Haber adquirido las competencias prácticas de las PEC. Para ello te recomendamos que mires las soluciones que se publicaron en el aula y las compares con las tuyas.
- Tener asimilados los conocimientos básicos del lenguaje de programación Java trabajados durante el semestre. Para ello, te sugerimos repasar aquellos aspectos que consideres oportunos en la Guía de Java.

### Objetivos

Con esta Práctica el equipo docente de la asignatura busca que:

- Sepas analizar un problema dado y codificar una solución a partir de un diagrama de clases UML y unas especificaciones, siguiendo el paradigma de la programación orientada a objetos.
- Seas capaz de utilizar un *software* libre para la realización de diagramas de clases.
- Te enfrentes a un programa de tamaño medio basado en un patrón de arquitectura como es MVC (Modelo-Vista-Controlador)..

### Resultados de aprendizaje

Con esta Práctica debes demostrar que eres capaz de:

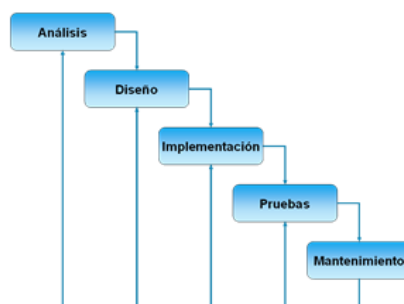
- Crear un diagrama de clases UML que explote los conceptos y mecanismos de la orientación a objetos para dar solución a un problema dado.
- Codificar un programa basado en un patrón de arquitectura como es MVC.
- Codificar en Java un diagrama de clases UML siguiendo el paradigma de programación orientada a objetos y el patrón de arquitectura MVC.
- Usar ficheros de test en JUnit para determinar que un programa es correcto.
- Usar con cierta soltura un entorno de desarrollo integrado (IDE) como IntelliJ.

# Introducción

El Equipo Docente considera oportuno que, llegados a este punto, relacionemos esta asignatura con conceptos propios de la ingeniería del software. Por ello, a continuación vamos a explicarte la metodología de desarrollo denominada “en cascada” (en inglés, *waterfall*) y el patrón de arquitectura software “Modelo-Vista-Controlador (MVC)”. Creemos que esta información, además de contextualizar esta Práctica, puede ser interesante para alguien que está estudiando una titulación afín a las TIC.

## Metodología en cascada o *waterfall*

La metodología clásica para diseñar y desarrollar software se conoce con el nombre de metodología en cascada (o en inglés, *waterfall*). Aunque ha sido reemplazada por nuevas variantes, es interesante conocer la metodología original. En su versión clásica esta metodología está formada por 5 etapas secuenciales (ver siguiente figura).



La etapa de **análisis** de requerimientos consiste en reunir las necesidades del producto. El resultado de esta etapa suele ser un documento escrito por el equipo desarrollador (encabezado por la figura del analista) que describe las necesidades que dicho equipo ha entendido que necesita el cliente. No siempre el cliente se sabe expresar o es fácil entender lo que quiere. Normalmente este documento lo firma el cliente y es “contractual”.

Por su parte, la etapa de **diseño** describe cómo es la estructura interna del producto (p.ej. qué clases usar, cómo se relacionan, etc.), patrones a utilizar, la tecnología a emplear, etc. El resultado de esta etapa suele ser un conjunto de diagramas (p.ej. diagramas de clases UML, casos de uso, diagramas de secuencia, etc.) acompañado de información textual. Si nos decantamos en esta etapa por hacer un programa basado en programación orientada a objetos, será también en esta fase cuando usemos el paradigma *bottom-up* para la identificación de los objetos, de las clases y de la relación entre ellas.

La etapa de **implementación** significa programación. El resultado es la integración de todo el código generado por los programadores junto con la documentación asociada.

Una vez terminado el producto, se pasa a la etapa de **pruebas**. En ella se generan diferentes tipos de pruebas (p.ej. de test de integración, de rendimiento, etc.) para ver que el producto final hace lo que se espera que haga. Evidentemente, durante la etapa de

implementación también se hacen pruebas a nivel local –test unitarios (p.ej. a nivel de un método, una clase, etc.)– para ver que esa parte, de manera independiente, funciona.

La última etapa, **mantenimiento**, se inicia cuando el producto se da por terminado. Aunque un producto esté finalizado, y por muchas pruebas que se hayan hecho, siempre aparecen errores (*bugs*) que se deben ir solucionando a posteriori.

Si nos fijamos en la figura, siempre se puede volver atrás desde una etapa. Por ejemplo, si nos falta información en la etapa de diseño, siempre podemos volver por un instante a la etapa de análisis para recoger más información. Lo ideal es no tener que volver atrás.

En esta asignatura hemos pasado, de alguna manera, por las cuatro primeras fases. Así pues, la etapa de análisis la hemos hecho desde el Equipo Docente. Nosotros nos “hemos reunido” con el cliente y hemos analizado/documentado todas sus necesidades (enunciados de los ejercicios de las PEC). A partir de estas necesidades, hemos ido tomando decisiones de diseño. Por ejemplo, decidimos que el software se basaría en el paradigma de la programación orientada a objetos y que usaríamos el lenguaje de programación Java. Una vez decididos estos aspectos clave, hemos ido definiendo, a partir de la identificación de objetos, las diferentes clases (con sus atributos y métodos, i.e. datos y comportamientos) y las relaciones entre ellas a partir de las necesidades del cliente confirmadas en la etapa de análisis y de entidades reales que aparecen en el problema (i.e. objetos). Para ello hemos usado un paradigma *bottom-up*. Como puedes imaginar, la etapa de diseño es una de las más importantes y determinantes de la calidad del software, ya que en ella se realiza una descripción detallada del sistema a implementar. Es importante que esta descripción permita mostrar la estructura del programa de forma que su comprensión resulte sencilla. Por ese motivo es frecuente que la documentación de la fase de diseño vaya acompañada de diagramas UML, entre ellos los diagramas de clases. Estos diagramas además de ser útiles para la implementación, también lo son en la fase de mantenimiento. En esta práctica ahondaremos en la fase de diseño mediante la elaboración de un diagrama de clases UML.

La etapa de implementación (también conocida como desarrollo o codificación) la hemos trabajado durante todo el semestre y la vamos a abordar con énfasis en esta práctica.

Finalmente, la etapa de test/pruebas la hemos tratado durante el semestre con los ficheros de JUnit que se proporcionaban con los enunciados de las PEC y con alguna prueba extra que hayas hecho por tu cuenta. Estos ficheros nos permitían saber si las clases codificadas se comportaban como esperábamos. En esta práctica también trabajarás esta fase.

Evidentemente, la metodología en cascada no es la única que existe, hay muchas más: por ejemplo, prototipado y las actualmente conocidas como metodologías ágiles. En este punto podemos decir que en las PEC hemos seguido, en parte, una metodología TDD (*Test-Driven Development*), en cuya fase de diseño se definen los requisitos que definen los test (te los hemos proporcionado con los enunciados) y son estos los que dirigen la fase de implementación. Si quieres saber más sobre metodologías para desarrollar software (donde

se incluyen los patrones) y UML, te animamos a cursar asignaturas de Ingeniería del Software. A estas metodologías de desarrollo de software se les debe añadir las estrategias o metodologías de gestión de proyectos, como SCRUM, CANVAS, así como técnicas específicas para la gestión de los proyectos, p.ej. diagramas de Gantt y PERT, entre otros.

## Patrón Modelo-Vista-Controlador (MVC)

En esta Práctica usaremos el patrón de arquitectura de software llamado MVC (Model-View-Controller, i.e. modelo, vista y controlador). El patrón MVC es muy utilizado en la actualidad, especialmente en el mundo web. De hecho, con el tiempo han surgido variantes como MVP (P de *Presenter*) o MVVM (Modelo-Vista-VistaModelo). En líneas generales, MVC intenta separar tres elementos clave de un programa:

- **Modelo:** se encarga de almacenar y manipular los datos (y estado) del programa. En la mayoría de ocasiones esta parte recae sobre una base de datos y las clases que acceden a ella. Así pues, el modelo se encarga de realizar las operaciones CRUD (i.e. Create, Read, Update y Delete) sobre la información del programa, así como de controlar los privilegios de acceso a dichos datos. Una alternativa a la base de datos es el uso de ficheros de texto y/o binarios. El modelo también puede estar formado por datos volátiles que se crean en tiempo real y desaparecen al cerrar el programa.
- **Vista:** es el conjunto de “pantallas” que configura la interfaz con la que interactúa el usuario. Cada “pantalla” o vista puede ser desde una interfaz por línea de comandos hasta una interfaz gráfica, diferenciando entre móvil, tableta, ordenador, etc. Cada vista suele tener una parte visual y otra interactiva. Esta última se encarga de recibir los inputs/eventos del usuario (p.ej. clic en un botón) y de comunicarse con el/los controlador/es del programa para pedir información o para informar de algún cambio realizado por el usuario. Además, según la información recibida por el/los controlador/es, modifica la parte visual en consonancia.
- **Controlador:** es la parte que controla la lógica del negocio. Hace de intermediario entre la vista y el modelo. Por ejemplo, mediante una petición del usuario (p.ej. hacer clic en un botón), la vista –a través de su parte interactiva– le pide al controlador que le dé el listado de personas que hay almacenadas en la base de datos; el controlador solicita esta información al modelo, el cual se la proporciona; el controlador envía la información a la vista que se encarga de procesar (i.e. parte interactiva) y mostrar (i.e. parte visual) la información recibida del controlador.

Gracias al patrón MVC se desacoplan las tres partes. Esto permite que teniendo el mismo modelo y el mismo controlador, la vista se pueda modificar sin verse alteradas las otras dos partes. Lo mismo, si cambiamos el modelo (p.ej. cambiamos de gestor de base de datos de MySQL a Oracle), el controlador y las vistas no deberían verse afectadas. Lo mismo si modificáramos el controlador. Así pues, con el uso del patrón MVC se minimiza el impacto de futuros cambios y se mejora el mantenimiento del programa. Si quieres saber más, te recomendamos ver el siguiente vídeo: <https://youtu.be/UU8AKk8Slqg>.

## Enunciado

Esta Práctica contiene 3 ejercicios. Esta se basa en el patrón MVC. Concretamente se pide:

- **Ejercicio 1 - Diseño del modelo (5 puntos):** elaboración del diagrama de clases del modelo que dé respuesta al problema planteado.
- **Ejercicio 2 - Codificación del modelo y controlador (4.5 puntos):** siguiendo el patrón MVC, implementación en Java del modelo propuesto en el diagrama de clases del Ejercicio 1 y de los `TODO` del controlador proporcionado siguiendo el paradigma de programación orientada a objetos.
- **Ejercicio 3 - Vista (0.5 puntos):** mejora de una de las vistas proporcionadas.

Para cada ejercicio debes entregar tu solución en el formato que se especifica en el propio ejercicio y/o en el apartado “Formato y fecha de entrega” de este enunciado.

## Aspectos generales a tener en cuenta

En este apartado queremos resaltar algunas cuestiones que consideramos importantes.



1. Para hacer esta práctica **deberás tener en cuenta las especificaciones que se indiquen en este enunciado y en los test. Recuerda que los test tienen prioridad en caso de contradicción.**

2. Antes de seguir leyendo, **lee los criterios de evaluación** de esta Práctica.

3. Cuando tengas problemas, relee el enunciado, busca en los apuntes y en Internet. Si aún así no logras resolverlos, **usa el foro del aula antes que el correo electrónico.** Eso sí, **en el foro no puedes compartir código.**

4. **La realización de la Práctica es individual.** Si se detectan indicios de plagio o el uso de herramientas de inteligencia artificial, el Equipo Docente se reserva el derecho de contactar con el estudiante para aclarar la situación. Si finalmente se ratifica la falta de originalidad y autoría, la asignatura quedará suspendida con un 0 y se iniciarán los trámites para abrir un expediente sancionador tanto al estudiante que ha copiado como al que ha facilitado la información.

5. **La fecha límite indicada en este enunciado es inaplazable.** Puedes hacer diversas entregas durante el período de realización de la Práctica. El/La PDC corregirá la última entrega. **Pasada la fecha límite no se aceptarán entregas.**



## Ejercicio 1 - Diseño (5 puntos)

Antes de empezar con este ejercicio debes tener en cuenta los siguientes aspectos:



1. En este ejercicio se pide el diagrama de clases del modelo. No existe una solución única, así que no te preocupes.
2. Para realizar este ejercicio debes utilizar el editor de diagramas [Dia](#) o [Drawio](#). Para Dia te damos una breve guía con este enunciado (ver anexo).
3. Como en el Ejercicio 2 codificarás en Java, el diagrama de clases UML que hagas debe ser para dicho lenguaje. Así pues, si algún atributo/método usa alguna clase/interfaz de la API de Java, entonces su nombre debe aparecer como si fuera un tipo primitivo (no hay que hacer una relación binaria, agregación, etc.). En cambio, si la clase/interfaz de Java forma parte de una relación de herencia, sólo es necesario que la caja contenga su nombre.
4. El enunciado de este ejercicio puede contener información que no sea necesaria para el diseño del programa a nivel de diagrama de clases. Cuando hablamos con un cliente, éste nos da información que puede abordarse en diferentes fases del producto, p.ej. diseño, codificación, etc. Hay que ser capaz de discernir cuándo una información es útil y cuándo no para cada fase.
5. Debes entregar un fichero editable de tipo `.dia` o `.drawio` con el diagrama de clases UML que contiene la solución que propones para dar respuesta a las necesidades y especificaciones del problema planteado.
6. Un fichero `.png` que represente en formato imagen el diagrama de clases UML que contiene el fichero editable anterior.

A continuación describimos el problema para el cual debes dar una solución en forma de diagrama de clases UML:

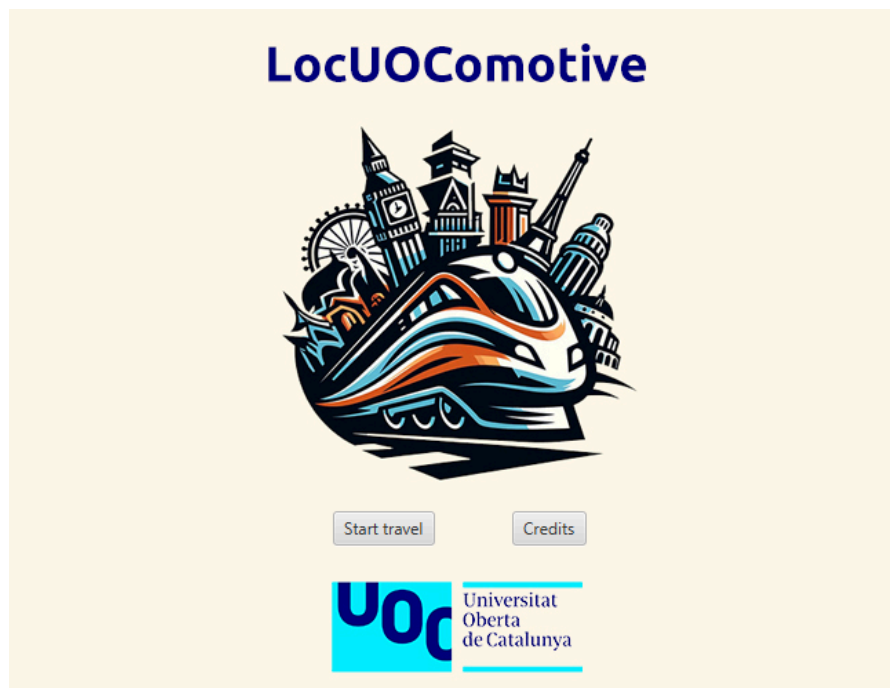
Este semestre queremos realizar una aplicación de gestión de viajes de tren. Dicha aplicación nos permitirá viajar por diferentes ciudades de Europa y poder tener un registro de todos los viajes realizados.

El prototipo que se va a desarrollar solo contiene 8 estaciones de tren, un total de 8 rutas y 8 modelos de tren distintos. No obstante, podéis añadir nuevos elementos en la aplicación si así lo deseáis para hacer más pruebas o simplemente por gusto al final del desarrollo.

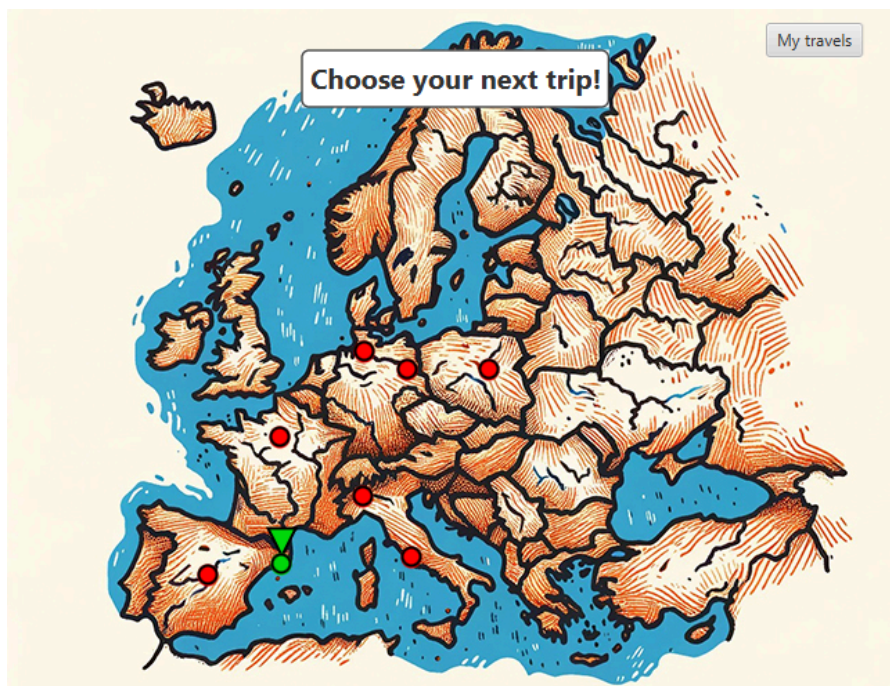
A continuación, os mostramos un par de capturas de pantalla para poder tener una idea inicial de la aplicación que vamos a diseñar y codificar.



Pantalla de inicio



Pantalla de selección del próximo destino del viaje en tren



*Las imágenes utilizadas para desarrollar esta aplicación han sido generadas con inteligencia artificial.*

Como se puede observar en la segunda imagen, los puntos corresponden a las estaciones de tren que hay repartidas por Europa. Estos puntos son botones pulsables que permiten seleccionar el siguiente destino al que vamos a viajar. Asimismo, el punto verde (exclamación) corresponde a la ubicación actual del usuario que va a efectuar la compra del viaje.

Para poder representar dicha situación, para cada estación es necesario almacenar su identificador, que debe ser único, el nombre de la estación, la ciudad donde se encuentra, el año de apertura, el tipo de estación que es según la ubicación de sus andenes (por encima del nivel del suelo, debajo del suelo o elevados) y sus coordenadas x e y en el mapa de la aplicación. Además, como la aplicación puede mostrar la imagen de la estación, es necesario almacenar el nombre del fichero JPG de la imagen de la misma. Toda la información que carga la aplicación sobre las estaciones la podéis encontrar en el fichero `stations.txt` en el directorio `/resources/data/`.



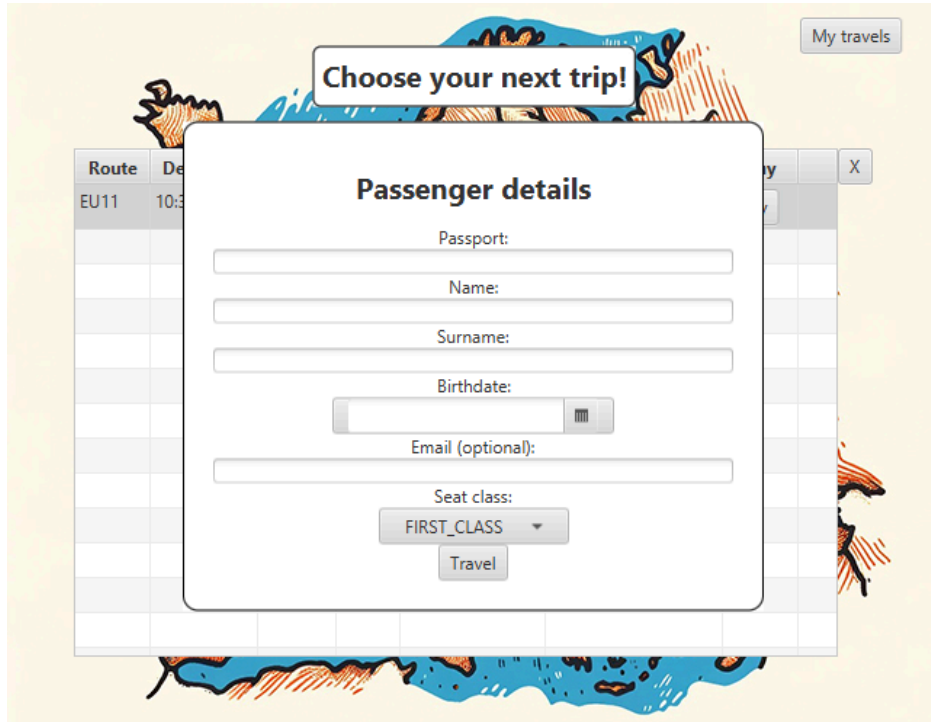
**Pista:** Considerad la sobrescritura de algunos métodos como `toString` durante el diseño y la codificación de las clases si consideráis que os pueden ser de utilidad para simplificar la implementación.

Como es lógico, la aplicación también guarda información sobre los trenes que hay disponibles. En este caso, la información relevante es el identificador único del tren, su modelo y el conjunto de vagones que tiene. Al igual que pasa con los trenes actuales de larga distancia, un tren puede tener uno o más vagones dedicados al servicio de restauración. Por lo tanto, los vagones de restauración (también llamados cafetería) no pueden disponer de asientos a la venta. En cambio, para los vagones de pasajeros es importante almacenar todos los asientos y tener conocimiento de qué pasajero va en cada asiento.

Además, no todos los vagones de pasajeros son de la misma categoría, ya que en total hay 3 clases de vagones, ofreciendo de esta manera un viaje en primera, segunda y tercera clase (`FIRST_CLASS`, `SECOND_CLASS` y `THIRD_CLASS`). Los de primera clase son los que menos asientos pueden tener por vagón, menos de 20; los de segunda, a partir de 20 y menos de 50; y los de tercera clase, que pueden tener 50 o más asientos por vagón. Todos los vagones, incluyendo los de restauración deben estar identificados con la letra C seguido de un número que va incrementando a medida que se van añadiendo vagones al tren. Por ejemplo, el primer vagón tendría como número "C1", el segundo "C2" y así sucesivamente. Toda la información que carga la aplicación sobre los trenes y sus vagones la podéis encontrar en el fichero `trains.txt` en el directorio `/resources/data/`.

Para terminar con los elementos principales de la aplicación, también es necesario almacenar la información referente a los pasajeros. Para comprender mejor la información que debe almacenarse, se proporcionará una captura de pantalla del formulario que contiene la aplicación.

### Pantalla para introducir los datos del pasajero y el tipo de asiento



Como se puede observar, para cada pasajero se debe almacenar su pasaporte, su nombre, sus apellidos, su fecha de nacimiento y, opcionalmente, un correo electrónico de contacto. Los campos obligatorios (todos menos el email) tienen que ser validados y no pueden ser nunca `null`, y en el caso de las cadenas de caracteres estas no pueden contener un texto vacío. En el caso del email (único campo opcional), en caso de ser informado, tiene que tener un formato de email correcto. Es decir, puede ser una cadena de caracteres vacía si no se informa, pero si contiene algún valor, este siempre debe ser un email válido. En este caso, se define un email válido bajo las siguientes condiciones:

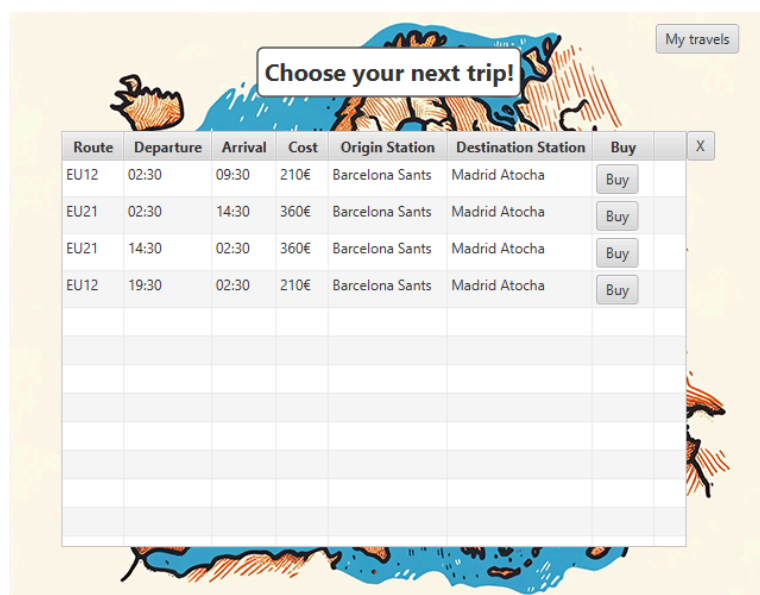
- Antes del @ debe haber como mínimo un carácter. Los caracteres permitidos son letras del alfabeto inglés (mayúsculas y minúsculas, sin caracteres como la ç o la ñ), números, puntos (.), guiones bajos (\_) o guiones (-).
- Debe contener siempre un símbolo @.
- Después del @ debe haber un texto formado como mínimo por un carácter. Los caracteres permitidos son letras del alfabeto inglés (solo en minúsculas, sin caracteres como la ç o la ñ), números, puntos (.) y guiones (-). El texto que va después del @ representa el nombre del dominio (o subdominio) del email.
- Después de este último conjunto debe contener una extensión de dominio, que siempre debe empezar con un punto (.) seguido 2 o 3 caracteres en minúscula.

En caso de que alguno de los campos sea incorrecto, debe devolver una excepción con un mensaje personalizado que se deberá mostrar por pantalla. Esta última parte (la de vincular lo que devuelve una función con el hecho de mostrarlo por la pantalla) ya viene codificada con el enunciado, pero es importante que seáis vosotros quien defináis estos mensajes.

A continuación, se explicará cómo funciona el sistema de compra a nivel de entidades del modelo.

Cuando un pasajero quiere comprar un billete de tren, debe poder ver qué rutas de tren y qué horarios disponibles hay en la estación donde se encuentra. Os recordamos que la aplicación siempre se plantea como si el pasajero se encontrara físicamente en la estación marcada de color verde. Por lo tanto, cuando selecciona otra estación a la que quiere viajar en el mapa, debe poder consultar todos los horarios de todas las rutas de tren que pasan por esa estación. Concretamente, las rutas están identificadas mediante un identificador único y deben tener las referencias de todas las estaciones por las que pasan, siendo la primera estación que almacenen la estación origen de la línea y, la última, la estación donde termina la línea. Toda la información que carga la aplicación sobre las rutas y los horarios la podéis encontrar en el fichero `routes.txt` en el directorio `/resources/data/`. En este archivo, veréis que el primer valor corresponde al identificador de la ruta, el que hay después del primer `|` corresponde al identificador del tren que hace esta ruta y, seguidamente, contiene los identificadores de todas las estaciones por las que pasa en orden y, además, los distintos horarios en los que está físicamente en esa estación (es decir, la hora de llegada a esa estación y, a la vez, la hora de salida que continúa el recorrido). En vuestro caso, veréis que todas las rutas tienen 2 horarios.

### Pantalla para poder seleccionar el horario para viajar a otra estación



Route	Departure	Arrival	Cost	Origin Station	Destination Station	Buy	X
EU12	02:30	09:30	210€	Barcelona Sants	Madrid Atocha	Buy	
EU21	02:30	14:30	360€	Barcelona Sants	Madrid Atocha	Buy	
EU21	14:30	02:30	360€	Barcelona Sants	Madrid Atocha	Buy	
EU12	19:30	02:30	210€	Barcelona Sants	Madrid Atocha	Buy	

Siguiendo con el caso de uso de compra de un billete, una vez el usuario ha seleccionado el horario para viajar a una estación (i.e. hace click en el botón `Buy`), aparece el formulario que se ha mostrado anteriormente, en el cual el usuario introduce sus datos y pulsa en el botón `Travel`. Si todos los datos son correctos, se considera que el usuario compra correctamente el billete y, además, se simula el viaje a la estación deseada.

Es importante tener en cuenta que los billetes de tren deben ser almacenados para poder ser consultados en el historial y, además, cada pasajero debe poder consultar sus billetes adquiridos. De cada billete, nos interesa saber el propietario de ese billete, la referencia del asiento seleccionado, su precio y la información del viaje (la hora y la estación de salida, la hora y la estación de llegada, y el asiento en el que se viajó —el cual se identifica por el número del vagón seguido de un guión y el número del asiento dentro del vagón).

Además, la asignación de asiento se produce durante el proceso de compra, eligiendo siempre el primer asiento disponible empezando por el primer vagón y recorriendo todos los asientos por número de asiento. Eso sí, es importante que el asiento asignado sea del mismo tipo que el pasajero especificó durante el formulario. Es decir, si el usuario especificó que quiere un asiento en primera clase, se deberá asignar el primer asiento que cumpla con dicha condición.

Una vez el pasajero adquiere el billete, automáticamente se simula el viaje y se encontrará en la estación de destino. Por lo tanto, el mapa debe actualizar la estación en la que se encuentra, dejando de color rojo y con un botón pulsable la estación de origen y marcando con la exclamación verde la estación de destino, ya que es la estación en la que se encuentra en ese momento el pasajero. Asimismo, si el pasajero pulsa en el botón de arriba a la derecha con el texto `My travels`, debe aparecer la información del billete como se observa en la siguiente captura de pantalla.

[illegible]



Para simplificar el funcionamiento de la aplicación, se considerará que todos los pasajeros bajan del tren cuando lo hace el usuario de la aplicación. Por lo tanto, una vez se llegue a la estación de destino, todos los asientos del tren deben quedar disponibles de nuevo para otro viaje.

Finalmente, como habréis observado en la pantalla del formulario, la aplicación solicita los datos del pasajero en cada compra. No obstante, internamente no se deben almacenar pasajeros duplicando su pasaporte. En caso de que se introduzca un pasaporte que ya esté registrado en la aplicación, debe coger su referencia y actualizar los datos guardados por aquellos nuevos que haya recibido durante ese proceso de compra.



A la hora de crear vuestro diagrama de clases, podéis añadir tantas clases y asociaciones como queráis siempre que consideréis que permitan representar de una mejor manera el escenario descrito. Además, también podéis crear las enumeraciones e interfaces que necesitéis.

## Ejercicio 2 - Codificación (4.5 puntos)

Este ejercicio se divide en 2 partes:

- Codificar el diagrama de clases para el modelo que has propuesto en el Ejercicio 1.
- Codificar los `TODO` del controlador que te proporcionamos en el fichero `.zip`.

Antes de empezar con este ejercicio debes tener en cuenta los siguientes aspectos:



1. Para hacer este ejercicio **deberás tener en cuenta las especificaciones que se indiquen en este enunciado y en los test. Los test tienen prioridad en caso de contradicción.**

2. Puedes usar cualquier clase, interfaz y enumeración que te proporcione la API de Java. Sin embargo, **no puedes añadir dependencias** (i.e. librerías de terceros) que no se indiquen en este enunciado.

3. En el controlador no podéis modificar las firmas de los métodos proporcionados con el enunciado. No obstante, podéis codificar métodos auxiliares, pero **bajo ningún concepto se pueden modificar los que vienen con el enunciado.**

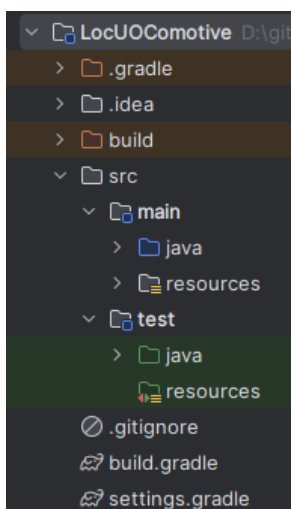
4. **Los test proporcionados no deben modificarse.** Asimismo, si se detectan trampas, p.ej. *hardcodear* código para superar los test, la nota final será 0.

### Entorno

Para esta práctica utiliza el siguiente entorno:

- JDK  $\geq 21$ .
- IntelliJ Community.
- Gradle, quien descargará las dependencias necesarias para el proyecto.

### Estructura general del proyecto



Si abres el `.zip` que se te proporciona con este enunciado, encontrarás el proyecto `LocUOComotive`. Si lo abres en IntelliJ, verás la estructura que se muestra en la siguiente imagen. De dicha estructura cabe destacar:

**src:** es el proyecto en sí, el cual sigue la estructura de directorios propia de Gradle (y Maven).

En `src/main/java` verás tres paquetes llamados `model` (que puedes dividir en los subpaquetes que consideres), `view` y `controller`. Lo hemos organizado así porque, como hemos comentado, usaremos el patrón MVC.



En `src/main/resources` encontrarás los estilos y pantallas que se utilizan en la vista gráfica del juego, así como los ficheros de configuración de la aplicación.

Por su parte, `src/test/main` contiene los ficheros de test JUnit. Asimismo, en `src/test/resources` encontrarás ficheros de configuración de la aplicación que son utilizados para testear el programa.

**build.gradle:** contiene toda la configuración necesaria de Gradle. En él hemos definido tareas específicas para esta Práctica con la finalidad de ayudarte durante su realización.

## Modelo

Dentro del package `edu.uoc.locuocomotive.model` codifica el diagrama de clases que has propuesto en el Ejercicio 1.

## Controlador

El controlador es quien maneja la lógica del negocio. En este caso, la lógica de la aplicación. Es decir, **el controlador es el responsable de decidir qué hacer con la petición que ha realizado el usuario desde la vista. Lo habitual es que el controlador haga una petición al modelo.**

En el paquete `controller` del proyecto verás una clase llamada `LocUOComotiveController`. Ésta es la clase controladora de la aplicación (un programa puede tener varias clases controladoras).



Desde los métodos del controlador con el comentario `//TODO` deberás usar los elementos definidos en el modelo. Las firmas de los métodos proporcionados con el enunciado no pueden ser modificadas. Podéis codificar funciones auxiliares adicionales, pero **bajo ningún concepto se pueden modificar los que vienen con el enunciado.**

Igualmente, hay métodos que ya os damos completamente codificados y que no debéis modificar, p.ej. `loadStations`.

Para esta clase queremos dar unas indicaciones adicionales para algunos métodos que debes codificar. Antes, habréis notado que el controlador no tiene ningún atributo y muchos métodos solo reciben y devuelven tipos de datos del lenguaje Java como `String`, es decir, no manejan instancias del modelo como estaciones u otros. Por lo tanto, no solo tenéis que implementar los métodos, si no que **tenéis que declarar todos los atributos necesarios** para el correcto funcionamiento de la aplicación.

### LocUOComotiveController (constructor)

El constructor del controlador debe inicializar todos los atributos necesarios para el correcto funcionamiento de la aplicación. Concretamente, la aplicación necesita almacenar las **estaciones, las rutas, los trenes y todos los billetes adquiridos**. Para todos estos datos, se

recomienda utilizar el tipo de dato `ArrayList`. Además, también debe poder almacenar todos los `pasajeros` que se hayan registrado en la aplicación. Para este caso, se recomienda utilizar una estructura de tipo `HashMap`.

Finalmente, **el constructor** también debe cargar los datos de los tres archivos mencionados anteriormente (`trains.txt`, `stations.txt` y `routes.txt`, en este orden). Además, **debe asignar como estación actual** (en la que se encuentra el pasajero usuario de la aplicación) **la primera estación que haya cargado del archivo `stations.txt`**, que en el fichero que os damos corresponde a la de Barcelona.



**Pista:** revisad los métodos ya implementados con el enunciado para simplificar la codificación de esta funcionalidad.

### ✓ **addStation**

Este método debe registrar la estación en la aplicación con los datos que recibe por parámetro.

### ✓ **addRoute**

Este método debe añadir una ruta a partir de la información recibida por parámetros. Para cada ruta hay que añadir todos los horarios para las estaciones por las que pasa. Es decir, las rutas registradas tienen que tener acceso a la referencia del horario por el que pasan por cada estación. Cuando se invoca este método, se presupone que el tren que hace la ruta así como las estaciones por las que pasa ya han sido añadidos a la aplicación previamente.

Los horarios recibidos tienen el siguiente formato:

```
stationId|hh:mm, hh:mm, ...]
```

Por lo tanto, deberéis utilizar el método `split` para poder separar los elementos como se ha hecho en otros métodos que podéis tomar de referencia.

### ✓ **addTrain**

Debe añadir un tren a la aplicación con los valores recibidos por parámetro. Es decir, no solo debe crear un tren, sino que también debe preparar todos los vagones y los asientos de cada uno de ellos.

### ✓ **getStationsInfo**

Debe devolver un `List` de Java de tipo `String` que contenga toda la información de todas las estaciones, en el siguiente orden:

```
id|name|city|image|x|y
```

### getSeatTypes

Debe devolver un `array` de Java de tipo `String` que contenga los nombres de los tipos de asientos que se hayan definido previamente en el modelo. Se recuerda que los tipos de asientos pueden ser de primera, segunda o tercera clase.

### getRoutesByStation

Recibe el identificador de una estación y debe devolver una `List` con todas las rutas con todos los horarios que tiene esa estación. Esta `List` debe estar ordenada según la hora de salida del tren. Cada elemento de la `List` será un `String` con los datos siguientes respetando el formato que se muestra a continuación:

```
departureTime|arrivalTime|routeId|ticketCost|departureStationId|
arrivalStationId|departureStationName|arrivalStationName
```

Asimismo, una de las informaciones que debe devolver es el coste del billete (elemento marcado en rojo). Para calcularlo, se aplica una fórmula donde se multiplican el número de horas completas de viaje por una constante (30€/hora completa de viaje). Por ejemplo, si un viaje de Barcelona a París tiene una duración de 3 horas y media, el coste será de 90€.

### addPassenger

Registra un pasajero en la aplicación con los datos recibidos por parámetro. Además, en caso de encontrar un pasaporte ya registrado (i.e. mismo pasaporte), no debe crear ningún nuevo registro, sino que debe actualizar los datos del pasajero con los que ha recibido en este último proceso de compra. En caso de que surja cualquier error, se debe lanzar una excepción.

### createTicket

Es el método que se encarga de crear un nuevo billete a partir de los datos recibidos. Para ello, deberá encontrar el primer asiento disponible en el tren de esta ruta y asignar el pasajero. Además, es el método encargado de actualizar la estación en la que se encuentra el pasajero, viajando a la estación destino y vaciar el tren de otros pasajeros. En caso de que surja cualquier error, se debe lanzar una excepción.

### buyTicket

Es el método que se encarga de gestionar todo el caso de uso completo de la compra de un billete. Por lo tanto, debe registrar el pasajero en la aplicación en caso de que no lo esté (o actualizar sus datos si ya existe su pasaporte) y crear el billete con las mismas especificaciones que se han especificado para el método anterior. En caso de que surja cualquier error, se debe lanzar una excepción. El mensaje de la excepción será mostrado por la pantalla como un error.

### getAllTickets

Debe devolver una `List` de Java de tipo `String` con la información de cada uno de los billetes registrados en la aplicación. Cada elemento de la list (i.e. un billete), será un `String` con el siguiente formato e información:

```
routeId|departureTime|departureStationName|
arrivalTime|arrivalStationName|carNumber-seatNumber|ticketCost
```

### getPassengerInfo

Este método recibe un pasaporte y devuelve la información del pasajero que coincide con ese pasaporte como una `String` con el siguiente formato e información:

```
passport|name|surname|birthday|email
```

En caso de no encontrar al pasajero, debe devolver una `String` vacía.

### getTrainInfo

Este método recibe un identificador de tren y devuelve la información del tren que coincide con ese identificador como una `String` con el siguiente formato e información:

```
trainId|model|numberOfCars
```

En caso de no encontrar ningún tren, debe devolver una `String` vacía.

### getPassengerTickets

Este método recibe un pasaporte y devuelve una `List` de Java de tipo `String` con la información de cada uno de los billetes que tiene el pasajero con el pasaporte recibido. Cada elemento de la list (i.e. un billete), será un `String` con el siguiente formato e información:

```
routeId|departureTime|departureStationName|
arrivalTime|arrivalStationName|carNumber-seatNumber|ticketCost
```

### getRouteDepaturesInfo

Este método recibe el identificador de una ruta y devuelve una `List` de Java de tipo `String` con la información de cada estación incluyendo sus horarios. Para cada estación, es decir, para cada elemento de la lista, será un `String` con el siguiente formato e información:

```
stationId|[hh:mm, hh:mm, ...]
```

Donde la primera hora corresponde al primer horario y, la segunda, al segundo horario.

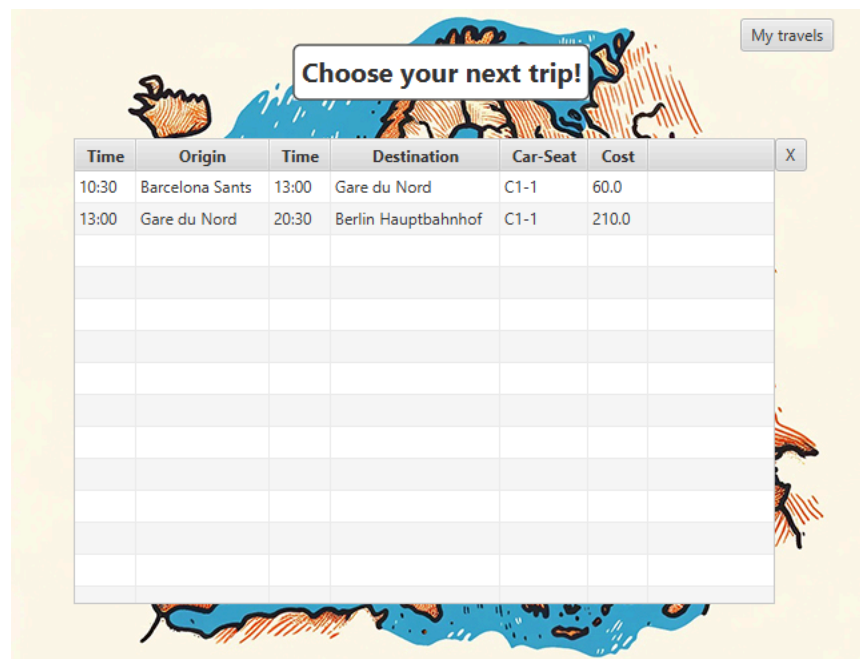
### getCurrentStationId

Este método devuelve el identificador de la estación en la que se encuentra el usuario de la aplicación. Es decir, devuelve el identificador de la estación que se encuentra de color verde. La función asume que el usuario siempre está en una estación.

## Ejercicio 3 - Vista (0.5 puntos)

Las vistas son las “pantallas” con las que interactúa el usuario. En este caso, tenemos una manera de interactuar, es decir, la aplicación en modo gráfico. Con el proyecto ya te damos las vistas/pantallas del programa “hechas”. Decimos “hechas” porque **te pedimos que añadas nuevas columnas en la tabla que contiene los viajes que ya se han hecho. Además, en la pantalla `credits.fxml`, debes añadir tu nombre y correo de la UOC.**

Puedes modificar el contenido de esta tabla desde el archivo `PlayViewController` del `package edu.uoc.locuocomotive.view`. Verás que en tu versión no están todas las columnas. **Deberás añadir las que faltan para dejar la tabla tal y como se ve en la siguiente captura de pantalla:**



Time	Origin	Time	Destination	Car-Seat	Cost	X
10:30	Barcelona Sants	13:00	Gare du Nord	C1-1	60.0	
13:00	Gare du Nord	20:30	Berlin Hauptbahnhof	C1-1	210.0	



Para hacer esta parte del enunciado te recomendamos leer el apartado 5.2 de la Guía de Java obviando las referencias a Eclipse. Como verás se sugiere usar el programa Scene Builder, el cual permite crear y modificar interfaces gráficas de manera WYSIWYG. Encontrarás Scene Builder en el siguiente enlace: <https://gluonhq.com/products/scene-builder/#download>. Si quieres vincular Scene Builder con IntelliJ (no es obligatorio, pero sí práctico), ves a File → Settings... Luego a Languages & Frameworks. Dentro escoge la opción JavaFX y en el lado derecho indica dónde está el fichero ejecutable de Scene Builder. A partir de aquí, dependiendo de la versión de IntelliJ, podrás hacer clic derecho en IntelliJ sobre un fichero `.fxml` y decirle que lo abra con Scene Builder. Igualmente, cuando se abre un fichero `.fxml` en IntelliJ, éste muestra dos pestañas, una con el código FXML y otra pestaña "Scene Builder" que integra Scene Builder dentro del IDE.

## Corolario

Si estás leyendo esto, es que ya has terminado la Práctica. ¡¡Felicidades!! Llegados a este punto, seguramente te estés preguntando: *¿cómo hago para pasarle el programa a alguien que no tenga ni IntelliJ ni JDK instalados?* Buena pregunta. La respuesta es que debes crear un archivo ejecutable, concretamente, un JAR (Java ARchive). Un `.jar` es un tipo de fichero –en verdad, un `.zip` con la extensión cambiada– que permite, entre otras cosas, ejecutar aplicaciones escritas en Java. Gracias a los `.jar`, cualquier persona que tenga instalado JRE (*Java Runtime Environment*) lo podrá ejecutar como si de un fichero ejecutable se tratase. Normalmente, los ordenadores tienen JRE instalado.

Para crear un fichero `.jar` para una aplicación JavaFX hay que tener presente que la clase principal (i.e. aquella que tiene el `main`) no puede heredar de `Application`. Si lo hace, el `.jar` no se ejecutará correctamente. Es por ello que la solución más sencilla es crear una nueva clase que llame al `main` de la clase que hereda de `Application`. Si miras el fichero `build.gradle`, verás que dentro de la configuración del plugin `application` usa como `main`, el que tiene `LocUOComotive`, mientras que la tarea `jar` invoca al `main` de la clase `Main`. Asimismo, debido a que JavaFX no pertenece al `core` de JDK desde la versión 11, debemos añadir los módulos que el programa necesita, de lo contrario, la ejecución del `.jar` fallará. Para indicar los módulos debemos hacer el proyecto modular, que no es más que añadir el fichero `module-info.java` al proyecto. Si te fijas, te hemos facilitado dicho fichero en `src/main/java`. En el apartado 4.3 de la Guía de Java damos una pincelada muy breve al tema de los módulos introducidos por JDK 9.

Para crear un fichero `.jar` que se ejecute en una máquina que tenga instalada JRE, debes descomentar la tarea `jar` que encontrarás dentro de `build.gradle`. Esta tarea está configurada para crear un *fat jar*, es decir, un fichero `.jar` que, además de las clases de nuestro programa, contiene también todas las clases de todas las librerías de las que depende. Así pues, es un fichero más grande (de ahí el uso del adjetivo *fat*) de lo que sería un `.jar` generado de manera normal. Una vez descomentada la tarea y actualizadas las tareas Gradle (recuerda darle al botón refrescar que aparece en el fichero `build.gradle`), sólo tienes que hacer doble click en la tarea `jar` y se creará el fichero `.jar` dentro de una carpeta llamada `build`. Más concretamente, está dentro de `build/libs`. Simplemente copia el fichero `LocUOComotive-1.0-SNAPSHOT.jar` (contiene todo: `.class` y recursos) y ejecútalo donde quieras (asegúrate que en el ordenador que utilices esté, como mínimo, la versión 21 de JRE). Puedes ejecutarlo haciendo doble click o usando el comando `java -jar LocUOComotive-1.0-SNAPSHOT.jar` en un terminal.

Quizás estés pensando: *¿qué sucede si en el ordenador en que se ejecuta el `.jar` no hay JDK ni JRE?* Pues, o bien lo instalas, o bien usas `jlink`. Lo que hace `jlink` es empaquetar el `.jar` junto con una versión *ad hoc* de JRE. Para ello necesita que el proyecto Java esté modularizado, puesto que, según los módulos que se indiquen en el



fichero `module-info.java`, el JRE *ad hoc* que cree será mayor o menor. Para usar `jlink` debes comentar, en `build.gradle`, la tarea `jar` que genera el *fat jar*. A continuación, descomentar la tarea `jlink` que encontrarás en `build.gradle`. Después, sólo tienes que hacer doble click en la tarea de Gradle llamada `jlink` que encontrarás dentro del grupo `build`. El resultado se creará en la carpeta `build/image`. Para ejecutar la aplicación debes ir a `image/bin` y ejecutar el fichero `LocUOComotive`, no sin antes copiar el directorio `levels` que hay en `resources` dentro del directorio `bin` (sinceramente, no sabemos por qué no funciona dejándolo en `resources`). A veces hay problemas para que la aplicación generada con `jlink` lea correctamente los ficheros añadidos en `resources`, así que si no os funciona, no os frustréis.

Cabe destacar que `jlink` es un comando propio de JDK y, por lo tanto, se puede ejecutar desde línea de comandos sin necesidad de usar Gradle (y el plugin correspondiente): <https://www.devdungeon.com/content/how-create-java-runtime-images-jlink>.

*¿Y si queremos un instalador?* Pues a partir de JDK 16 está disponible `jpackage`. Lee más sobre `jar`, `jlink` y `jpackage` en: <https://dev.to/cherrychain/javafx-jlink-and-jpackage-h9>.

De todas maneras, hoy en día se usan aplicaciones como Docker para distribuir programas.

# Evaluación

Esta Práctica se evalúa de la siguiente manera:

## Ejercicio 1 - Diseño (5 puntos)

Se evaluará la calidad de la propuesta así como el uso correcto del estándar UML para la creación de diagramas de clases. Asimismo, la no presentación del fichero `.png` exigido supondrá la pérdida de 0.5 puntos.

## Ejercicio 2 - Codificación (4.5 puntos)

Este ejercicio se evaluará mediante la superación de los test proporcionados.

Tipo de test	Peso	Comentarios
4 basic	2 pts.	<p>Estos test comprueban que los métodos básicos son funcionalmente correctos. Para probarlos haz:</p> <p style="text-align: center;"><code>Gradle → verification → testBasic</code></p> <p>La nota se calculará a partir de la siguiente fórmula:</p> <p style="text-align: center;"><math>(\#test\_basic\_pasados / \#test\_basic) * 2</math></p>
3 advanced	1.5 pts.	<p>Estos test comprueban que los métodos avanzados son funcionalmente correctos. Para probarlos haz:</p> <p style="text-align: center;"><code>Gradle → verification → testAdvanced</code></p> <p>La nota se calculará a partir de la siguiente fórmula:</p> <p style="text-align: center;"><math>(\#test\_advanced\_pasados / \#test\_advanced) * 1.5</math></p>
1 special	1 pts.	<p>Este test comprueba que el método más difícil de todos funciona correctamente. Para probarlo haz:</p> <p style="text-align: center;"><code>Gradle → verification → testSpecial</code></p>



`Gradle → verification → testAll` ejecuta todos los test.

Se evaluará la evaluación de la calidad del código entregado observando cuestiones como por ejemplo:

- Uso de las convenciones y buenas prácticas del lenguaje Java.
- Calidad de los algoritmos.
- Legibilidad/Claridad.
- Comentarios Javadoc para clases, interfaces, enumeraciones, atributos y métodos que forman parte del modelo. **No es necesario generar la documentación.**
- Comentarios en puntos críticos o de difícil comprensión para un tercero.

**El estudiante puede recibir una penalización de hasta 1 punto debido a la mala calidad de su código.**

## Ejercicio 3 - Vista (0.5 puntos)

En este ejercicio se evaluarán los siguientes ítems:

Número	Puntuación
Añadir nombre y login en créditos	0.10 puntos
Añadir columnas en la tabla de viajes	0.15 puntos
Visualización correcta de la información en las columnas añadidas	0.25 puntos

## Formato y fecha de entrega

Tienes que entregar un fichero \*.zip, cuyo nombre tiene que seguir este patrón: `loginUOC_PRAC.zip`. Por ejemplo: `dgarciaso_PRAC.zip`. Este fichero comprimido tiene que incluir los siguientes elementos:

- Un directorio llamado `ex1` en la raíz del fichero \*.zip que contenga el fichero editable `Dia/Drawio loginUOC_PRAC.dia` (o `loginUOC_PRAC.drawio`) así como el fichero `loginUOC_PRAC.png` generado, siendo `loginUOC` tu login UOC.
- El proyecto `LocUOComotive` completado siguiendo las peticiones y especificaciones de los enunciados de los Ejercicios 2 y 3.

El último día para entregar esta Práctica es el **24 de junio de 2024** antes de las 23:59. Cualquier Práctica entregada más tarde será considerada como no presentada.

## Anexo: Guía Dia

Dia es un software libre para hacer diferentes tipos de diagramas, entre ellos, diagramas de clases. Sabemos que existen otros programas (ArgoUML, MagicDraw, Rational Rose, etc.), pero Dia es muy sencillo y además es gratuito. Es importante que te acostumbres a usar software libre ya que en las empresas muchas veces no se pueden permitir pagar las licencias que suponen algunos programas.

### Instalación

El programa Dia podrás descargarlo en la siguiente página web:  
<http://dia-installer.de/download/index.html>.



Si estás usando MacOs y, más concretamente la versión Yosemite, debes seguir las indicaciones que se explican en uno de los siguientes enlaces para que el programa se ejecute correctamente:

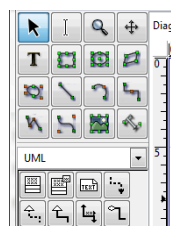
- <https://paletosdelaelectronica.wordpress.com/2015/02/23/dia-solucion-error-yosemite/>
- <http://navkirats.blogspot.com/2014/10/dia-diagram-mac-osx-yosemite-fix-i-use.html>

### Guía rápida de uso

Aunque Dia es un programa muy sencillo (y creemos que intuitivo) de usar, vamos a comentar cómo hacer algunas cosas:

#### Escoger elementos UML para dibujar

Dia permite dibujar diferentes tipos de diagramas. Por eso, tenemos que indicarle que queremos utilizar elementos UML. Para ello, vamos al panel izquierdo del programa y en el desplegable que aparece escogemos UML. En ese preciso momento, justo debajo del desplegable, aparecerán los elementos que podemos dibujar a la hora de crear nuestro diagrama de clases.



## Elementos básicos de dibujo



- Clicka en este botón y luego clicka en el lienzo para crear una caja, p.ej. para crear una clase. Haz doble click en la caja para que se te muestre una ventana de edición. En la pestaña “Atributos” puedes crear y modificar atributos, y en la pestaña “Operaciones” puedes crear y modificar métodos.



- Clicka en este botón y luego une dos cajas (de origen a destino; destino = punta de flecha) para crear una relación de dependencia. Si haces doble click en la línea, sólo tienes que preocuparte de escribir en el apartado “Nombre” para poner, por ejemplo, “throws”.



- Clicka en este botón y luego une dos cajas (de origen a destino; origen = rombo) para crear una asociación de agregación o composición. Si haces doble click en la línea, verás que aparece una ventana con dos columnas: “Side A” (rombo) y “Side B” (sin rombo). Aquí puedes escribir el rol (nombre del atributo), la multiplicidad (p.ej. \*), la visibilidad del rol (p.ej. si indicas “Privado” se añadirá “-” delante del rol) y si quieres mostrar la punta de la flecha (importante para relaciones unidireccionales).



- Clicka en este botón y luego une dos cajas para crear una asociación binaria. Si haces doble click en la línea, verás que aparece la misma ventana que con el botón anterior. De hecho, puedes usar cualquiera de los 2 botones para crear una asociación y escoger el tipo mediante la opción “Tipo” de dicha ventana. La opción “Nada” corresponde a una asociación binaria.



- Clicka en este botón y luego une dos cajas (de superclase a subclase) para crear una relación de herencia.



- Clicka en este botón y luego une dos cajas (de interfaz a clase) para crear una relación de implementación.

## Definir clases y métodos abstractos

- Para definir una **clase abstracta** (i.e. nombre en cursiva) debes hacer doble click en la clase, en la pestaña "Clase" marcar la opción "Abstracta" y luego en la pestaña "Estilo" poner el valor `Bold-Oblique` en el desplegable en el que interseccionan la fila "Clase abstracta" y la segunda columna de "Fuente".
- Para definir un **método abstracto** (i.e. nombre en cursiva) debes hacer doble click en la clase, elegir la pestaña "Operaciones", escoger el método que quieres que sea abstracto y en el desplegable llamado "Tipo de herencia", escoger la opción "Abstracta". A continuación, para que los métodos abstractos de la clase se vean correctamente, debes ir a "Estilo" y poner el valor `Oblique` en el desplegable en el que interseccionan la fila "Abstracta" y la segunda columna de "Fuente".

## Definir atributos y métodos estáticos

- Para definir un **atributo static** (i.e. nombre subrayado) debes hacer doble click en la clase, ir a la pestaña "Atributos", escoger el atributo que quieres hacer estático y abajo a la izquierda marcar la opción "Vista de clase". Si es el último atributo del listado, deberás añadir un atributo vacío con visibilidad "Implementación" para que se vea claramente la línea que subraya el atributo `static`.
- Para definir un **método static** (i.e. nombre subrayado) debes hacer doble click en la clase, ir a la pestaña "Operaciones", escoger el método que quieres hacer estático y justo debajo de "Tipo de herencia" marcar la opción "Vista de clase". Si es el último método del listado, deberás añadir un método vacío con visibilidad "Implementación" para que se vea claramente la línea que subraya el método `static`.

## Definir clases, atributos y métodos `final`

- Para definir una **clase final** puedes escribir como parte del nombre de la clase el texto `{leaf}` (p.ej. `Person {leaf}`). También puedes escribir `final` en el apartado "Estereotipo" de la pestaña "Clase".
- Para definir un **atributo final** debes escribir el nombre de dicho atributo en UPPERCASE con los espacios sustituidos por `_`.
- Para definir un **método final** debes, en la pestaña "Operaciones", escoger el método que quieres hacer `final` y justo debajo de "Tipo de herencia" marcar la opción "Consulta". Esto hará que se añada el texto `const` al lado del método.



## Definir una interfaz

- Una interfaz se define igual que una clase, pero escribiendo `"interface"` en el apartado "Estereotipo" de la pestaña "Clase".
- Si la interfaz no va a tener atributos, puedes esconder esta parte de la caja desmarcando la opción "Atributos visibles" de la pestaña "Clase".
- Recuerda que los métodos que no son `default` en la interfaz son abstractos.

## Definir una enumeración

- Una enumeración se define igual que una clase pero escribiendo `"enumeration"` en el apartado "Estereotipo" de la pestaña "Clase".
- Si la enumeración no va a tener métodos, puedes esconder esta parte de la caja desmarcando la opción "Operaciones visibles" de la pestaña "Clase".
- Los valores de la enumeración son atributos públicos, `final` y `static` sin tipo ni valor por defecto. Esto es así implícitamente, por lo que en este caso sólo tenemos que escribir el nombre de los valores en mayúscula (son como constantes, i.e. atributos `final`), y obviar el subrayado (para indicar que es `static`) y el modificador de acceso público (se puede poner el valor "Público" o "Implementación", que significa "por defecto o nada").
- Sabemos que las enumeraciones son muy potentes en Java, pero en UML son muy limitadas. Así pues, podremos añadirles atributos en el mismo apartado de los valores y, en la parte de operaciones, podremos poner constructores y métodos. Sin embargo no podremos indicar qué valores coge un valor de la enumeración para los atributos declarados. Como al final UML no es algo sagrado, podemos, si queremos, poner los diferentes valores como "valor por defecto" del atributo, pero esto debe ser algo consensuado con el equipo de trabajo ya que no es estándar.