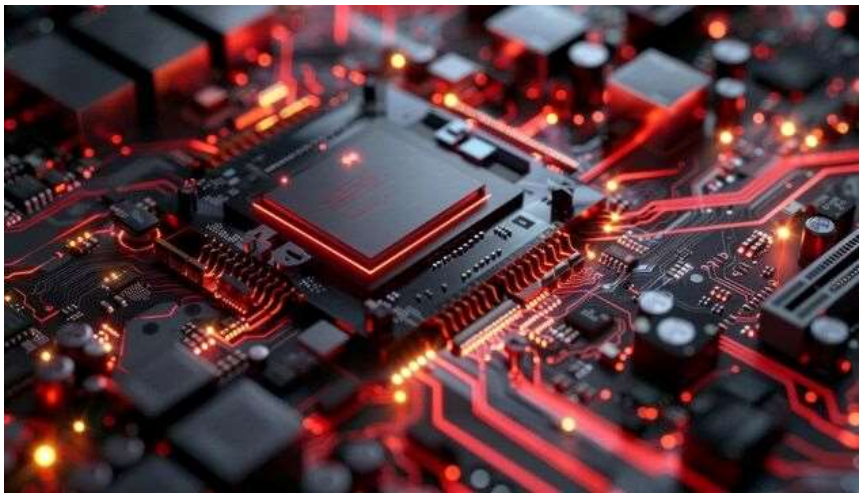
Programación Orientada a Objetos - PEC 2024-2025 -



Alumno: Pau Culla Loren

Correo electrónico: paucullaloren@gmail.com

Tutora: Teresa Alsinet

Año: 2024-2025

○ Análisis general del programa

La aplicación desarrollada, titulada **Sistema de Gestión de Movilidad Sostenible**, tiene un enfoque eminentemente académico. Ha sido diseñada específicamente para cumplir con los requisitos de la Prueba de Evaluación Continua (PEC) de la asignatura **Programación Orientada a Objetos** de la UNED, durante el curso 2024-2025.

El desarrollo de esta aplicación se ha basado en los contenidos del libro *Programación orientada a objetos con Java usando BlueJ* de David J. Barnes y Michael Kölling, que se utiliza como texto principal en la asignatura. A lo largo de este manual se proponen diversos proyectos prácticos, entre los cuales destaca el uso de simulaciones informáticas como herramienta para modelar situaciones del mundo real dentro de un entorno controlado y predecible.

Inspirándose en esta metodología, se decidió orientar el proyecto como una simulación. En concreto, la práctica consiste en representar un sistema de movilidad urbana sostenible en el que interactúan diversas entidades: **estructuras físicas** (como las bases), **vehículos** (motos, bicicletas y patinetes eléctricos) y **personas** (usuarios del servicio y trabajadores encargados del mantenimiento).

Cada una de estas entidades tiene comportamientos específicos y objetivos propios. Por ejemplo, los **usuarios** necesitan vehículos disponibles, en buen estado y con batería suficiente para desplazarse. Estos aspectos dependen del trabajo de los **empleados**, quienes se encargan del mantenimiento y la recarga. Sin embargo, estos trabajadores requieren a su vez una retribución económica, que se financia mediante las tasas cobradas a los propios usuarios.

Al programar las interacciones y condiciones de cada elemento, es posible ejecutar la simulación y analizar cómo se coordinan los distintos agentes dentro del sistema. Este enfoque permite observar de forma controlada las dinámicas internas de un servicio de movilidad sostenible, simulando escenarios reales con un alto grado de abstracción.

Las **simulaciones informáticas** son una herramienta fundamental en numerosos campos del conocimiento, desde las ciencias físicas hasta las ciencias sociales, pasando por la ingeniería, la economía o la planificación urbana. Su valor radica en la capacidad de **representar sistemas complejos** dentro de un entorno digital controlado, donde es posible estudiar comportamientos, anticipar problemas y experimentar con posibles soluciones sin necesidad de intervenir directamente en la realidad.

En el caso concreto de esta práctica, se ha simulado un sistema de movilidad sostenible en una ciudad, con el fin de observar cómo interactúan distintos elementos clave: los vehículos compartidos, las bases de estacionamiento, los usuarios que hacen uso del servicio y los trabajadores que lo mantienen en funcionamiento. Esta representación no solo permite comprobar que el modelo programado es funcional, sino que también abre la puerta a **reflexionar sobre cómo podría organizarse un sistema similar en la vida real**, identificando cuellos de botella, distribuciones ineficientes o desequilibrios entre oferta y demanda.

El valor de una simulación no reside únicamente en su fidelidad al detalle, sino en su **capacidad de abstraer lo esencial** del sistema que se quiere estudiar. Sin embargo, uno de los aspectos más profundos —y quizá más filosóficos— de las simulaciones es que, dentro de un entorno de programación, **el nivel de detalle es teóricamente ilimitado**. Es posible definir reglas, condiciones, excepciones y comportamientos con el grado de precisión que se desee. Cada nueva capa de complejidad —por ejemplo, considerar el desgaste mecánico según el tipo de trayecto, las preferencias individuales de los usuarios o los turnos laborales de los trabajadores— nos acerca a una mayor verosimilitud, a un comportamiento emergente que se asemeja cada vez más al del mundo real.

No obstante, conviene tener claro que alcanzar una simulación absolutamente fiel a la realidad es **inalcanzable**. La **complejidad inherente al mundo físico y social** impone un límite natural a cualquier modelo, por sofisticado que sea. Aun así, este intento constante por aproximarse a lo real, por representar lo inabarcable mediante código, es también un ejercicio intelectual de interés, pues nos lleva a pensar con claridad, a definir las dinámicas de los sistemas, a anticipar interacciones imprevistas y a cuestionar supuestos.

En este sentido, las simulaciones son también **una herramienta de pensamiento**, una forma de explorar cómo funcionan las cosas cuando no podemos experimentar directamente con ellas. Y en el caso particular de los sistemas de movilidad urbana, donde intervienen numerosos factores interdependientes —infraestructura, tecnología, economía, comportamiento humano—, simular es no solo útil, sino prácticamente imprescindible como paso previo a la implantación de soluciones reales.

En el siguiente apartado de la memoria se procederá a describir en detalle las principales clases que componen la aplicación, analizando las responsabilidades de cada una, su relación con el resto del sistema y la manera en que dichas responsabilidades se llevan a cabo en el código.

Este análisis no es meramente estructural, sino que busca poner en valor el enfoque de la **Programación Orientada a Objetos (POO)** adoptado durante todo el desarrollo. Lejos de

limitarse a una simple agrupación de funcionalidades, la POO ha permitido diseñar un sistema modular, extensible y coherente, en el que cada clase representa un concepto del dominio del problema —como usuario, vehículo, base o trabajador— encapsulando tanto su estado como su comportamiento.

Gracias a principios fundamentales de la POO como la **encapsulación**, **abstracción**, **herencia** y **polimorfismo**, ha sido posible modelar relaciones complejas de forma natural y clara. Por ejemplo, los distintos tipos de vehículos heredan características comunes de una clase base, al mismo tiempo que definen comportamientos específicos. Este diseño favorece la reutilización del código y facilita su mantenimiento y evolución futura. El sistema se ha construido de forma que cada clase asuma una **única responsabilidad bien definida** (siguiendo el principio de responsabilidad única del diseño orientado a objetos), lo que permite entender, modificar o ampliar cada componente de manera aislada y predecible.

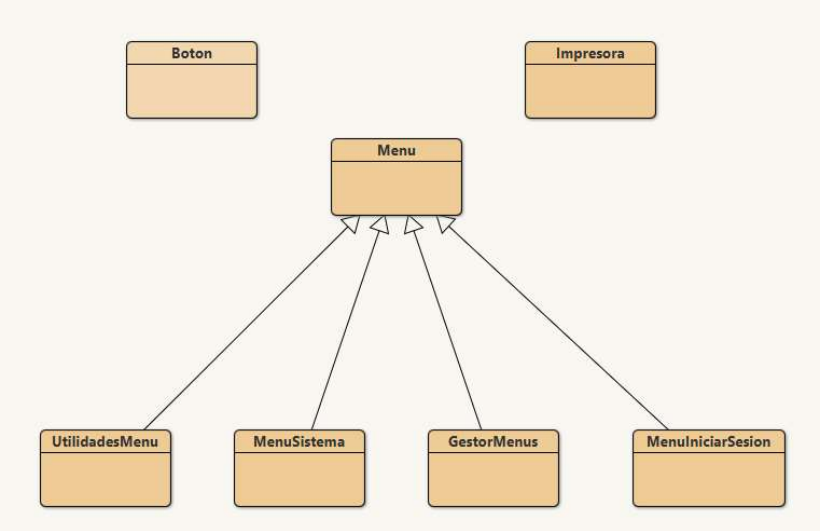
Este enfoque orientado a objetos no solo ha servido como medio técnico, sino también como herramienta pedagógica para interiorizar los conceptos clave de la asignatura, al tiempo que se da forma a una simulación funcional con relevancia práctica.

o Clases

Para explicar las clases las dividiremos en múltiples agrupaciones y las analizaremos por separado, dando una breve descripción de las responsabilidades de cada una.

Menús

Para el desarrollo de la aplicación se optó por diseñar una interfaz de usuario amigable e interactiva. Para ello se usó programación orientada a objetos. Este enfoque permitió estructurar el código de manera modular, facilitando la reutilización, el mantenimiento y la escalabilidad del sistema.



Boton

La clase botón no tiene una enorme relevancia en el programa en general, pero sirvió para estandarizar el uso de botones en la interfaz. Dado que se asume que el estilo, tamaño y funcionalidad de los botones será compartido a lo largo de la aplicación, se tralada toda la lógica posible a esta clase para que luego añadir botones en un menú u otro sea cuestión de unas pocas líneas de código.

Aparte de los fines prácticos, estandarizar el uso de botones también promete una homogeneización de la interfaz, lo que también contribuye a fines estéticos.

Los botones tienen:

- Un nombre (que es el que se muestra encima del botón)
- Una acción (que es un ActionListener que describe qué ocurre al pulsar el botón)
- Una estética (fuente, color de fondo, tamaño…).

Impresora

Otra clase secundaria de la aplicación es Impresora. Esta clase se creó en el momento en que se empezó a dar un uso serio a la terminal para mostrar información en tiempo real del estado de la simulación.

A medida que se mostraban más y más mensajes en la terminal del mismo color la información cada vez era menos clara, y era complicado saber si un mensaje correspondía a una entidad u otra. Para ello se decidió que cada mensaje correspondiente a una entidad se mostraría en el mismo color de la entidad. Eso se podría haber solventado de diversas formas, pero en este caso se prefirió crear una clase Impresora con un método llamado printColorClase, que recibe como parámetros la clase que hace el print y un string con el mensaje. Entonces la propia clase se encarga de agregar el color necesario e imprimir el mensaje en la terminal.

Esta solución simplifica el uso de esta estrategia de mostrar cada mensaje en un color distinto dependiendo de la clase. Además, reduce el acoplamiento, dado que si se quisiera cambiar el color de una clase no sería necesario ir print por print cambiando el color, sino que se modificaría en la clase Impresora y se aplicaría a todos los prints automáticamente.

Menu

La clase Menu tiene cuatro subclases asociadas. El principal propósito de su existencia es definir las características comunes que todo menú debe tener. Esto es, todo menú tiene un conjunto de botones, todo menú tiene un tamaño, un nombre, un JPanel y un JFrame o JDialog asociado.

UtilidadesMenu

Esta subclase tiene la intención de implementar opciones que se usan recurrentemente en los otros menús. Para ello usa a su vez menús, y es por eso por lo que hereda de la clase menú.

Por ejemplo, supongamos que en el menú de usuarios se quiere hacer que un usuario determinado alquile un vehículo determinado. Para ello necesitamos un menú en el que se indique el tipo de vehículo, luego otro que pregunte la id del vehículo por alquilar. Todas esas cuestiones son facilitadas por UtilidadesMenu.

Otra utilidad de importancia es mostrar mensajes de error o éxito según la interacción con un menú haya funcionado o no. UtilidadesMenu también tiene código que implementa estas funcionalidades, con lo cual los otros menús solamente tienen que llamar a un método para hacer aparecer un mensaje de éxito o de error, sin necesidad de responsabilizarse de ello.

GestorMenus

Esta subclase sería la implementación del menú principal de la aplicación. Desde este gestor se puede acceder al resto de menús (MenuIniciarSesion y Menu Sistema). Este menú crea una ventana en la que se permite navegar al resto de menús. Como curiosidad tiene una implementación de botón de navegación a la pestaña anterior, donde se hace uso de una pila.

Desde el menú de GestorMenus se puede acceder a MenuIniciarSesion.

MenuIniciarSesion

Esta subclase corresponde a un único submenú. Este es responsable de que la persona que interactúa con la interfaz sea capaz de iniciar sesión en una entidad de la simulación, para poder tomar control de esta.

Desde el menú de MenuIniciarSesion se puede acceder a MenuSistema.

MenuSistema

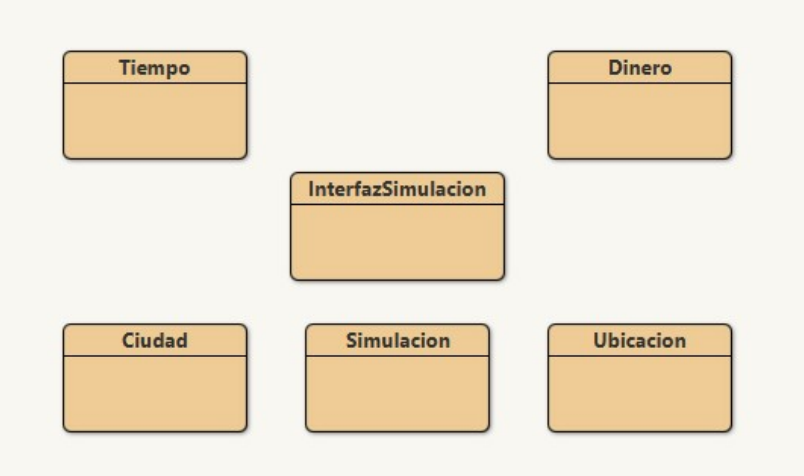
Esta subclase contiene la mayoría del código del apartado de los menús. Se trata de la sección donde se implementan todas las funcionalidades de la interfaz, como el hecho de crear entidades y añadirlas a la ciudad, dar órdenes a las entidades para que hagan una u otra acción o acceder a estadísticas e información general de la simulación.

En base al tipo de persona que ha accedido al menú (lo que se conoce a través de MenuIniciarSesion) se muestran unas opciones u otras. Cuando todavía no existe ninguna entidad en la ciudad, es necesario iniciar sesión como Administrador para agregar entidades.

Lógica y Simulación

Esta sección se encarga de implementar la lógica central que rige el funcionamiento de la simulación, exceptuando aquellas acciones que son responsabilidad directa de las entidades del sistema.

En un entorno simulado como este, no todo el comportamiento proviene de las propias entidades (como vehículos, usuarios o trabajadores). Existen también componentes externos que **orquestan y coordinan** el conjunto, gestionando aspectos globales como el avance del tiempo, la generación de eventos o el manejo de la economía. Esta sección abarca precisamente **todos esos elementos estructurales** que permiten que la simulación funcione de forma coherente y fluida, proporcionando el contexto en el que las entidades individuales pueden interactuar según las reglas del sistema.



Simulacion

Simulación es una de las clases más importantes de todo el programa. Contiene instancias de todas las clases que se presentarán a continuación en esta sección, y posee el bucle principal de ejecución de la simulación, implementado en el método runSimulacion.

Simulacion posee objetos ciudad, tiempo y dinero. Mediante ellos controla la distribución física de las entidades, el avance y retroceso del tiempo y el estado de la economía general, respectivamente.

En adición posee un historial de estados, que permite poder dar marcha atrás en la simulación y visitar escenarios anteriores. Aparte posee un objeto InterfazSimulacion, que se explica a continuación.

InterfazSimulacion

La simulación se ejecuta internamente en la máquina, pero resulta fundamental disponer de una representación visual que permita observar en tiempo real todo lo que ocurre en el sistema. Esta responsabilidad recae en la clase InterfazSimulacion.

Esta clase proporciona los mecanismos necesarios para interactuar con la simulación y visualizar su evolución. Entre sus funcionalidades más relevantes se encuentra el control de la velocidad del tiempo simulado, tanto para acelerarlo como para ralentizarlo o incluso retrocederlo, facilitando así el análisis de situaciones concretas.

Además, la interfaz muestra información contextual de gran utilidad: descripciones detalladas de las entidades presentes en una ubicación específica, la hora del día simulada en cada instante, y el movimiento dinámico de los distintos agentes a lo largo del entorno urbano. Todo ello permite al usuario entender de forma clara y accesible cómo se desarrolla la simulación y cómo interactúan los distintos elementos del sistema.

Ubicación

La clase **Ubicación** se encarga de gestionar de manera uniforme las posiciones de todas las entidades dentro de la simulación. Dado que el entorno urbano se modela como una **matriz bidimensional**, cada entidad debe estar asociada a una fila y una columna, lo que garantiza un seguimiento preciso de su ubicación en todo momento.

Esta clase tiene varias responsabilidades clave, como la asignación de ubicaciones a las entidades, el acceso a las mismas para consultas posteriores y la comparación de ubicaciones para determinar si dos entidades coinciden en el mismo lugar en un momento determinado. Además, facilita otras operaciones relacionadas con el control espacial de las entidades, asegurando que su posición dentro del sistema esté siempre correctamente gestionada.

Ciudad

La clase **Ciudad** es un componente fundamental del programa, ya que gestiona la lista principal de entidades dentro de la simulación y se encarga de asignar a cada una de ellas una ubicación específica dentro de la matriz que representa el entorno urbano. Su responsabilidad central es manejar y actualizar la lista de entidades de manera eficiente y consistente, asegurando que todas las interacciones con la simulación sean correctas.

Una de las decisiones clave en el diseño de esta clase fue el uso del tipo de datos **CopyOnWriteArrayList** para almacenar las entidades. Este tipo de lista se seleccionó por su capacidad para evitar errores por concurrencia, ya que cada vez que se añade un nuevo elemento, se crea una copia de la lista antes de hacer la modificación. Esta característica es crucial en el contexto de la simulación, donde la lista de entidades se accede de forma constante para realizar diversas operaciones. Si bien el acceso es secuencial en su mayoría, existe la posibilidad de que nuevas entidades sean añadidas mientras la lista está siendo recorrida en otros puntos del programa. Este comportamiento podría generar fallos fatales si no se gestiona correctamente.

Gracias al uso de **CopyOnWriteArrayList**, se elimina el riesgo de colisiones entre lecturas y escrituras, garantizando que la simulación se ejecute sin problemas, y lo hace de manera **eficiente**, ya que la velocidad de acceso a los elementos de la lista no se ve afectada.

La clase **Ciudad** ofrece métodos esenciales como agregarEntidad, que permite añadir nuevas entidades a la simulación, y métodos para obtener sublistas de entidades basadas en diferentes criterios, como la clase de las entidades o su identificador único (ID). Además, proporciona funcionalidades como la detección de entidades con fallos mecánicos o la verificación de si una ubicación específica está ocupada por una entidad de un determinado tipo.

En resumen, la clase **Ciudad** es crucial para la gestión de las entidades y sus ubicaciones físicas en la simulación. A través de sus métodos, otras clases pueden obtener información clave sobre el estado del sistema y manipular las entidades de forma controlada y segura.

Tiempo

La simulación transcurre a lo largo de una dimensión temporal que puede avanzar o retroceder, lo que requiere una gestión precisa del tiempo para mantener la coherencia de los eventos. Para ello, se diseñó la clase **Tiempo**, que tiene la responsabilidad de controlar el tiempo actual en diferentes unidades: segundos, minutos, días, meses y años.

Además de gestionar el paso del tiempo, la clase Tiempo proporciona a otras partes del sistema la capacidad de acceder al valor actual del tiempo. Esto incluye funcionalidades como calcular el intervalo temporal entre dos instantes distintos de tiempo, lo cual es crucial para diversas operaciones dentro de la simulación.

La clase también interactúa con la **InterfazSimulacion**, enviando de forma constante información sobre la hora del día simulada para que la interfaz de usuario se actualice en consecuencia. En particular, modifica el color de los elementos de la interfaz según la hora, permitiendo una representación visual dinámica del paso del tiempo a medida que avanza o retrocede la simulación.

En resumen, la clase **Tiempo** no solo gestiona la **dimensión temporal** de la simulación, sino que facilita la interacción con otros componentes del sistema, asegurando que el tiempo transcurra de manera controlada y proporcionando información relevante a la interfaz de usuario en tiempo real.

Dinero

Al igual que en la simulación existe una dimensión temporal que regula el avance y retroceso de los eventos, es igualmente esencial contar con una dimensión económica que maneje las transacciones financieras dentro del sistema. En este entorno simulado, coexisten dos tipos de agentes: los usuarios, que consumen los servicios de movilidad, y los trabajadores, que ofrecen dichos servicios a través de tareas como la reparación de vehículos o la recarga de baterías.

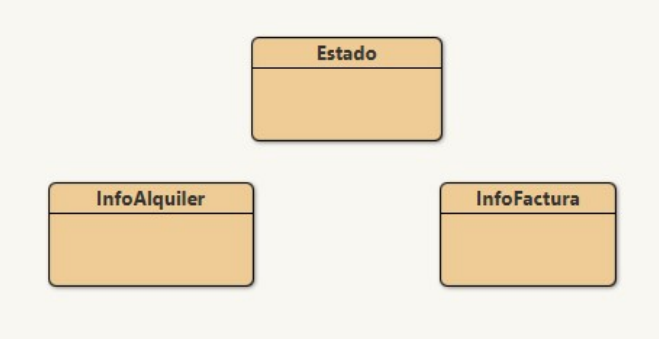
Cada vez que un trabajador realiza una tarea, ya sea reparando una entidad o recargando un vehículo, cobra una tarifa por sus servicios. Este ingreso debe ser cubierto por los fondos del sistema, los cuales provienen de las tasas de uso periódicas que los usuarios deben pagar por utilizar los vehículos. Estas tasas son una fuente constante de ingresos para el sistema, y su valor puede ser ajustado según las necesidades del administrador.

Desde el menú del Administrador, es posible modificar tanto la frecuencia de cobro de las tasas como el precio, lo que ofrece flexibilidad en la gestión económica del sistema y permite simular diferentes escenarios de precios y estructura de costos. Además, el sistema asume la existencia de un Estado que cobra impuestos sobre las operaciones del servicio, lo que agrega una capa de realismo económico a la simulación.

En resumen, la dimensión económica no solo permite gestionar las finanzas internas del sistema, asegurando que los trabajadores reciban su pago por los servicios prestados, sino que también refleja la interacción entre los usuarios, los trabajadores y el Estado, simulando así un entorno económico realista y controlable.

Estructuras de información

En esta sección, se abordarán tres clases distintas que, aunque no comparten una relación de herencia, están agrupadas debido a que su misión es común: **almacenar información sobre los eventos ocurridos durante la simulación** para diversos fines. Estas clases desempeñan un papel clave en el seguimiento y análisis de la dinámica del sistema, permitiendo registrar, gestionar y consultar los eventos que tienen lugar a lo largo del desarrollo de la simulación.



Estado

La clase **Estado** es fundamental para el registro y seguimiento del avance de la simulación. Cada vez que la simulación realiza una actualización, también conocida como un **"step"**, se crea un nuevo objeto de tipo **Estado** que se almacena en una pila, permitiendo así llevar un control detallado de la evolución del sistema a lo largo del tiempo.

Los objetos de la clase **Estado** tienen dos campos principales, que son esenciales para almacenar la información clave de cada paso de la simulación. Estos campos son:

- **estadoCuadricula**: Es una matriz que refleja el estado actual de la ciudad, es decir, la distribución de las entidades en la ciudad en ese momento. Cada celda de la matriz almacena la ubicación de las entidades presentes, proporcionando una representación espacial precisa en cada instante.
- **estadoEntidades**: Es un **CopyOnWriteArrayList** que guarda todas las entidades activas en ese momento de la simulación. Cada entidad dentro de esta lista es una estructura de información compleja que contiene todos los detalles necesarios sobre su estado que dependen de la naturaleza de la clase.

La existencia de esta clase permite una funcionalidad clave: **rebobinar la simulación**, es decir, retroceder en el tiempo y recuperar el estado de la simulación en pasos previos. Aunque esta

capacidad no era estrictamente necesaria para el funcionamiento básico del sistema, resultó ser muy útil durante las fases de testeo, ya que permite observar cómo se comportan las entidades y el sistema en general bajo distintas condiciones, facilitando la identificación de errores y la mejora del rendimiento.

InfoAlquiler

La clase **InfoAlquiler** tiene la misión de generar y almacenar informes detallados sobre los alquileres realizados por los usuarios. Cada vez que un usuario alquila un vehículo, se crea un nuevo objeto de esta clase, el cual contiene información clave sobre el alquiler realizado. Los principales datos almacenados en un objeto **InfoAlquiler** son:

- **Usuario y vehículo alquilado**: Se registra el usuario que ha realizado el alquiler, así como el vehículo específico que ha sido alquilado.
- **Tiempo inicial y final**: Se almacena tanto el **inicio** como el **fin** del alquiler, lo que permite calcular de manera precisa la duración del alquiler.
- **Mes**: Se guarda el mes en el que se realizó el alquiler, lo que facilita el análisis de la frecuencia y patrón de uso de los vehículos a lo largo del tiempo.

Esta información es **esencial** para el sistema de **promociones** de usuarios. Para determinar si un usuario puede ser ascendido a la categoría **premium**, es necesario saber cuántos vehículos ha alquilado en los últimos meses. Este proceso se lleva a cabo recorriendo la lista de entidades del sistema, seleccionando aquellas que sean de la clase **Usuario**, y luego accediendo a su correspondiente registro de objetos **InfoAlquiler** para obtener la información relevante.

El uso de esta clase permite gestionar de manera eficiente las promociones basadas en el comportamiento de los usuarios, asegurando que se ofrezcan incentivos a aquellos que han demostrado un uso frecuente de los vehículos en el sistema.

InfoFactura

La clase **InfoTrabajo** tiene una función similar a la clase **InfoAlquiler**, pero en lugar de registrar alquileres de vehículos, se encarga de registrar la información de los trabajos realizados por los **trabajadores** del sistema, ya sean **mecánicos** encargados de la reparación de vehículos o **técnicos de mantenimiento** responsables de la carga de baterías y otras tareas de mantenimiento. Los datos clave almacenados en un objeto **InfoTrabajo** son los siguientes:

- **Trabajador**: Se registra el trabajador que ha realizado la tarea, proporcionando información sobre la entidad que ha realizado el trabajo.
- **Duración del trabajo**: Se guarda el **tiempo** que ha durado el servicio realizado, lo que puede ser útil para calcular costos y analizar la eficiencia del trabajador.
- **Entidad que ha recibido el trabajo**: Este campo registra la entidad específica sobre la cual se ha realizado el trabajo, como una base reparada o un vehículo cuya batería ha sido recargada.
- **Precio del servicio**: Se almacena el **precio** que cobra el trabajador por el servicio, calculado en base a la duración del trabajo, las tarifas del trabajador y la naturaleza del trabajo realizado.

Esta información es crucial para gestionar tanto los pagos a los trabajadores como el análisis del rendimiento del sistema. Además, permite realizar un seguimiento detallado de las tareas realizadas, facilitando la gestión económica del sistema y la evaluación de la eficiencia de cada trabajador, así como la generación de informes estadísticos generales.

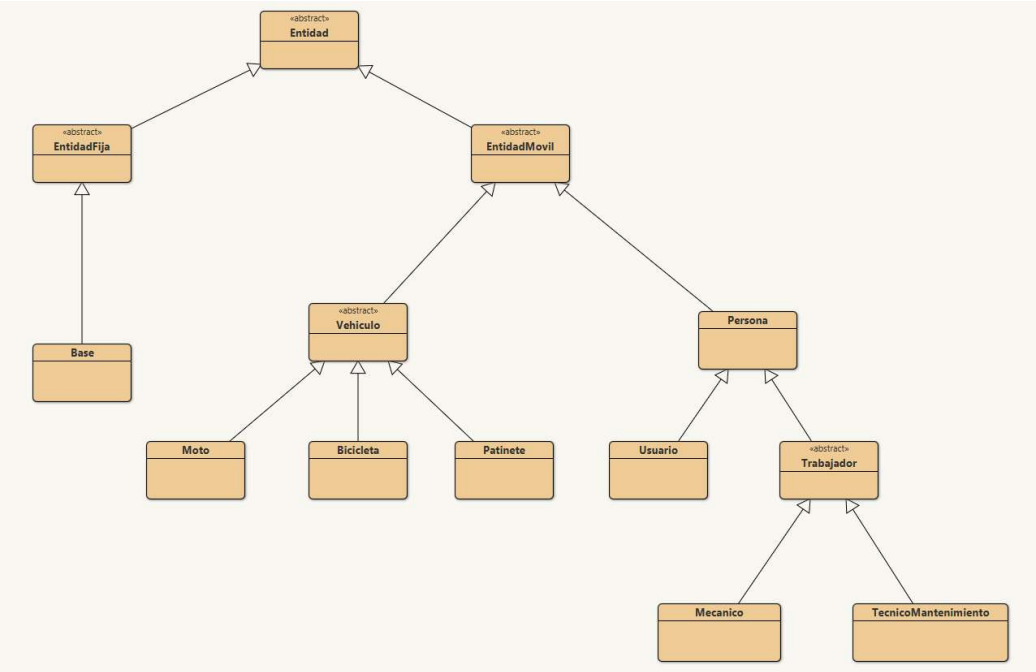
Entidades

En esta sección, se explorará la **jerarquía de clases** que conforman las **entidades** dentro de la simulación. El concepto de entidad, desde un punto de vista filosófico, se puede definir como **“todo aquello que tiene existencia, sea real, imaginario o conceptual”**.

En el contexto de la simulación, las clases más superiores de esta jerarquía son de carácter conceptual y abstracto, lo que permite una mayor flexibilidad y adaptabilidad en el sistema. Cuanto más abstracta sea una clase, más casos podrá abarcar, permitiendo que las clases inferiores hereden propiedades y comportamientos comunes sin necesidad de redefinirlas cada vez. Este enfoque proporciona una base sólida sobre la que construir la funcionalidad del sistema.

A medida que descendemos en la jerarquía, las clases se van volviendo más concretas y específicas, conectándose de manera más directa con la realidad de la simulación. Finalmente, en las clases hoja, llegamos a las representaciones más específicas y reales de las entidades, como los **vehículos (motos, bicicletas, patinetes)** o los **trabajadores (mecánicos, técnicos de mantenimiento)**.

Esta estructura jerárquica, basada en la abstracción progresiva, es clave para mantener un sistema **modular** y **escalable**, donde las clases más generales pueden ser extendidas y especializadas a medida que se desciende en la jerarquía, permitiendo la inclusión de nuevas entidades y comportamientos sin alterar el diseño central del sistema.



Entidad

La clase **Entidad** actúa como la clase base de la jerarquía de entidades dentro de la simulación, y de ella heredan todas las clases que representan a las diferentes entidades del sistema. Esta clase se encarga de regular las propiedades compartidas por todas las entidades de la simulación, independientemente de su tipo específico (por ejemplo, vehículos, trabajadores, etc.).

Entre las responsabilidades principales de la clase **Entidad**, se encuentran:

- **ID**: Cada entidad posee un identificador único (**ID**) que permite diferenciarla de otras entidades de su misma tipología dentro del sistema.
- **Edad**: La clase maneja la **edad** de la entidad, un atributo importante para simular la **depreciación** o el **envejecimiento** de los vehículos o incluso de los trabajadores en el sistema.
- **Color**: Esta propiedad se utiliza para asignar un **color** visual a la entidad, lo que resulta útil tanto para la visualización de la simulación como para la impresión de mensajes en la consola mediante la clase Impresora.
- **Ubicación actual**: La clase mantiene la **ubicación** de la entidad en la ciudad, permitiendo conocer su posición exacta en la matriz que representa el entorno. Aunque existan entidades con ubicaciones móviles y otras estáticas, está claro que toda entidad tiene una posición física en un momento determinado.
- **Estado mecánico**: Este atributo refleja el estado físico de la entidad, ya sea un vehículo o una base, y se utiliza para determinar si se encuentra en condiciones óptimas para su uso. El estado mecánico se degrada irremediablemente a medida que el tiempo avanza, y es usado en una función que establece una relación inversa: cuanto menor es el valor del estado mecánico, mayor es la probabilidad de que, en cada step de la simulación, surja un fallo mecánico. Este fallo anula completamente la operatividad de la entidad hasta que sea reparada. Cuando una entidad sufre un fallo mecánico, se activa una alerta de fallo, y un mecánico debe intervenir para reparar la entidad, restaurando su estado mecánico al 100%. Una vez completada la reparación, la alerta de fallo mecánico se desactiva, y la entidad vuelve a estar disponible para su uso.
- **Fallo mecánico**: Tal y como se ha explicado en el punto anterior, cuando surje un fallo mecánico la entidad se vuelve inutilizable hasta que un mecánico la repara.

- **Alerta de fallo mecánico**: A diferencia de lo que podría pensarse, un vehículo que sufre un fallo mecánico no activa automáticamente su alerta. El fallo permanece latente hasta que un usuario intenta utilizar el vehículo y descubre que no funciona correctamente. En ese momento, el usuario abandona el trayecto, deja el vehículo en su ubicación actual y notifica el problema al sistema a través de la aplicación. Es entonces cuando se activa la alerta de fallo mecánico, lo que permite que el cuerpo de mecánicos registre el incidente y se movilice para gestionarlo. Un trabajador es asignado para acudir a la ubicación del vehículo averiado, trasladarlo a una base si fuera necesario, y proceder a su reparación.

La clase **Entidad** proporciona una **base común** que facilita la gestión y el control de todas las entidades del sistema, asegurando que compartan los mismos atributos esenciales y comportamientos fundamentales, mientras que las clases derivadas pueden especializarse y añadir propiedades y métodos específicos según el tipo de entidad.

Entidad Fija

EntidadFija es una clase intermedia en la jerarquía de herencia que, actualmente, no contiene lógica ni atributos propios. Su función principal es estructural y preventiva: aunque en esta versión del programa solo existe una entidad fija (la base), se ha creado esta clase pensando en una posible expansión futura. De este modo, si en el futuro se introducen más tipos de entidades fijas, podrán heredar de esta clase común, lo que facilitará la organización del código y su mantenibilidad.

Base

En la simulación, las bases cumplen una función central dentro del ecosistema urbano. Son el punto de almacenamiento predeterminado para bicicletas y patinetes eléctricos cuando no están siendo utilizados por usuarios. Además, las bases son el único lugar habilitado para llevar a cabo tareas de mantenimiento: tanto la reparación de fallos mecánicos como la recarga de baterías deben realizarse obligatoriamente en una base, ya que se asume que es ahí donde se dispone del equipamiento técnico necesario.

Cuando un vehículo sufre una avería —ya sea por un fallo mecánico o por batería agotada— un trabajador del sistema acude a su ubicación, lo traslada manualmente a la base más cercana y, una vez allí, realiza la intervención correspondiente. Tras ello, el vehículo queda nuevamente

disponible para su uso. Este flujo de trabajo refuerza la idea de la base como centro logístico operativo en el sistema de movilidad simulada.Las bases también poseen estado mecánico y pueden tener fallos mecánicos, si eso ocurre se vuelven inutilizables y es necesario que un trabajador la repare para seguir con su normal funcionamiento.

Las **bases** cuentan con dos arrays fundamentales para la gestión de vehículos:

- **vehiculosDisponibles**: contiene todos los vehículos almacenados en la base que están en condiciones óptimas de uso —es decir, sin fallos mecánicos y con batería suficiente—. Son los únicos que pueden ser alquilados por los usuarios.
- **vehiculosInhabilitados**: almacena aquellos vehículos que han sido trasladados a la base para su reparación o recarga. Mientras se encuentren en esta lista, los usuarios no pueden acceder a ellos, ya que están fuera de servicio hasta que un trabajador los ponga nuevamente en estado operativo.

EntidadMovil

La clase **EntidadMovil** es, sin duda, una de las clases que más código concentra en toda la aplicación. Sin embargo, sus responsabilidades están claramente delimitadas: se encarga de gestionar el movimiento de las entidades a lo largo del entorno simulado, es decir, dentro de la clase **Ciudad**.

Las entidades móviles son aquellas que pueden **desplazarse por el mapa**, y engloban a **todas las entidades** del sistema **excepto las bases**, que permanecen estáticas.

El sistema de movimiento se articula en torno a dos mecanismos principales:

- **Trayecto planeado**: la entidad selecciona una **ubicación de destino** y genera un trayecto hacia ella desde su posición actual. En cada paso de la simulación, la entidad avanza una casilla siguiendo la ruta calculada. Este mecanismo permite, por ejemplo, que un usuario se desplace hasta una moto antes de alquilarla, o que un trabajador se dirija a una entidad averiada para su reparación.
- **Seguimiento de otra entidad**: algunas entidades móviles pueden sincronizar su movimiento con el de otra. Es el caso de los usuarios que han alquilado un vehículo: mientras dura el trayecto, el usuario no calcula su propia ruta, sino que se acopla al

desplazamiento del vehículo. En términos de programación, esto se logra mediante el atributo entidadSeguida. Cuando este atributo no es nulo, la entidad "seguidora" imita en tiempo real la ubicación de la entidad seguida, simulando que viajan juntas. Este seguimiento se mantiene activo hasta que el trayecto del vehículo finaliza.

Vehiculo

Los **vehículos** son una subclase de EntidadMovil que incorporan nuevas funcionalidades específicas, entre las que destaca el sistema de batería. Para desplazarse, los vehículos consumen energía eléctrica, almacenada en una batería interna. La batería se descarga progresivamente a medida que el vehículo realiza desplazamientos, hasta agotarse por completo. Cuando esto sucede, el vehículo queda inhabilitado y necesita ser transportado a una base por un técnico de mantenimiento, donde podrá ser recargado.

Dentro del sistema se contemplan **tres tipos de vehículos**:

- **Moto**
- **Bicicleta**
- **Patinete eléctrico**

Las **bicicletas** y **patinetes** comparten un comportamiento casi idéntico, diferenciándose solo en algunos parámetros internos, como la capacidad máxima de la batería. Ambos tipos tienen una restricción importante: solo pueden desplazarse de una base a otra. Esto simula la lógica de un sistema de estacionamiento estructurado, donde el usuario debe recoger y dejar estos vehículos en puntos específicos.

Las **motos**, en cambio, ofrecen una mayor libertad de movimiento. Pueden dirigirse a cualquier casilla del entorno simulado, sin necesidad de limitarse a las bases, reflejando una flexibilidad operativa mayor dentro del sistema de movilidad.

Persona

Las personas constituyen una subclase de las entidades móviles. Cada instancia de esta clase recibe un nombre generado aleatoriamente, dado que toda persona en el mundo real posee un identificador único y personal.

En la simulación se distinguen **dos tipos principales de personas**, con funciones y responsabilidades muy diferenciadas:

- **Usuario**

Los usuarios son las personas que utilizan los servicios ofrecidos por el Sistema de Gestión de Movilidad Sostenible. Son las únicas entidades capaces de utilizar vehículos para desplazarse por la ciudad.

Cuentan con atributos propios, como su saldo en el sistema, el registro de alquileres realizados, y el estado de usuario estándar o premium, el cual se determina según su historial de actividad. Además, los usuarios son clave para el proceso de detección de fallos mecánicos: cuando intentan usar un vehículo con problemas, abortan su trayecto y notifican la incidencia a través de la aplicación, activando así la correspondiente alerta de fallo.

- **Trabajador**

Los trabajadores se encargan del mantenimiento tanto de la infraestructura del sistema (bases) como de los vehículos. Son imprescindibles para garantizar la continuidad del servicio: sin su intervención, el uso continuado llevaría inevitablemente a fallos mecánicos y al agotamiento de las baterías, dejando inoperativos a muchos componentes del sistema.
Cada trabajador puede atender únicamente una entidad a la vez, y registra un historial de los trabajos realizados, incluyendo duración y remuneración. Cada uno posee una tarifa base y un precio por hora.

Existen dos especializaciones de trabajador, con funciones claramente definidas:

  - **Mecánico**

    Son responsables de reparar fallos mecánicos, tanto en bases como en vehículos. No tienen permiso para intervenir en la gestión energética de los vehículos, es decir, no pueden recargar baterías.

- **Técnico de mantenimiento**

Su función principal es la recarga de baterías de los vehículos eléctricos. Cuando un vehículo agota su batería, esta envía automáticamente una notificación gracias a su conectividad. El técnico recibe esta alerta, se desplaza hasta la ubicación del vehículo, lo traslada a la base más cercana y realiza la recarga.
Dado que las fallas en las bases pueden comprometer seriamente el funcionamiento del sistema (por ejemplo, impidiendo la reparación de otros vehículos), los técnicos también pueden encargarse de reparar bases si no hay mecánicos disponibles.

○ Conclusión

La realización de este proyecto ha sido clave para comprender los principios de la Programación Orientada a Objetos (POO). Durante el desarrollo del Sistema de Gestión de Movilidad Sostenible se aplicaron de manera práctica los conocimientos adquiridos en la asignatura.

Un aspecto significativo ha sido modelar las entidades y sus interacciones de forma abstracta, representando un sistema real mediante objetos y clases. La creación de clases que simulan vehículos, trabajadores, bases, usuarios y otros componentes ha permitido cumplir con los objetivos académicos y experimentar con la flexibilidad que la POO ofrece para resolver problemas reales.

Además, la inclusión de simulaciones ha permitido explorar la modelización de sistemas dinámicos. A través de la simulación de un sistema de movilidad urbana, se ha podido observar cómo los componentes interactúan entre sí, cómo el tiempo, el estado mecánico de las entidades y las baterías afectan al rendimiento global, y cómo los eventos inesperados, como los fallos mecánicos pueden gestionarse de manera controlada por los propios agentes del sistema, sin necesidad de intervención externa.

A lo largo de mi experiencia en programación durante los últimos años he desarrollado proyectos similares en términos de funcionalidad y complejidad, pero siempre sin utilizar la **POO**. Esto provocó dificultades en el mantenimiento y expansión del código, ya que a medida que las aplicaciones crecían, el código se volvía más desorganizado y caótico, dificultando la integración de nuevas funcionalidades y la resolución de problemas sin generar más inconvenientes.

Este proyecto ha evidenciado las ventajas de trabajar con un enfoque orientado a objetos, ya que se ha logrado organizar mejor las entidades, sus interacciones y comportamientos, facilitando la expansión y la implementación de nuevas funcionalidades de manera ordenada. De modo que la abstracción y la modularidad han demostrado no solo mejorar la organización del código, sino también la flexibilidad y escalabilidad de los proyectos a largo plazo.

En resumen, este proyecto ha sido una excelente oportunidad para consolidar los conceptos de POO, entender su aplicación en escenarios prácticos y gestionar la complejidad de sistemas reales mediante la programación. La simulación de un sistema de movilidad sostenible ha sido enriquecedora desde el punto de vista técnico y filosófico, ayudando a comprender la relación entre teoría y práctica en la programación, además de proporcionar una experiencia valiosa para futuras aplicaciones más complejas.