Imagen en blanco y negro

Descripción generada automáticamente con confianza media

**Patrones Software – Sistema de peajes**

Integrante 1:

Víctor Sanavia Valdeolivas

03202543T

[victor.sanavia@edu.uah.es](mailto:victor.sanavia@edu.uah.es)

Integrante 2:

Adrián Rodríguez Hurtado

09064004A

[a.rodriguezh@edu.uah.es](mailto:a.rodriguezh@edu.uah.es)

ÍNDICE

[Enunciado y Requisitos 3](#_Toc186377719)

[Introducción 3](#_Toc186377720)

[Descripción de la aplicación 3](#_Toc186377721)

[Objetivos 3](#_Toc186377722)

[Características principales 4](#_Toc186377723)

[Requisitos funcionales y técnicos 4](#_Toc186377724)

[Innovaciones 5](#_Toc186377725)

[Manual de usuario 6](#_Toc186377726)

[Concurrencia 6](#_Toc186377727)

[Distribuida 8](#_Toc186377728)

[Diseño de la aplicación 10](#_Toc186377729)

[Patrones de creación 10](#_Toc186377730)

[Patrón Factory Method 10](#_Toc186377731)

[Patrón Singleton 12](#_Toc186377732)

[Patrones estructurales 13](#_Toc186377733)

[Patrón Adapter 13](#_Toc186377734)

[Patrón Facade 15](#_Toc186377735)

[Patrón Flyweight 17](#_Toc186377736)

[Patrones de comportamiento 19](#_Toc186377737)

[Patrón Iterator 19](#_Toc186377738)

[Patrón State 21](#_Toc186377739)

[Patrón Strategy 23](#_Toc186377740)

[Aplicación 25](#_Toc186377741)

[Clases de concurrencia 25](#_Toc186377742)

[Clases de distribuida 26](#_Toc186377743)

[Uso de Java 27](#_Toc186377744)

# Enunciado y Requisitos

## Introducción

Nuestro proyecto tiene como objetivo la implementación de un programa que simule el funcionamiento de un sistema de peaje de carreteras. A través de este software, se busca representar el flujo de vehículos que pasan por el peaje y la interacción con los empleados que trabajan en las cabinas. El sistema contará con diferentes tipos de vehículos y estaciones de peaje, permitiendo observar el proceso de cobro, la ocupación de cabinas y el comportamiento de los empleados. Además, tendrá una interfaz gráfica y la opción de un módulo de consulta remota.

## Descripción de la aplicación

La aplicación es una simulación en Java del sistema de peaje, en el que múltiples vehículos (coches, camiones y ambulancias) se dirigen hacia las cabinas para realizar el pago correspondiente. Los vehículos se distribuyen en cabinas específicas de acuerdo con su tipo y a las modalidades de pago, y son atendidos en orden de llegada. En particular, el sistema otorga prioridad a las ambulancias para que accedan de manera inmediata en situaciones de emergencia.

El comportamiento de los vehículos se registra en un archivo de log y se visualiza en la interfaz. Además, el sistema incluye un módulo de consulta remota que permite visualizar el estado del peaje en tiempo real desde una ubicación externa, así como habilitar o deshabilitar cabinas de pago con tarjeta.

## Objetivos

* **Desarrollar** una simulación funcional del sistema de peaje, que represente vehículos y empleados como hilos.
* **Implementar** una interfaz gráfica intuitiva que permita observar el estado de cada cabina y de cada vehículo en el sistema.
* **Crear** un módulo de consulta remota que permita supervisar el peaje desde un cliente externo.
* **Registrar** en un log el comportamiento de cada vehículo y empleado, facilitando el análisis del sistema.
* **Flexibilizar** el programa para la aplicación de distintos patrones software.

## Características principales

* **Simulación por hilos**: Cada vehículo y empleado se representa mediante un hilo, permitiendo una simulación realista de concurrencia.
* **Asignación de cabinas**: Los vehículos son dirigidos a cabinas específicas, con distinción de tipo de pago (manual o automático).
* **Registro de eventos**: Cada evento en el sistema se guarda en un log con el momento exacto de la acción, facilitando el análisis de comportamiento.
* **Interfaz gráfica**: Se cuenta con una interfaz que muestra el estado de cada cabina, la posición de los vehículos y el estado de cada empleado.
* **Módulo de consulta remota**: Permite la supervisión remota del sistema en tiempo real, actualizando cada segundo.

## Requisitos funcionales y técnicos

**Requisitos Funcionales:**

* **Simulación del flujo de vehículos** hacia el peaje, incluyendo la asignación y cobro de acuerdo con su tipo.
* **Prioridad para vehículos de emergencia**, como las ambulancias, que siempre avanzan en la fila.
* **Interfaz gráfica** para visualizar en tiempo real el estado del peaje, cabinas y vehículos en espera.
* **Log de eventos** para almacenar todos los movimientos y operaciones realizadas en el peaje.
* **Módulo de consulta remota** que permita observar el sistema desde otro dispositivo, con la capacidad de abrir o cerrar cabinas.

**Requisitos Técnicos:**

* **Programación en Java**, usando hilos para simular el comportamiento concurrente.
* **Uso de NetBeans** como entorno de desarrollo.
* **Sincronización de hilos** para evitar conflictos en el acceso a las cabinas y en la interacción con los empleados.
* **Mecanismos de comunicación distribuida** para el módulo de consulta remota.
* **Estructura de logs** en un archivo de texto, que registre cada evento con la fecha de la acción.

## Innovaciones

* **Generación dinámica de ambulancias**: Mediante un botón en la interfaz gráfica, el usuario puede simular emergencias, añadiendo ambulancias a la fila de vehículos de manera prioritaria.
* **Módulo de consulta en tiempo real**: La implementación de un sistema de consulta remota permite supervisar el sistema de manera externa, actualizando la información cada segundo.
* **Control de cabinas de manera remota**: A través del módulo de consulta, es posible abrir o cerrar cabinas de pago con tarjeta, ajustando el flujo de la simulación en tiempo real.

# Manual de usuario

El usuario tiene dos formas de ejecutar el programa. Una de ellas es mediante concurrencia, donde el usuario va a poder observar las distintas cabinas de coches y camiones.

## Concurrencia

Interfaz de usuario gráfica

Descripción generada automáticamente

Figura . Interfaz concurrente

Dentro de esta interfaz, el usuario puede parar el programa, congelando así a todos los vehículos que se encuentran dentro de las cabinas y los que están esperando a entrar. Además, existe la opción de poder generar una ambulancia, la cual se situará directamente la primera para entrar debido a la preferencia.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Figura 2. Cabina de coche

También existe la posibilidad de confirmar si un vehículo ha pasado el peaje o no, y comprobar la factura que ha pagado.

Interfaz de usuario gráfica

Descripción generada automáticamente con confianza baja

Figura 3. Coche que ha pasado el peaje

Forma

Descripción generada automáticamente con confianza media

Figura 4. Coche que todavía no ha pasado el peaje

## Distribuida

Interfaz de usuario gráfica

Descripción generada automáticamente

Figura 5. Interfaz distribuida

Dentro de esta interfaz, el usuario puede abrir o cerrar las cabinas automáticas. Si el usuario decide cerrar una cabina, el vehículo que se encuentre dentro de ella finalizará su ejecución y procederá a cerrar ésta, inutilizándola hasta que se vuelva a abrir.

Interfaz de usuario gráfica

Descripción generada automáticamente

Figura 6. Interfaz distribuida con las cabinas automáticas cerradas

Interfaz de usuario gráfica

Descripción generada automáticamente

Figura 7. Interfaz concurrente con las cabinas automáticas cerradas

# Diseño de la aplicación

## Patrones de creación

### Patrón Factory Method

**Factory Method** es un patrón de diseño creacional que proporciona una interfaz para crear objetos en una superclase, mientras permite a las subclases alterar el tipo de objetos que se crearán.

Se utiliza cuando se quiere encapsular la creación compleja, permite flexibilidad/extensibilidad, separa responsabilidades, trabaja con abstracciones y facilita la inyección de dependencias.

El patrón ha sido utilizado en la aplicación para poder crear las distintas cabinas (automáticas y manuales) pero dejando que sean las cabinas las que deciden que clase instanciar.

Diagrama

Descripción generada automáticamente

Figura 8. Diagrama genérico del Factory Method



Figura 9. Diagrama de nuestra aplicación del Factory Method

* FactoriaDeCabinas (Creador): Es la clase encargada de crear el tipo de cabina, ya se automática o manual, en función del tipo que le llegue.
* CabinaManual (Producto concreto): Es la clase que se encarga de todo lo referido a la cabina manual. Para ello, se controla que el empleado entre a la cabina, que el empleado vuelva, que haya un vehículo en la cabina o que el empleado salga a descansar. Además, se controla el ciclo de vida del vehículo en el peaje, el pago del vehículo y el posterior abandono de éste.
* CabinaAutomatica (Producto concreto): Es la clase encargada de controlar la cabina automática, pero al no tener empleado, solo es necesario controlar al vehículo que entre, el respectivo cobro y posterior abandono del vehículo.

### Patrón Singleton

**Singleton** es un patrón de diseño creacional que nos permite asegurarnos de que una clase tenga una única instancia, a la vez que proporciona un punto de acceso global a dicha instancia.

Se utiliza cuando se necesita asegurar que una clase tenga una única instancia accesible globalmente, gestionando recursos compartidos, estados globales o configuraciones centralizadas.

El patrón ha sido utilizado para poder crear un *logger* donde se almacene un historial de los vehículos que han pasado por el peaje con su respectiva hora.

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente con confianza media

Figura 10. Diagrama genérico del Singleton



Figura 11. Diagrama de nuestra aplicación del Singleton

* Log (Singleton): En esta clase de define una instancia la cual será accesible por el resto de las clases sin necesidad de modificar el código

## Patrones estructurales

### Patrón Adapter

**Adapter** es un patrón de diseño estructural que permite la colaboración entre objetos con interfaces incompatibles.

Se utiliza cuando se necesita hacer compatible una interfaz existente con otra sin modificar su código, permitiendo que clases con interfaces incompatibles trabajen juntas.

El patrón ha sido utilizado para poder realizar una conversión de euros a libras esterlinas en la factura.

Diagrama

Descripción generada automáticamente

Figura 12. Diagrama genérico del Adapter



Figura 13. Diagrama de nuestra aplicación del Adapter

* Recibo (Objetivo): Es la interfaz donde simplemente tiene un método para obtener el precio.
* AdaptadorDinero (Adaptador): Es la clase encargada de recibir la cantidad de dinero a pagar y convertirla a libras (multiplicando por 0.83).

### Patrón Facade

**Facade** es un patrón de diseño estructural que proporciona una interfaz simplificada a una biblioteca, un framework o cualquier otro grupo complejo de clases.

Se utiliza cuando se necesita proporcionar una interfaz simplificada y unificada para un sistema complejo de subsistemas, facilitando su uso y reduciendo la dependencia directa de las clases internas.

El patrón ha sido utilizado para facilitar el acceso a la factura generada por el peaje, definiendo 3 clases del subsistema, de las cuales la clase fachada se hará responsable.

Diagrama

Descripción generada automáticamente

Figura 14. Diagrama genérico del Facade



Figura 15. Diagrama de nuestra aplicación del Facade

* FachadaFactura (Fachada): Es la clase fachada del sistema. Esta clase conoce las clases del subsistema y se encarga de delegar las peticiones en los objetos del subsistema.
* TarifaFactura (Clase del subsistema): Esta clase se encarga de devolver de manera aleatoria el tipo de tarifa que será (estándar, reducida o premium).
* GeneradorPrecio (Clase del subsistema): Esta clase es la encargada de generar el precio base en función del tipo de vehículo, ya que para coches será de 3 y para camiones será de 5.
* GeneradorFactura (Clase del subsistema): Esta clase simplemente se encarga de generar la factura con el precio que ha sido asignado.

### Patrón Flyweight

**Flyweight** es un patrón de diseño estructural que te permite mantener más objetos dentro de la cantidad disponible de RAM compartiendo las partes comunes del estado entre varios objetos en lugar de mantener toda la información en cada objeto.

Se utiliza cuando se necesita minimizar el uso de memoria al compartir datos comunes entre múltiples objetos similares, optimizando recursos en aplicaciones con gran cantidad de objetos.

El patrón ha sido utilizado para poder crear los distintos tipos de vehículos en el caso de que no existan, optimizando así la gestión de los vehículos, compartiendo recursos comunes y minimizando el uso excesivo de memoria.

Diagrama

Descripción generada automáticamente

Figura 16. Diagrama genérico del Flyweight



Figura 17. Diagrama de nuestra aplicación del Flyweight

* FactoriaVehiculos (Fábrica de pesos ligeros): Es la clase encargada de la gestión de la creación y reutilización de los objetos de tipo vehículo basados en su tipo (coche o camión) y su identificador.

## Patrones de comportamiento

### Patrón Iterator

**Iterator** es un patrón de diseño de comportamiento que te permite recorrer elementos de una colección sin exponer su representación subyacente (lista, pila, árbol, etc.).

Se utiliza cuando se necesita recorrer elementos de una colección de forma secuencial y uniforme, sin exponer detalles internos de la estructura de datos. Es ideal cuando deseas encapsular la lógica de iteración en un objeto separado y permitir operaciones de acceso a elementos mediante un enfoque consistente.

El patrón ha sido utilizado para poder acceder secuencialmente a todos los vehículos, ya que se encuentran dentro de una lista, por lo que tendremos una forma uniforme, coherente e independiente de desplazarnos por los elementos de una colección.

Diagrama

Descripción generada automáticamente

Figura 18. Diagrama genérico del Iterator



Figura 19. Diagrama de nuestra aplicación del Iterator

* Agregado (Agregado): Esta interfaz es utilizada para poder definir el método de fabricación para crear un iterador.
* AgregadoVehiculos (Agregado concreto): Esta clase implementa la interfaz Agregado y crea los iteradores concretos para los vehículos.
* Iterador (Iterador): Esta interfaz es utilizada para definir los métodos de iteración. En nuestro caso, hemos definido los métodos hayMas y siguiente.
* IteradorVehiculos (Iterador concreto): Es la clase que implementa la interfaz Iterador para proporcionar una implementación específica de la iteración sobre una colección de vehículos.

### Patrón State

**State** es un patrón de diseño de comportamiento que permite a un objeto alterar su comportamiento cuando su estado interno cambia. Parece como si el objeto cambiara su clase.

Se utiliza cuando una clase necesita cambiar su comportamiento en función del estado en el que se encuentra. Permite definir un objeto con varios estados distintos y transitar entre ellos de manera controlada, proporcionando una forma de encapsular la lógica específica de cada estado sin depender de estructuras condicionales complejas.

El patrón ha sido utilizado para poder controlar el estado de la cabina, ya que ésta se puede encontrar tanto abierta como cerrada.

Diagrama

Descripción generada automáticamente

Figura 20. Diagrama genérico del State



Figura 21. Diagrama de nuestra aplicación del State

* Estado (Estado): Esta interfaz define los métodos que dependen del estado del objeto (abierto o cerrado).
* EstadoCabinaAbierta (Estado concreto): Esta clase implementa el comportamiento específico de la cabina abierta.
* EstadoCabinaCerrada (Estado concreto): Esta clase implementa el comportamiento específico de la cabina cerrada.

### Patrón Strategy

**Strategy** es un patrón de diseño de comportamiento que te permite definir una familia de algoritmos, colocar cada uno de ellos en una clase separada y hacer sus objetos intercambiables.

Se utiliza cuando se necesita variar o extender el comportamiento de una clase en función de diferentes algoritmos o estrategias, permitiendo a las clases delegar su lógica a diversas implementaciones específicas.

El patrón ha sido utilizado para poder definir tres tipos de estrategias para ajustar el precio de la factura.

Diagrama

Descripción generada automáticamente

Figura 22. Diagrama genérico del Strategy



Figura 23. Diagrama de nuestra aplicación del Strategy

* Contexto (Contexto): Esta clase utiliza las diferentes estrategias para las distintas tareas. Mantiene una referencia a la instancia Estrategia, y tiene un método para reemplazar la actual instancia de Estrategia.
* Estrategia (Estrategia): Esta interfaz define todos los métodos disponibles (en nuestro caso, el método es calcular).
* EstrategiaEstandar (Estrategia concreta): Esta clase implementa el comportamiento específico de la estrategia estándar (añadir el 21% al precio).
* EstrategiaPremium (Estrategia concreta): Esta clase implementa el comportamiento específico de la estrategia premium (añadir el 5% al precio).
* EstrategiaReducida (Estrategia concreta): Esta clase implementa el comportamiento específico de la estrategia reducida (añadir el 10% al precio).

# Aplicación

## Clases de concurrencia



Figura 24. Diagrama de concurrencia

## Clases de distribuida



Figura 25. Diagrama de distribuida

## Uso de Java