Programación Orientada a Objetos 2
UNQ

Trabajo Final
Sistema de Estacionamiento Medido
1er. Sem. 2024

Integrantes:
- Gallippi Valentina (valentina.gallippi@gmail.com)
- Iglesias Melina (melinaiglesias03@gmail.com)
- Roldan Marcos (guillermoroldan0@gmail.com)

Decisiones de diseño:

- La lista de estacionamientos se encarga de almacenar todos los estacionamientos del día, independientemente de si están vigentes o no.

- El modo de la aplicación por defecto será el manual, con la asistencia desactivada.

- El modo de la aplicación del conductor puede ser cambiado mientras no tenga un estacionamiento en curso.

- El saldo de un celular nunca será negativo, ya que se calcula la hora máxima en función de su saldo.

- La carga de un celular nunca será un número negativo.

- Los tickets de compra no pueden ser modificados después de haber sido creados.

- Un inspector no puede cambiar su número de legajo después de haber sido instanciado a través de su clase.

- Asumimos que el estado inicial por defecto será "walking" porque, una vez que se instancia la aplicación, recibirá mensajes de `walking()` o `driving()` de acuerdo a su estado actual. Es decir, mantendrá el estado correcto sin ocasionar acciones externas que afecten al protocolo.

- Asumimos que `iniciarEstacionamiento` y `finalizarEstacionamiento` se van a utilizar dentro del rango de horas dadas por el SEM

Detalles de implementación:

- No contemplamos los casos en los que una búsqueda no pueda devolver ningún elemento. Siempre asumimos que al realizar búsquedas se encontrará el objeto deseado, evitando así el uso de excepciones. Esta decisión fue acordada con el profesor Cano.

- Los test que verifican si un estacionamiento está vigente, darán error luego de las 20:00hs por el uso del `LocalDate.now()`.

- El método `notificarEntidades()` queda a disposición para ser utilizado en una extensión, ya que no está especificado en qué momento debe realizarse la notificación.

- El número de celular del estacionamiento puntual retorna 0, ya que estacionamientoPuntual no tiene ningún celular asociado. Necesitamos que el método `getCelular()` esté definido en esta clase para el correcto funcionamiento de una búsqueda implementada. De todas maneras, el mensaje `getCelular()` nunca llegará a estacionamientoPuntual debido a la filtración previa de una lista.

Patrones de diseño utilizados y sus roles

En este trabajo, se aplicaron los patrones Observer, Strategy y State, los cuales se explicarán a continuación junto con sus roles específicos:

Observer

Este patrón define una relación de dependencia uno-a-muchos entre objetos, de manera que cuando un objeto cambia de estado, todos sus dependientes son notificados y actualizados automáticamente. Esto sucede en la aparición de entidades que deben ser notificadas ante un evento que suceda en SEM. De este modo, SEM toma el rol de observable, manteniendo una lista de observadores, las entidades.

subject: SEM
observers: Entidad.

Permite mantener actualizados a los objetos observadores con el estado de un sujeto.

State

El patrón de diseño State permite a un objeto cambiar su comportamiento cuando su estado interno cambia. Esto se ve en el estado que verifica la aplicación, driving y walking, donde toman los siguientes roles:

context: AppConductor.
state: Estado
concreteStateA: Walking.
concreteStateB: Driving.

Permite que AppConductor adapte su comportamiento dinámicamente según su estado interno.

Strategy

El patrón de diseño Strategy permite definir una familia de algoritmos, encapsular cada uno de ellos y hacerlos intercambiables. En este trabajo, lo aplicamos en dos casos:

En primer lugar, se puede cambiar el modo de la aplicación al deseado, ya sea manual o automático, aplicando el método `setModo()`, lo que ejemplifica el uso de este patrón de diseño.

context: AppConductor
strategy: ModoDeAppConductor
concreteStrategyA: ModoManual
concreteStrategyB: ModoAutomatico

Permite que AppConductor cambie su comportamiento seleccionando el modo adecuado de funcionamiento según las necesidades.

En segundo lugar, tenemos el mismo contexto, AppConductor quien gestiona una asistencia que puede ser activada o desactivada:

context: AppConductor
strategy: Asistencia
concreteStrategyA: AsistenciaActivada
concreteStrategyB: AsistenciaDesactivada

Permite a AppConductor alternar entre diferentes estados de asistencia, asegurando que el comportamiento de la aplicación se adapte a las necesidades del usuario.