Informe TP final

Objetos 2

Integrantes

- Brian Ramirez

Repositorio

https://github.com/ramirez7358/tp-final-obj-2

Propósito de este documento

El propósito de este documento es explicar las decisiones de diseño que fueron tomadas en el desarrollo de este trabajo práctico. También hacer una lista de los patrones de diseño implementados, junto a una explicación de los roles de cada clase dentro del patrón.

Desarrollo

El primer paso en el desarrollo del trabajo práctico fue hacer el diseño de la solución utilizando el diagrama de clases UML. El primer diseño era muy distinto al que está ahora, sin embargo me sirvió para poder entender mejor qué es lo que se esperaba del sistema.

Una vez terminada la primera versión del UML comenzó a implementar la solución en Java. En el proceso de la implementación en código surgieron muchos cambios, ya que encontramos algunos puntos en los que mi diseño no estaba siendo útil.

En un comienzo la clase centro del sistema llamada **ParkingSystem** tenía muchas responsabilidades por lo cual comencé primero por empezar a sacar algunas de las responsabilidades llevándolas a otras clases.

Otros de los cambios que surgieron a partir de la implantación fue el uso de la clase **AppSEM** como punto de entrada del sistema por app. Al comienzo tenía una clase **Cellphone** que representaba a un celular el cual tenía aplicaciones. El mismo no tenía ninguna responsabilidad en particular sino que solamente funcionaba como una especie de clase intermedia entre el usuario y la aplicación en sí misma.

Finalmente uno de los cambios más importantes es que en un principio no se había detectado el patrón State para manejar los estados de la aplicación cuando el usuario está manejando o caminando. El uso de este patrón para mi solución se va a explicar más adelante en el apartado de los patrones detectados en este trabajo.

Patrones de diseño encontrados

A continuación voy a listar los patrones de diseño encontrados en mi solución para este sistema. Luego una explicación de cada uno de ellos indicando los roles de cada clase participante en el patrón.

- Observer
- Strategy
- State
- Singleton
- Builder
- Template Method

Observer

Aplique este patrón para manejar las alertas de inicio y fin de estacionamiento, tanto para los estacionamiento que se hacen por compra puntual como para los que se hacen por aplicación. La suscripción también permite recibir alertas cuando se compra crédito.

Los roles son los siguientes:

- Clase ParkingSystem : Concrete subject
- Interfaz AlertListener : Observer
- Clase AlertManager : Concrete observer

Strategy

Este patrón fue utilizado para resolver qué hacer cuando se detecta un cambio de estado en la app (de manejando a caminando o viceversa). Dependiendo la estrategia que tenga la clase **AppSEM** se hacía algo distinto. Gracias a este patrón puede ejecutar un comportamiento distinto dependiendo si la App está en modo manual o en modo automático.

Los roles son los siguientes:

- Clase AppSEM : Context
- Interfaz ModeStrategy : Strategy
- Clases ManualModeStrategy e AutomaticModeStrategy : Concrete strategies

State

El patrón state fue utilizado para representar los estados de la app. Los estados son “caminando” o “manejando” dependiendo en qué estado esté la app, se cambia a la otra y además se ejecuta un comportamiento distinto dependiendo del modo.

Los roles son los siguientes:

- Clase AppSEM : Context
- Interfaz MovementState : State
- Clases DrivingState y WalkingState : Concrete States

Singleton

Este patrón fue utilizado para controlar que la instancia que se maneja del **ParkingSystem** sea siempre la misma. Con resto garantizo un punto de acceso global en la cual se obtendrá la misma instancia de la clase para no reiniciar sus datos, ya que solo me interesa que haya un ParkingSystem.

Los roles son los siguientes:

- Clase ParkingSystem : Singleton

Template method

Si bien en un principio no lo había detectado como un patrón, surgió para poder solucionar un fragmento de código que estaba repitiendo. Lo use en la clase abstracta Parking, la cual es padre de los parkings que se crearon por app o por compra puntual. Ambos para calcular si son vigentes tomaban como cota inferior a la fecha de creación. Por lo tanto lo único que cambiaba en este algoritmo era la fecha de finalización para poder armar el rango. Entonces con un método abstracto delegue a las clases hijas que decidan cuál es su cota superior.

Los roles son los siguientes:

- Clase Parking : Clase abstracta
- Metodo inForce : Template method
- Clases ParkingPerApp y ParkingPerPurchase : Clases concretas
- Método timeLimit : operación primitiva a implementar

Aclaraciones

- La única función de la clase **TimeUtil** es ser un wrapper para los métodos estáticos de la librería LocalDate y LocalDateTime. El motivo fue porque estaba teniendo muchos problemas a la hora de poder mockear estos métodos para poder hacer test.
- En el método **sendNotificacion** de la clase **AppSEM** uso un **System.out** solamente para poder mostrar algo. En este método debería estar el código pertinente para poder mostrarle una notificación al usuario.
- En la clase **AppSEM** uso una lista de **Activity** la cual funciona como un historial de movimientos del usuario.