SISTEMA DE ESTACIONAMIENTO MEDIDO

Grupo 10

INTEGRANTES:

Trejo Julian Daniel:
trejojulian998@gmail.com
De Maio Nicolás:
nicolas.demaio19@gmail.com
Lopez Ulises:
uliseslopez1112@gmail.com

Índice:

Aclaración Sobre el UML:

En el UML al darnos cuenta que había relaciones entre clases e interfaces que se encontraban a mucha distancia una de otra, como no queríamos que quede desprolijo al cruzar todas las flechas, optamos por realizar un duplicado de la interfaz y colocarla cerca de la clase que posee una relación con la misma. Es por esta razón que se observarán algunas interfaces duplicadas.

Modelo y Decisiones de diseño:

Visión general:

Podríamos decir que las clases más importantes de nuestro modelo vienen a ser el SEM (Sistema de estacionamiento medido), la AppUsuario, el PuntoDeVenta y la AppDelMunicipio. La gran mayoría de las funcionalidades del sistema tienen su origen en una de esas 4 clases.

Nuestro mayor desafío fue quizás encontrar una manera en la que todas ellas estuvieran, de alguna forma, interconectadas. Esa conexión surgió en forma de las clases que llamamos Gestores.

Estos gestores son instancias que consideramos únicas en nuestro modelo. La función principal de estas clases es la de abstraer la forma de acceder a los objetos que administra, por ejemplo, tenemos gestores de suscriptores, otro de estacionamientos, y así. Para asegurarnos que estos gestores sean realmente únicos en nuestro sistema lo que hicimos fue lo siguiente, cuando una Clase, para funcionar, debe conocer a una de estas clases administradoras (gestores), recibe como argumento, en su constructor, al SEM tipado con una interfaz que le permita acceder a las referencias que requerirá a lo largo de su ciclo de vida. Estas interfaces son llamadas interfaces de servicio, y su objetivo es no permitir que se pueda acceder a las funcionalidades totales del SEM desde una clase que no tiene por qué conocerlas.

Al ser recibidas desde la clase SEM, todas las instancias de las clases van a poseer la misma referencia a los gestores, logrando así la sensación de unicidad anhelada aunque no se trate de un singleton.

Gestor De saldo:

En el caso de la consulta de saldo, no nos pareció muy alocado el hecho de retornar 0 en caso que el celular no se encuentre registrado. Detener el flujo del programa en esa situación no nos parecía muy buena idea porque en sí, es verdad que el celular que consultó tiene 0 de saldo en el sistema, nunca cargo dinero, y para nuestros fines no nos rompería nada. Recordar que cuando se quiere iniciar un estacionamiento se consulta si el saldo es mayor al precio de estacionamiento de una hora, por ende para estos casos, retornar 0 ya es suficiente para prevenir que se quiera estacionar con un celular no registrado.

En el caso de la disminución del saldo sí nos parece importante retornar una excepción puesto que en este caso el sistema si se ve afectado puesto que se va a querer disminuir saldo a una cuenta que realmente no existe. En este caso sí vemos necesario la utilización de una excepción.

Se asume que el saldo del usuario no puede bajar por debajo de 0. No nos parecía bien contar con un saldo negativo. Además nos basamos en la idea que el dinero que perdería el SEM si, por ejemplo, una persona tiene dinero para 1 hora de estacionamiento, y lo deja por 10 horas, la recuperaría mediante la infracción labrada por el inspector al vehículo.

Estacionamientos:

Los estacionamientos por compra puntual asumimos que se finalizaban solos puesto que conocen su hora de fin. Esto nos permite definir dinámicamente su vigencia.

Los estacionamientos mediante app, en cambio, requieren de un "trigger" para su finalización como bien dice el enunciado. Siendo este "trigger" el SEM cuando finaliza todos los estacionamientos o bien la propia app del usuario cuando finaliza el estacionamiento. Al momento de finalizar todos los estacionamientos nuestro SEM simplemente le indica a las aplicaciones que finalicen los mismos, puesto que los estacionamientos por compra puntual son finalizados de manera automática y asumimos que estos son vendidos por personas en los locales mencionados, y esta persona no vendería un estacionamiento con una hora de fin que sobrepase la hora límite de estacionamientos (ellos deben saber sobre el funcionamiento del sistema).

Como no conocemos su hora de fin hasta que se concrete la finalización del mismo, optamos por indicar que esta es la hora mínima entre la establecida por el SEM y la hora máxima que podría costear el usuario con el saldo que posee. De esta forma el campo no queda en null en su inicialización.

Como decidimos "setear" la hora de fin como mencionamos anteriormente necesitábamos un mecanismo para hacerle saber al inspector si el automóvil que estaba chequeando poseía o no un estacionamiento vigente. Por esta razón se nos ocurrió que en lugar de preguntar solo por vigencia, este preguntaría si se encuentra en regla. Encontrarse en regla significa que el estacionamiento está vigente y además la hora actual es menor a la hora de fin. ¿Por qué nos tomamos las molestias de hacer esto? Porque si no, una persona podría estacionar con 40 pesos de saldo, el sistema la dejaría porque posee saldo para una hora, le indicaría que su estacionamiento finaliza en una hora, pero esta persona podría dejar su automóvil por 10 horas en ese lugar. Para el sistema ese estacionamiento estaría vigente, entonces el inspector no podría multarlo, y eso nos parecía una falencia en el sistema. De esta forma, al preguntar si está vigente y además está en el horario estipulado podemos multar a la gente que estaciona y no posee saldo suficiente para la totalidad de su estadía en el lugar.

Interfaz del Celular:

Decidimos colaborar con una interfaz celular que pueda responder a mensajes que fueron mencionados directa o indirectamente a lo largo del enunciado del TP.

Como se establece que no debíamos preocuparnos por la implementación del GPS y de cómo el celular obtiene su ubicación optamos por convertirlo en una interfaz que representa algo externo a nuestro modelo con lo que vamos a colaborar. Al ser una interfaz también nos permite poder "mockearlo" y testear normalmente nuestra porción del código.

Ejemplo de extracto del enunciado en el que vemos lo recién mencionado:

“asuma que el teléfono puede conocer su posición a través de su GPS”

AppUsuario:

En el caso de la AppUsuario nos vimos obligados a asumir ciertas cosas sobre el enunciado. En el mismo, se dice que cuando se inicia un estacionamiento se le debe indicar a la app la patente. Esto optamos por obviarlo puesto que de hacerlo de esa forma la aplicación no podría funcionar en modo automático porque necesitaría del input de la patente para poder iniciar el estacionamiento. Habiendo visto esto, optamos por guardarnos la patente del vehículo a la hora de instanciar la aplicación y luego utilizamos esa referencia para todos los métodos que la requieran. Algo similar sucedió con el número de celular a la hora de finalizar el estacionamiento.

¿Por qué LocalDateTime?

Por mucho tiempo, los desarrolladores se han quejado de la falta de soporte adecuado de los paquetes relacionados a las fechas y al tiempo, por ejemplo java.util.Date. Entre las razones para sus críticas encontramos que eran propensos a tener problemas de concurrencia, algo con lo que un desarrollador no desea tener que lidiar a la hora de escribir código relacionado al manejo de fechas.
Durante este período, que está contenido durante las versiones de Java previas a Java 8, muchos desarrolladores o bien optaban por crear su propio sistema de fechas, o tenían que utilizar el sistema previamente mencionado con las fallas que acarreaba. Así fue cómo surgió Joda Time, una librería externa que venía para solucionar los problemas mencionados anteriormente.

"The standard date and time classes prior to Java SE 8 are poor. By tackling this problem head-on, Joda-Time became the de facto
Standard date and time library for Java prior to Java SE 8."

Sin embargo, a partir de Java SE 8 estos problemas fueron solucionados con la llegada de java.time(JSR-310) que fue añadida al núcleo de la JDK para solucionar oficialmente todos los problemas relacionados a las clases de Tiempo /Fechas.

Desde su llegada, se convirtió en el standard de facto y es utilizado por la gran mayoría de proyectos. Por esta razón, nosotros no vimos la necesidad de cubrirnos contra el Open Close Principle en este caso. Como es la clase standard a ser utilizada, no vimos conveniente tener que cambiar el modelo para poder adaptarlo para posibles futuros cambios porque justamente, LocalDateTime es una de las clases estándares más utilizadas a partir de Java SE 8 para representar lo relacionado a fechas y tiempo.

Misceláneo:

Notarán que tenemos varias interfaces en Clases que se encargan de almacenar objetos, por ejemplo infracciones. Esto es así porque de momento, como no sabemos sobre persistencia en bases de datos, nos remitimos a guardar los datos en listas o Maps, pero en un futuro quizá

estos roles podrían ser cubiertos por bases de datos. Entonces, para poder cumplir el Open Close Principle, decidimos dejar nuestro sistema abierto a posibles implementaciones futuras.

Si bien clases como Infracción pueden parecer tener síntomas de una data class, consideramos que el enunciado guía hacia una solución como tal. Se describe a la infracción como algo con muchos atributos, pero en ningún momento se habla de ningún comportamiento que pueda tener asociado. Tampoco surge la necesidad de realizar un "Extract Method" en alguno de los métodos de las clases que colaboran con la misma para luego transferir dicho método a la clase Infracción. Nosotros tomamos la decisión de mantener a la infracción como una clase a pesar de lo previamente mencionado porque pensábamos que de esta forma obtendríamos un mejor diseño al modelar mejor las entidades de nuestro dominio. Además, no la vemos tanto como una data class sino como una clase cuyos métodos todavía no han "florecido", es decir, al contar con ella, cualquier nuevo comportamiento a futuro que esté relacionado con infracciones va a poder ser incorporado sin ningún inconveniente en nuestro modelo.

Optamos por guardar una referencia a la zona en el estacionamiento porque en el enunciado se hacer referencia a que el SEM querrá explotar la información geo-temporal de estacionamientos para censar la congestión de estacionamientos en distintas zonas de la ciudad y así mejorar las políticas del sistema. Con la referencia ya incluida en nuestro modelo, esto hará muy sencillo una implementación futura requerida por el cliente.

También tomamos recaudos en lo referido al horario de inicio y fin de los estacionamientos, y además con el precio de los mismos. Dejamos abierto nuestro modelo a cambios en horarios de estacionamientos y cambios en la tarifa cobrada por los mismos. Estos datos están contenidos en el gestor de estacionamientos, y si en un futuro se deseara modificar algo relacionado a los mismos solo tendríamos que tocar allí, no tendríamos que ir a tocar todo nuestro código por haber "hardcodeado" los valores en cada una de las clases. Es decir, de esta forma nos estamos protegiendo del smell Shotgun Surgery.

Cuando en el gestor de zonas se quiere pedir una zona cuando no se encuentra en un punto contenido en alguna Zona, decidimos lanzar una excepción en tiempo de ejecución. Esto es así dado que si se le pasa un punto, el cual no está contenido por ninguna zona que se encuentre en el gestor de zonas, debería lanzar una excepción indicando que no hay una zona para dicho punto.

En cuanto a las notificaciones, como no había mucha información de las mismas en el enunciado nos tomamos la libertad de modelarlas con un método toString() para que los suscriptores puedan tener un acceso rápido a la información de cada una de las notificaciones que reciben.

En lo respectivo a las Respuestas, decidimos hacerlas clases para obtener un mejor diseño orientado a objetos y poder contener los datos que cada una requiere. Al ser clases se puede observar que tienen cierta importancia en el dominio. Esta importancia queda también plasmada en la sección del enunciado que habla de los eventos de inicio y fin de estacionamiento. A su vez, como en la clase Notificación, incluimos un método toString() para brindarle al usuario un acceso rápido al contenido de la misma.

Para modelar el área de las zonas y la inclusión en las mismas decidimos utilizar la clase Polygon. Esta clase ya nos proveía de una manera de chequear la inclusión de un punto dentro de sí misma y nos pareció interesante para nuestro modelo. Lo único que teníamos que hacer para poder utilizarla era pedir que el GPS del celular nos proveyera de un Point para que todo

el modelo funcione correctamente. Decidimos utilizar la clase Point2D ya que es una clase Abstracta que nos permitía mantener nuestros mensajes abiertos a cualquier clase que extienda a la misma.

Patrones de Diseño utilizados:

Observer Pattern:

Si bien no seguimos al pie de la letra la descripción del patrón por razones explicadas a continuación, utilizamos la lógica detrás del patrón observer para solucionar la problemática de notificar a las entidades externas al SEM que desean ser notificadas del inicio/fin de estacionamiento y la carga de saldo.

Nuestra visión:

Nuestro primer enfoque fue hacer que el punto de venta y las aplicaciones de usuario heredarán de Observable, y que los que quisieran ser suscriptores implementaran la interfaz Observer. Sin embargo, este modelo no nos terminó convenciendo mucho puesto que los suscriptores se tendrían que suscribir a 2 observables distintos, y además cada app y cada punto de venta tendrían que poseer a los observers en una lista interna, es decir, tendríamos referencias a los observers en muchas localizaciones al mismo tiempo.

La solución planteada, que fue la que quedó plasmada en el modelo final, fue la siguiente. Decidimos nuclear a todos los suscriptores en un solo lugar, la clase que llamamos GestorDeSuscriptores. El objetivo de esa clase es servir como una especie de “Base de datos” de todos los suscriptores y a su vez actuar como nexo de comunicación entre todos los puntos de ventas y aplicaciones con los suscriptores, los cuales se encontrarán referenciados una única vez en nuestro modelo, dentro del gestor.

La lógica de funcionamiento es la siguiente. Por ejemplo, cuando una aplicación registra un nuevo estacionamiento le envía al gestor de suscriptores que notifique a los suscriptores del evento ocurrido. Este, luego notifica a su lista de observers del cambio ocurrido.

Por lo tanto, podría decirse que basándonos en los roles del libro de Gamma et.al, la Aplicación/Punto de venta y el Gestor de suscriptores compartirían el rol de observable.

- La Aplicación/Punto de venta posee el estado de interés del cual los suscriptores desean ser notificados.
- El Gestor de Suscriptores conoce a los observers y provee un protocolo para añadirlos y quitarlos.
- El envío de la notificación podría definirse como una acción conjunta entre ambas clases.

Por su parte, la interfaz suscriptor es la encargada de cumplir el rol de Observer. Es una interfaz que provee el protocolo necesario para poder ser notificado de los cambios de estado que se generen en el observable. Cuando cada suscriptor concreto implemente esa interfaz, ese suscriptor cumplirá el rol de Observer Concreto.

Si bien no seguimos la receta del patrón al pie de la letra, consideramos que utilizamos nuestro conocimiento del mismo para encontrar una solución que se adecuaba a nuestro modelo.

Strategy Pattern:

En la Clase AppUsuario, nos encontramos que el comportamiento de alertar al usuario del inicio/fin de estacionamiento variaba en base a si la aplicación se encontraba en modo automático o en modo manual. Uno a simple vista podría decir que con un condicional ya es suficiente, pero no, puesto que luego podrían implementarse más modos o "estrategias" distintas, y deberíamos expandir nuestro código siguiendo añadiendo ifs, algo que en no es bien visto en la programación orientada a objetos. Por lo tanto, decidimos utilizar el patrón strategy para encapsular el comportamiento de cada uno de los modos en su propia estrategia. Si en un futuro, apareciera el "modo semiautomático" nosotros simplemente deberíamos añadir una nueva estrategia. Esto también respeta al OCP, y a su vez no expone la lógica de las estrategias, que en algunos casos puede resultar compleja, a los usuarios.

Esta razón de uso se corresponde a una de las del Gamma et.al. :

"A class defines many behaviors, and these appear as multiple conditional statements in its operations. Instead of many conditionals, move related conditional branches into their own Strategy class."

Roles de acuerdo a la definición del libro:

Strategy: El modo. Declara una interfaz común para todos los algoritmos.

ConcreteStrategy: ModoAutomático y ModoManual. Estas son las clases que implementan el algoritmo usando la interfaz Modo.

Context: La AppUsuario. Posee una referencia a una estrategia en todo momento y se encuentra configurado con una de las estrategias.

Elegimos Strategy por sobre State en este caso porque el modo automático y manual no son los responsables de cambiar el estado del contexto. El modo cambia con un mensaje enviado a voluntad del usuario a la AppUsuario.

Del libro:

"Strategy and Context interact to implement the chosen algorithm. A context may pass all data required by the algorithm to the strategy when the algorithm is called. Alternatively, the context can pass itself as an argument to Strategy operations. That lets the strategy call back on the context as required."

En nuestro caso, decidimos optar por la segunda implementación mencionada, enviamos al contexto (AppUsuario) como argumento en la operación de la strategy. Al usar esta estrategia, el contexto debe implementar una interfaz que permita lograr una estrecha relación entre el contexto y la estrategia. Esta interfaz en nuestro modelo viene a ser ModeClient.

State Pattern:

El enunciado menciona que a la app le van a estar llegando constantemente mensajes de driving() o walking() al implementar la interfaz MovementSensor. En cuanto se reciban dos mensajes distintos de manera consecutiva, por ejemplo driving() -> walking() o viceversa, se debería actuar de acuerdo al caso. Esto nos llevó a pensar que podríamos tener un estado que represente a estar caminando y otro que represente estar manejando. Estos dos estados sabrían que hacer cuando les llegara el mensaje que reprensente su estado opuesto, y procedería a realizar dicha acción y luego a cambiar de estado a la aplicación.

Por ejemplo, si el estado actual es Caminando y a la app le llega el mensaje driving(), la app le delega a su estado, y este al haber recibido el mensaje que representa a su estado opuesto, actúa en consecuencia. En este caso, como el estado es Caminando y le llega driving(), quiere decir que debería comunicarle a la app que desencadene la cadena de acciones referidas a finalizar el estacionamiento (si están dadas las condiciones) y además le actualiza el estado a Manejando. Si le llega el mensaje walking(), simplemente no hace nada.

Como el EstadoDeMovimiento, como los llamamos en nuestro modelo, es capaz de saber en qué momento se debe cambiar de estado al contexto, y es responsable de hacerlo, se debe utilizar el patrón State.

Roles de acuerdo a la definición del libro:

Context: AppUsuario.

State: EstadoDeMovimiento.

ConcreteState subclasses: Caminando/Manejando.

Aclaración sobre Mockito y la Versión de JDK utilizada:

Estamos utilizando la versión Mockito 3.6.0 para poder mockear métodos estáticos utilizados de clase LocalDateTime. Al correr los test vimos que salta una warning en consola pero luego de investigar concluimos que no es perjudicial y los tests corren normalmente.
Para el desarrollo del trabajo se utilizó la versión de JDK 14.0.2.