Universidad Nacional de Quilmes

TRABAJO PRÁCTICO FINAL

SISTEMA DE ESTACIONAMIENTO MEDIDO

PROGRAMACIÓN CON OBJETOS II

INFORME

INTEGRANTES:

Díaz, Federico	Legajo 53218
Ferreyra, Valentín	Legajo 55437
Lo Surdo, Braian	Legajo 53036

FECHA DE RE-ENTREGA:

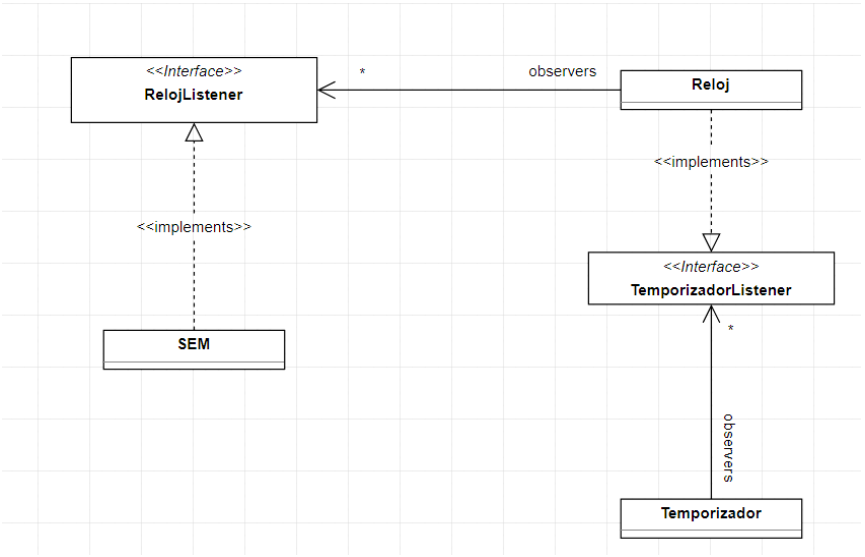
2 de diciembre de 2021

Decisiones de diseño:

- Jerarquía Estacionamiento: Originalmente la jerarquía de Estacionamiento fue planteada con dos clases concretas que heredaban de una abstracta. Sin embargo a medida que iba quedando implementado el código la clase que se había nombrado como *EstacionamientoPuntoVenta* estaba completamente implementada en su clase abstracta ya que los comportamientos que consideramos como default figuraban allí y solo la clase concreta de Estacionamiento App tenía comportamiento distintivo. Si bien se planteó programar métodos completamente abstractos en *EstacionamientoVigente*, para así no tener una clase concreta vacía, nos pareció, para este caso, más prolijo usar overrides sobre los comportamientos default que tener código escrito de más. Si bien se sacrificó escalabilidad hay dos razones por las cuales se decidió que no valía la pena: 1) El proyecto no va a crecer y 2) aunque quisiéramos suponer que el proyecto podría escalar los refactors necesarios para volver a obtener una clase abstracta y las concretas que extendieron de la misma son mínimos.
- Sobre la interface *Aviso Genérico* y las Data Class que la implementan. Hubiésemos preferido hacer una implementación donde una Entidad que escucha al *SEM Monitoreo* ante el evento simplemente guarde en una colección para estacionamientos iniciados o en la de estacionamientos finalizados según corresponda. Sin embargo, interpretamos que la consigna esperaría un eventual comportamiento de ese Aviso genérico y nos sentimos obligados para poder separar a esos estacionamientos asignándoles dentro de una clase Aviso que los representara.
- En el trabajo también dejamos comentarios de ciertas decisiones de las cuales nos pareció buena resolver con un patrón de diseño determinado.

Observer:

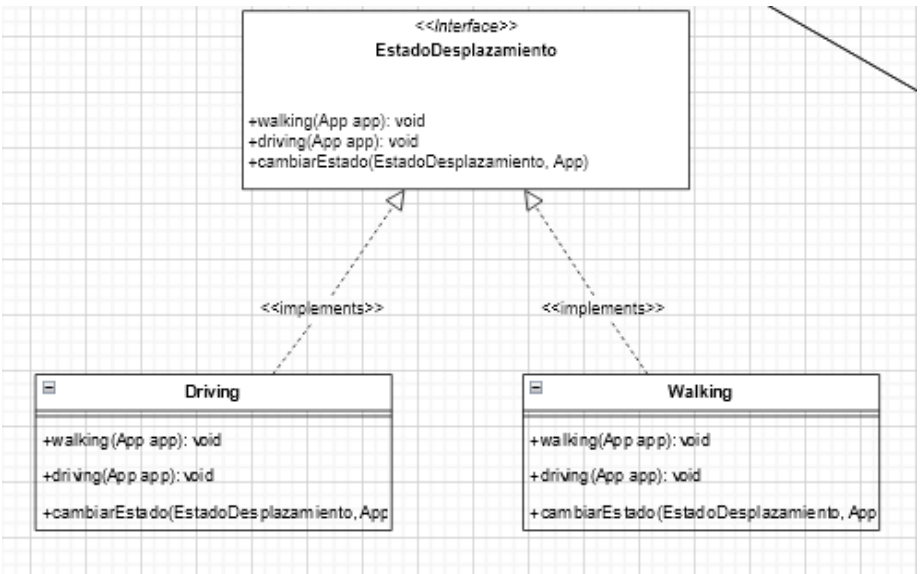
Como uno de los patrones de diseño más importantes que implementamos en nuestro trabajo, fue que el SEM sea un *RelojListener*, ya que al SEM le interesa saber todo el tiempo qué hora es, tanto para estar al tanto del horario para registrar estacionamientos, como el horario para finalizarlos.



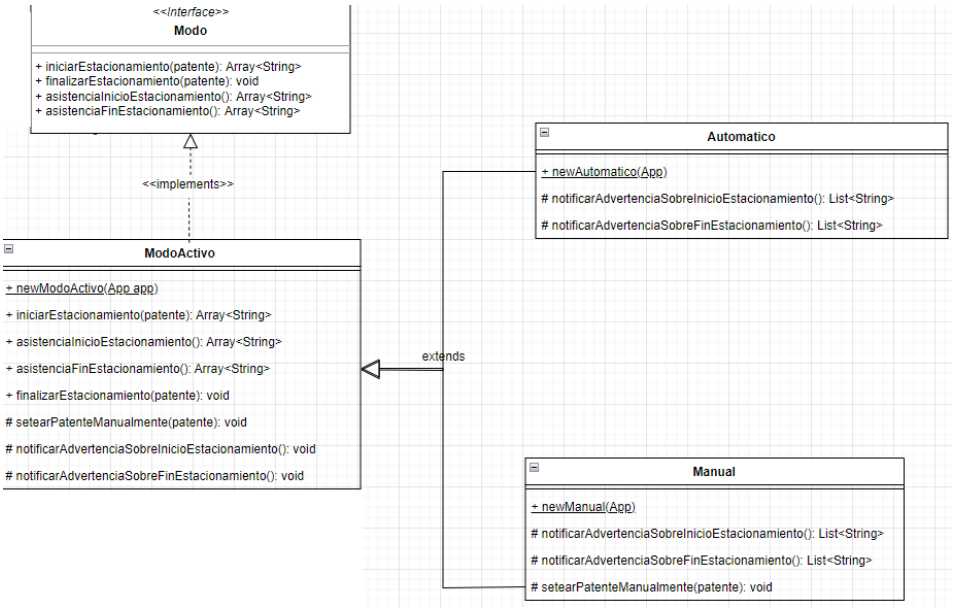
Además, decidimos que Reloj sea un *TemporizadorListener,* lo cual nos ayudó a la hora de armar y desarrollar los test: necesitábamos un objeto que se encargue de hacer transcurrir el tiempo para, por ejemplo, testear el caso en el que son las 20hs y se deben finalizar todos los estacionamientos vigentes. Para ello, gracias al Temporizador podemos simular el transcurso del tiempo, y como el Reloj es un listener, actualiza la hora del SEM.

State:

Se cambió la forma en la que la App interactuaba al recibir los mensajes walking() y driving(), ahora delega el estado en el que se encuentra el usuario (ya sea que esté caminando o conduciendo) a una clase que se encargue de su respectivo comportamiento, por medio del uso del patron de diseño State, en el cual, según el estado actual, enviará el mensaje correspondiente y esta clase se encargará del cambio de estado y la acción



correspondiente, o no hará nada (dependiendo el caso). Por ejemplo, si la clase Walking recibe el mensaje walking(), quiere decir que el usuario continúa caminando, por lo cual no hará nada; en cambio, si recibe el mensaje driving(), quiere decir que hubo un cambio en el movimiento, por lo cual realizará la acción correspondiente (la cual estará delegada a otras clases que se encargarán de su comportamiento específico ya sea que estuviese la App en modo manual o en modo automático) y cambiará el estado de la App a driving, de esta manera, si se vuelve enviar un mensaje de walking() o driving, esta vez será trabajo de la clase driving() realizar las acciones correspondientes.



Template Method:
Aprovechamos un poco las ventajas del patrón de Template Method para realizar el comportamiento de los modos automaticos/manual, de esta forma, mediante la clase ModoActivo, la cual es una clase abstracta, incorporamos toda la lógica principal, las operaciones primitivas correspondientes y un *hook method* el cual nos permite dar el comportamiento necesario para la clase Manual que hereda de este ModoActivo.

Problemas/Retos a la hora de diseñar el código:

Uno de los primeros problemas surgidos de la base del primer UML que hicimos, fue que la clase principal que manejaria todo, el SEM, estaba teniendo demasiadas responsabilidades y métodos que no le correspondían, lo cual implicaba una clara violación del principio Single Responsibility, por suerte la solución fue inmediatamente resuelta, mediante la división de las responsabilidades del SEM en diferentes clases que trabajan independientemente, y las cuales, el SEM delega segun el tipo de

comportamiento que deba ejecutar, de esta manera, el SEM principal pierde la responsabilidad múltiple que tenía, quedandole la unica responsabilidad de delegar a otros las tareas necesarias.

Otro reto que se presentó fue cómo hacer para transcurrir el tiempo y de esta manera poder realizar los diversos test correctamente.
Para esto, la solución que logramos implementar fue que el SEM sea un observer del reloj, para así poder actualizar el tiempo cada vez que pasen las horas, asimismo reloj es un observer de temporizador, de esa forma pudimos simular y adelantar horas para testear casos donde, por ejemplo, necesitábamos que sean las 20hs para finalizar los estacionamientos, o que transcurra 1 hora para verificar que se le cobra una hora más de estacionamiento a los usuarios correctamente.

Sobre el Diseño del UML:

El desarrollo del UML fue el primer paso que dimos como grupo a la hora de discutir de qué manera íbamos a diseñar la solución de nuestro código. Colocamos la mayoría de los métodos que usan nuestras clases, sin embargo, tomamos la decisión de no incluir los métodos getters y setters de las clases en nuestro diagrama para que los comportamientos de ellas sea clara y legible y evitar que se “ensucie” con estos métodos; sí incluimos métodos privados de las clases porque lo vimos necesario y ayudaría a entender el comportamiento de cada uno.
Desde ya, creemos que con esta decisión quien lea el UML asumirá la existencia de los mismos en los casos donde los vea necesario.

Sobre la actividad en GitHub:

El desarrollo del SEM principal y la creación de los distintos gestionadores del SEM fueron realizados al inicio en modo PeerWorking, es decir, trabajando juntos a la par en la implementación de los mensajes básicos, por lo cual se vé reflejado en la actividad como que uno aportó muchas líneas de código, siendo parte de estas en realidad hechas en conjunto.

Además, al distribuir las clases en carpetas, esto provocó que la actividad de GitHub agregada y eliminara código y lo sumara en el número mostrado.