*Universidad Nacional de Quilmes*

*Tecnicatura en Programación Informática*

*Programación con Objetos II - Año 2020*

# Programación con Objetos II

## Trabajo Final: Sistema de Estacionamiento Medido

### Integrantes


- Gonzalo Cortave - `gonza.cortave@hotmail.com`
- Matias Tixeira - `matiastixeira.3240@gmail.com`
- Juan Francisco Perez - `juanf.perez.97@gmail.com`


### Diseño

#### El Sistema De Estacionamiento Medido

Ya que el Sistema De Estacionamiento Medido (SEM) reúne demasiadas tareas y responsabilidades diferentes decidimos separarlo en partes. Esto nos facilita la comunicación entre dichas partes, permite que el sistema sea más flexible y abierto.

El SEM interactúa con interfaces que le permiten conocer todos los elementos registrados de cierto tipo (por ejemplo compras o infracciones), así como operar sobre ellos. Adicionalmente, en varios casos optamos por crear una interfaz especial que exponga sólo la parte de la funcionalidad requerida externamente, pero no permita acceder directamente a todos los elementos registrados. Ejemplo: IControlSaldo.

Esto tambíen permite respetar el principio de *Interface Segregation*, según el cuál agentes externos no deberían conocer funcionalidad que no usan.

La separación en partes del SEM permite también dejar que los agentes externos se comuniquen sólo con la parte que tenga la funcionalidad que necesiten. Por eso optamos por no re-implementar los mensajes de cada parte del SEM en el mismo.

#### El server de estacionamiento

Implementamos un server para que sirva como intermediario entre el SEM y la app. De esta forma, toda la funcionalidad requerida por la app pero no necesariamente por el SEM puede ser ubicada en el server, y se evita que la app conozca varias partes del server, así como que esas partes del server se conozcan entre sí (en caso de que no existiera la necesidad previamente).

La responsabilidad de traducir los pedidos de la app a efectos en el SEM recae ahora sobre el server, y no sobre el SEM ni la app.

Otra característica del server es su modo de responder a los pedidos de inicio y fin de estacionamiento de la app: las respuestas. Ya que necesitabamos un objeto polimórfico que el server pudiera enviar a la app tanto cuando el inicio del estacionamiento es exitoso como cuando falla, implementamos las respuestas.

Las respuestas también sirven para el modo automático de la app: cuando el usuario comienza a caminar o manejar, pero no está en una situación en la que deba iniciar o finalizar un estacionamiento, el modo automático, a través de una respuesta, sabrá si debe comunicarse con el server, y en caso contrario evitará enviar una notificación innecesaria de intento de estacionamiento fallido al usuario.

## Patrones

### State

Usado para modelar los estados de movimiento de la app.

Implementamos el patrón State para el estado de movimiento de la app entendiendo que los métodos `isDriving()` e `isWalking()` sólo modofician el estado de la app, pasando al siguiente estado e indicandole a la app dicho cambio de estado (en caso de ser necesario). De esta forma, la respuesta de la app al cambio de estado (en este caso los mensajes `comenzoAManejar()` y `comenzoACaminar()` ) puede estar separada del mecanismo de cambio de estado en sí.

#### Roles

- **Context** -> `AppEstacionamiento`
- **State** -> `EstadoDeMovimiento`
- **ConcreteState** -> `Manejando` y `Caminando`

### Strategy

En el caso de la activación y del modo de alerta de la App, nos encontramos con que dependiendo en que modo elija el usuario estar, la app debe comportarse de manera diferente. Usamos Strategy ya que nos facilita definir un algoritmo para cada modo y configurar la app tal que se responda a los cambios de estado de la manera deseada.

La razón por la que usamos Stratagy en lugar de State en estos casos es porque los distintos tipos de modos pueden ser configurados en cualquier momento desde la app, y este cambio en la configuración no conlleva una respuesta inmediata. A diferencia del caso del estado de moviemiento, donde una acción específica dispara el cambio de estado, y este no puede ser realizado arbitrariamente. El estado de movimiento de la app debe ser consistente con la realidad,

mientras que los modos son a voluntad del usuario.

#### Roles en ModoDeActivacion

- **Context** -> `AppEstacionamiento`
- **Strategy** -> `ModoDeActivacion`
- **ConcreteStrategy** -> `ModoManual` y `ModoAutomatico`

#### Roles en ModoDeAlerta

- **Context** -> `AppEstacionamiento`
- **Strategy** -> `ModoDeAlerta`
- **ConcreteStrategy** -> `AlertaActivada` y `AlertaDesactivada`

### Observer

Decidimos utilizar un patrón Observer para gestionar las suscripciones debido a que todos los suscriptores son los observadores que aguardan ser notificados de algún evento. El observado en este caso es el gestor de suscripciones, que es capaz de alertar de distintos eventos a lo largo del SEM: tan sólo es necesario que luego de producido el evento, se le envíe un mensaje al gestor, y este a su vez enviará una alerta a los suscriptores.

La flexibilidad del patrón permite que cualquier objeto externo se acople al sistema, tan sólo implementando una interfaz ( `Suscriptor` ) con un único mensaje, e indicando al gestor de que quiere ser notificado.

### Roles

- **Subject** -> `IGestorDeSuscripcion`
- **ConcreteSubject** -> `GestorSuscripcion`
- **Observer** -> `Suscriptor`
- **ConcreteObserver** -> las entidades que deseen suscribirse e implementen la interfaz `Suscriptor`