Trabajo Práctico N°2 - GPS Challenge

[75.07/95.02] Algoritmos y Programación III - Cátedra Suarez
Primer cuatrimestre de 2022
Facultad de Ingeniería de la Universidad de Buenos Aires

GRUPO N°21		
Apellido y nombre	**Padrón**	**Email**
Gallino, Pedro	107587	pgallino@fi.uba.ar
Campillay, Edgar Matías	106691	ecampillay@fi.uba.ar
Abuin, Aquiles	107742	aeabuin@fi.uba.ar
Brizuela, Valentin	108071	vabrizuela@fi.uba.ar

Repositorio GITHUB	algo3_tp2
Correctora	Maia Naftali

Índice

1. Introducción
2. Supuestos
3. Modelo de dominio y detalles de implementación
4. Diagramas de clase
5. Bibliografía y webgrafía

INTRODUCCIÓN

El presente informe reúne la documentación de la solución del trabajo práctico número 2 de la materia Algoritmos y Programación III que consiste en desarrollar un juego llamado **GPS *Challenge*** en Java utilizando los conceptos, pilares y patrones propios del paradigma de la programación orientada a objetos estudiados a lo largo del curso.

GPS Challenge es un juego de estrategia por turnos, donde el escenario es una ciudad y el objetivo del juego, es guiar un vehículo a la meta en la menor cantidad de movimientos posibles. En cada turno el usuario debe decidir hacia cuál de las cuatro esquinas adyacentes disponibles avanzará.

SUPUESTOS

1. Las calles son de **doble circulación**.
2. Se desarrollan los casos de uso pedidos donde se presentan escenarios reducidos para hacer posible su ejecución.
3. Independientemente de la presencia de obstáculos y/o sorpresas, trasladarse de una esquina a otra suma un movimiento. En el caso específico del Piquete, si bien el el auto y la camioneta no pueden pasar, se les suma un movimiento de todas formas.
4. Existen varios tipos de partidas, **fácil, medio y difícil**. Donde varía el tamaño del mapa y la cantidad de obstáculos y sorpresas presentes.

MODELO DE DOMINIO Y DETALLES DE IMPLEMENTACIÓN

Generación del juego

Para la generación del juego se eligió utilizar una adaptación del **patrón de diseño creacional *Builder***, dicho uso del patrón permite al desarrollador acceder a una mayor personalización del juego. Haciendo uso de una clase directora `JuegoDirector` y los *setters* propios de la clase constructora, `JuegoConstructor`, se da la posibilidad de modelar en una instancia de `Juego` con escenarios iniciales distintos. Además, con el uso de un *getter* se permite al usuario obtener la instancia del juego sobre la cual se realizaron los *seteos*.

Ejemplo de creación de partida normal utilizando un auto como vehículo inicial.

```
JuegoDirector director = new JuegoDirector(raking);
director.crearPartidaFacil();
Juego juego = director.obtenerPartida();
```

Código dentro de la clase `director` que permitió generar una partida de dichas características.

```
public void configurarPartidaFacil() {
   constructor.asignarLongitudMapa(10)
           .agregarPozos(5)
```

```
            .agregarPiquetes(5)
            .agregarControlesPoliciales(5)
            .agregarSorpresasFavorables(10)
            .agregarSorpresasDesfavorables(5)
            .agregarSorpresasCambioDeVehiculo(5)
            .agregarMetaEn(COORDENADA_META)
            .asignarRanking(this.ranking)
            .asignarVehiculoInicial();
}
```

Diagrama de secuencia de la operación antes mencionada.

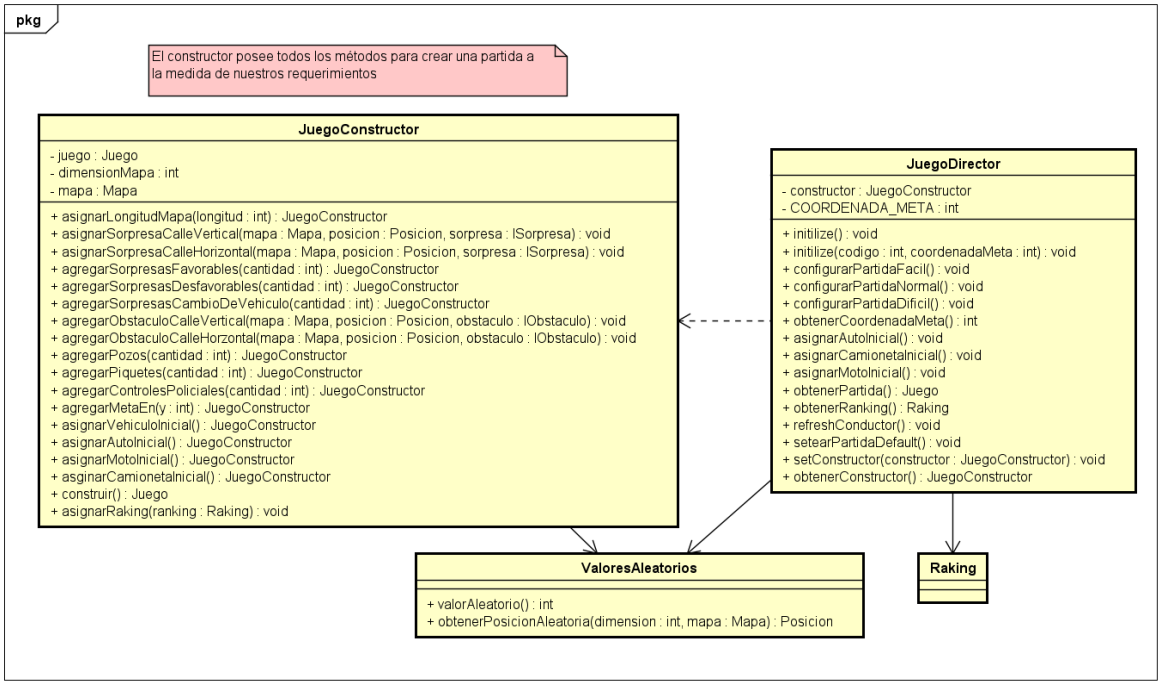


Diagrama de clase aislado de la clase `Constructor` y la `Director`

Utilizando los métodos propios de `Director`, `configurarPartidaDificil()`, `configurarPartidaFacil()`, se puede generar partidas con mayor y menor dificultad, considerando el cambio de dificultad como la variación entre los distintos tamaños del mapa, la cantidad de sorpresas y obstáculos disponibles en el juego (estos últimos generados en posiciones aleatorias generadas por la función `obtenerPosicionAleatoria(dimension)`).

Creación de obstáculos

Para la generación de `SorpresasPuntaje` se aplicó el patrón de diseño creacional ***Factory Method*** en conjunto con una implementación de ***Null Object Pattern***.

Factory method

A través de las interfaces `ISorpresaFabrica` e `ISorpresa` se lleva a cabo de una manera más ordenada dentro del uso del constructor del juego, `JuegoConstructor`, la instanciación de sorpresas modificadoras de puntaje. Estas permiten a través de una única sorpresa genérica `SorpresaPuntaje`, obtener las sorpresas favorables y desfavorables con solo una función.

```
public class SorpresaFavorableFabrica implements ISorpresaFabrica {
   private static final double VALOR_SORPRESA_FAVORABLE = 1.25;

   @Override
   public ISorpresa crearSorpresa() {
       SorpresaPuntaje sorpresa = new SorpresaPuntaje();
       sorpresa.asignarValor(VALOR_SORPRESA_FAVORABLE);
       return sorpresa;
   }
}
```

Diagrama de clases de los obstáculos, sorpresas y las relaciones entre esta con las clases tipo "fábrica".

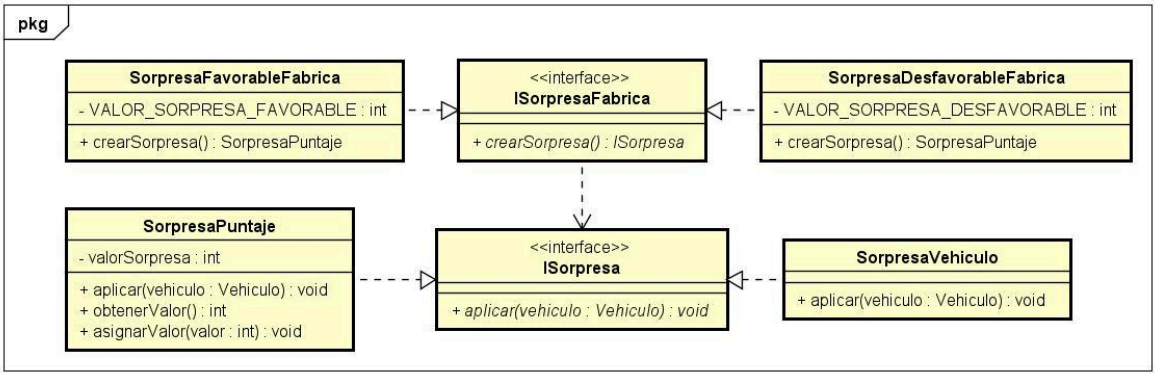


Diagrama de clases aislado de los obstáculos y sus respectivas fábricas

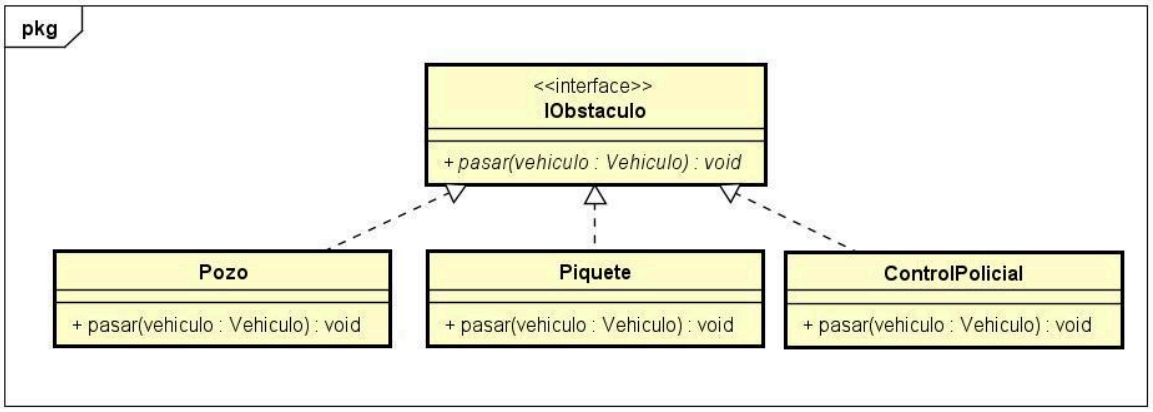


Diagrama de clases aislado de las sorpresas y sus respectivas fábricas

Null Object Pattern

Para la representación de vacío dentro del mapa, y evitar problemas de compatibilidad de tipos al implementar la ausencia de obstáculos y sorpresas dentro del mapa, se implementaron las siguientes clases, `VacioObstaculo`, `VacioSorpresa`, ambas implementadas utilizando la interfaz `IObstaculo` e `ISorpresa` respectivamente. Ambas contienen comportamiento nulo para que el jugador pueda pasar la calle sin problemas al no haber un obstáculo y/o sorpresa.

```
public class VacioSorpresa implements ISorpresa {
   @Override
   public void aplicar(Vehiculo vehiculo) {
       vehiculo.aplicarVacio();
   }
}
```

Donde `Vehiculo` y `Estado` implementan.

```
//Código presente en la clase Vehículo
public void aplicarVacio() {
   estado.pasarVacio();
}
//Código presente en la clase Estado, donde se manifiesta la ausencia de
comportamiento en sí.
public void pasarVacio() {}
```

De manera análoga se da el proceso con el `VacioSorpresa`.

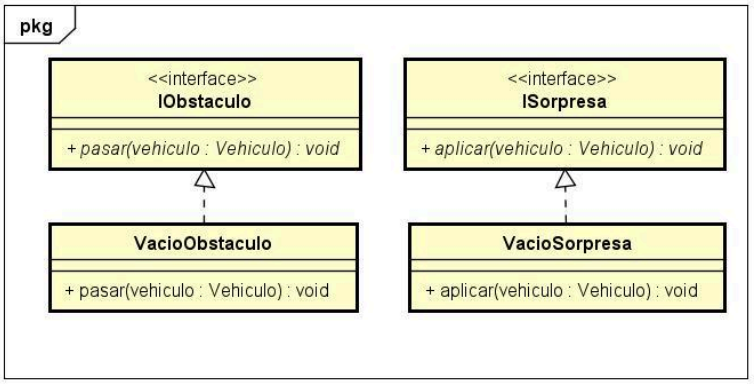


Diagrama de clases de los distintos tipos de sorpresas y sus relación con las interfaces.

Implementación del mapa

La ciudad se representa a través de un `Mapa` que contiene dos listas enlazadas bidimensionales, donde cada una de ellas representa las calles horizontales y verticales, las columnas y filas de cada una de las matrices ayudan a representar las coordenadas de cada calle.

Hay un objeto llamado `Juego` que contiene a los objetos `Mapa` y `Vehiculo`, esto nos permite reiniciar el juego una vez que el jugador haya llegado a la meta.

Cada `Calle`, está relacionada con el obstáculo y/o la sorpresa que se presente mediante composición. Si no existiera la calle, no tendría sentido la existencia de un obstáculo/sorpresa.

`Obstaculo` y `Sorpresa` son interfaces que implementan los verdaderos obstáculos y sorpresas, estos tienen el método `aplicar(vehículo)` que haciendo *double dispatch* con el vehículo se aplican obstáculos y las sorpresas en el.

Interacción distintos tipos de vehículo con los obstáculos

Para resolver la interacción entre el `Vehiculo` y los `Obstaculos/Sorpresas` aplicamos el patrón de diseño de comportamiento ***State***.

Para la implementación creamos una clase abstracta, `Estado`, e hicimos que `Vehiculo` contenga un objeto descendiente de ella , estos objetos pueden ser `Auto`, `Moto` o `Camioneta`. De esta forma se facilita el cambiar de vehículo en el juego (al aplicar sorpresas de cambio de vehículo) y adaptar las distintas interacciones entre los vehículos y los obstáculos. La mayor ventaja del uso de este patrón fue que el mismo permite evitar el uso de condicionales *if else* al momento de pasar un obstáculo.

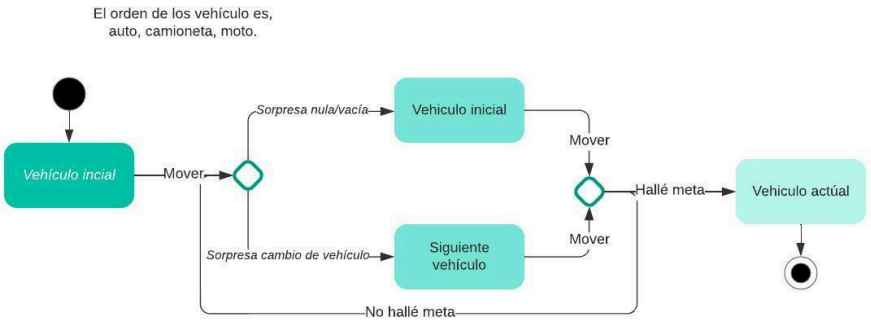


Diagrama de estados del vehículo durante su movimiento en el mapa.

Movimiento del vehículo

`Vehiculo` contiene una `Posicion` que tiene como atributos una `x` e `y` que son las coordenadas en el mapa.

Cuando queremos mover el `Vehiculo` llamamos al método `mover(Direccion)` propio de `Juego` pasándole como parámetro un objeto hijo de la clase `Direccion`. `Juego` lo que hace es llamar al método mover del `Vehiculo` que recibe como parámetro la `Direccion` recibida en Juego, el `Vehiculo` le dice a la `Direccion` que se mueva y esta llama al método `mover(Vehiculo)` que corresponda del objeto `Posicion` que contiene el `Vehiculo`.

`Posicion` delega a `Calle` (obtenida mediante coordenadas) el paso del vehículo por los obstáculos y/o sorpresas que contenga. Además, tiene métodos para calcular la próxima coordenada a visitar, y actualizarla al moverse.

El único Obstaculo que condiciona el movimiento del Vehiculo es el Piquete. Para que el Vehiculo no pase la Calle, `Piquete` delega a `Posicion` modificar la siguiente coordenada que ya había calculado, para restablecerla a la coordenada que se encuentra actualmente. De esta forma, al actualizar las coordenadas luego, el valor no varía y el Vehiculo se queda en su lugar.

Tablero de clasificaciones

Luego para el *ranking* creamos un objeto `Ranking` que se pasa por parámetro a `Juego`. Este utiliza un *heap* para ordenar los jugadores por puntaje. Para ello creamos un objeto `Jugador` contenido en el juego. Este tiene como atributo el nombre y la cantidad de movimientos, y agregamos `CompareTo` como uno de sus métodos para que se pudiera comparar en la cola del *ranking*.

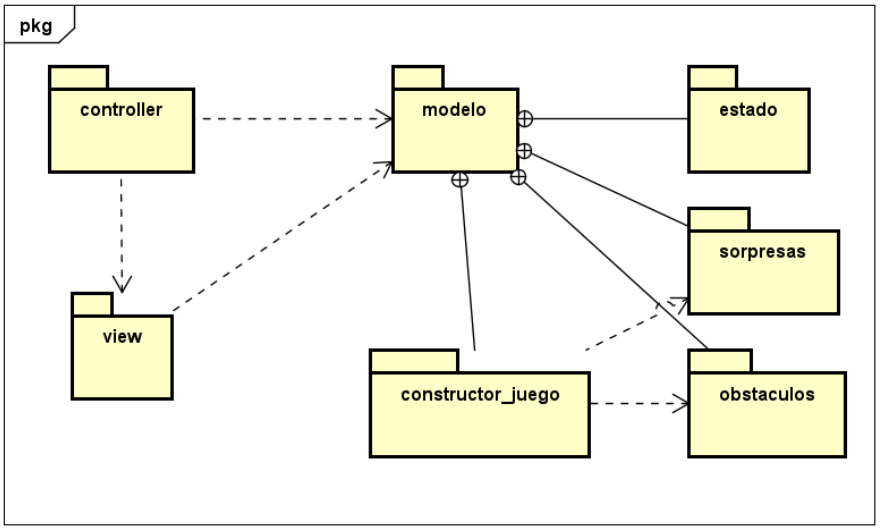
Meta cómo sorpresa

Para implementar la meta, la consideramos una sorpresa más. Al pasar por ella se activa el método ganar() de Vehiculo, cargando el puntaje final en el ranking activo.

Valores Aleatorios

Se creo una clase con el fin de calcular los distintos valores aleatorios que se necesiten durante la partida, llamada ValoresAleatorios. Cuenta con los metodos valorAleatorio() : int y obtenerPosicionAleatoria(dimension : int, mapa : Mapa) : Posicion. El primero devuelve un número del 0 al 9 para la probabilidad del control policial, y el segundo devuelve posiciones aleatorias del mapa para agregar obstáculos/sorpresas de forma random.

DIAGRAMA DE PAQUETES



DIAGRAMAS DE CLASE

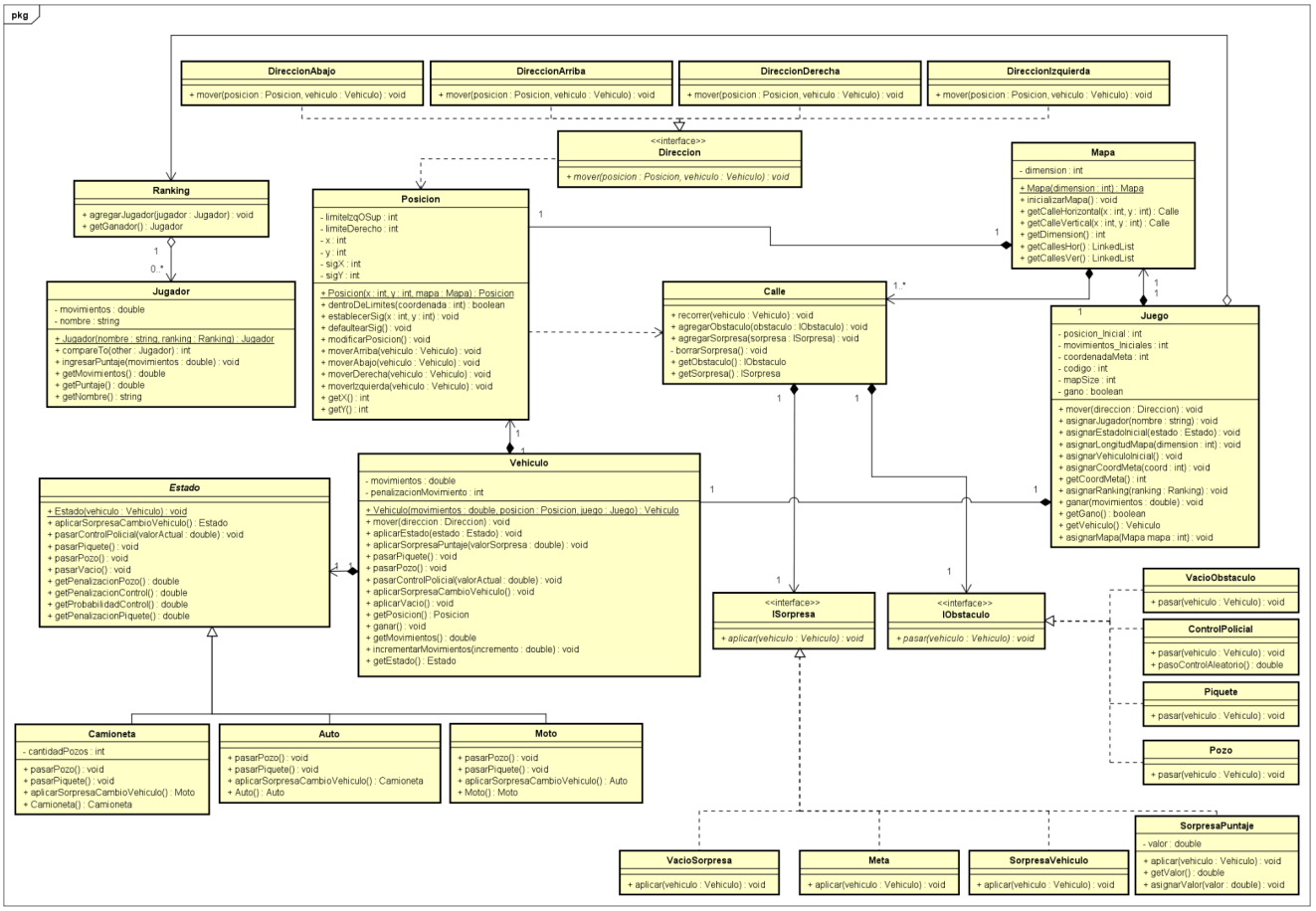


Diagrama de clases del modelo completo.

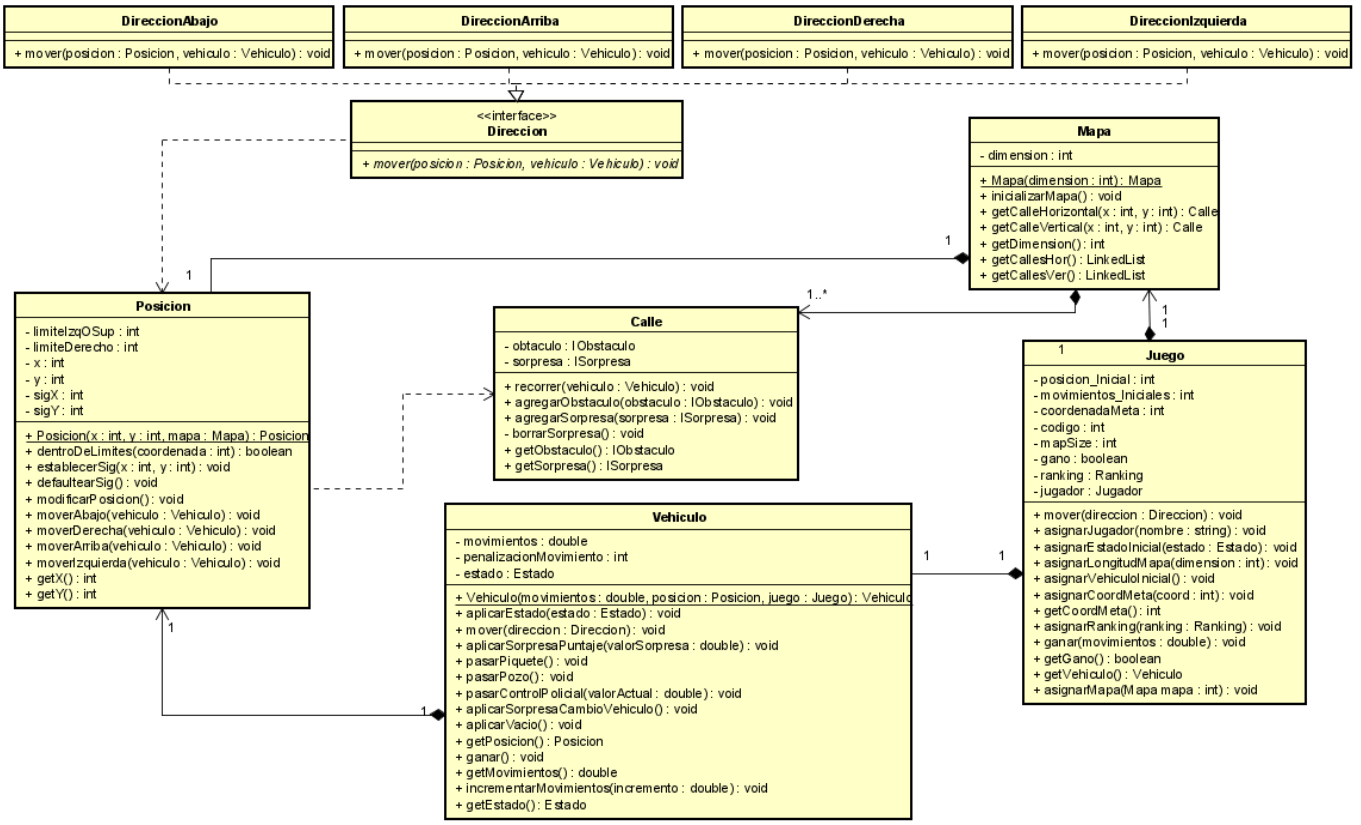


Diagrama de clases reducido a la lógica del movimiento.

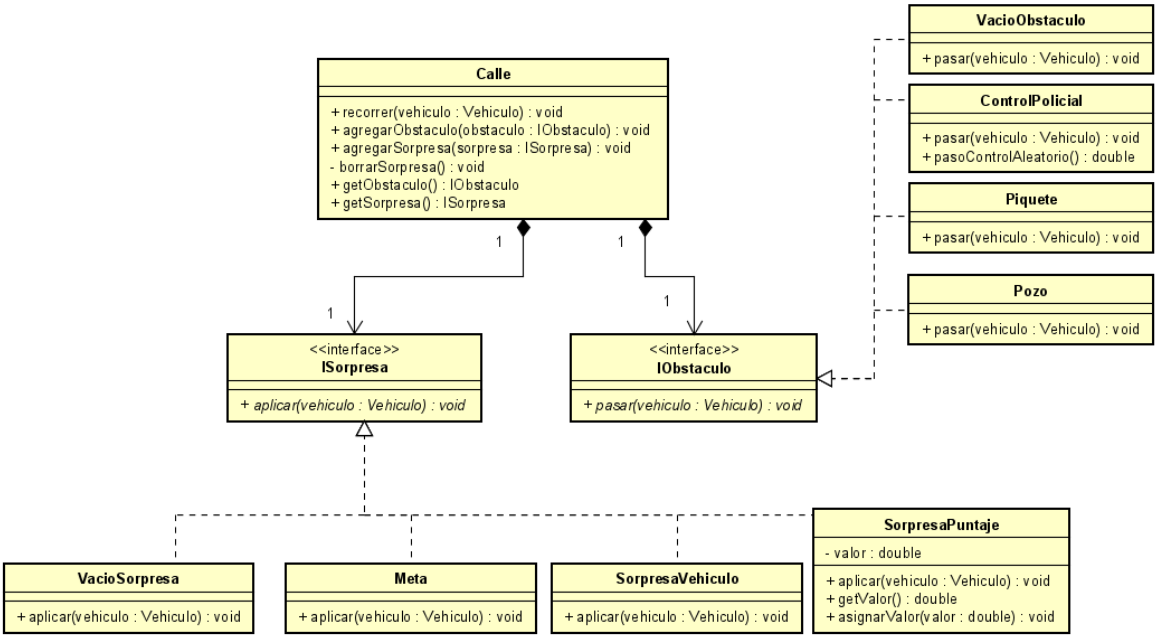


Diagrama de clases reducido a la calle y las interfaces Obstáculo y Sorpresa.

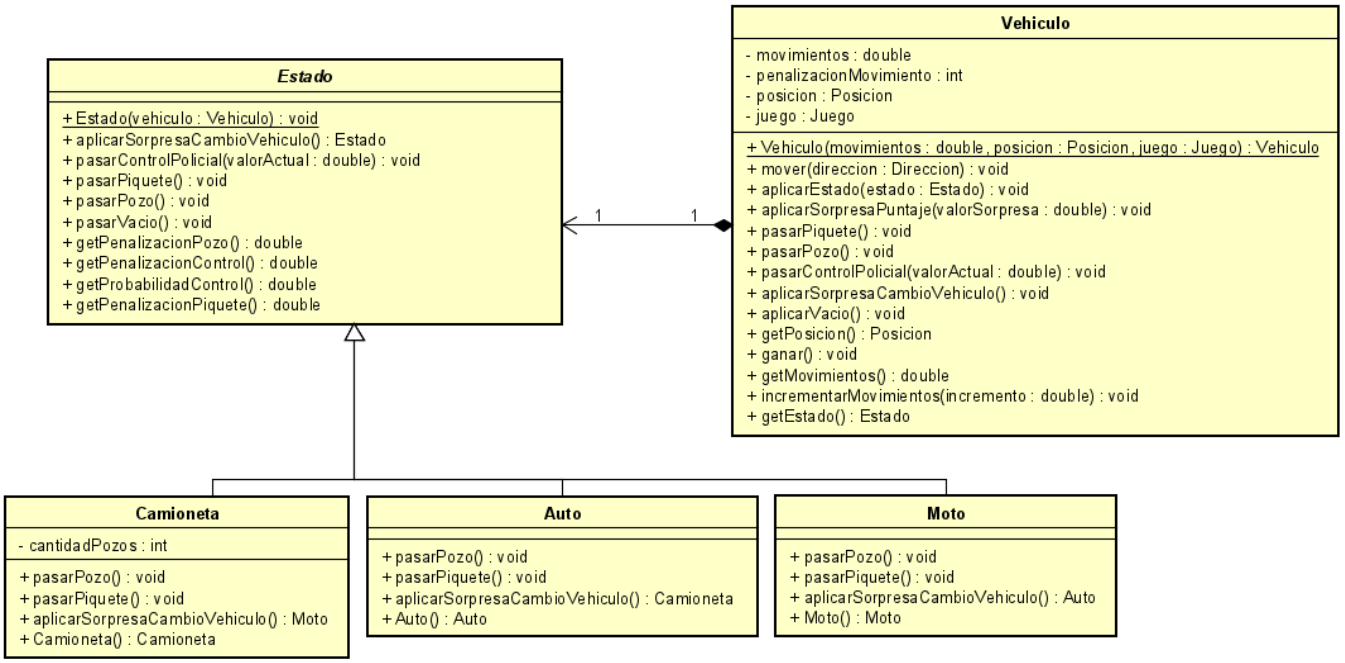


Diagrama de clases reducido al Vehículo y Estado.

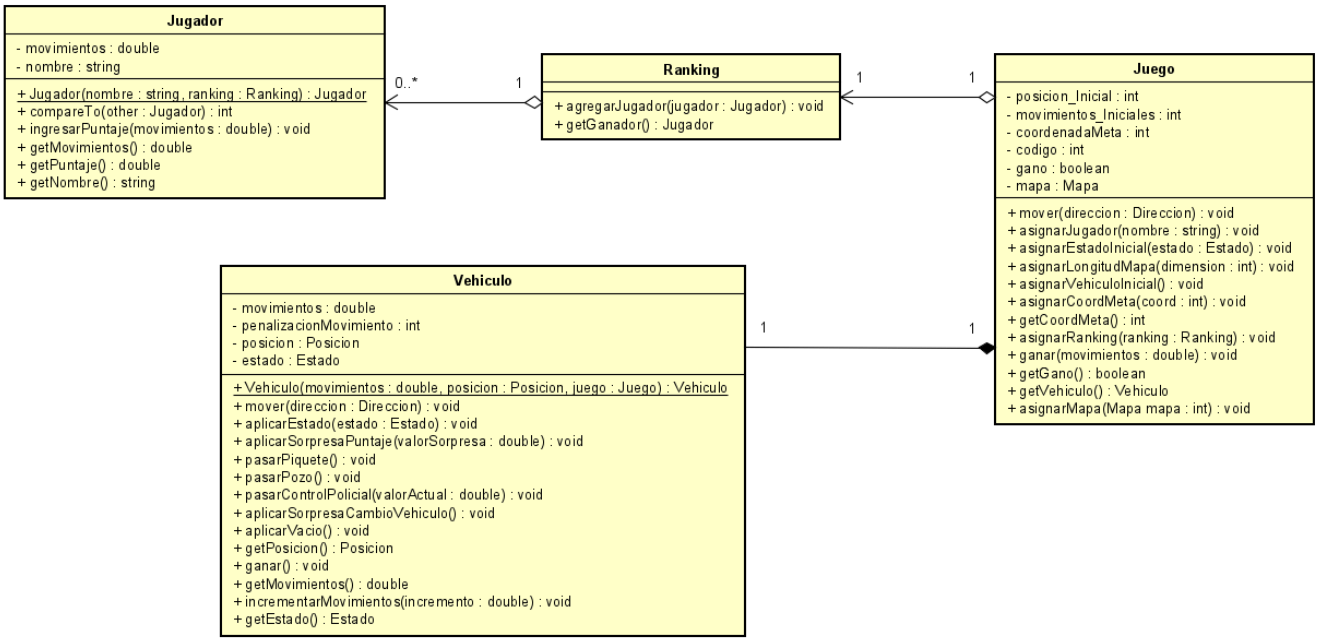


Diagrama de clases reducido al Ranking y Jugador.

DIAGRAMAS DE SECUENCIA

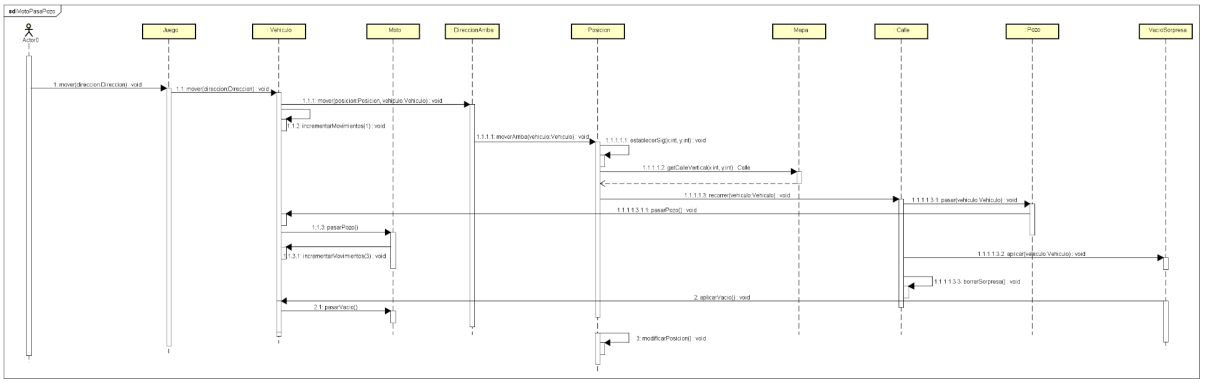


Diagrama de secuencia MotoPasaPozoYEsPenalizada

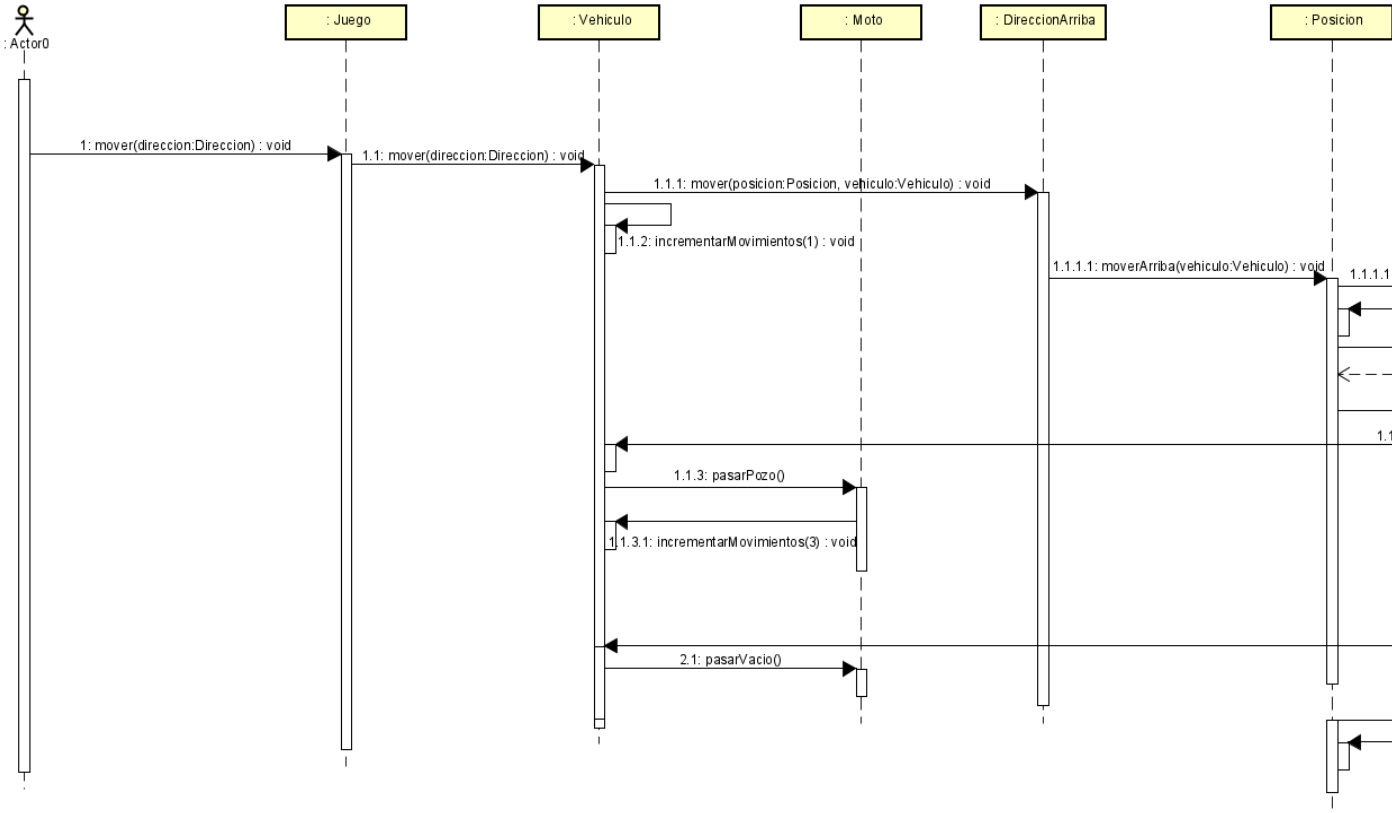


Diagrama de secuencia MotoPasaPozoYEsPenalizada 1ra mitad.

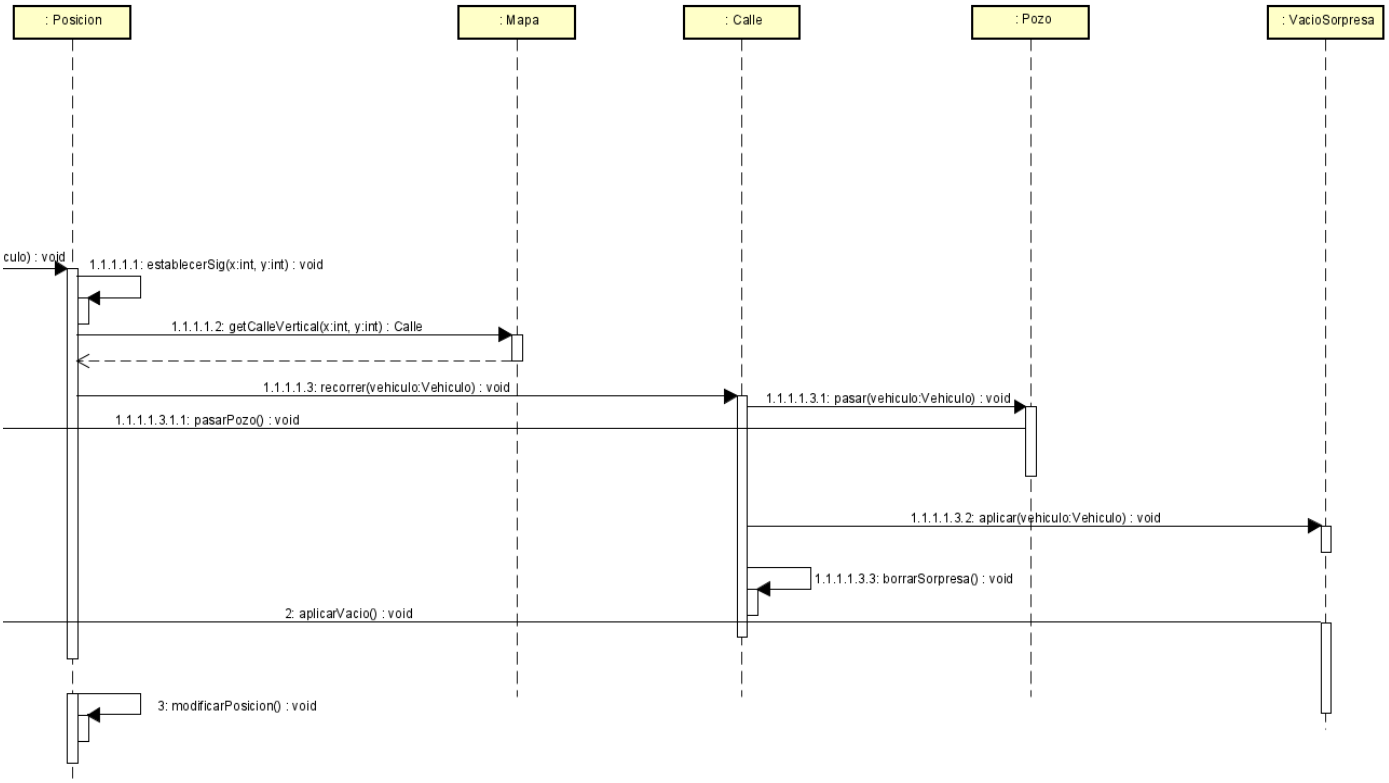


Diagrama de secuencia MotoPasaPozoYEsPenalizada 2da mitad.

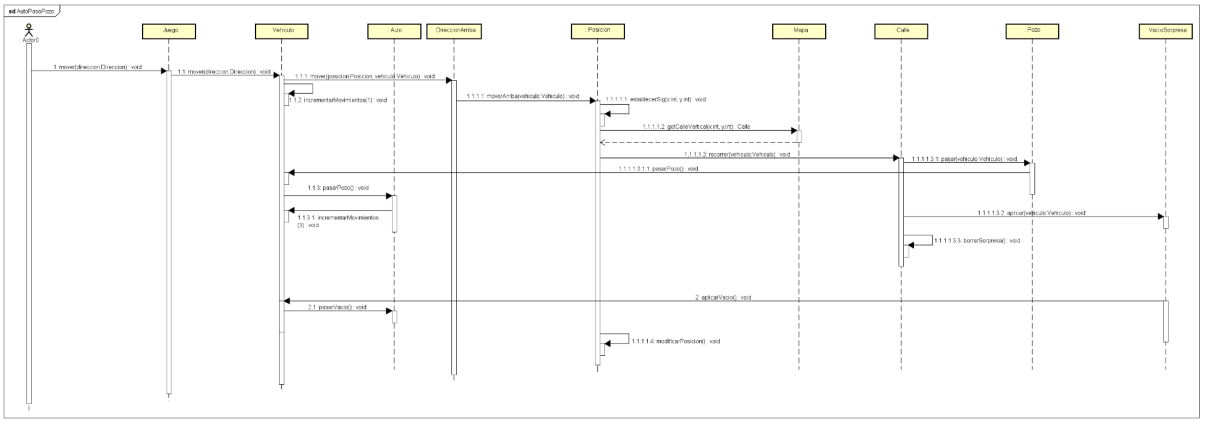


Diagrama de secuencia Auto PasaPozoYEsPenalizada

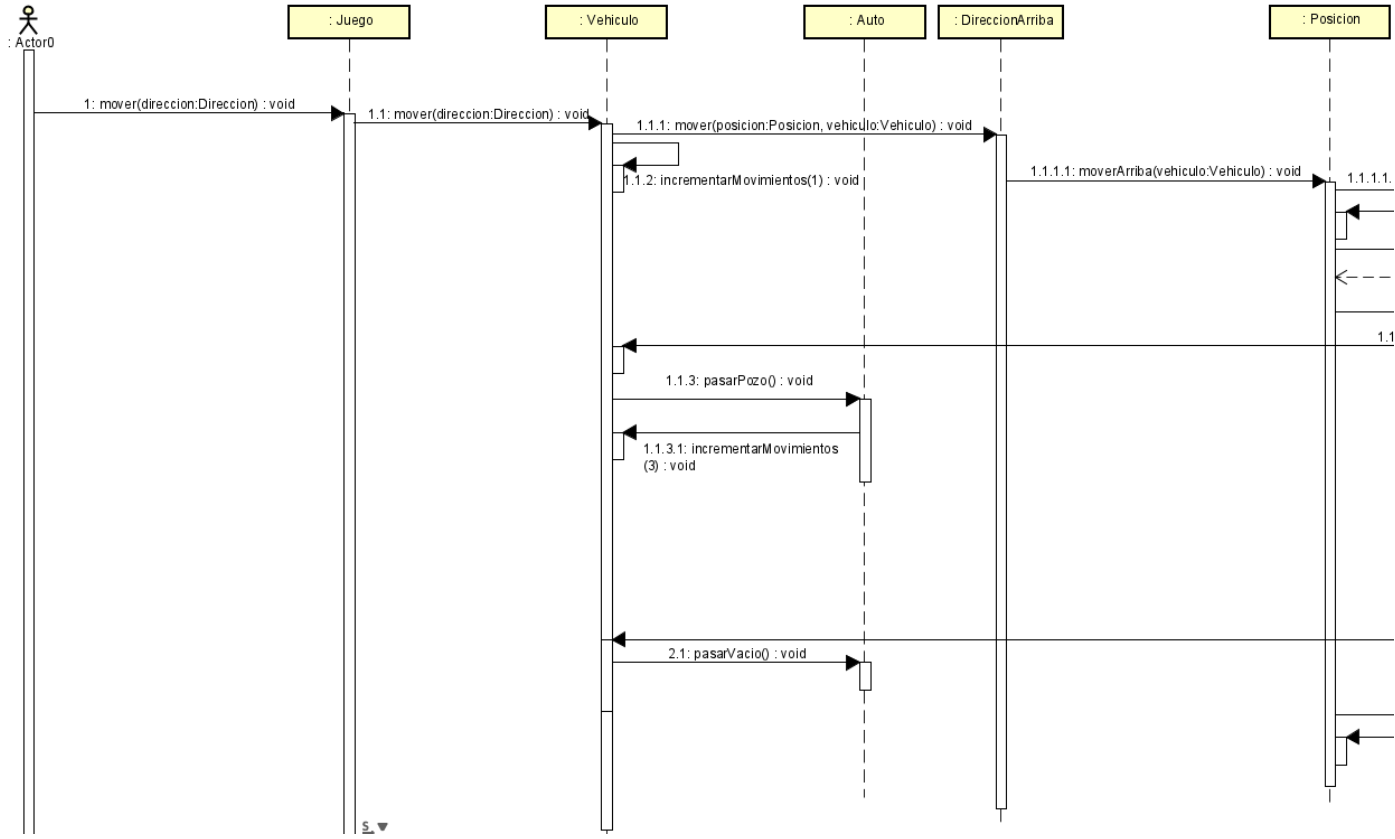


Diagrama de secuencia Auto PasaPozoYEsPenalizada 1era mitad.

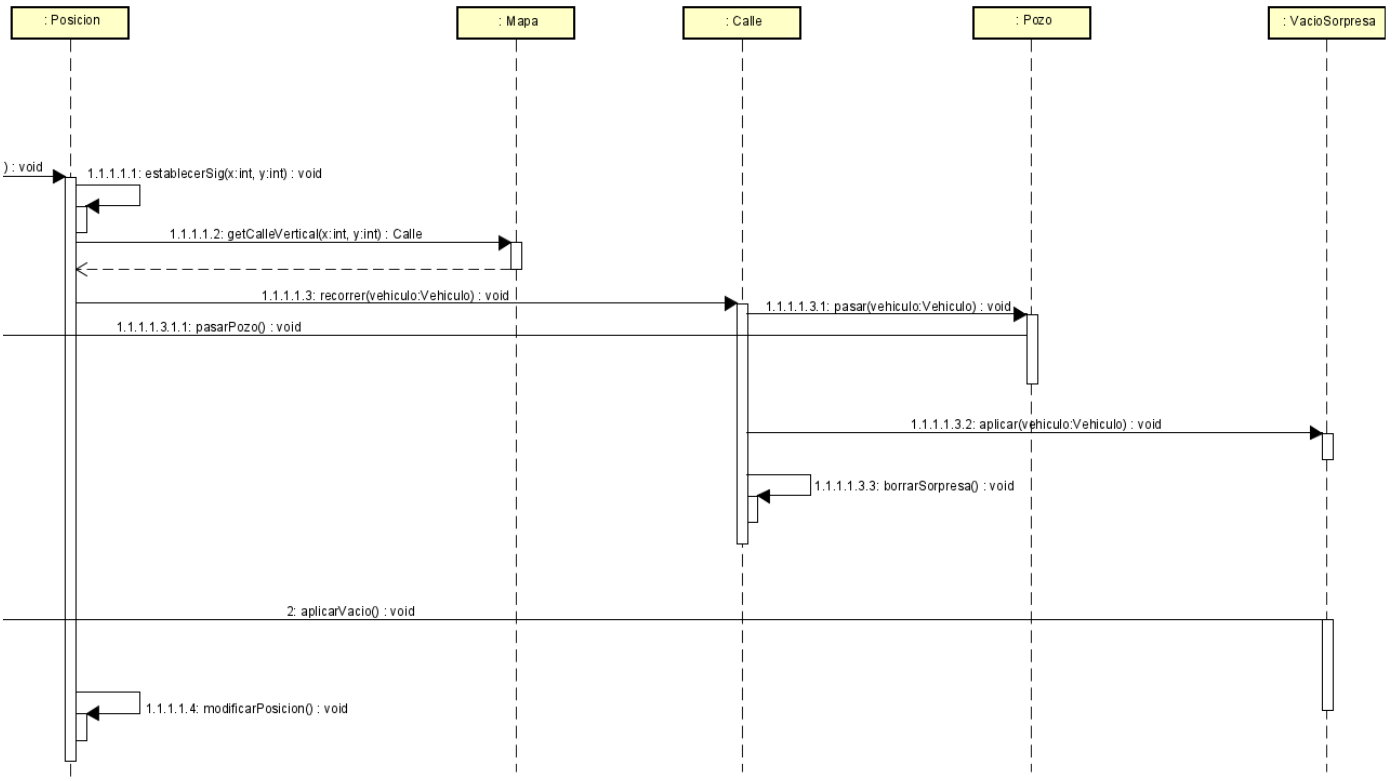


Diagrama de secuencia Auto PasaPozoYEsPenalizada 2da mitad.

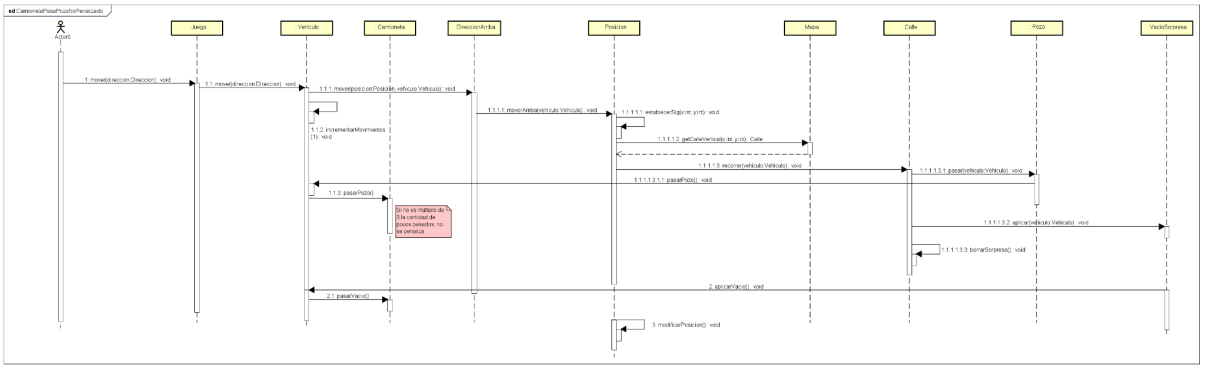


Diagrama de secuencia CamionetaPasaPozoYnoEsPenalizada

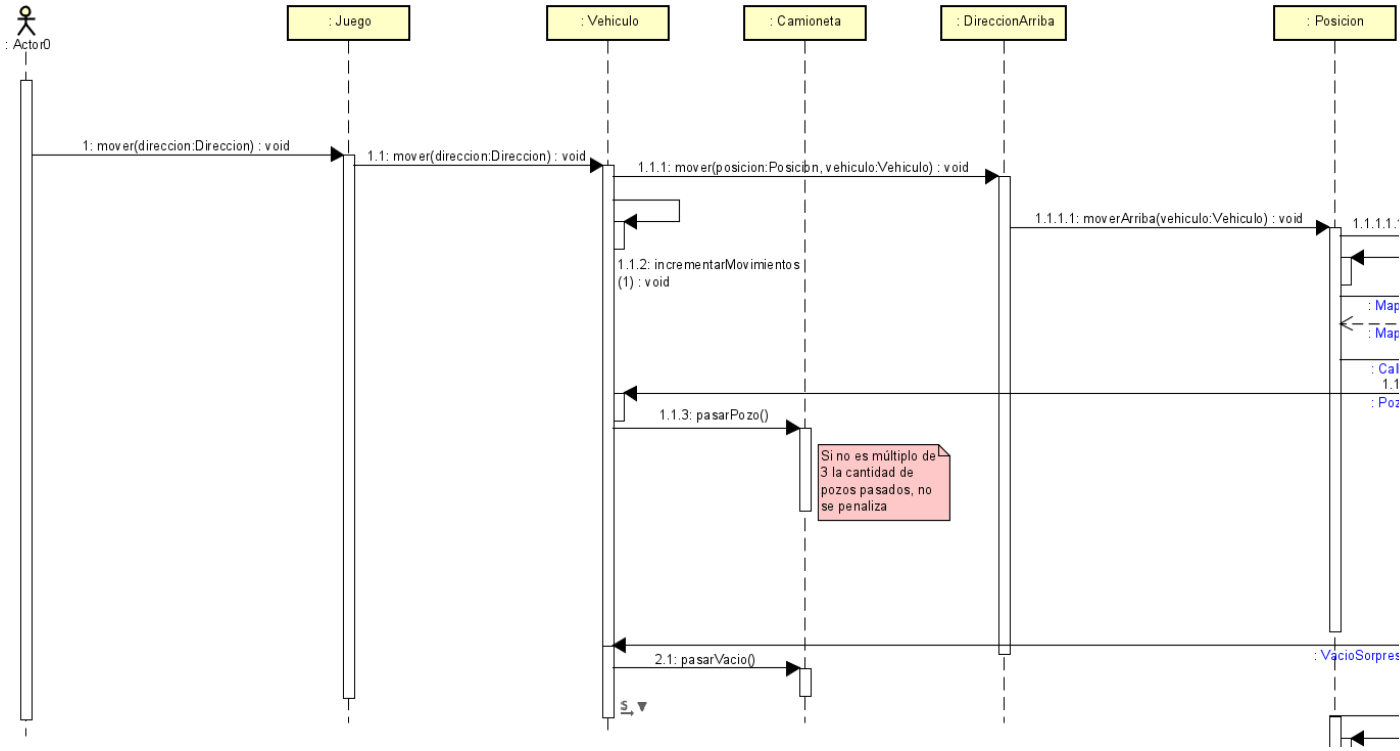


Diagrama de secuencia CamionetaPasaPozoYnoEsPenalizada 1era mitad.

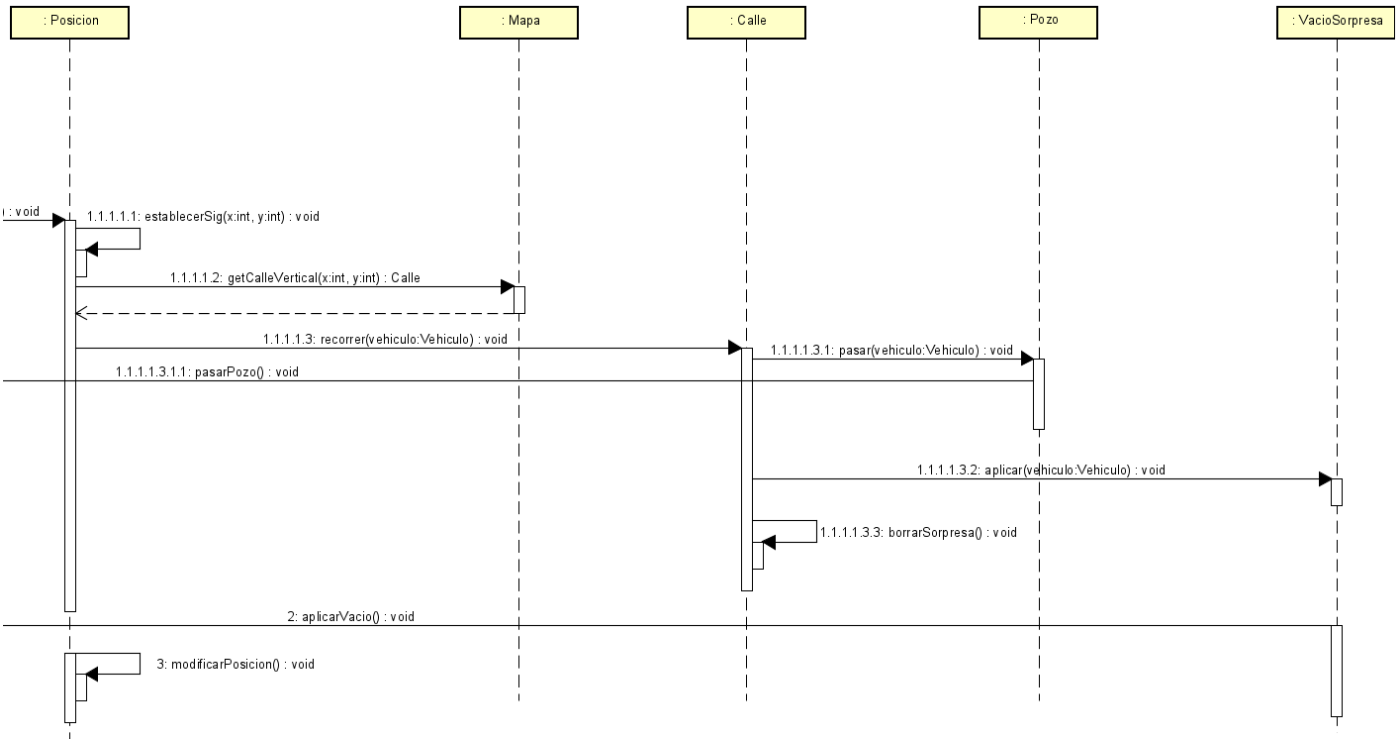


Diagrama de secuencia CamionetaPasaPozoYnoEsPenalizada 2da mitad.

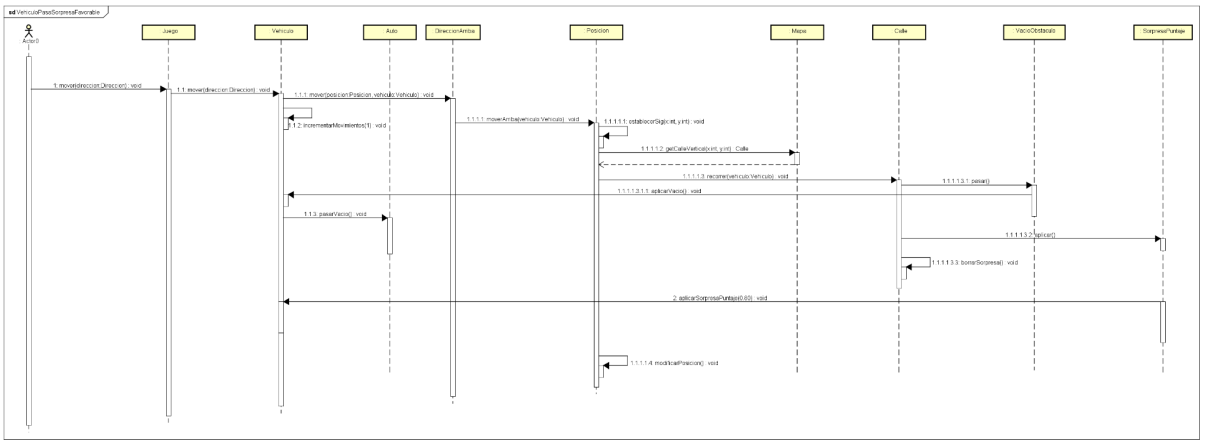


Diagrama de secuencia VehiculoEncuentraSorpresaFavorable.

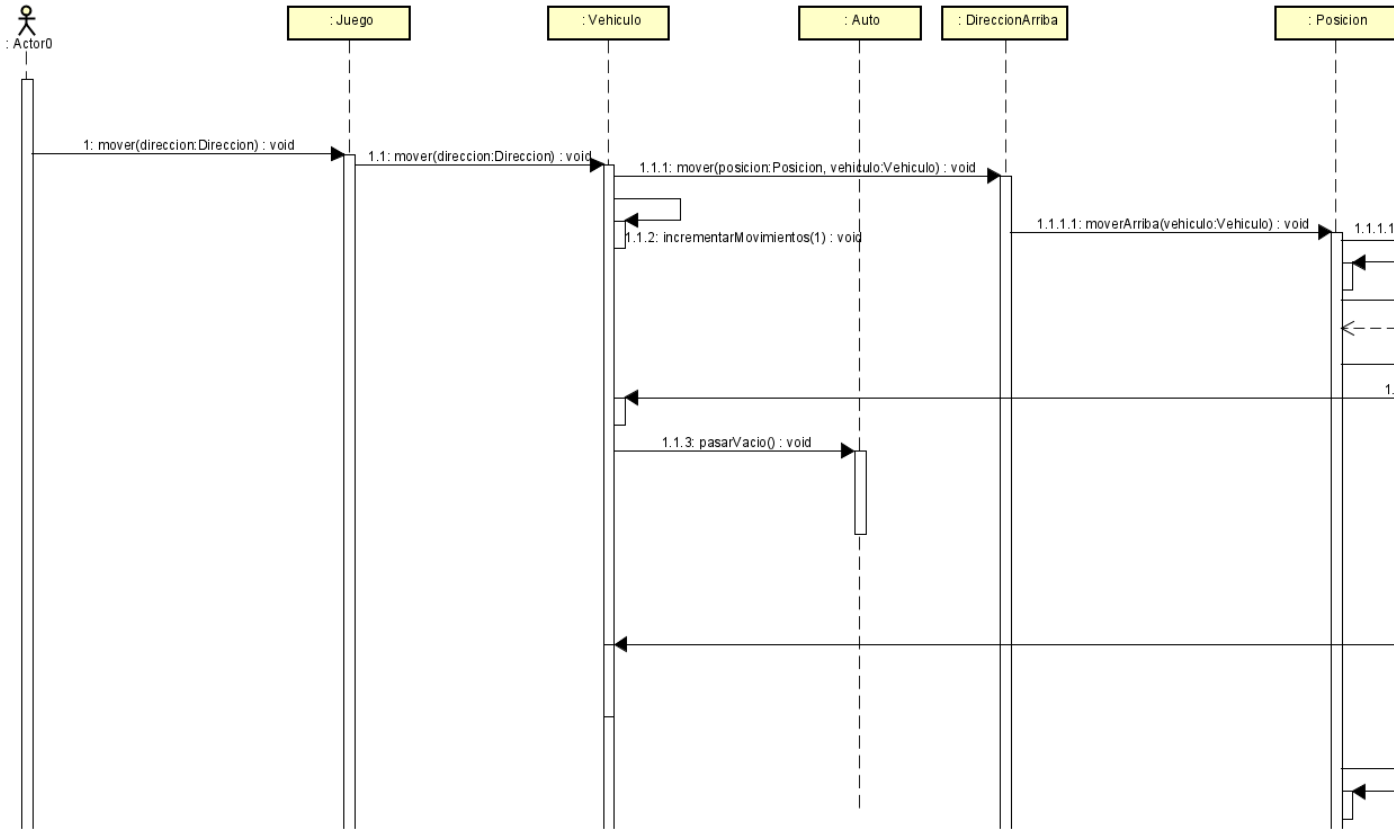


Diagrama de secuencia VehiculoEncuentraSorpresaFavorable 1era mitad.

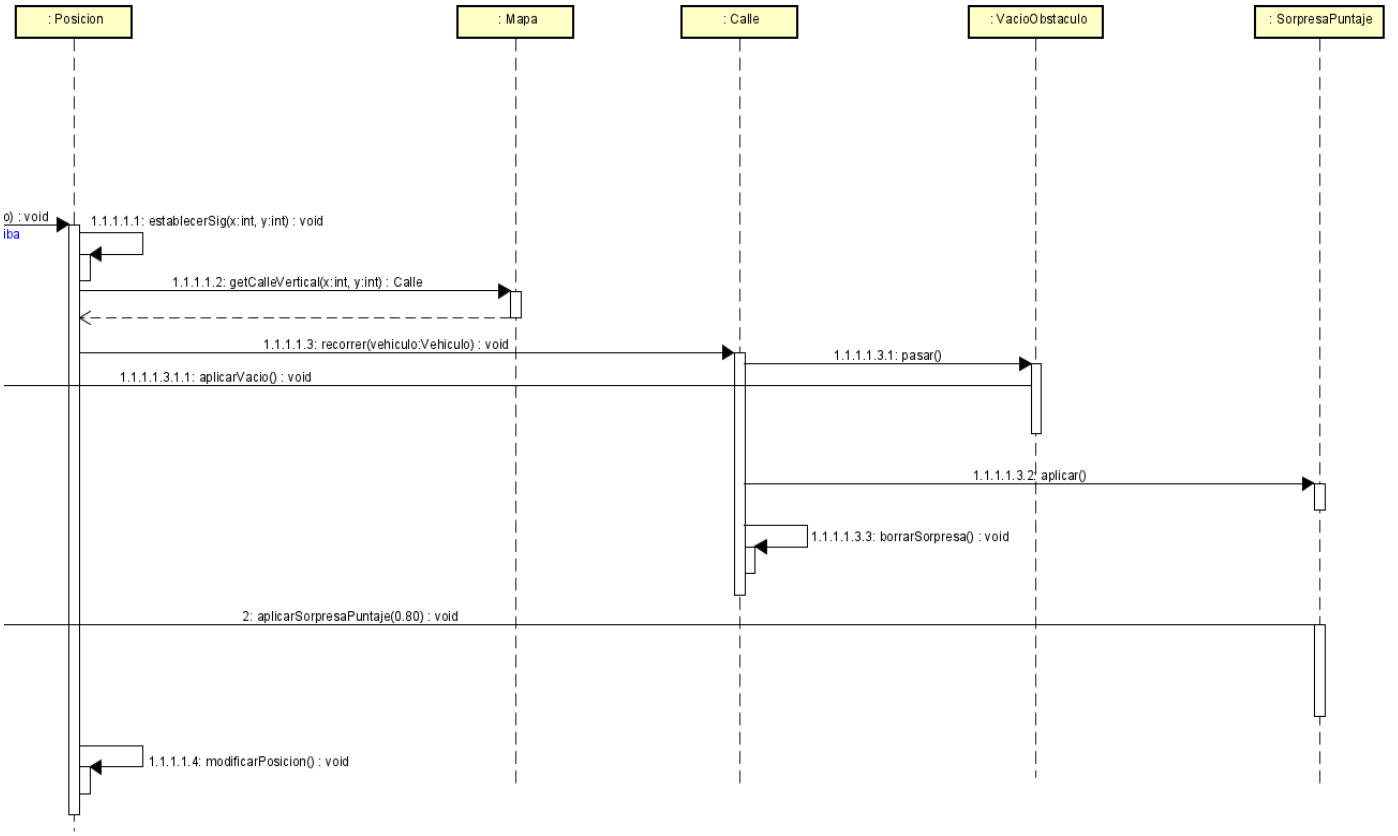


Diagrama de secuencia VehiculoEncuentraSorpresaFavorable 2da mitad.

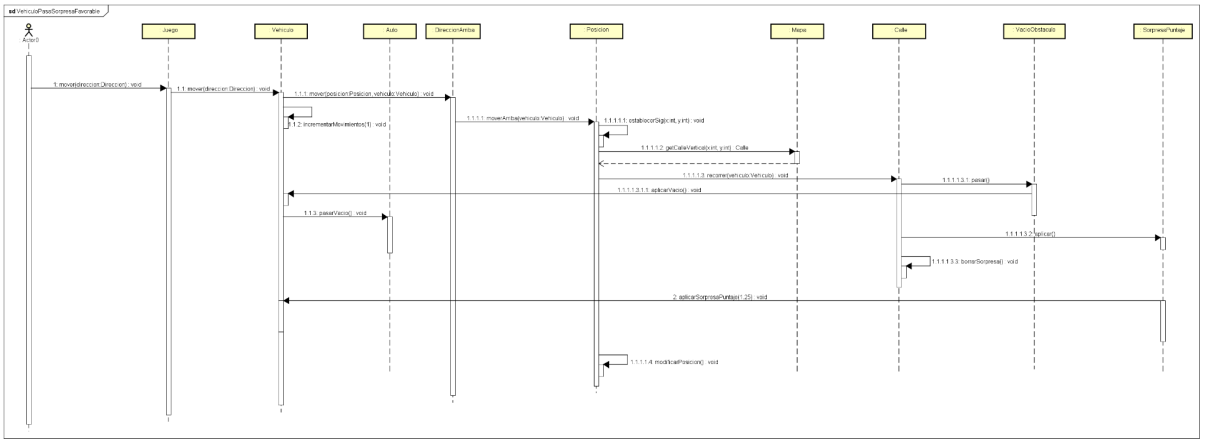


Diagrama de secuencia VehiculoEncuentraSorpresaDesfavorable.

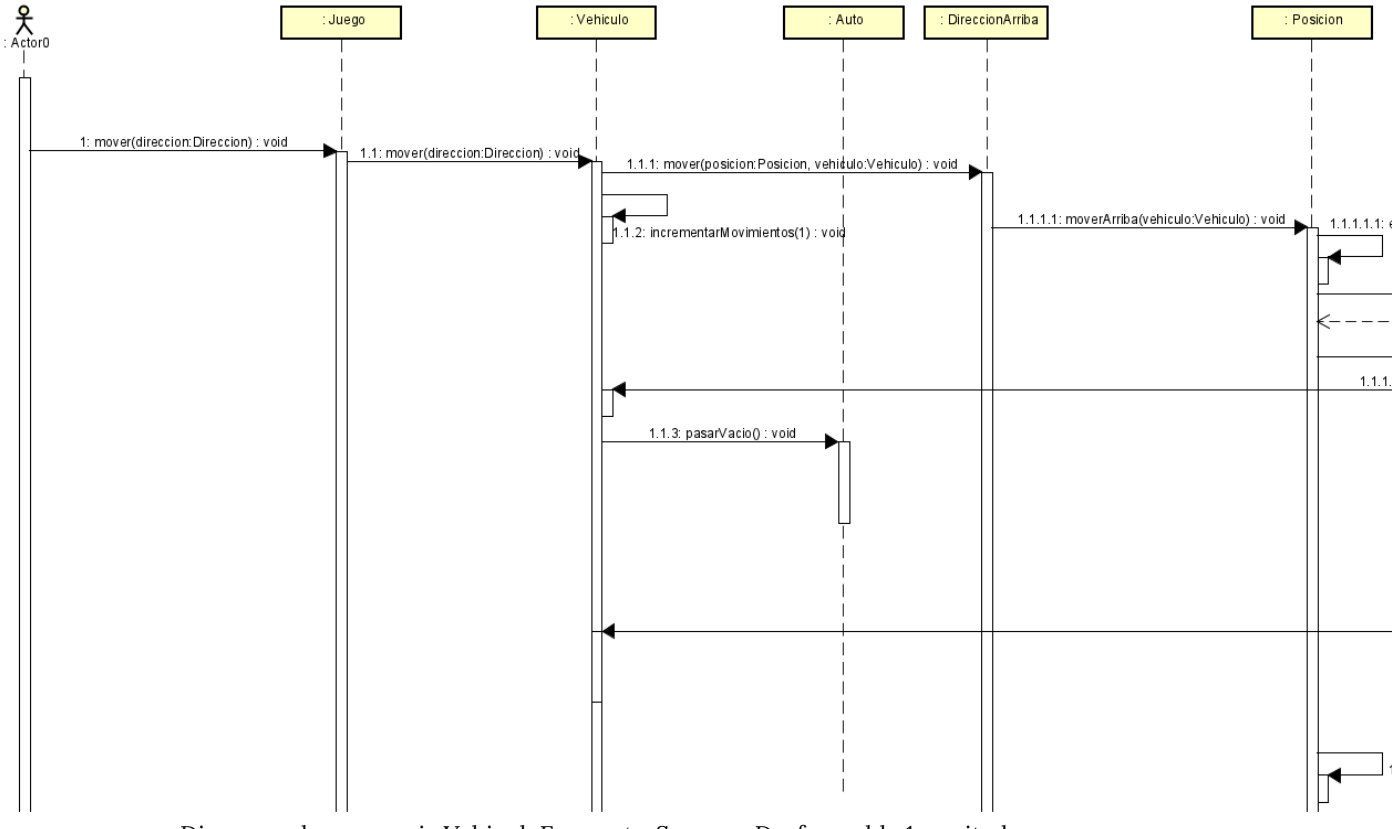


Diagrama de secuencia VehiculoEncuentraSorpresaDesfavorable 1ra mitad.

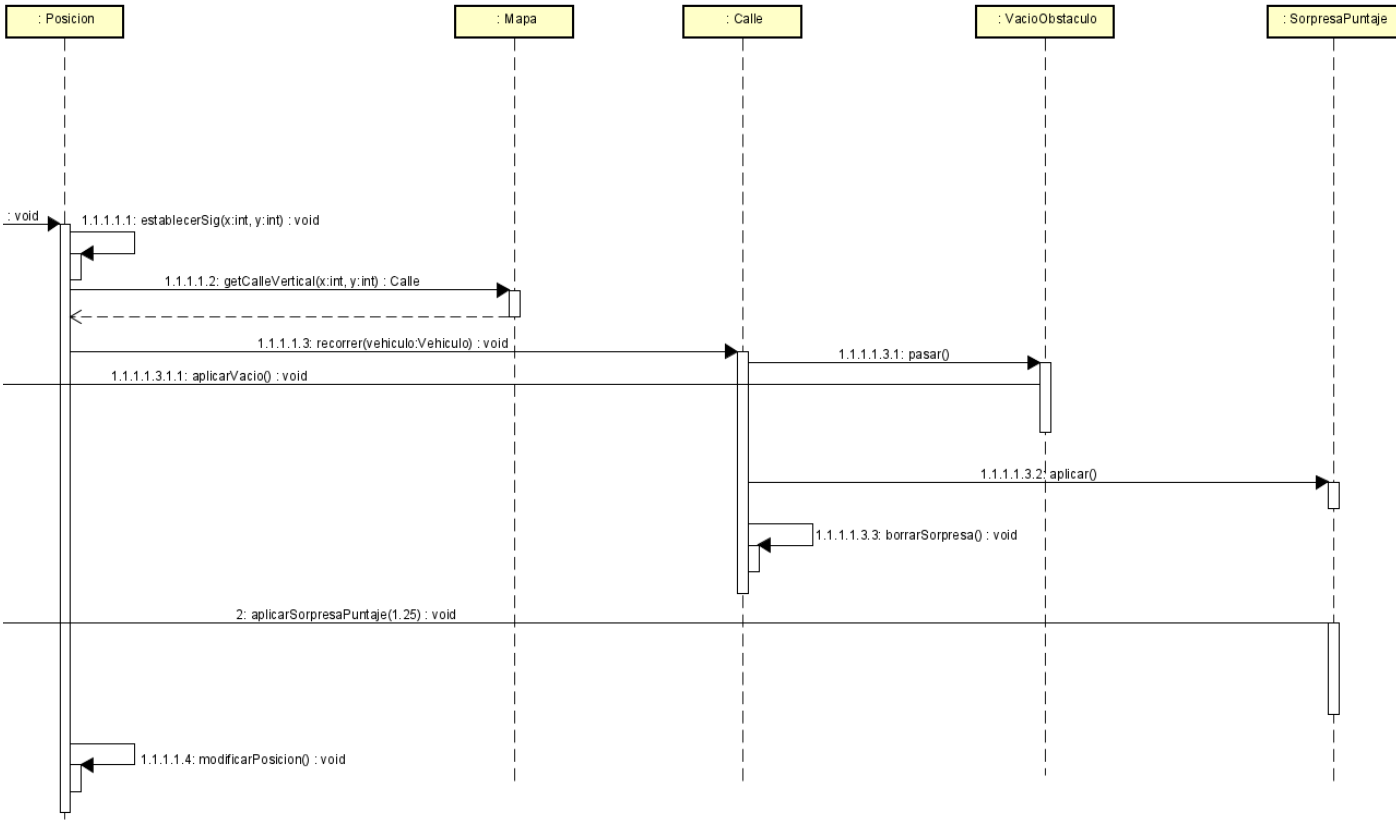


Diagrama de secuencia VehiculoEncuentraSorpresaDesfavorable 2da mitad.

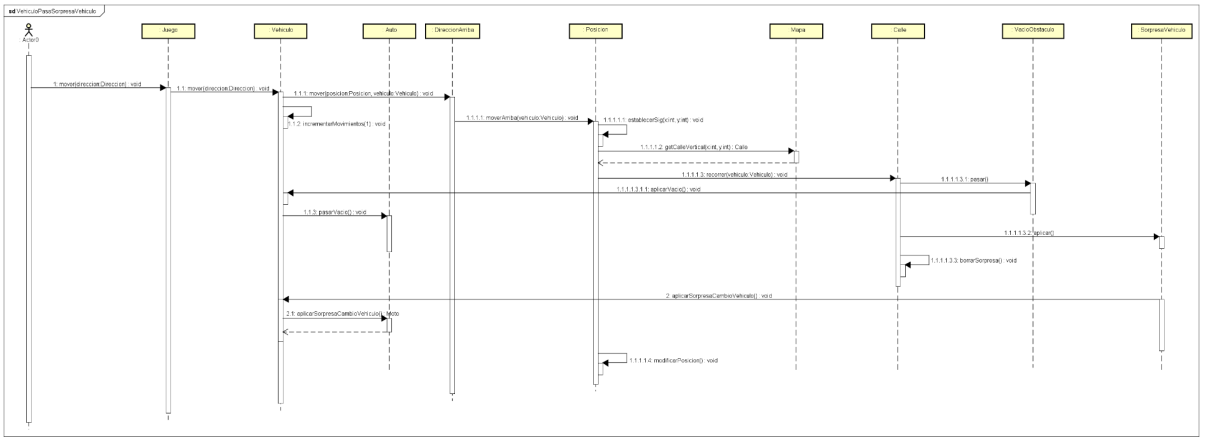


Diagrama de secuencia VehiculoEncuentraSorpresaCambioVehiculo.

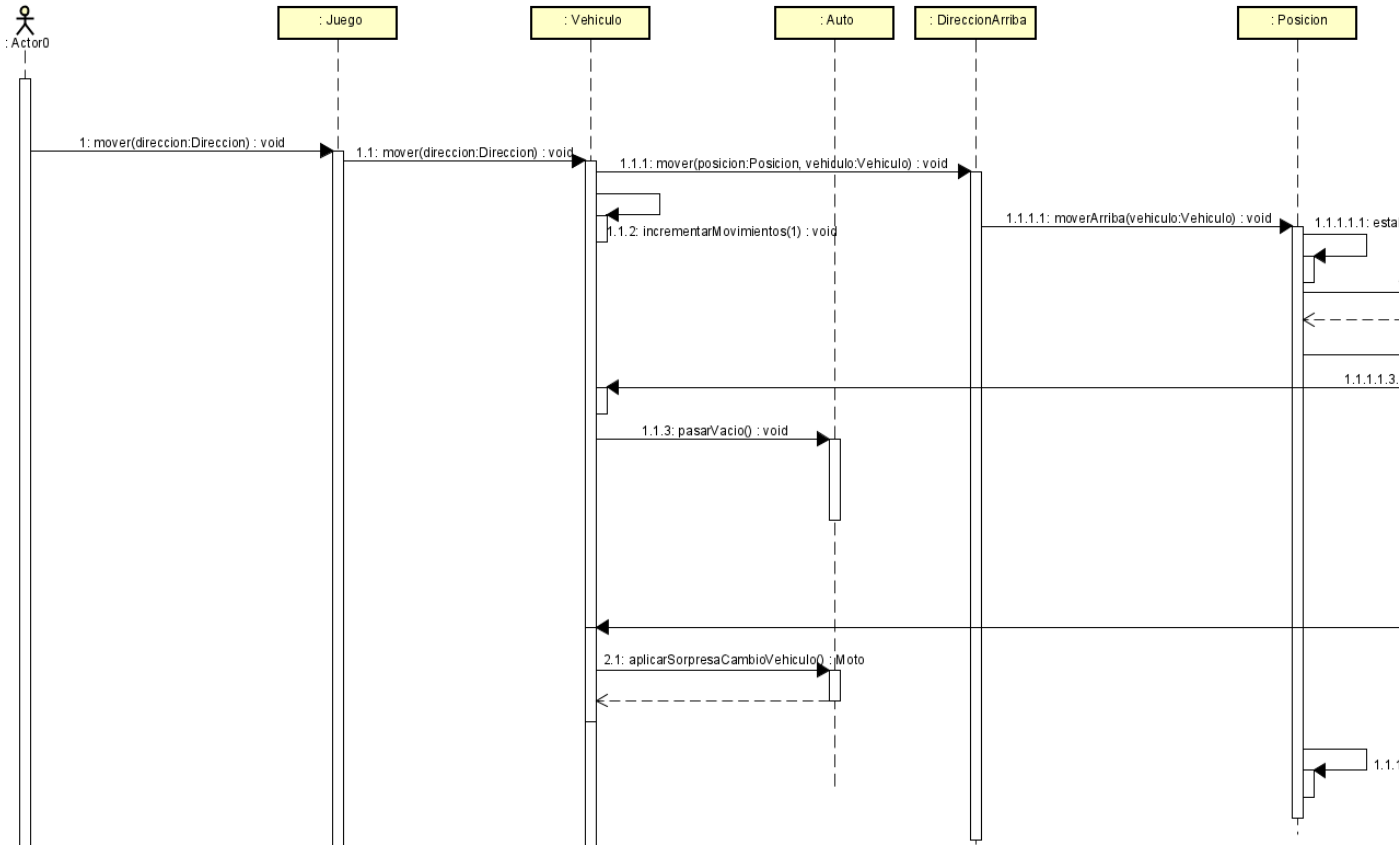


Diagrama de secuencia VehiculoEncuentraSorpresaCambioVehiculo 1ra mitad.

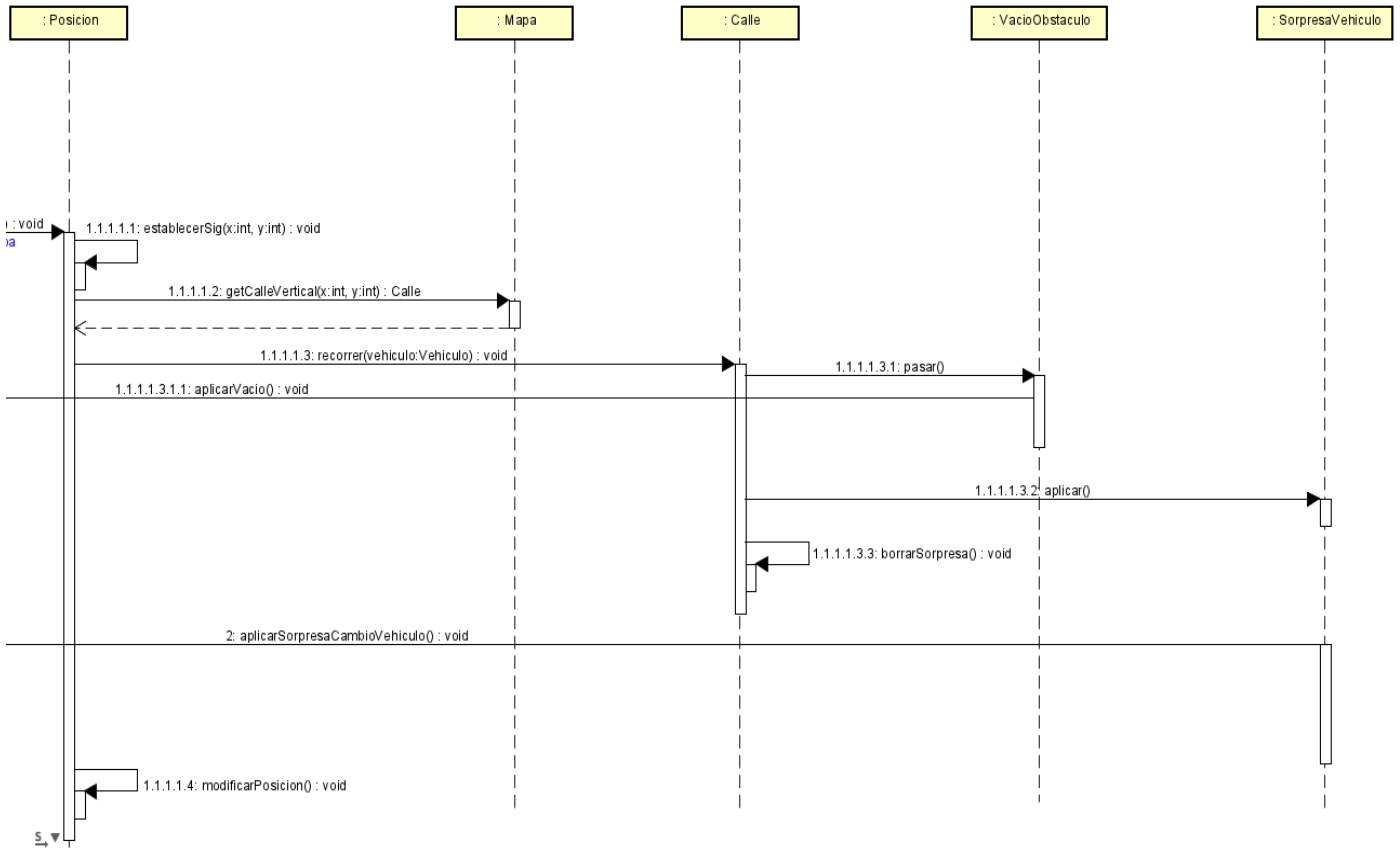


Diagrama de secuencia VehiculoEncuentraSorpresaCambioVehiculo 2da mitad.

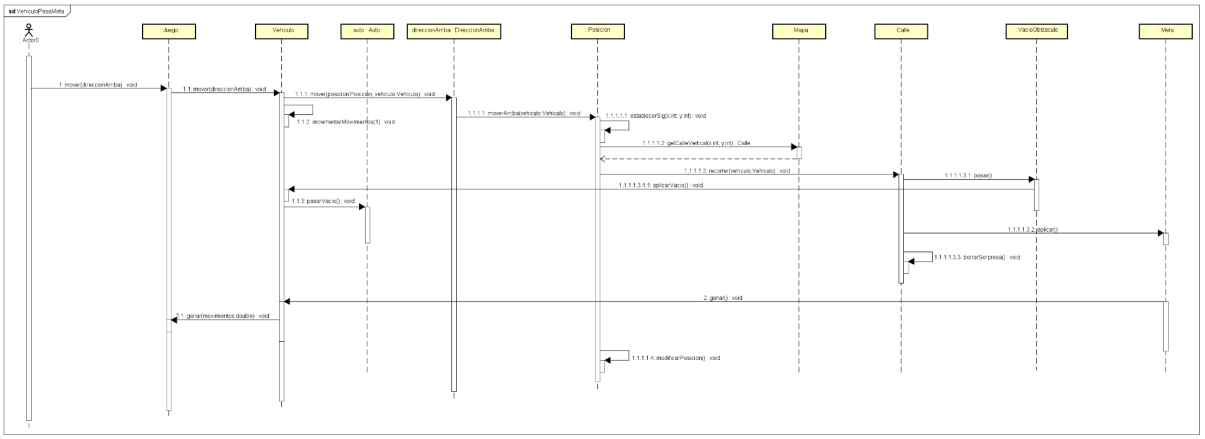


Diagrama de secuencia VehiculoPasaMeta

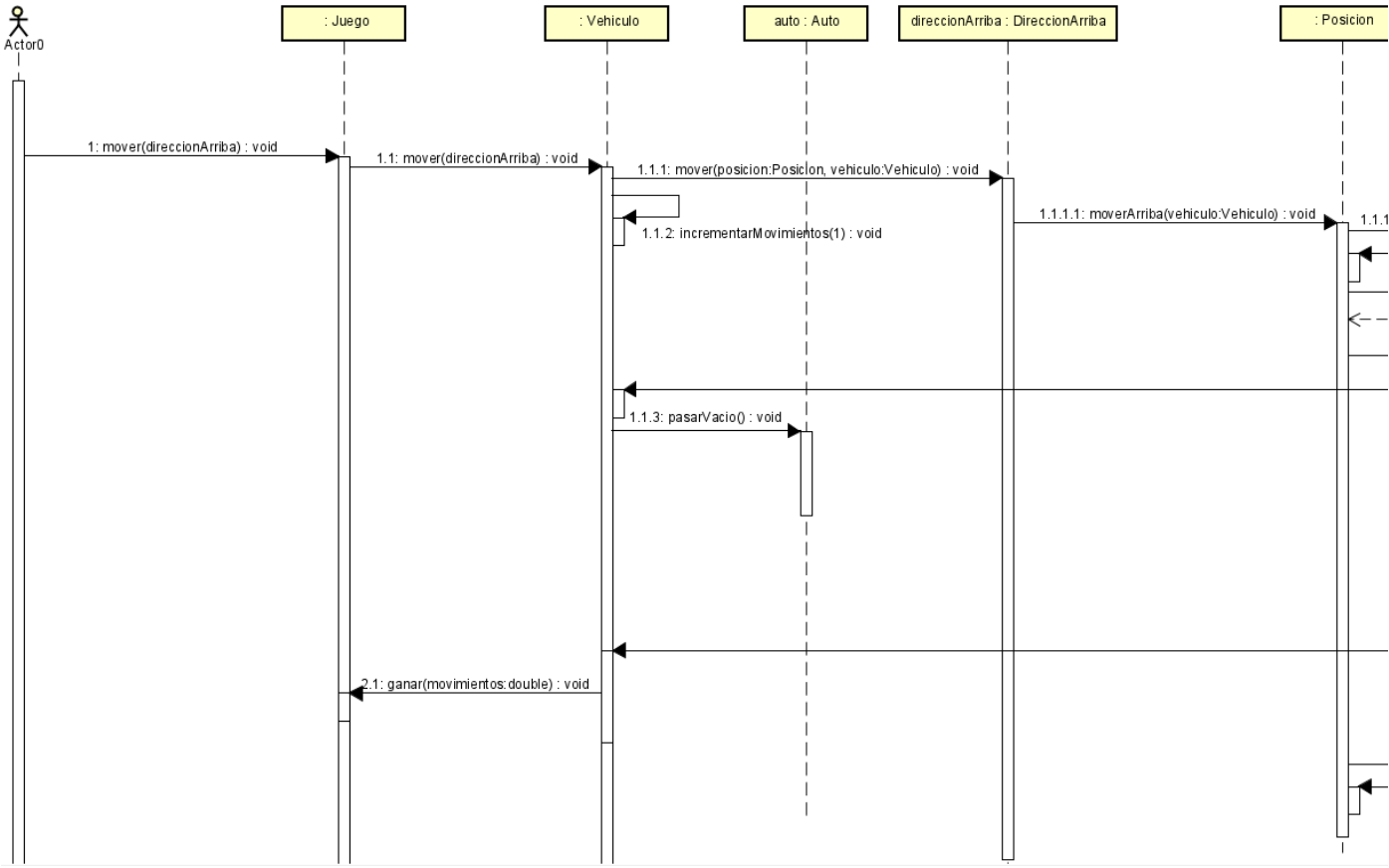


Diagrama de secuencia VehiculoPasaMeta 1ra mitad.

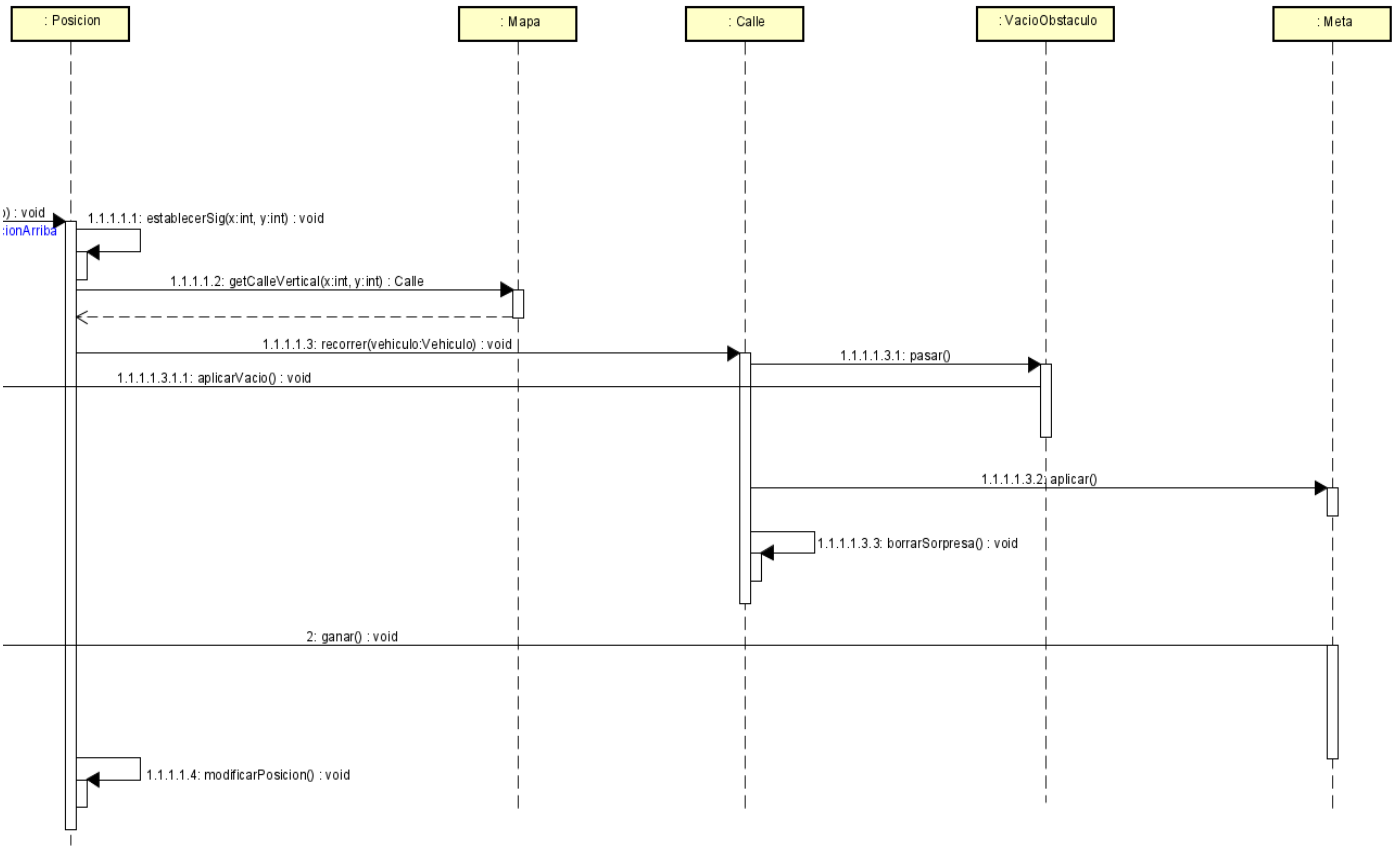


Diagrama de secuencia VehiculoPasaMeta 2da mitad.

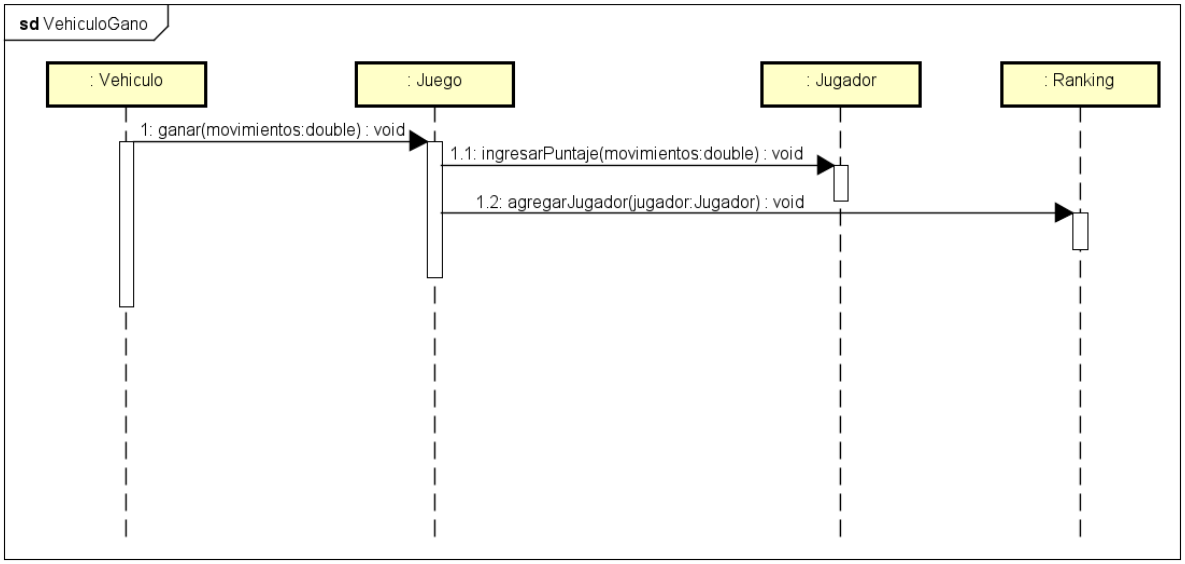


Diagrama de secuencia VehiculoGano

BIBLIOGRAFÍA Y WEBGRAFÍA

- Nicolás Paez - Double dispatch

- Refactoring Guru - Builder Method
- Refactoring Guru - Factory Method
- Refactoring Guru - State pattern
- Source Making - Null Object Pattern