

Trabajo Práctico 2 — GPS Challenge

[75.07/95.02] Algoritmos y Programación III
Primer cuatrimestre de 2022

Alumno	Padron	Email
CASTILLO, Carlos	108535	ccastillo@fi.uba.ar
DEALBERA, Pablo Andres	106585	pdealbera@fi.uba.ar
RECCHIA, Ramiro	102614	rrecchia@fi.uba.ar

Corrector	Email
GOMEZ, Joaquin	gjoaquin@fi.uba.ar
VALDEZ, Santiago	vsantiago@fi.uba.ar

Índice

1. Intruducccion	2
2. Supuestos	2
3. Diagramas de clases	2
3.1. Vehiculo	3
3.2. Sorpresas	3
3.3. Obstaculos	4
3.4. Mapa	4
3.5. Juego	5
3.6. Jugador	6
4. Diagrama de paquetes	7
5. Diagramas de secuencia	7
5.1. Interaccion Jugador - Sorpresa Cambio de Vehiculo	8
5.2. Interaccion Jugador - Sorpresa Favorable	8
5.3. Interaccion Jugador - Elemento	9
5.4. Jugador avanza y se encuentra con un Elemento	10
6. Diagramas de estado	10
6.1. Cambio de Vehiculo del Jugador	11
6.2. Vehiculo Pisa Obstaculo	12
7. Detalles de implementación	12
7.1. Vehiculo	12
7.2. Elemento	12
7.2.1. Meta	13
7.3. Mapa	13
7.4. Direccion	13
7.5. Interaccion Vehiculo-Obstaculo	13
7.6. Ranking y Persistencia	14
7.7. Juego	14
7.8. Modelo-Vista-Controlador (MVC)	14
7.8.1. Controladores	15
7.9. Null-Object Pattern	16
7.10. Inyeccion de Dependencias	16
7.11. Programacion contra Abstracción	16
7.12. Principio de Responsabilidad Unica	16
8. Consideraciones	16
8.1. Cambios a futuro:	16
9. Excepciones	17
9.1. (No) Excepcion cuando el usuario intentar ir fuera del Mapa	17
10. Conclusion	17

1. Intruduccion

Este trabajo consiste en diseñar un juego con interfaz gráfica. GPS es un juego de estrategia por turnos. El escenario es una ciudad y el objetivo, guiar un vehículo a la meta en la menor cantidad de movimientos posibles.

El juego se jugará por turnos, y en cada turno el usuario decide hacia cuál de las 4 esquinas posibles avanzará. Se tienen distintos vehículos, obstáculos y sorpresas con distintas funcionalidades.

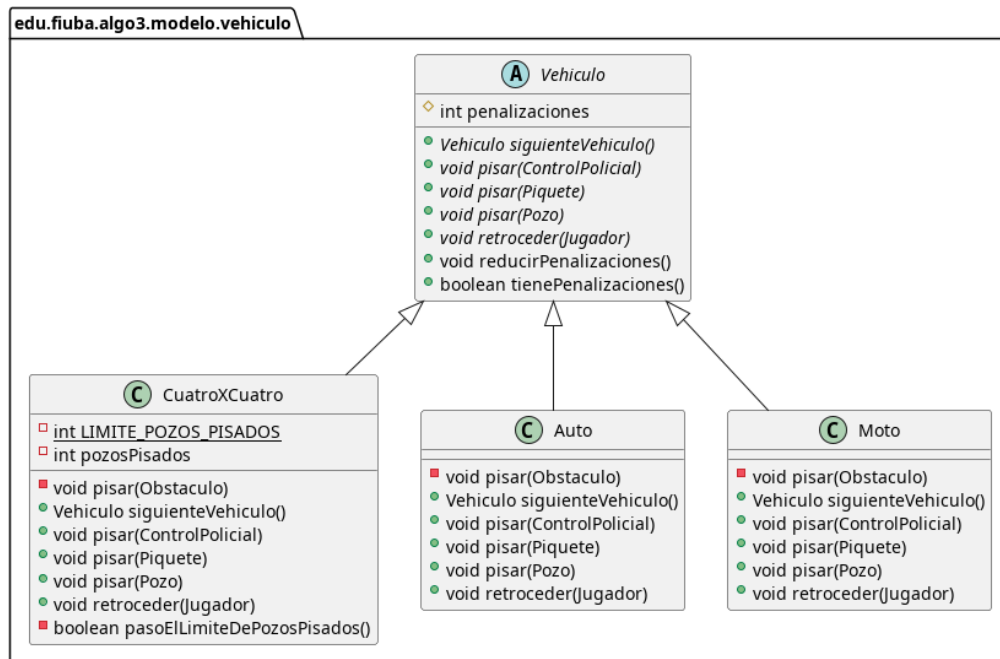
2. Supuestos

El trabajo se realizó con los siguientes supuestos ya que no eran especificaciones dentro de las consignas.

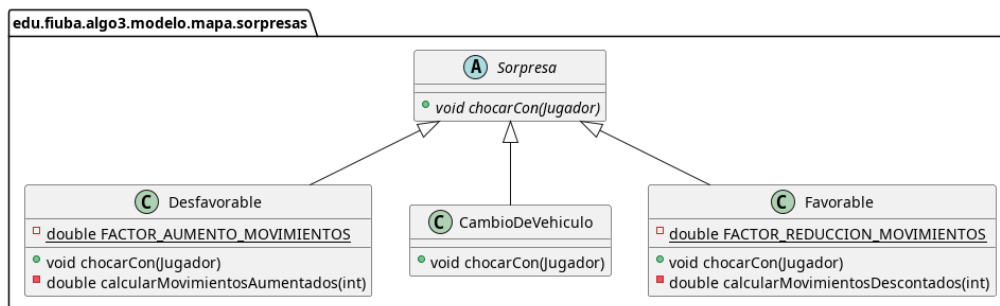
- Cuando se atraviesa un obstáculo aumentan las penalizaciones y por lo tanto el jugador no se puede mover.
- Para iniciar el juego el usuario tiene la posibilidad de agregar jugadores. Cada juego es secuencial, por lo tanto, cuando llegue uno a la meta comienza el turno del otro.
- Si el jugador se encuentra parado en una posición vacía se encuentra sobre un ElementoNulo, es decir, siempre hay un elemento en una posición.
- El jugador comienza con una moto como vehículo.
- Puede haber dos jugadores con el mismo nombre.
- El mapa se genera aleatoriamente cada vez que se abre la aplicación.
- El mejor puntaje es el jugador que tenga menor movimientos.
- Cuando un jugador llega a los límites del mapa y trata de salirse se queda en la misma posición y aumenta en uno sus movimientos.
- Cuando el vehículo es un auto o una 4x4 y quiere pasar por un piquete, estos no podrán avanzar y se sumarán movimientos.
- La posición inicial del jugador es siempre la misma, en la esquina superior izquierda.
- La posición de la meta se genera en una posición aleatoria de la última fila.
- No se generan obstáculos o sorpresas sobre la posición inicial del jugador ni sobre la meta.
- No puede haber dos elementos en una misma posición.
- Un jugador no se puede mover si tiene penalizaciones.

3. Diagramas de clases

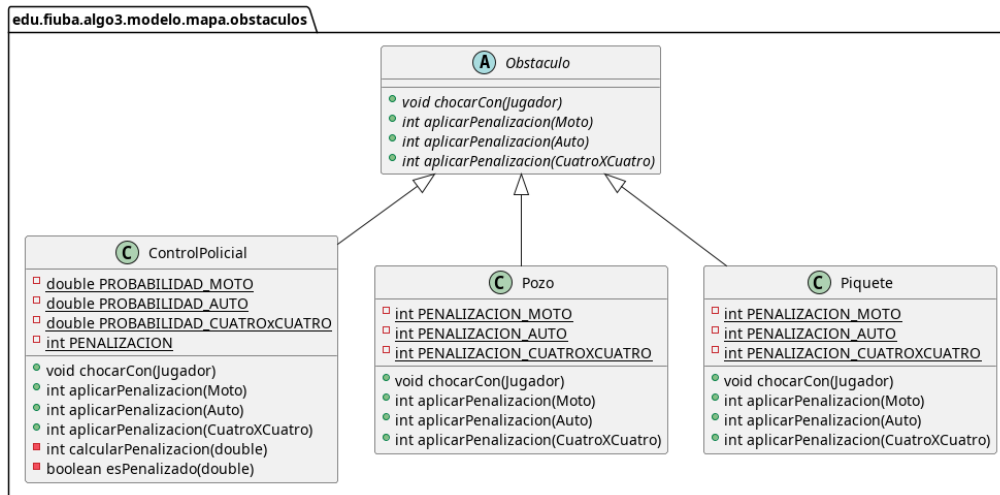
3.1. Vehiculo



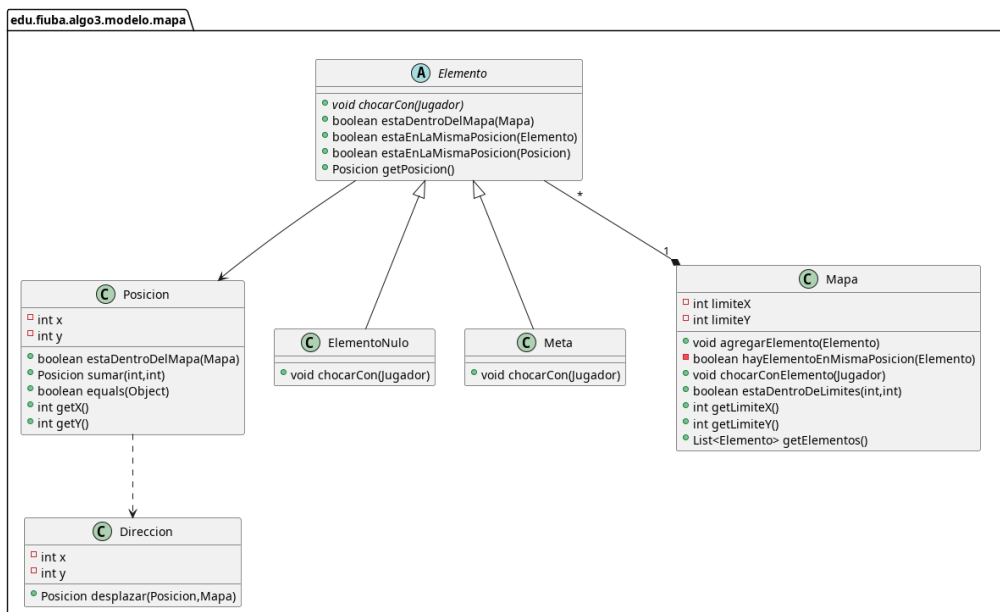
3.2. Sorpresas



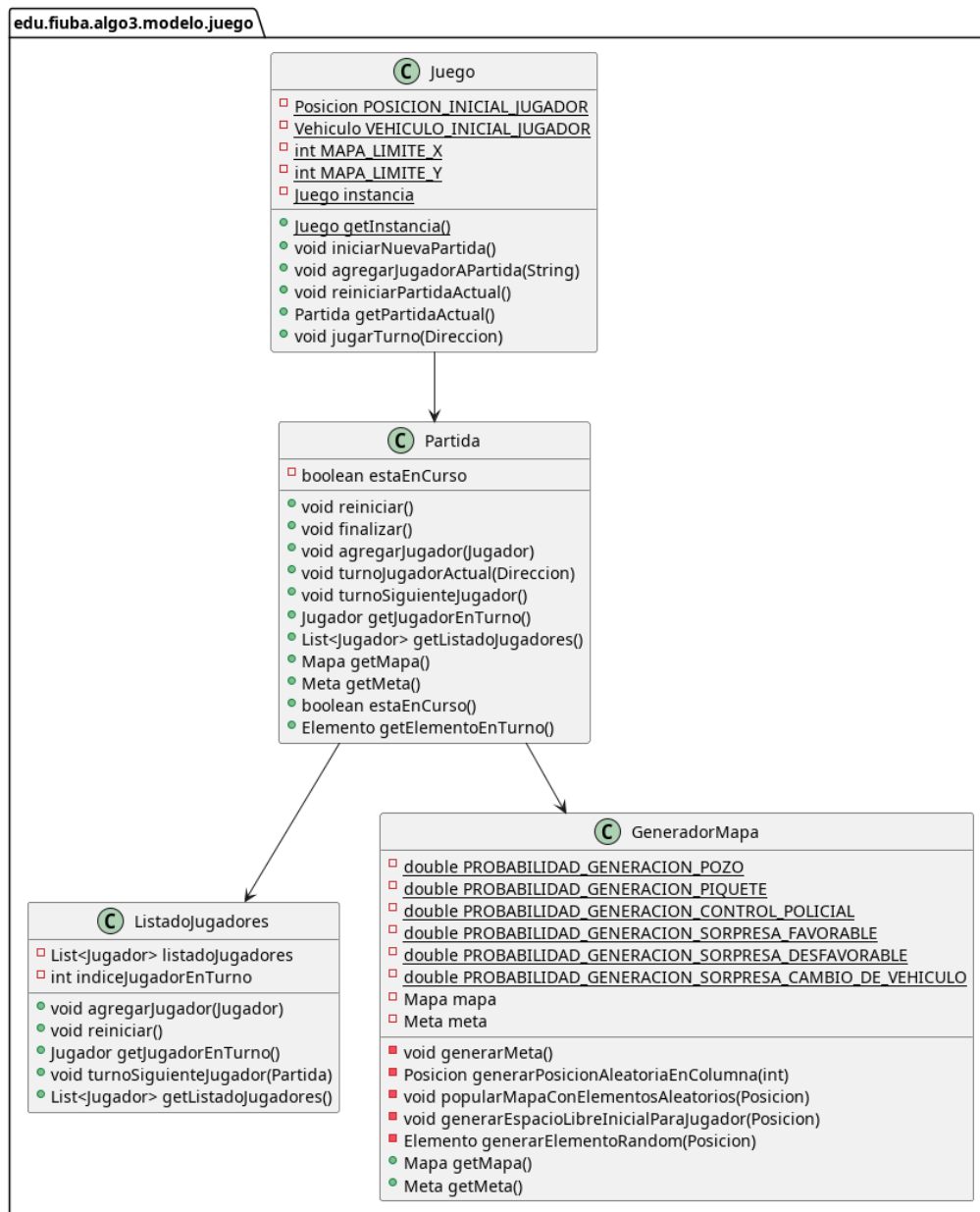
3.3. Obstaculos



3.4. Mapa












3.5. Juego



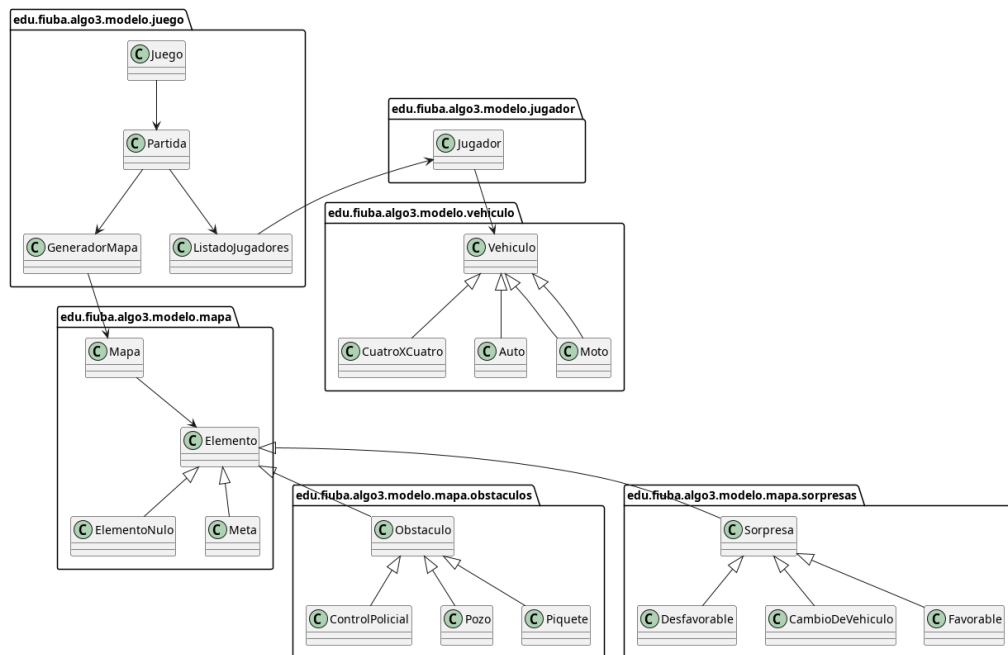
3.6. Jugador

edu.fiuba.algo3.modelo.jugador

Jugador

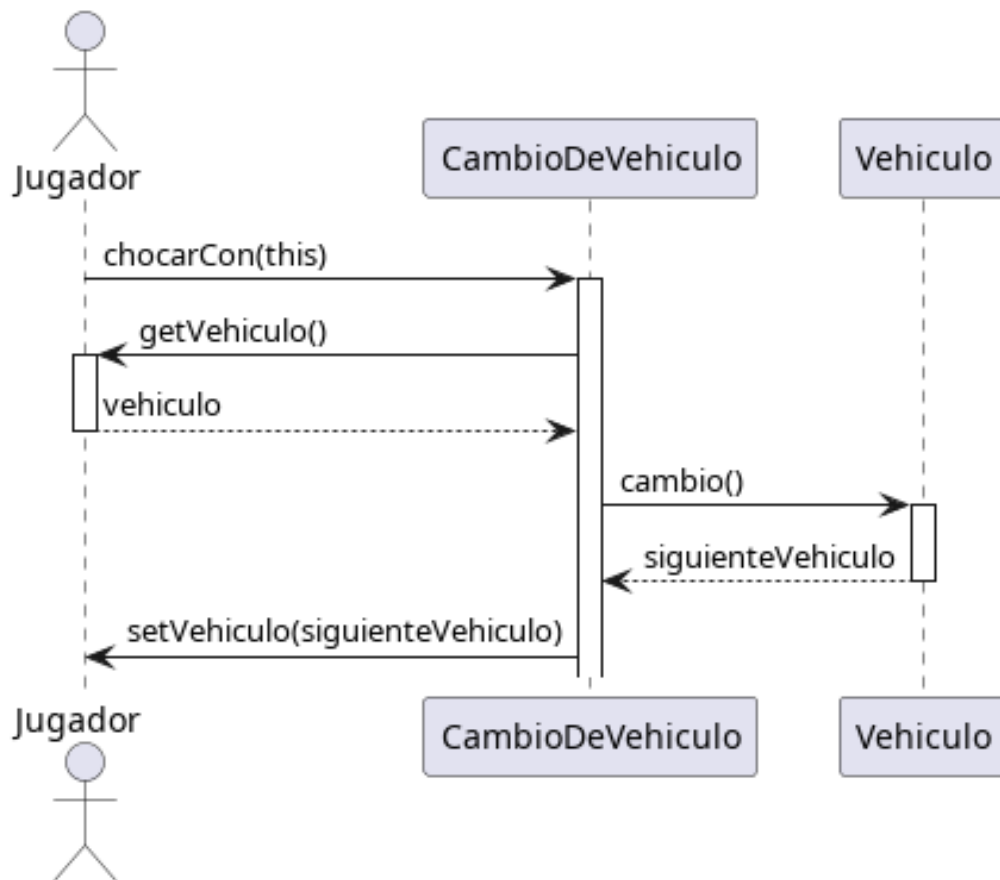
- ☐ String nombre
 - ☐ Posicion posicionInicial
 - ☐ Vehiculo vehiculoInicial
 - ☐ Vehiculo vehiculo
 - ☐ Posicion posicion
 - ☐ Posicion posicionAnterior
 - ☐ int movimientos
-
-  void avanzar(Direccion, Mapa)
 -  void retroceder()
 -  void reiniciar()
 -  void cambiarVehiculo()
 -  Vehiculo getVehiculo()
 -  Posicion getPosicion()
 -  int getMovimientos()
 -  void setMovimientos(int)
 -  String getNombre()

4. Diagrama de paquetes

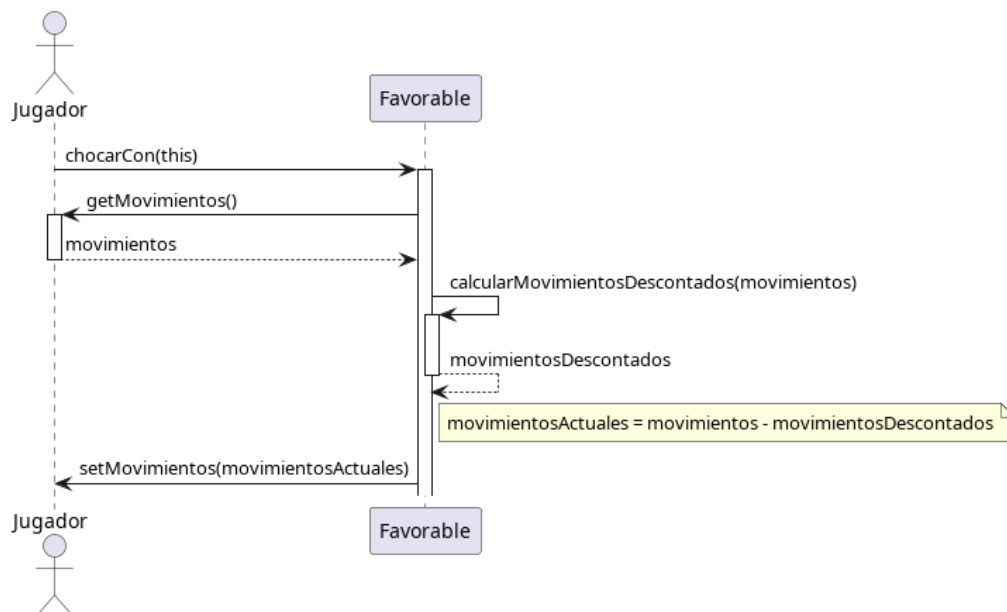


5. Diagramas de secuencia

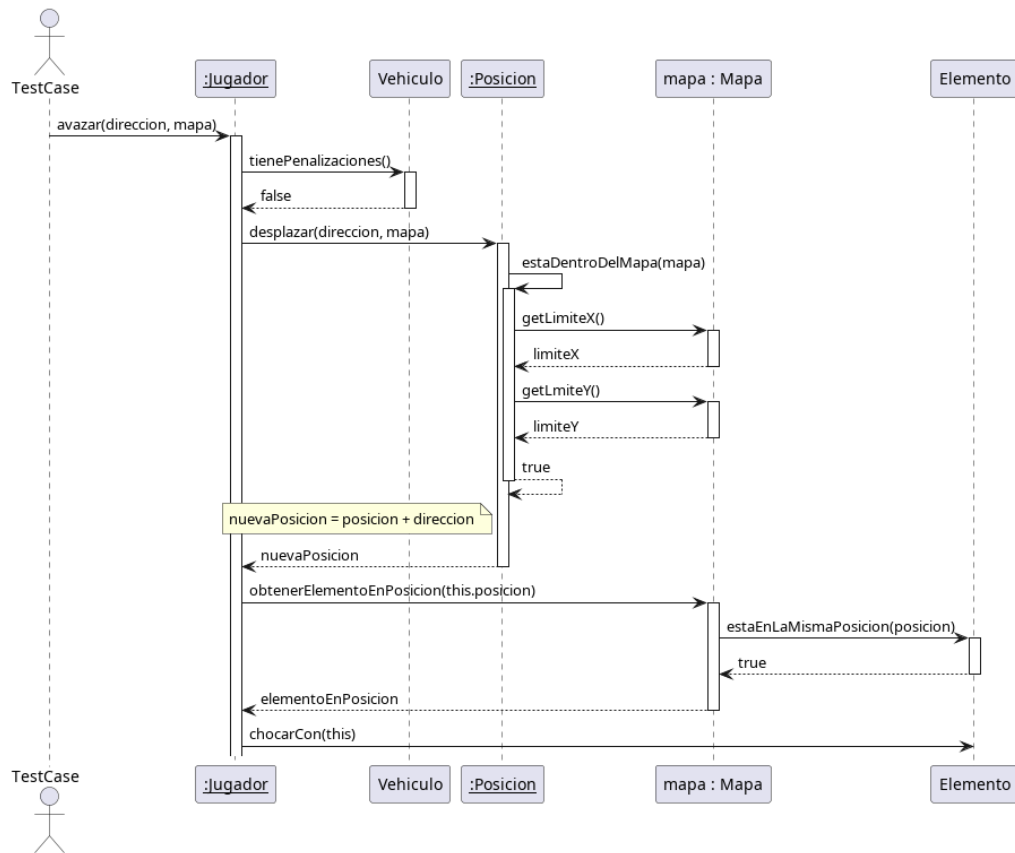
5.1. Interaccion Jugador - Sorpresa Cambio de Vehiculo



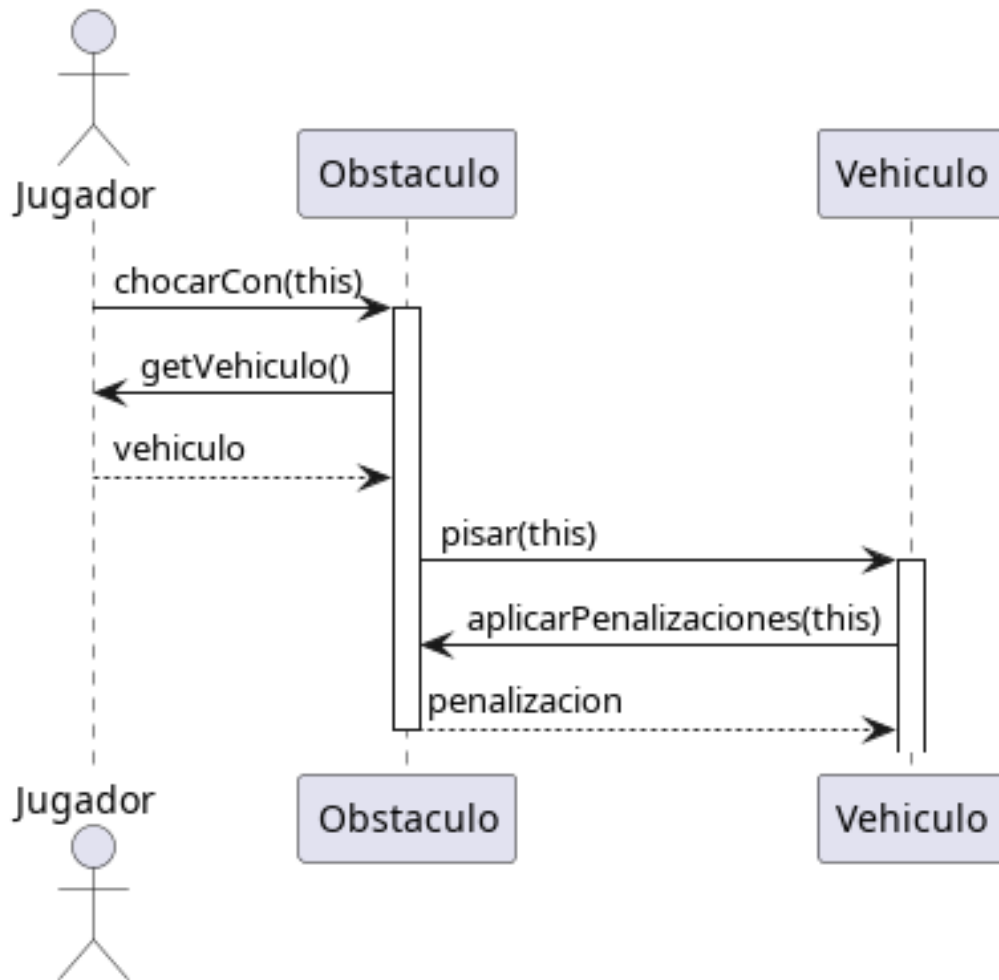
5.2. Interaccion Jugador - Sorpresa Favorable



5.3. Interaccion Jugador - Elemento



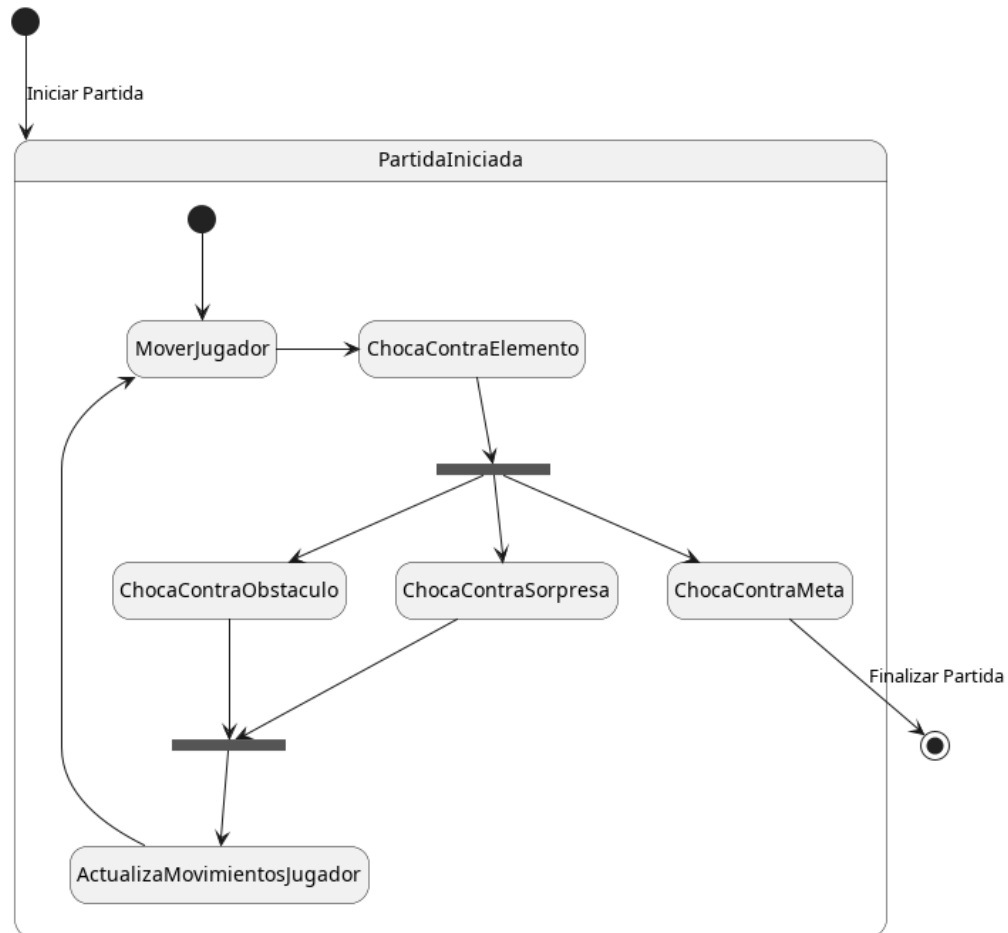
5.4. Jugador avanza y se encuentra con un Elemento



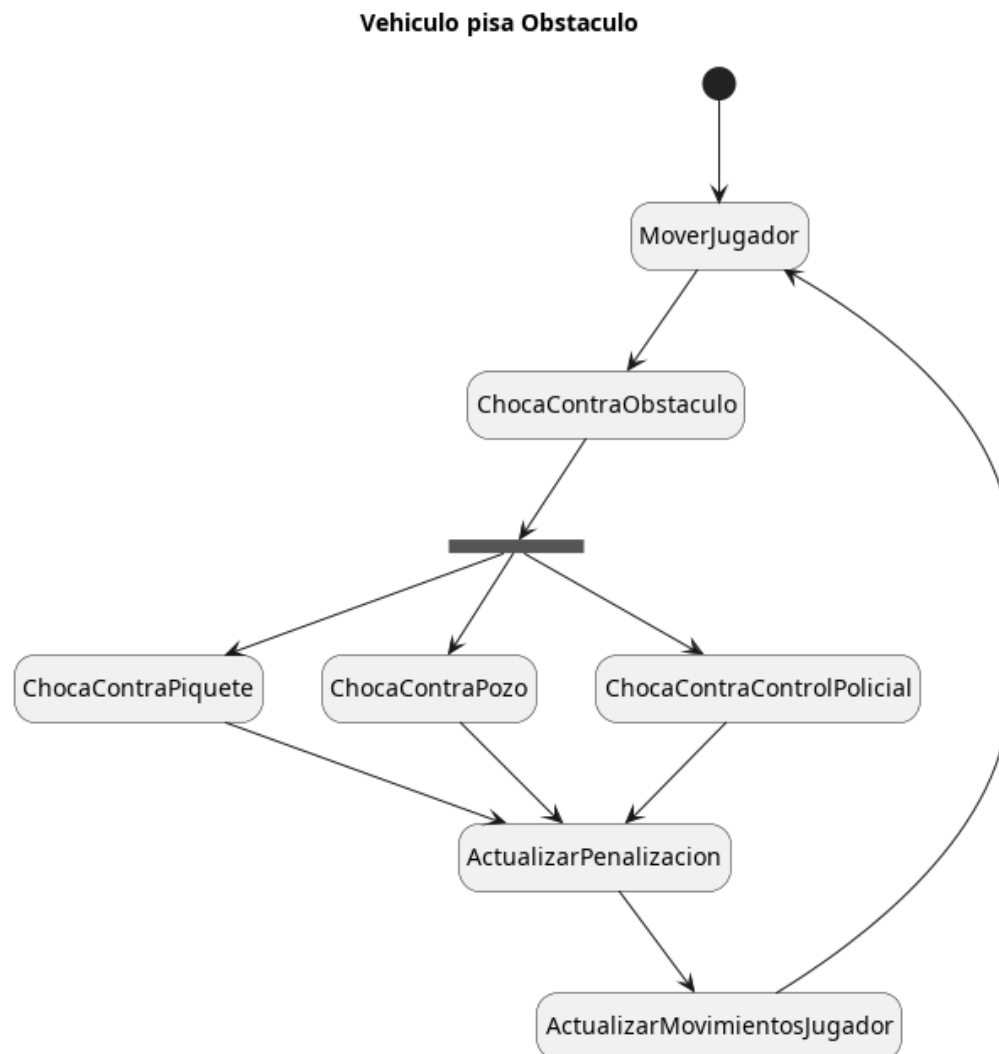
6. Diagramas de estado

6.1. Cambio de Vehiculo del Jugador

Movimiento del Jugador y Choque contra Elemento



6.2. Vehiculo Pisa Obstaculo



7. Detalles de implementación

7.1. Vehiculo

En principio tenes una clase abstracta llamada *Vehiculo* y usamos herencia para abstraer comportamiento comun entre su tres clases hijas: Moto, Auto y CuatroXCuatro.

Los autos y las 4x4 no pueden pasar los piquetes. Cuando avanzan hacia un piquete se posicionan sobre este pero como es una posición que no está permitida para dichos vehículos estos retroceden a su posición anterior. Una vez que sucede esto se actualiza la vista y como la posición se mantiene lo único que cambia es la cantidad de movimientos que se muestran en pantalla.

Tanto para los vehículos como para los elementos se utilizó herencia ya que se cumple la relación "es un". Además, se necesita que contengan los mismos atributos y métodos en común. También fue necesario sobrescribir algunos métodos.

7.2. Elemento

Es una clase abstracta de la cual heredan dos clases:

- Obstaculo
 - Pozo, Piquete y Control Policial.
- Sorpresa
 - Favorable y Cambio de Vehiculo
- Meta

Utilizamos esta clase para definir comportamientos que los distintos Elementos tienen en comun, como por ejemplo que pueden **chocaCon** un jugador, y algunas funciones de ayuda para saber si el elemento esta adentro del mapa, si esta en la misma posicion que otro elemento o una posicion arbitraria, etc.

7.2.1. Meta

La clase meta es un elemento y cuando se lo choca comienza el turno del siguiente jugador. Si es el último, se finaliza la partida.

7.3. Mapa

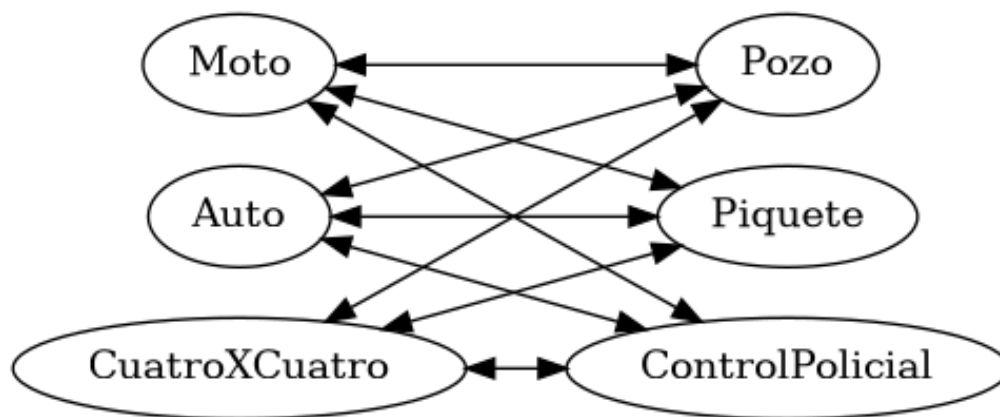
El mapa contiene una lista de elementos y cada elemento posee una posición.

7.4. Direccion

Se implementó la clase **Dirección** encargada de delegar el desplazamiento del jugador a la clase **Posición**.

7.5. Interaccion Vehiculo-Obstaculo

Para la interaccion Vehiculo-Obstaculo decidimos usar el patron *Double Dispatch* de forma ya que tenemos una interaccion de muchos a muchos entre los hijos de ambas clases abstractas:



Ademas de esto teniamos la necesidad de modelar implementaciones especificas como el caso de CuatroXCuatro-Pozo donde la CuatroXCuatro debe pisar tres pozos para recibir una penalizacion, cosa que no sucede en ninguna de las otras interacciones.

Para esto los Vehiculos tienen firmas segun cada implementacion de Obstaculo. Y cada implementacion de Obstaculo tiene firmas para cada Vehiculo.

7.6. Ranking y Persistencia

Para el ranking usamos un `HashMap<String, Long>` con el que almacenamos como clave el nombre del jugador y como valor la cantidad de movimientos minimo que hizo.

Esto se maneja en el `ControladorHistorialPartidas` que tiene dos metodos que hacen uso de la libreria Gson para crear un JSON del `HashMap`, escribirlo en un archivo `ranking.json` y otro metodo para obtener el historial en siguientes ejecuciones del programa *deserializando* el JSON parseado como un `HashMap` de nuevo.

```
1 {  
2   "Pablo": 24,  
3   "Ramiro": 20,  
4   "Carlos": 30  
5 }
```

7.7. Juego

El punto de entrada la aplicación es la clase `Juego`, que provee al usuario de las funciones principales para iniciar una partida y que cada jugador se mueva. También permite a algunas partes de la vista obtener información sobre el estado actual del juego.

La clase `Juego` es un singleton ya que solo puede haber una instancia del juego. Dicha clase contiene un generador del mapa y una partida. Las partidas contienen un listado de jugadores. Cada vez que se inicia el juego se crea una partida. Se pueden agregar varios jugadores a una partida (modo multijugador). Se puede reiniciar una partida y el mapa generado se conserva en este caso.

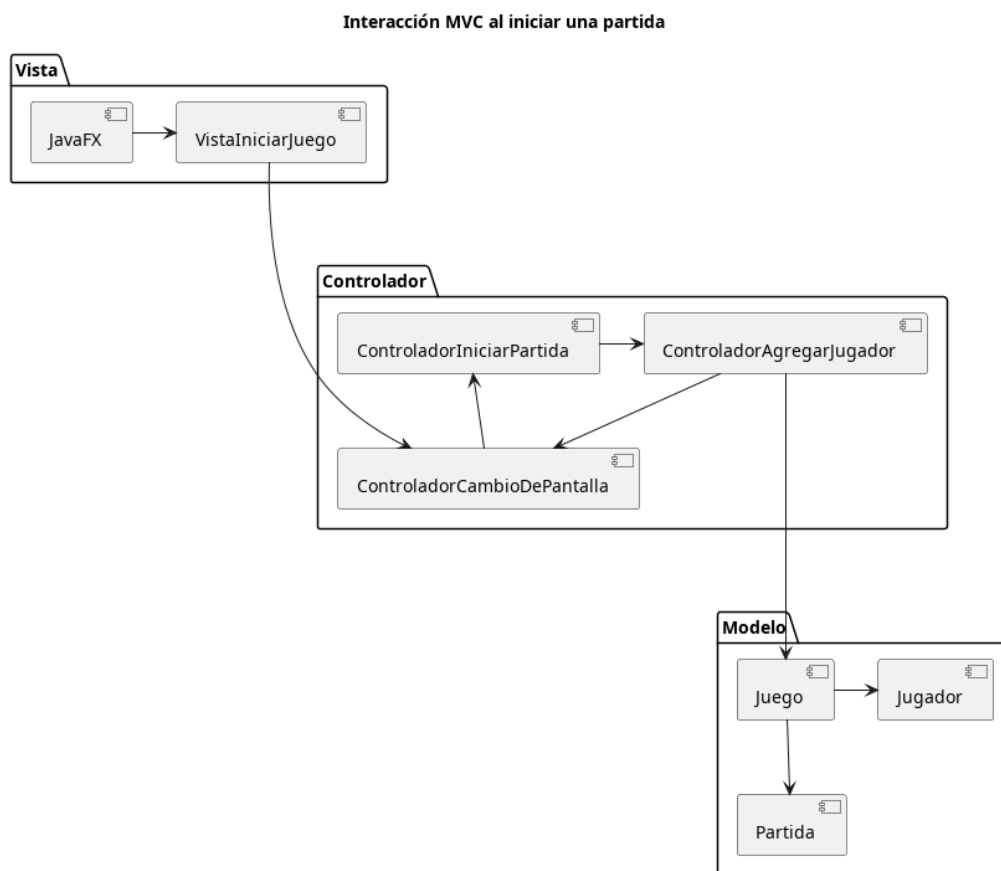
Desde los controladores se obtiene la instancia del juego y se manipula para iniciarlo, reiniciarlo y agregar jugadores.

El juego genera un mapa aleatorio a través de la clase `GeneradorMapa`. Si bien el mapa contiene una lista de elementos y cada elemento tiene su posición, se recorre el mapa vacío como una matriz para generar distintos elementos con distinta probabilidad.

7.8. Modelo-Vista-Controlador (MVC)

Aplicamos este patrón de diseño de software para separar la lógica del funcionamiento del juego de la interfaz.

Se usaron controladores para definir *evento* que ocurren durante el juego, por ejemplo, iniciar o finalizar una partida, o agregar un nuevo jugador. La mayoría de estos controladores extienden la clase `EventHandler` de JavaFX, por lo que son asignables a botones en la interfaz.



7.8.1. Controladores

1. ControladorCambioDePantallas

Se utilizó un controlador llamado **ControladorCambioDePantallas** para lidiar con el cambio de escenas sobre una misma *stage* de JavaFX. Este controlador permite concentrar todas las transiciones entre diferentes vistas (referidas como *pantallas*) para hacer estos cambios de manera más ordenada. Así por ejemplo, cualquier botón cuya tarea fuese mostrar la pantalla de ayuda, utiliza un único controlador para hacer esta transición, lo mismo para cualquier botón que retroceda a la pantalla de inicio, etc. Este controlador permite ahorrarnos estar pasando una referencia al *stage* principal para cada vista que fuese a necesitar interactuar con dicho *stage*, lo que hacía el código algo más caótico. También se utiliza este mismo controlador para interactuar con el *stage* cuando se alterna entre la pantalla completa y minimizada.

2. ControladorPostTurnoJugador

El controlador **ControladorPostTurnoJugador** se encarga de realizar todas las tareas de comprobación de finalización de turno luego de cada movimiento del jugador. Este controlador es referenciado dentro del main *event loop* que escucha cada presionar de tecla del usuario durante la partida en la clase **VistaPantallaPartida**.

Este controlador también se encarga de que las vistas se vuelvan a renderizar luego de cada turno, para actualizar las posiciones y puntajes de los jugadores. Para esto las vistas que muestran información que necesita ser actualizada durante el juego proveen los métodos correspondientes que toman la información del estado actual del juego y también vuelven a generar las figuras que estas vistas contienen con esta información actualizada.

La mayoría de los estilos fueron definidos utilizando CSS para simplificar la estructura de las clases de vista y reutilizar los estilos comunes a varios componentes del juego que tienen la misma estética, como los botones, por ejemplo.

3. ControladorHistorialPartidas

El controlador `ControladorHistorialPartidas` se encarga de tomar la información de la partida recientemente finalizada y la agrega el registro de partidas previamente guardadas en un archivo con formato JSON.

7.9. Null-Object Pattern

Se utilizó el patrón *Null Object* para hacer polimórfico el comportamiento de choque del jugador al moverse hacia cualquier posición, independientemente de que haya un obstáculo o sorpresa con algún efecto en esa posición.

7.10. Inyección de Dependencias

En varias clases se hicieron las dependencias inyectables, de tal forma que fuera fácil reemplazar el comportamiento de dichas clases. Por ejemplo, al crear un jugador se pueden definir tanto su posición inicial como su vehículo inicial como dependencias.

7.11. Programación contra Abstracción

También se programó contra abstracción en vez de contra clases concretas en donde se vió óptimo. Por ejemplo, en el caso de los vehículos y los obstáculos se diseñó todo de tal forma que al jugador no le importase contra qué obstáculo estaba chocando y qué vehículo tenía ese momento, permitiendo que el comportamiento fuese polimórfico y facilitando la adición de nuevos obstáculos y vehículos.

7.12. Principio de Responsabilidad Única

En el caso del diseño e implementación de la clase `Mapa`, se hicieron cambios durante el diseño del modelo para que esta clase tuviera una única responsabilidad, respetando el principio de única responsabilidad. Inicialmente esta clase era vista como un "administrador de elementos" (como se había descrito en alguna de las entregas semanales), pero finalmente terminó siendo únicamente una colección de elementos dentro de unos límites.

8. Consideraciones

8.1. Cambios a futuro:

- Implementar la funcionalidad multimedia. Lamentablemente debido al sistema operativo que utilizamos la mayoría de integrantes del grupo, no se pudo implementar dicha funcionalidad por problemas de la librería utilizada. Esto es culpa de JavaFX por no soportar Linux.
- Agregar una opción para que cada jugador pudiese elegir su carácter al momento de iniciar una partida de entre las imágenes de jugadores disponibles.
- Poner más imágenes de Messi.
- Implementar múltiples idiomas y múltiples temas de colores. Agregar funcionalidad de cambio de nivel. Con la implementación de `GeneradorMapa` esto se hace más sencillo pues se puede utilizar el patrón factory para cambiar el algoritmo de generación de elementos en el mapa de una partida en particular.

- Refactorizar y organizar la implementación de la vista y el controlador para evitar la duplicación de código.

Dentro de los cambios positivos del trabajo consideramos que la implementación del controlador `ControladorCambioDePantallas` fue beneficiosa ya que manipula el stage y no es necesario pasarlo como parámetro entre los distintos controladores y métodos.

9. Excepciones

9.1. (No) Excepcion cuando el usuario intentar ir fuera del Mapa

Una observacion que tuvimos durante el desarrollo fue la posibilidad de agregar una excepcion cuando el usuario intenta ir fuera de los bordes del mapa. Nosotros como supuesto elegimos no tratar esto como un error y directamente el modelo maneja esta posibilidad y no permite al usuario avanzar por fuera de los limites del mapa.

10. Conclusion

Utilizar prácticas de *extreme programming* y *agile* nos permitieron mantener un desarrollo colaborativo eficiente. Aplicar programación orientada a objetos en la implementación nos permitió dividir tareas sin manipular los mismos archivos. Aplicar *pair programming* e integración continua fue beneficioso para avanzar rápidamente en el desarrollo del trabajo. Además, facilitó las refactorizaciones realizadas sin tener que modificar significativamente el código ya implementado.