FIUBA - 7507

Algoritmos y programación 3

Trabajo práctico 2: GPS Challenge

2do cuatrimestre, 2013

(trabajo grupal)

Alumno:

Nombre	Padrón	Mail
Hernan Arroyo	91257	hernan_15_25@hotmail.com
Nicolas Carolo	95031	nico.carolo@hotmail.com
Ezequiel Grillo	93436	ezequiel.grillo@gmail.com
Martin Stancanelli	95188	mastancanelli@outlook.com

Fecha de entrega final:

Tutor:

Nota Final:

Tabla de contenidos

Introducción

Objetivo del trabajo

Aplicar los conceptos enseñados en la materia a la resolución de un problema, trabajando en forma grupal y utilizando un lenguaje de tipado estático (Java)

Consigna general

Desarrollar la aplicación completa, incluyendo el modelo de clases e interface gráfica. La aplicación deberá ser acompañada por prueba unitarias e integrales y documentación de diseño. En la siguiente sección se describe la aplicación a desarrollar.

Descripción de la aplicación a desarrollar

Contexto

La empresa Algo Ritmos SA dedicada al desarrollo de video juegos a decidido contratar a un grupo de programadores para implementar el juego GPS Challenge

Objetivo del juego

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.

Dinámica del juego

El juego se jugará por turnos, y en cada turno el usuario decide hacia cual de las 4 esquinas posibles avanzará.

Vehículos

El jugador podrá optar por tres diferentes tipos de vehículos.

- moto
- auto
- 4x4

Obstáculos

Al atravesar una cuadra el jugador se podrá encontrar con alguno de los siguientes obstáculos:

- Pozos: Le suma 3 movimientos de penalización a autos y motos, pero no afecta a las 4x4.
- Piquete: Autos y 4x4 deben pegar la vuelta, no pueden pasar. Las motos puede pasar con una penalización de 2 movimientos.
- Control Policial: Para todos los vehículos la penalización es de 3 movimientos, sin embargo la probabilidad de que el vehículo quede demorado por el control y sea penalizado es de 0,3 para las 4x4, 0,5 para los autos y 0,8 para las motos ya que nunca llevan el casco puesto.

Sorpresas

También se podrán encontrar diferentes tipos de sorpresas:

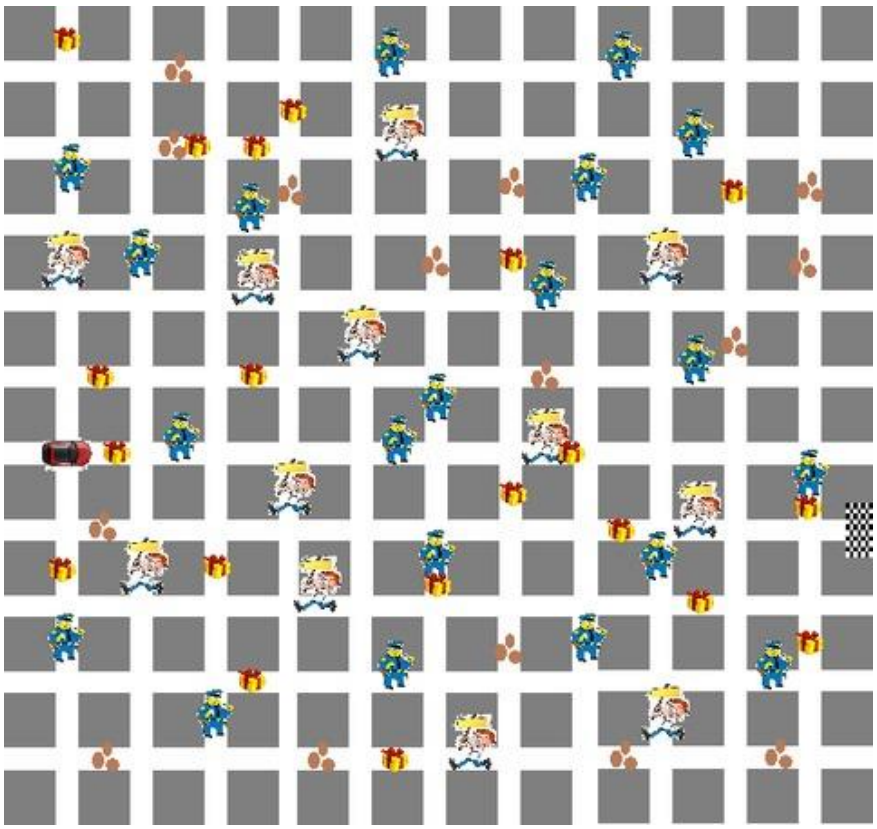
- Sorpresa Favorable: Resta el 20% de los movimientos hechos.
- Sorpresa Desfavorable: Suma el 25% de los movimientos hechos.
- Sorpresa Cambio de Vehículo: Cambia el vehículo del jugador. Si es una moto, la convierte en auto. Si es un auto lo convierte en 4x4. Si es una 4x4 la convierte en moto.

Las sorpresas figuraran en el mapa como un regalo y no se sabrá que es hasta que el vehículo la accione.

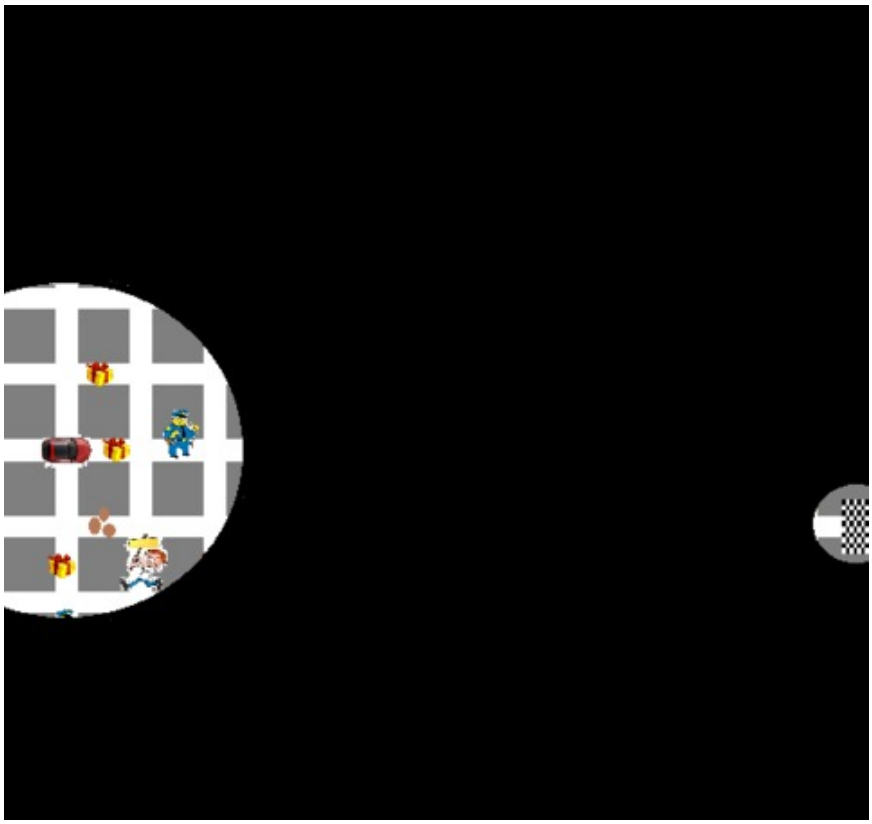
Escenario

Para hacerlo más interesante y jugable, el jugador no podrá ver más que dos manzanas a la redonda de la posición de su vehículo y la bandera a cuadros que marca la meta. El resto del mapa permanecerá en sombras.

El tamaño del escenario no será fijo, y tendrá un punto de partida y una meta.



Ejemplo de escenario



Ejemplo de escenario como lo visualiza el jugador

Puntajes altos

Se debe almacenar un ranking donde figuren los mejores puntajes asociados a un nickname que indique el usuario.

Entregables

- Código fuente de la aplicación completa, incluyendo también: código de la pruebas, archivos de recursos
- Script para compilación y ejecución (ant)
- Informe, acorde a lo especificado en este documento

Forma de entrega

A coordinar con el docente asignado.

Fechas de entrega

Se deberá validar semanalmente con el docente asignado el avance del trabajo. El docente podrá solicitar ítems específicos a entregar en cada revisión semanal.

La entrega final deberá ser en la semana del 13 de diciembre, en la fecha del curso en que se está inscripto.

Informe

Supuestos

-Decidimos no permitir al usuario ver todas las partidas jugadas.

-Los tamaños del mapa son fijos, solo varían eligiendo la dificultad del juego:

Difícil: 8x8 (alto x ancho)

Intermedio: 5x5

Fácil: 3x3

-Definimos los movimientos máximos al usuario teniendo en cuenta la distancia entre bandera y vehículo, sumándole una cantidad dependiendo la dificultad del juego.

-Asumimos que una Esquina pueda tener solo una Sorpresa y un solo Obstáculo.

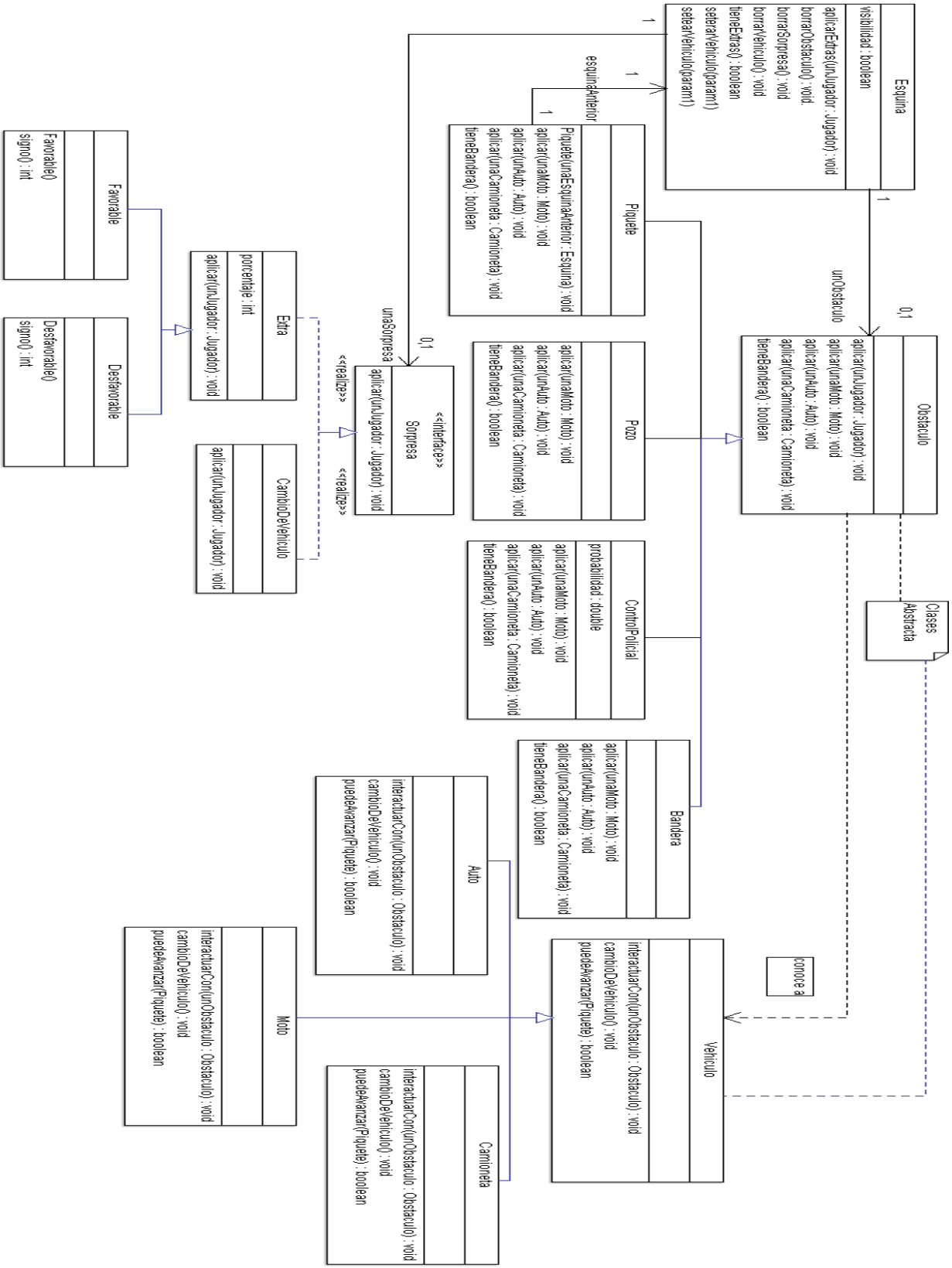
Modelo de dominio

Al comenzar el trabajo práctico decidimos hacer un bosquejo de clases que desde un principio tendrían que estar obligatoriamente. Así fue que comenzamos pensando donde se movería el vehículo, y nos dimos cuenta que lo central era que el vehículo se movía a través de esquinas, por lo tanto diseñamos la clase ***Mapa*** formada por esquinas, que van a tener una ***Posición*** por la que van a ser identificadas y van a contener al vehículo en cuestión, a objetos que implementen a ***Sorpresa*** y a los ***Obstáculos***. Luego mediante herencia para con ***Vehículo*** definimos las clases ***Auto***, ***Moto*** y ***Camioneta***, estos objetos tienen referencia a la esquina en la que están, al jugador al que pertenecen y a movimientos permitidos (definidos para todos los tipos de vehículos como 1, aunque en un principio se pensaba en que cada tipo maneje distintas distancias por movimiento). Posteriormente creamos la clase ***Jugador*** que va a tener: un nombre, al vehículo con el que va a jugar, la cantidad de movimientos hechos y un estado que permitirá saber si gano o no, en complemento como queríamos encapsular la forma en que se mueve el vehículo, pensamos en la clase ***Dirección*** que va a estar en Jugador y que es clase madre de ***Arriba***, ***Abajo***, ***Izquierda***, ***Derecha***, cuyo funcionamiento podrá ser seteado en Jugador para que luego al jugar se observe su estado y se dirija el movimiento. A continuación, extendimos el significado de la clase Obstáculo y la interfaz ***Sorpresa***, ya que creamos las clases ***Pozo***, ***ControlPolicial*** y ***Piquete*** (hijas de ***Obstáculo***), y ***CambioDeVehiculo*** y ***Extra*** (implementan ***Sorpresa***) y a su vez, ***Favorable*** y ***Desfavorable*** (hijas de ***Extra***). Para cumplir con el problema del radio de visibilidad del ***Vehículo***, implementamos la clase ***CambiadorDeVisibilidad***, que para el modelo va a ser utilizada y funciona correctamente, pero en cuestión de funcionalidad creemos que es mejor manejar el radio de visibilidad desde la interfaz gráfica. Por último, creamos la clase ***Juego*** para englobar el funcionamiento de todas las demás, es por eso, que esta clase va a tener a los elementos principales del juego, el ***Mapa*** y el ***Jugador***, para separar tres tipos de juegos posibles, creamos ***JuegoDificil***, ***JuegoIntermedio*** y ***JuegoFacil*** que heredan de ***Juego***. Y además también va a contar con la posición de donde está la bandera y la posición inicial del vehículo, que van a ser definidas por la clase ***Randomizador***, y luego utilizadas para definir la cantidad de movimientos posibles del jugador.

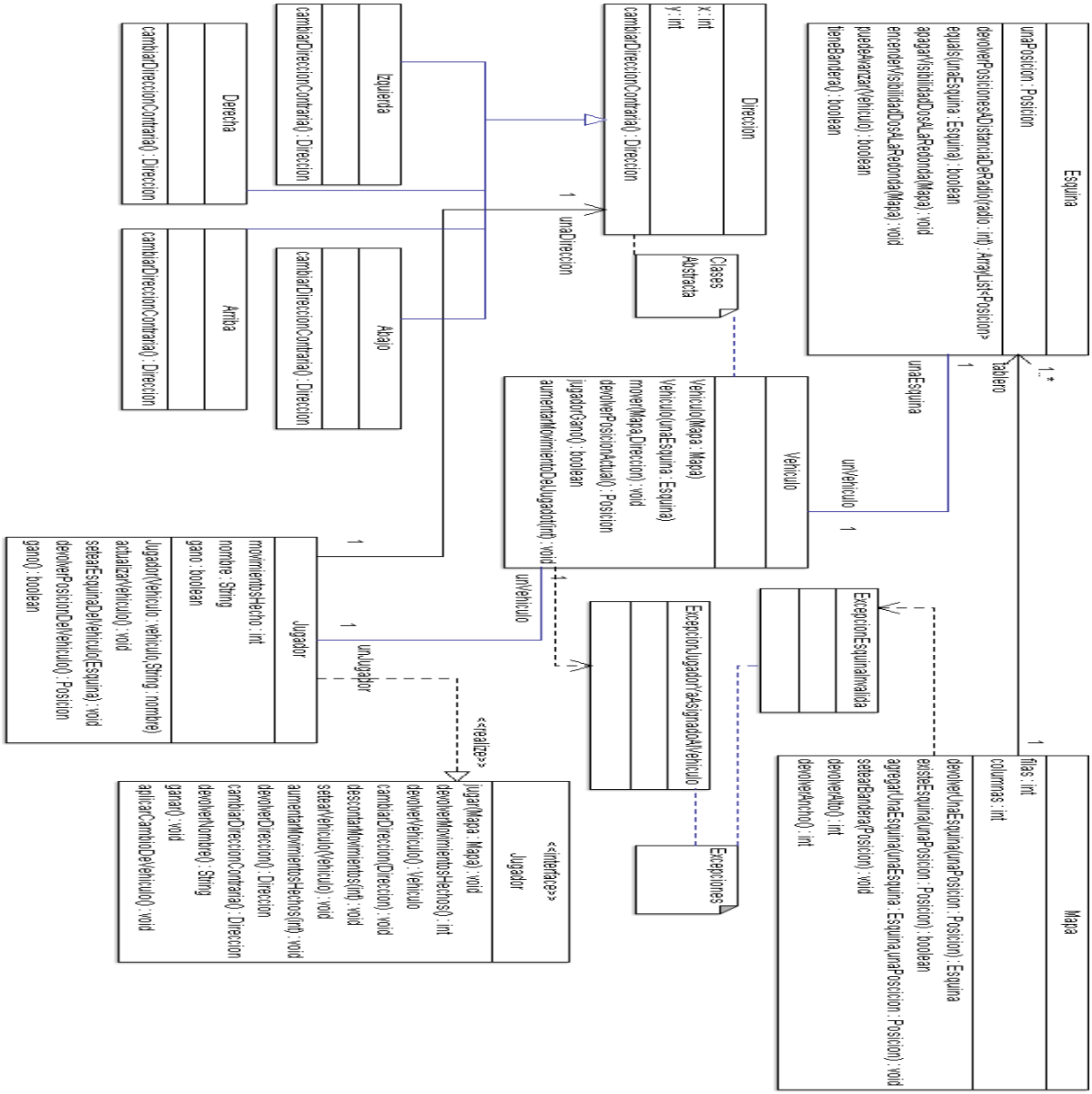
Para culminar, no queremos dejar de mencionar el problema más grande que tuvimos: ***Observador***-***Observado***, y aunque sabemos que no lo resolvimos de la mejor manera creemos llegamos a alcanzar un funcionamiento correcto.

Diagramas de clases

Los diagramas pueden ser vistos mejor en el .zip entregado.



Juego
cantidadDePiquetes : int
cantidadDeControlPoliciales : int
cantidadDePozos : int
cantidadDeFavorables : int
cantidadDeDesfavorables : int
cantidadDeCambiosDeVehiculos : int
movimientosMaximos : int
posicionBandera : Posicion
unMapa : Mapa
jugadorImplementacion : Jugador
posicionInicialVehiculo : Posicion
JuegoJString : nombreJuegador,int: tamanioMapa,Posicion : posicionVehiculo,Vehiculo : vehiculo)
JuegoJugadorImplementacion : unJugador,Mapa : unMapa,Posicion : posicionBandera)
completarMapaConExtras(Posicion : posicionBandera) : void
moveJugadorEnDireccion(Direccion : unaDireccion) : void
jugar(Direccion : unaDireccion) : void
seTermino() : boolean
calcularPuntajeFinal() : int
setearCantidadSorpresas'Obstaculos() : void
ExcepcionJuegoTerminado
<<interface>>
Randomizador
obtenerNumeroRandom() : double
<<realize>>
RandomizadorImplementacion
obtenerNumeroRandom() : double
obtenerNumeroEntre(min : int,max : int) : int
JuegoFacil
JuegoFacil(String : nombreJuegador,int: tamanioMapa,Posicion : posicionVehiculo,Vehiculo : vehiculo) : void
JuegoFacil(JugadorImplementacion : unJugador,Mapa : unMapa,Posicion : posicionBandera)
setearCantidadSorpresas'Obstaculos() : void
calcularPuntajeFinal() : int
JuegoDificil
JuegoDificil(String : nombreJuegador,int: tamanioMapa,Posicion : posicionVehiculo,Vehiculo : vehiculo) : void
JuegoDificil(JugadorImplementacion : unJugador,Mapa : unMapa,Posicion : posicionBandera)
setearCantidadSorpresas'Obstaculos() : void
calcularPuntajeFinal() : int
JuegoIntermedio
JuegoIntermedio(String : nombreJuegador,int: tamanioMapa,Posicion : posicionVehiculo,Vehiculo : vehiculo) : void
JuegoIntermedio(JugadorImplementacion : unJugador,Mapa : unMapa,Posicion : posicionBandera)
setearCantidadSorpresas'Obstaculos() : void
calcularPuntajeFinal() : int



Detalles de implementación

Para solucionar el problema que tuvimos en cuanto a la integración del modelo con la vista y el controlador creamos las clases ObjetoObservable y ObjetoObservador las cuales se comunican con los vehículos para poder saber cuándo cambian su estado. Y así poder comunicarse con la vista para saber en qué momento dibujarlos.

Para las demás clases del dominio como las Sorpresas, los Obstáculos, el Jugador, etc. usamos la interfaz Observer y la clase Observable definidas en Java.

Por otro lado, se utilizó la clase Randomizador, RandomizadorImplementacion y RandomizadorTesteador para poder simplificar la ejecución de los tests.

Con el mismo fin se crearon las clases JugadorImplementacion y JugadorTesteador usadas en las diferentes clases de tests.

Los patrones utilizados durante el desarrollo del trabajo práctico fueron los siguientes:

• Double-Dispatch: Se utilizó este patrón de diseño para implementar los Obstáculos y las Sorpresas. De manera que el vehículo al encontrarse con alguno de ellos reaccione de acuerdo a lo esperado.

• Singleton: Aplicamos este patrón de diseño en la clase ObjetoObservable, ya que este objeto debe tener una única instancia durante la ejecución del juego. Se aplicó porque cuando se instancia un Auto, Moto o Camioneta, el constructor llama al constructor de Vehículo, y este a su vez llama al constructor de ObjetoObservable. Esto hacia que se borre la lista de observadores.

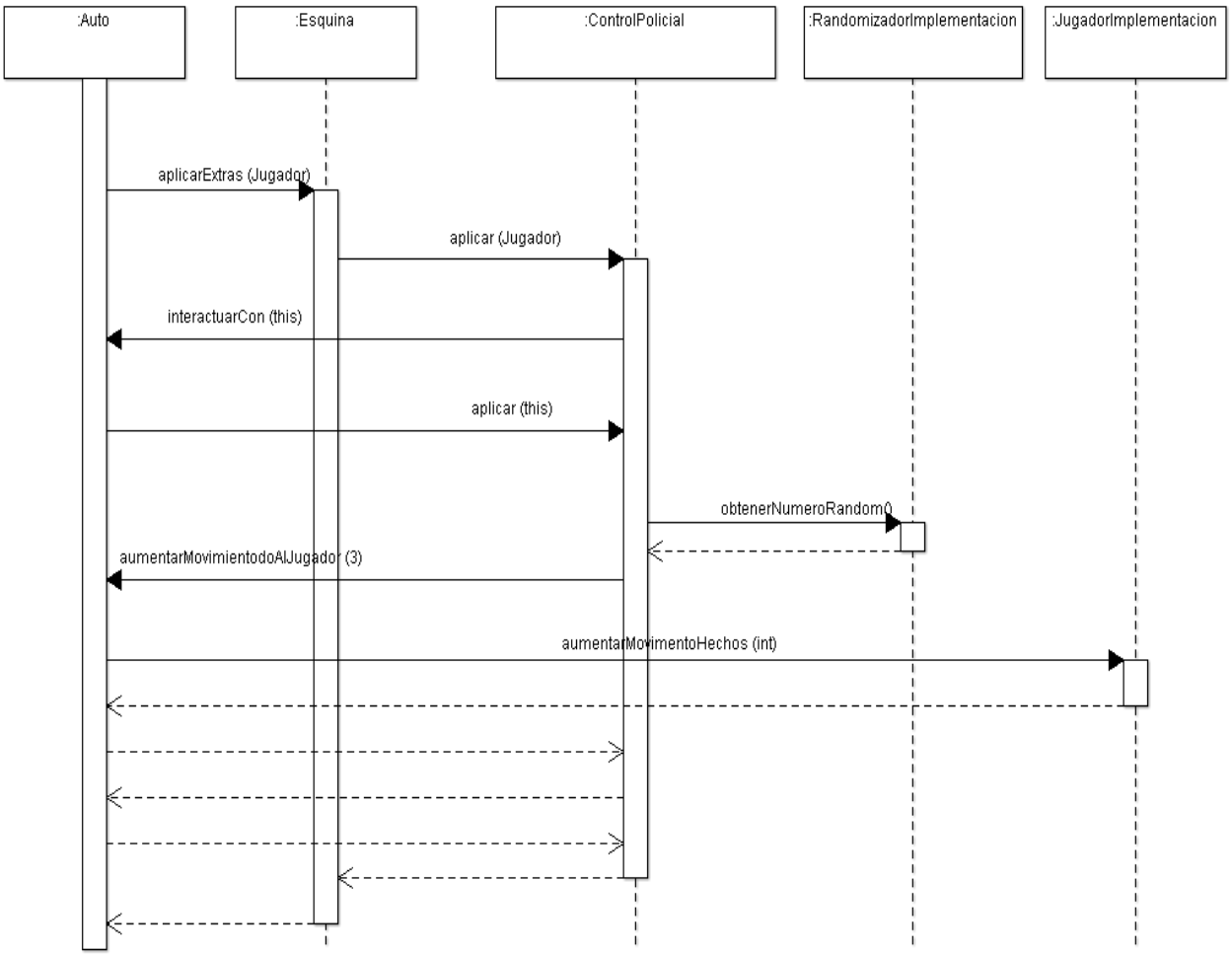
• MVC: Para diseñar la interacción entre la vista, el controlador y el modelo se usó este patrón de diseño. Aunque el resultado final no fue del todo correcto, se intentó aplicarlo en un principio.

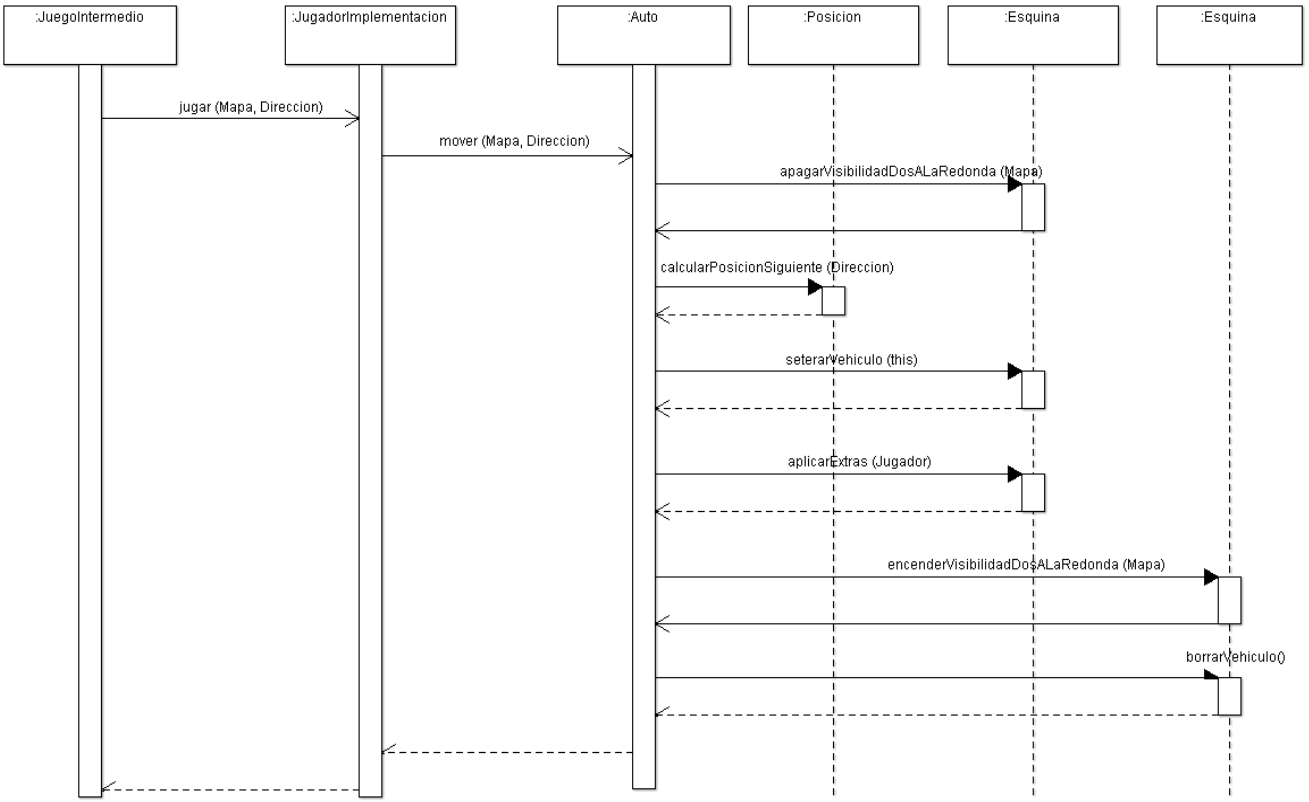
Excepciones

Se crearon las siguientes excepciones:
• ExcepcionEsquinaInvalida: Si bien la existencia de una esquina es chequeada por el mapa y desde el código siempre que se le pide a una esquina en el mapa se chequea si existe anteriormente, se creó esta excepción en caso de que por algún evento excepcional se saltee ese chequeo.
• ExcepcionJuegoTerminado: Esta excepción también se creó en caso de que ocurra un evento inesperado, ya que se chequea que el jugador pueda jugar antes de que este efectué la jugada.
• ExcepcionJugadorYaAsignadoAlVehiculo: En nuestro modelo tenemos un método llamado setearJugadorAlQuePertenece, que le setea al vehículo el jugador que le corresponde. Como solo hay un jugador en todo el juego, para evitar que el jugador asignado al vehículo pueda ser modificado durante la partida, se creó la excepción mencionada.

Diagramas de secuencia

Al igual que los diagramas de clase, estos también se pueden ver más claramente en el .zip entregado.


:Auto
:Esquina
:ControlPolicial
:RandomizadorImplementacion
:JugadorImplementacion
aplicarExtras (Jugador)
aplicar (Jugador)
interactuarCon (this)
aplicar (this)
obtenerNumeroRandom()
aumentarMovimientodoAlJugador (3)
aumentarMovimentoHechos (int)


:JuegoIntermedio
:JugadorImplementacion
:Auto
:Posicion
:Esquina
:Esquina
jugar (Mapa, Direccion)
mover (Mapa, Direccion)
apagarVisibilidadDosALaRedonda (Mapa)
calcularPosicionSiguiente (Direccion)
seterarVehiculo (this)
aplicarExtras (Jugador)
encenderVisibilidadDosALaRedonda (Mapa)
borrarVehiculo()

Checklist de corrección

Esta sección es para uso exclusivo de la cátedra, por favor no modificar.

Carpeta

Generalidades

- ¿Son correctos los supuestos y extensiones?
- ¿Es prolija la presentación? (hojas del mismo tamaño, numeradas y con tipografía uniforme)

Modelo

- ¿Está completo? ¿Contempla la totalidad del problema?
- ¿Respeta encapsulamiento?
- ¿Hace un buen uso de excepciones?
- ¿Utiliza polimorfismo en las situaciones esperadas?

Diagramas

Diagrama de clases

- ¿Está completo?
- ¿Está bien utilizada la notación?

Diagramas de secuencia

- ¿Está completo?
- ¿Es consistente con el diagrama de clases?
- ¿Está bien utilizada la notación?

Diagrama de estados

- ¿Está completo?
- ¿Está bien utilizada la notación?

Código

Generalidades

- ¿Respeta estándares de codificación?
- ¿Está correctamente documentado?