Facultad de Ingeniería

Universidad de Buenos Aires

Departamento de Computación

75.07 Algoritmos y Programación III

Curso 2

1^{er} cuatrimestre 2022

GPS CHALLENGE

Autores:

Facundo Baez (97733)
fbaez@fi.uba.ar

Cristian Torales (95549)
ctorales@fi.uba.ar

Agustin Piperno (96857)
apiperno@fi.uba.ar

Matias Viñas (99705)
mvinas@fi.uba.ar

23 de junio de 2022

Índice

1. Introducción

Durante este Trabajo Practico N°2 se desarrolló una aplicación denominada GPS CHALLENGE, un juego de estrategia por turnos, donde el escenario es una ciudad y el objetivo es guiar a un vehículo a la meta en la menor cantidad de movimientos posibles.

El mismo se implementó con un lenguaje de tipado estático (Java), y fue llevado a cabo utilizando el paradigma de programación orientada a objetos (POO), TDD (Test Driven Development), e integración continua mediante el uso de GitHub Actions.

Se desarrolló la aplicación por completo, incluyendo el modelo de clases e interfaz gráfica. La aplicación es acompañada por pruebas unitarias e integrales y documentación de diseño.

2. Supuestos

El jugador podrá optar por tres diferentes tipos de vehículos:

- moto
- auto
- 4x4

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. Para una 4x4 penaliza en 2 movimientos luego de atravesar 3 pozos.
- Piquete: Autos y 4x4 deben pegar la vuelta, no pueden pasar. Las motos pueden 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.

También, cuando un vehículo este en circulación 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.

El juego se desarrolla en un escenario que representa a una ciudad con sus calles y esquinas. En cada turno el usuario elige una de las cuatro direccionas para avanzar con su vehículo. Las sorpresas y los obstáculos que afectan al vehículo se encuentran en las calles que unen a las esquinas.

El jugador no puede visualizar más de dos manzanas a la redonda de la posición de su vehículo y la bandera que marca la meta. El resto del mapa permanecerá en sombras.

El tamaño del escenario no es fijo: pueden definirse sus dimensiones al comienzo del juego. Por otro lado, el escenario tiene un punto de partida y una meta. Y las sorpresas y los obstáculos se generan de manera aleatoria por distintos puntos del escenario.

3. Desarrollo

3.1. Archivos

El desarrollo del trabajo práctico consta de la documentación y los siguientes archivos:

3.2. Método de Desarrollo

El desarrollo se realizo dividiendo por módulos para obtener tareas que puedan ser trabajas de forma individual, esto se logro utilizando branches de github y luego realizando un cuidadoso merge entre los trabajos realizados. Y además buena parte del código se realizo trabajando de apares, esto es mientras un programador codea el otro lo guía.

Además el desarrollo se realizo de forma iterativa con 4 entregas 1 por semana, más una entrega cero donde se realizo las configuración del entorno de programación:

- Repositorio de código creado en Github
- Servidor de integración continua configurado con Github Actions
- Al menos un commit realizado por cada integrante, actualizando el README.md del repositorio.

Mientras que para las entregas 1 y 2 la cátedra proporciono una serie de test que el modelo debía cumplir, a los que se debió adicionar otros creados propios.

Para las entregas 3 y 4 se tuvo el modelo del videojuego terminado y la interfaz grafica desarrollada.

A su vez el desarrollo fue incremental, esto es que el codigo se fue mejorando, complejizando y añadiendole más funciones a medida que pasaban las entregas.

Para la escritura del código de la aplicación se utilizaron prácticas agiles tales como TDD (test driven Development), refactorizacion e integracion continua. Para ello primero se implementaron los tests unitarios y se desarrollaron las clases para poder hacer pasar los tests. Luego se refactorizó el código, para aportar mayor legibilidad, mantenibilidad y compatibilidad con los pilares de POO.

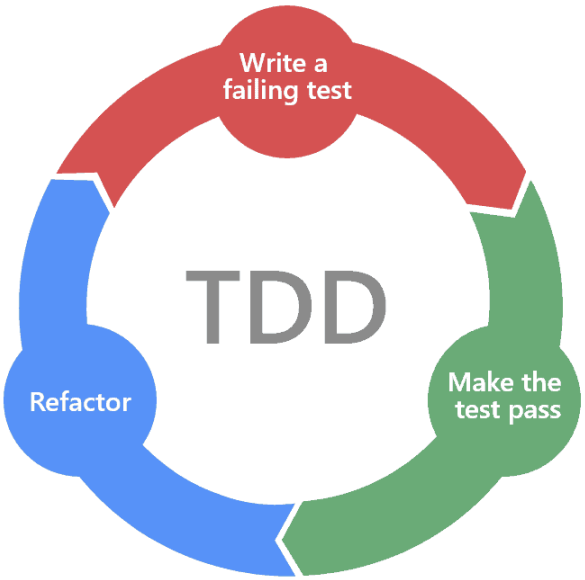

Figura 1. secuencia TDD

Todo el proceso descrito se realiza de forma iterativa. Además, se agregaron los tests por clase.

Para los test unitarios se utilizo el framework de mockito, con el fin de que para las clase que poseían una fuerte dependencia se puedan testear por separado. Por ultimo se realizaron pruebas integrales que prueben que estas dependencias funcionen correctamente.

3.3. Clases utilizadas

Las siguientes clases son las forman parte de la solución de los requerimientos de la aplicación GPS CHALLENGE:

- Juego: Representa al juego en si, posee el listado de jugadores y administra sus turnos, además cuenta con un listado de los jugadores históricos con el puntaje más alto. Interactúa con la interfaz gráfica para mover de posición a los vehículos de los jugadores.

- Jugador: Posee su nickname configurable al inicio de la partida, su puntaje y un Vehículo el cual podrá mover hacia distintas direcciones con el fin de llegar a la meta
- Grilla: Es la representación del mapa de la cuidad, conformada por una matriz de posiciones y un listado de Ubicables (Punto de Partida, Meta, Sorpresas y Obstáculos) con su posición, estos ubicables pueden ser generados de forma aleatorea en una posición también aleatorea. Cada vez que un vehículo se mueve se verifica si "piso.ªlgún ubicable y si esto sucedió se le aplica al vehiculo.
- Vehículo: Cada uno de los jugadores posee uno. Puede cambiar de posición relativa según cada dirección proporcionada.Contiene un cantidad de movimientos acumulados y tiene como atributo a un TipoDeVehiculo y delega en ella como le afectan los distintos tipos de ubicables que se pueden encontrar por la grilla.
- TipoDeVehiculo: Clase abstracta, puede ser tanto una Moto, un Auto o una Cuatro por Cuatro, en alguna de estas clases herederas delega el vehículo su comportamiento cuando se topa con un Obstáculo.
- Moto: Es un tipo de vehículo, cuando el vehículo se tope con Pozo se calculara un penalización de 3 movimientos, para un Control Policial habrá un gran probabilidad de ser atrapado (porque siempre van sin casco) y penalizado con 3 movimientos, es el único capaz de pasar por un piquete, aunque con una penalización de 2 movimientos. El siguiente tipo de vehículo es un Auto.
- Auto: Es un tipo de vehículo, cuando el vehículo se tope con Pozo se calculara un penalización de 3 movimientos, para un Control Policial habrá un probabilidad de ser atrapado y penalizado con 3 movimientos, en el caso de intentar pasar por un piquete se lanzara una excepción avisando que esto no posible.El siguiente tipo de vehículo es una cuatro por cuatro.
- CuatroPorCuatro: Es un tipo de vehículo, cuando el vehículo se tope con Pozo a diferencia de los demás no recibirá una penalización, a menos que haya pasado por 2 piquetes previamente, para un Control Policial habrá un probabilidad de ser atrapado y penalizado con 3 movimientos, en el caso de intentar pasar por un piquete se lanzara una excepción avisando que esto no posible. El siguiente tipo de vehículo es una moto.
- Ubicable: Es una interfaz por lo que no tiene estado (no posee atributos). Las clases que implementen a la interfaz deben implementar serEncotradoPor (unVehiculo).
- Obstaculo: Es una interfaz por lo que no tiene estado (no posee atributos). Son elementos que afectan negativamente a un vehículo
- ControlPolicial: Posee una Posición, puede ser encontrado por un vehículo, tiene una probabilidad de afectar negativamente la cantidad movimientos del vehículo, dependiendo del tipo de vehículo.
- Pozo: Posee una Posición, puede ser encontrado por un vehículo, afecta negativamente a la cantidad de movimientos, a excepción de 4x4 según que condiciones.
- Piquete: Posee una Posición, puede ser encontrado por un vehículo, no permite que el vehículo pase por el arrojando una excepción, a menos que sea una moto.
- Sorpresa: Es una interfaz por lo que no tiene estado (no posee atributos). Son elementos que afectan a un vehículo de forma aleatorea, no se sabe de que tipo es hasta encontrarla.
- SorpresaFavorable: Es una implementación de sorpresa, posee una Posición y el vehiculo que la encuentra vera reducida su cantidad de movimientos en un 20 por ciento.
- SorpresaNoFavorable: Es una implementación de sorpresa, posee una Posición y el vehiculo que la encuentra vera aumentada su cantidad de movimientos en un 25 por ciento.
- CambioDeVehiculo: Es una implementación de sorpresa, posee una Posición y el vehículo que la encuentra cambiara de tipo de vehiculo según su tipo actual.
- Posición: Se trata de una coordenada en X y otra en Y, puede obtener la suma de 2 posiciones y comparar posiciones.
- Dirección: Es una interfaz por lo que no tiene estado (no posee atributos).Las clases que implementan esta interfaz son Derecha, Izquierda, Arriba y Abajo, y estan obligadas a implementar posicionRelativa que devuelve una posicion que sumada a la posicion de un vehiculo se obtendra el desplazamiento del mismo.

3.4. Movimiento de piezas

Para el movimiento de las piezas se decidió que los Vehículos realizarán movimientos de dos posiciones de la Grilla y ésto representa a un movimiento del mismo en el juego. Esto se realizó de esta manera debido a que antes de que el Vehículo llegue a la posición siguiente, tiene que atravesar un elemento ubicable, por ejemplo un obstáculo o sorpresa.

Como al atravesar las posiciones de elementos ubicables, podría suceder el caso en que no tenga ningún elemento, para esto se utilizó el patrón de diseño "Null Object"que permite crear un objeto nulo que represente a esos elementos que no deberían afectar.

Para evitar respetar los principios de encapsulamiento, al momento de realizar comparaciones se realizaron comparaciones entre clases. Con esto se evitó la utilización de getters para comparaciones. Por ejemplo: se utilizó esta estrategia para comparar posiciones y puntajes, para el caso de los puntajes se realizaron las comparaciones entre jugadores ya que éstos tienen como atributo a los puntajes.

3.5. Pilares de POO

Los pilares del paradigma que se emplearon para modelar las clases fueron: abstracción, encapsulamiento, polimorfismo, delegación y herencia.

La abstracción expresa las características esenciales de un objeto, las cuales distinguen al objeto de los demás. Además de distinguirlos, provee límites conceptuales. Este enfoque consiste en aislar un elemento de su contexto o del resto de los elementos que lo acompañan para un estudio específico y considerando solo las propiedades esenciales para dicho análisis.

Encapsulamiento: Es el proceso de almacenar en una misma sección los elementos de una abstracción que constituyen su estructura y su comportamiento; sirve para separar el interfaz contractual de una abstracción y su implantación. Por ejemplo la clase abonoNacional, posee y puede a sus atributos que le permiten calcular el gasto de una llamada cualquiera; una mala implementación seria que una instancia de esta clase, deba para cumplir con esta responsabilidad, consultarle a otro objeto el precio por minuto durante la hora hábil. Otro forma de romper encapsulamiento seria al consultar por los atributos de un objeto, en un método de una instancia de una clase distinta a la que se le realizo la consulta, para evitar esto se pasan los datos por parámetro en el metodo directamente, de forma tal que viendo dentro desde el objeto no se sepa que atributos tiene el otro objeto.
El encapsulamiento es el ocultamiento de la implementación. Cada objeto es responsable de responder a los mensajes que recibe, sin que quien le envía el mensaje tenga que saber cómo lo hace. Las ventajas que conlleva este enfoque es que pueden haber implementaciones alternativas para una misma operación y, por otro lado, se pueden realizar cambios en la implementación sin afectar al cliente que utiliza el servicio.

La delegación es el mecanismo por el cual un objeto delega una funcionalidad a otros objetos para luego tomar una decisión. Un objeto debe conocer al objeto que le va a delegar una funcionalidad. En la delegación, generalmente, se cumple la relación “contiene”, “hace referencia” y “es parte de”. En ocasiones, se aplica como sustitución de la herencia. La delegación es conveniente si se reutiliza la interfaz sin mantenerla.

La herencia es una relación entre clases, por la cual se define que una clase puede ser un caso particular de otra (llamada clase hija). Cuando hay herencia, todas las instancias de la clase hija son también instancias de la clase madre. Esta característica permite reutilizar código dado que al definir un comportamiento en la clase madre y usar ese comportamiento sin tener que repetir el código en cada clase hija. La herencia es conveniente si se reutiliza la interfaz tal cual está. En otras palabras, la herencia permite programar por diferencia. En la herencia, generalmente, se cumple la relación “es un”.

El polimorfismo es la capacidad de respuesta que tienen distintos objetos de responder de maneras diferentes a un mismo mensaje. Esto permite aumentar de la reutilización, hacer el código más legible y ocultar detalles de la implementación al interactuar con un grupo de clases diferentes a través de una interfaz común.

3.6. Patrones utilizados

Singletton Estrategie Double dispach

3.7. Excepciones

3.8. Diagramas UML

3.8.1. Diagramas de Clase

3.8.2. Diagramas de secuencia

3.9. Testeo del código

Se realizaron Test unitarios para cada una de las clases del Trabajo Practico, con el fin de aislar a las clases y probar que las instancias tengan el comportamiento esperado, esto es que cumplan con el contrato.

En el caso del uso no esperado/no deseado de una clase también se realizan pruebas para estos casos y se verifican que se lancen las excepciones correspendientes.

3.9.1. Pruebas de mutación

4. Conclusiones

Se comenzó a trabajar con TDD lo permitió implementar el software realizando incrementos pequeños en la complejidad del código, con el fin de no introducir bugs y además de permitir organizar el trabajo. La refactorización para realizar mejoras del código sin cambiar contrato/funcionalidad resulto fundamental.

5. Bibliografía

”Growing object-oriented software, guided by tests ” - Freeman Pryce - 2011

Programación orientada a objetos y técnicas avanzadas de programación - Carlos Fontela - 2003