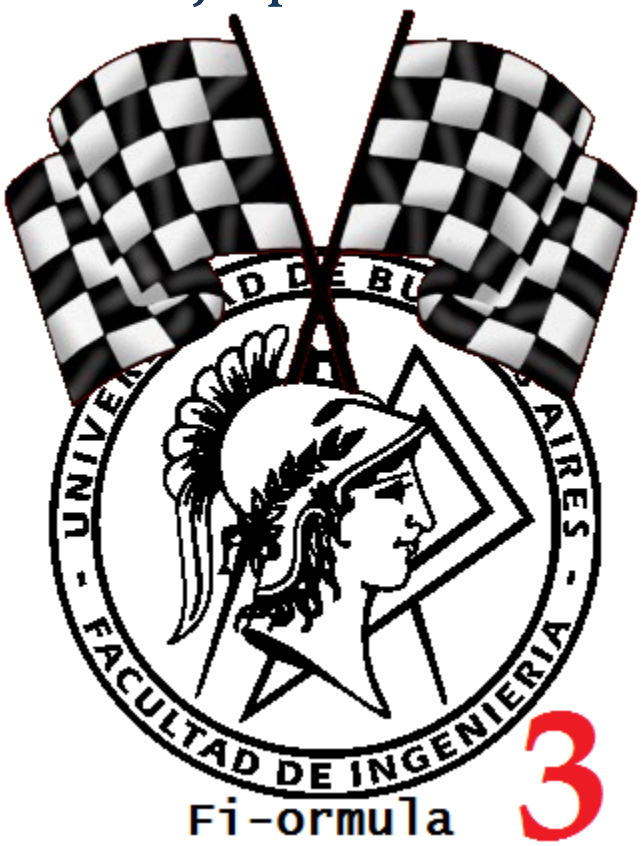
Trabajo práctico n°2



Autores

- Benítez, Demián Agustín; #91142
- Bogado, Sebastián Javier; #91707
- Luchetta, Constantino Amadeo Fabricio; #90967
- Torres, Miguel Ángel; #91396

Contenido

Introducción

La idea de la información aquí presentada es ayudar al corrector a seguir la misma lógica que nosotros seguimos. Nunca pretendió ser perfecta, pero sí la mejor que pudimos.

Se divide en distintos apartados, puntos que consideramos clave, que merecían un desarrollo más allá de lo que hay en *.java.

Los diagramas están en hipervínculos porque algunos eran demasiado grandes como para incluirlos en el documento y que se leyeran bien. Si bien otros entraban, elegimos una forma general para presentarlos.

Supuestos

- Al pasar por los obstáculos a menos de 30km/h, no producen daño.
- Las lomas de burro no ocupan todo el ancho de la pista.
- La discretización de la pista es de cuadraditos de $1m^2$. Entonces, si cada 1km se desgasta un porcentaje (según la guía del TP), podemos asumir que cada 1m se gasta una milésima de ese porcentaje

Cambios entre la primera entrega y la segunda

Para la primera parte habíamos decido que la clase Entorno se encargue de enviar los mensajes para que el auto doble, acelere y frene, es decir, que también sea parte del Control. Además, también generaba el gameloop. Pero luego de analizarlo decimos desechar esa implementación, ya que considerábamos que el Entorno ya conllevaba demasiadas responsabilidades. Y nos parecía mal que el entorno estuviera en las áreas de Modelo y Control.

Si se hubiera conservado esa implementación, luego alguna posible refactorización o cambio de implementación seria mas complicada si el proyecto ya estuviera terminado.

Nos pareció un buena decisión que el entorno no tenga demasiadas responsabilidades ya que logro una simplificación para la misma y para el proyecto en sí. Y reforzó la idea: representa el mundo físico que relaciona al auto y la pista.

Modelo de dominio

La clase Entorno implementa la interfaz ObjetoVivo y reescribe el método “vivir” donde allí se va a realizar las operaciones principales del juego.

Donde las operaciones son las siguientes.

- Actualizar posición del auto: que se asigna al auto su nueva posición en función de la velocidad que tiene en ese instante, es decir, solamente mueve al auto un diferencial de tiempo, ya que entre llamado y llamado hay un intervalo de tiempo muy pequeño.

- Chequear colisiones: se comprueba si la posición del auto coincide con el obstáculo siguiente, si llegaran a coincidir, se le envía un mensaje al auto informándole que deben dañarse los neumáticos.

- Desgaste por Terreno: se le avisa al auto el tipo de terreno en que se encuentra metro a metro, que según el terreno debe haber un determinado aumento en el daño de las ruedas. Aquí se implemento el patrón Double Dispatch.

- Chequear daño del auto: corrobora que el daño del auto sea menor que 100, en caso contrario finaliza el juego informando que se "ha perdido". Se informa al Controlador de Carrera que se ha perdido la carrera.

- Comprobar Fin De Pista: comprueba si se ha llegado al final de la pista, en el caso de que sea así, se informa al Controlador de Carrera que se termina la carrera enviándole el tiempo invertido y la pista donde se corrió.

Todas esas operaciones se realizan en cada iteración del Loop del Juego, que es realizado por el Controlador de Juego.

Diagrama de clases – Relaciones en el modelo

Diagrama de clases – Métodos importantes

Diagramas de secuencia del corazón del modelo

Como se aprecia en los siguientes diagramas, esto sucede en la clase Entorno:

Ciclo central

Actualizar posición

Chequear colisiones:

Si pasó el obstáculo

Si no pasó el obstáculo, y lo choca

Informar colisión

Chequear daño del auto

Informar terreno

Chequear fin de pista (no merece diagrama, es muy simple)

Modelo de servicio

Lo más resaltable del paquete de servicio, son las fábricas. Lo único que se usa desde afuera, es la FabricaDeAutos. Tiene sólo un método estático para instanciar un auto, sin necesidad de crear una instancia de la misma. Sólo necesita dos strings, una para el motor y otro para las ruedas. Si bien suele relacionarse a las strings con poca extensibilidad, para disminuir esto, tenemos una clase, NombresDeFabricables[1], donde está todo nombre que referencie a las clases que implementen Autoparte.

Otra variante de la implementación de las fábricas, es con reflexión: creando los motores desde las strings, cuidándose siempre que sea el nombre exacto de la clase.

Diagrama de clases de las fábricas

Hay también en el paquete, vectores Velocidad, Posición; Colisionador, que se encarga de verificar si un obstáculo y un auto chocan; CuerpoExtenso, que le da las propiedades de largo y ancho a los obstáculos y el auto; y Tiempo, que almacena los récords de las pistas (nombre y tiempo que le tomó al jugador).

Menú

Esta sería la clase de la Vista en donde se encuentra el menú principal del Juego. En donde se seleccionan el tipo de Motor, Neumáticos y la Pista. Desde donde luego se iniciará el juego con lo elegido anteriormente.

El Menú es una clase que hereda de la clase JFrame y que contiene los botones donde se crean las ventanas de selección de Motor, Neumáticos o Pista, y con un botón “Salir” que cierra la aplicación y con otro “Comenzar”, que iniciar la carrera pero se habilita solo cuando se ha elegido todo lo que necesita para que inicie la carrera.

[1] Originalmente, se llamaba NombresDeAutopartes, pero fue modificada por una potencial extensión para fabricar terrenos y obstáculos. Nunca sucedió.

Cada ventana de selección tiene botones con las opciones que se pueden elegir, excepto cuando se selecciona la pista, allí se abre un Browser donde se abrirá la pista que se desee. Naturalmente, heredan de una ventana de selección básica, que luego es "personalizada".

Los botones de selección de autoparte, podrían haber sido generalizados en un listener de selección, como los botones que abren las subventanas del menú. En una futura versión, podría ser así, aunque con un patrón command nos daría mayor extensibilidad.

Cuando se selecciona la opción de comenzar se cargan las instancias de Automóvil (fábrica) y Pista según lo especificado antes, y se envían esas instancias al ControladorDeCarrera, que inicia la VentanaDeInformacion (donde se muestra velocidad, distancia recorrida, tiempo y daño) y la Ventana Principal (donde se vera la pista con el auto).

Serialización de la pista

Todos los elementos de la pista conocen cómo serializarse. Convenimos entonces:

- La pista almacena las listas de pozos, lomas de burro, terrenos y mejores tiempos
- La lista de pozos almacena: cantidad de pozos, y los pozos
- Cada pozo almacena su posición en x e y
- La lista de lomas de burro funciona de manera similar
- La lista de terrenos almacena la cantidad de terrenos, y los terrenos
- Los terrenos almacenan su principio y fin
- Los mejores tiempos almacenan el nombre del jugador y el tiempo en que logró la pista

Partiendo de esas premisas, implementamos utilizando JDOM.

Excepciones

- **ExcepcionLimitesIncorrectosEnElTerreno**: esta excepción salta cuando el principio del terreno es más grande que el final. Se captura y se informa por pantalla desde JOptionPane.showMessageDialog(…).
- **ExcepcionPistaFinalizada:** la lanza Entorno.chequearFinDePista() cuando termina la pista. Es obligatorio atraparla, y la atrapa el mismo Entorno, para luego llamar a ControladorDeCarrera.terminarCarrera.

Podríamos haber agregado excepciones para las fábricas, que informaran que lo que le pedimos no existe. En principio, sería imposible que sucediera por la forma en que está hecho, pero si agregamos una clase motor Solaris y nos olvidamos de agregarlo en la fábrica, nos enteraríamos que estamos trabajando con algo nulo recién al empezar la carrera.

Traspaso de la lógica a interfaz gráfica

Para poder realizar la interfaz grafica con el Titiritero tuvimos que hacer reducir nuestro grado de libertad de decisión de cómo implementarla y ajustarnos al modo que esta diseñado el Titiritero.

Tuvimos que hacer que la clase Entorno implemente la interfaz ObjetoVivo , que dispone del método vivir, que allí se van a ir realizando todas las operaciones de actualizaciones de posición y verificaciones de colisiones.

Para los objetos que serán dibujados en la pantalla, creamos dos clases VistaAutomovil y VistaPista, una para que represente la imagen del auto, y la otra valla mostrando toda la pista. Estas dos clases implementan la interfaz Dibujable de Titiritero. Dentro de la VistaPista se encuentran los obstáculos que deben ser dibujados y terrenos a mostrar, ya que ella sabe cuando y donde debe dibujarlos, pues que si se pintaran en la Ventana por separado la relación entre ellas (las clases) sería más complicada y su implementación sería algo más dificultosa para que se coordinen correctamente.

Uno de los problemas que se encontró fue, que el modelo y la vista tenían por así decirlo dos “sistemas de coordenadas” diferentes, uno era referido a la parte inferior izquierda de la pista (Modelo) y el otro era respecto al auto (Vista), y a eso se le sumo un poco de complejidad por el sistema de coordenadas que ya llevan consigo las Ventanas. Para solucionar este problema creamos la clase CambioDeCoordenadas que se encarga de hacer este trabajo, que según la posición que tenga el obstáculo en el modelo la transforma en una posición de la ventana, en la que se dibujara la misma.

Suponemos que si no hubiésemos contado con el Titiritero tal vez nuestro grado de libertad para elegir de qué forma implementar la vista seria más amplio, pero ello conllevaría hacer un nuevo Diseño para ello y con pruebas, en cambio ya con el Titiritero disponible esa etapa no nos incumbiría. Entonces se trataría de una relación costo-beneficio, de alguna forma se nos reduciría trabajo en un sentido pero nos restaria grado de libertad para las decisiones de implementación.

Usuario vs el modelo

Los mensajes recibidos por el Modelo desde el Control, son realizados por la clase OyenteTeclado. Esta tiene la funcionalidad de hacer todo lo concerniente a datos de entrada a través del teclado.

Se encarga de decirle al auto si tiene que doblar, frenar, acelerar y también si se debe pausar el juego. Ya que se tiene una referencia al Automóvil y al ControladorJuego.

La clase OyenteTeclado implementa la interfaz KeyListener, donde se reescribe el método keyPressed, que en el momento que se realiza un evento, cuando alguna tecla fue presionada se llama al mismo y se comprueba si la que se presiono tiene alguna funcionalidad y luego se realiza la tarea correspondiente respecto a ello.

Creemos que esta fue una de las soluciones más óptimas que pudimos diseñar, por su simplicidad y teniendo una fácil implementación. Y hasta el momento no se nos ocurrió otra que tenga mayores beneficios.

Patrones de diseño

El principal patrón aplicado en el TP (era uno de los objetivos), fue el **MVC**. Hicimos un esfuerzo enorme para que nos quedaran bien discriminadas las partes en distintos paquetes.

- control: están prácticamente todo aquello que implementa ActionListener. Además, hay unas clases que controlan el desarrollo del juego
- modelo: ahí está nuestro modelo de dominio entero. Hay una clase importante, modelo.Entorno, que representa la relación entre la pista y el auto. Le da un sentido físico al juego. Además, está el paquete modelo.servicio, donde se alocan las clases que solo nos interesan por su funcionalidad.
- vista: paquete compuesto por las ventanas, y por la representación gráfica de cada elemento del dominio (auto, loma de burro, etc.)

También tenemos el paquete titiritero, que tiene una mezcla de control y vista.

Otro patrón aplicado fue el **double dispatch** al momento de chocar con un obstáculo, dado que el desgaste producido a los neumáticos varía dependiendo del tipo de ruedas y del obstáculo. Básicamente, cuando la rueda se entera que chocó, le informa al obstáculo que está siendo chocado, y éste contesta diciéndole a la rueda con quién está chocando. Más formal está mostrado en un diagrama de secuencia.

El último patrón implementado conscientemente fue el de las fábricas. Son dos, en realidad: **abstract factory** y **factory**, pero porque la primera se apoya en la segunda. Las usamos para la creación de los autos. Podría haber sido de instancia única (singleton), pero preferimos la idea de que, para crear al auto, no se necesitara más que un método. Conseguimos un auto recién salido del horno con sólo hacer:

Automovil auto = FabricaDeAutos.instanciarAuto(metadataAuto);

Las subfábricas abstractas (de motores y ruedas) tienen un mapa que guarda la clave, y la referencia a una fábrica concreta. Una forma más fácilmente extensible de hacer esto podría haber sido con reflexión, y nos ahorrábamos un par de fábricas.

Indirectamente, usamos patrones similares al **observer** (desde el MVC)**, command** (desde el menú, donde todos implementan ActionListener) e **iterator** (para iterar sobre las listas de terrenos, obstáculo, etc., en la pista), pero no merecen mayor mención.

Build

El build del Ant incluido tiene los siguientes comandos:

- ant limpiar: borra lo compilado
- ant compilar: limpia, y recompila el programa
- ant junit: compila el programa y corre los tests
- ant correr: corre el jar. Falla si no hay jar
- ant correrlimpio: recompila el programa y corre el jar
- ant generarentrega: genera un zip con todo lo que hay en el proyecto

La herramienta que nos presentaron fomenta la automatización de los proyectos. Sin embargo, no pudimos hacer funcionar el junit. Tenía problemas para encontrar las clases.

Además, el .jar generado sólo funcionaba si lo corríamos desde el Ant. Manualmente, desde la carpeta, no funcionaba.

En fin, estuvimos ejecutando correrlimpio, o correr, si no hubo cambios.

Extras

Hubo dos cosas que queremos comentar, pero que no entran en las otras categorías.

La primera: en el menú, en las ventanas de selección de motores y ruedas, al pasar el mouse por un botón sale un ToolTip Text con la descripción del autoparte. Para que esos simples botones no tuvieran que instanciar, por ejemplo, un Paler9 (obligadamente llamando a las fábricas), hicimos uso de la reflexión. Como cada botón está directamente asociado al nombre de la clase, con esa herramienta conseguimos las descripciones.

La segunda: un bug *intrackeable* producido al iniciar la carrera desde el menú. Nos llevó mucho tiempo, y no tiene nada que ver con el diseño, pero le encontramos la vuelta igual.

En el momento que ya se llamaba al inicio de la carrera desde el menú principal la Ventana de la Pista se tornaba completamente blanca, pero seguía con su funcionamiento aunque sin ninguna imagen mostrándose.

Pensamos que esto era un error del IDE o incluso del JDK, ya que aun todavía desconocemos la fuente o el motivo del error, pero sí sabemos en dónde es que se produce. El inconveniente era cuando se iniciaba la carrera mediante alguno método de algún ActionListener o en alguno que iniciara la carrera, y que el método mismo sea llamado del ActionListener, es decir el problema se presentaba cuando se iniciaba la carrera en algún método que respondía a la acción de algún evento.

La solución que planteamos fue que el inicio de la carrera no dependa únicamente del llamado en un ActionListener, para ello hicimos que la carrera se corriera en un Thread diferente al del menú. Cuando se instanciara el menú también se crearía el Thread, que se iniciara también en ese momento su ejecución con la salvedad de que entraría en bucle que lo mantiene dormido durante 1000 milisegundos mientras se mantenga la condición de carreraInciada en falso.

Luego de que se seleccionan todas la autopartes y la pista, se cargan al Thread y se asigna el valor verdadero a la variable de control del bucle. Comenzando así correctamente la carrera.

Es decir que la solución que hicimos fue que la carrera se “Pre-cargue” cuando se inicia el Menú.

Aun seguimos sin conocer el porqué del error que se nos interpuso pero sabemos que se encuentra en los Listener, ya que lo comprobamos haciendo diferentes tipos de pruebas llegando a la conclusión de que allí se encuentra la causa del error, aunque desconozcamos su motivo.