FIUBA - 7507

Algoritmos y programación 3

Trabajo práctico 2: Algo42 Full

Informe Parcial

1er cuatrimestre, 2011
(trabajo grupal)

Nombre	Padrón	E-Mail
Awad, Lucas Javier	92277	lucasawad@gmail.com
Martínez, Gaston Alberto	91383	gaston.martinez.90@live.com.ar
Rossi, Federico Martín	92086	federicomrossi@gmail.com

Índice

Introducción

Objetivo del trabajo

Aplicar los conceptos enseñados en la materia a la resolución de un problema, trabajando en grupo utilizando Java o C#.

Consigna

Desarrollar la aplicación completa, incluyendo el modelo de clases, la interfaz con el usuario, con sus pruebas correspondientes, de un juego cuyas reglas se muestran a continuación, y se pedirá pruebas unitarias y de integración completas, y elementos de documentación que se detallan en este enunciado.

Introducción

La empresa Rezagos Militares Software Inc., creadora de grandes éxitos en el campo de juegos de video, está encarando un proyecto para realizar un simulador de batalla de aviones para ser utilizado como software de entrenamiento de la Fuerza Aérea Argentina y ha elegido a su grupo de trabajo para realizar el diseño y desarrollo de su nueva e innovadora idea.

El juego consiste en un avión de combate que debe combatir naves enemigas.

Los escenarios deben tener 2 componentes principales: el fondo y los elementos móviles, como aviones enemigos, armas enemigas, objetos especiales, el avión del jugador y las armas del jugador.

A continuación se describe el enunciado general con las características funcionales de la aplicación a desarrollar:

> Corre el año 2042 y nuestro país debe defenderse de una invasión extranjera que busca el control de las fuentes de agua potable de nuestras provincias. Nuestra flota aérea consta de 2 aviones, uno de los cuales no funciona por falta de mantenimiento. La flota extranjera es muy poderosa y está compuesta de miles de aviones que comienzan a sobrevolar nuestro territorio y amenazan con controlarlo por completo. Pero aun queda una esperanza si nuestro único avión (cuyo nombre clave es "Algo42") pudiera llegar hasta el porta-aviones enemigo y arrojarse en picada sobre él, destruyendo el cuartel de control de los invasores.
>
> Para cumplir con su cometido, el Algo42 deberá cumplir una serie de misiones. En cada misión se enfrentara a una flota distinta de aviones invasores. Las flotas de los enemigos están conformadas por distintos tipos de aviones.
>
> La cantidad de aviones de las flotas es variable (mínimo 15 aviones). Cada flota cuenta con un avión Guía que coordina al resto de los aviones de la flota. En caso de destruirse el avión Guía los demás aviones detienen sus disparos instantáneamente y huyen del campo de

batalla.

Una implementación de una empresa competidora puede verse en el siguiente link, y sirve para entender la dinámica del juego:

http://www.youtube.com/watch?v=xQIB-O0DZm4

Los enemigos cuentan con los siguientes modelos de naves:

Nombre	**Armas**	**Estrategia de vuelo**	**Observaciones**	**Puntos por destrucción**
Avionetas	Lasers	Idas y vueltas en línea recta	Son los aviones más rápidos.	20
Bombarderos	Lasers, cohetes y torpedos rastreadores	Zig/Zag	Son los más poderosos pero al mismo tiempo los más lentos. Al ser destruidos, el Algo42 puede tomar sus armas.	30
Exploradores	No tiene	En círculos.	Vuelan en círculos amplios recorriendo toda la superficie aérea, en búsqueda de chocar al Algo42.	50
Cazas	Torpedos simples	En grupo formando una V	Al ser destruido su tanque de energía pude ser tomado por Algo42.	30
Cazas (Nuevo)	Torpedos adaptables	En grupo formando una V	Al ser destruido su tanque de energía pude ser tomado por Algo42.	Quita la mitad de los puntos al Algo42.

Por su parte el Algo42 es un avión escalable. En la versión base solo cuenta con lasers, pero puede escalar aumentado su poderío apropiándose de las armas y energía de los aviones que destruye.

Consideraciones generales:

- Todo avión tiene una fuente de energía, la cual disminuye a medida que es atacado. Cuando dicha energía llega a cero el avión es destruido.
- El Algo42 va sumando puntos para su misión a medida que destruye aviones enemigos. Al llegar a 1000 puntos termina el nivel y pasa al siguiente.
- Los lasers no se gastan, pero los torpedos y cohetes sí.
- El espacio no esta vacío, además de las flotas enemigas hay aviones civiles (pasan en línea recta a poca velocidad, el Algo42 debe evitar destruirlos ó chocarlos, caso contrario pierde 300 puntos por cada avión civil destruido) y helicópteros de la policía federal (se mueven en círculos pero tienen orden de no disparar, también debe evitarse su destrucción o se pierden 200 puntos por cada helicóptero).

Entregables

Se deberá desarrollar la aplicación completa, incluyendo la interfaz gráfica. Deberá poder grabarse los puntajes altos (los n mas altos, configurable) y grabar el estado del juego para retomarlo nuevamente en otro momento.

Deberá entregarse:

- todas las clases con sus métodos , organizados en paquetes/namespaces según criterio del alumno.
- conjunto de pruebas unitarias que muestren el uso del modelo y su correcto funcionamiento.
- documentación completa del código fuente
- documentación completa del diseño de clases (ver siguientes secciones del enunciado)
- la aplicación deberá poder correrse desde consola, para lo cual deberá proveerse el archivo de Ant o NAnt correspondiente.

Forma de entrega

Este documento se deberá completar con las secciones correspondientes. Deberá acordarse con su ayudante la forma de entrega de los elementos que se evaluarán.

Fechas de entrega

Entrega 1 (semana del 24 de mayo de 2011): se deberá entregar la primer versión de la documentación con todos correspondientes y el modelo de datos con sus pruebas unitarias.

Entrega final (semana del 21 de junio de 2011): se deberá entregar la aplicación completa junto con la documentación revisada y corregida según los comentarios realizados por el ayudante asignado.

Pruebas

Todas las clases deberán contar con sus pruebas unitarias **COMPLETAS**. La aplicación deberá contar con pruebas de integración **COMPLETAS**.

Documentación

Supuestos

A continuación se enuncian todos los supuestos que completarán las especificaciónes en la forma de funcionamiento y desempeño del software como producto final:

- Suponemos que todos los objetos móviles del escenario tienen vida propia a lo largo del tiempo (inclusive el Algo42). En el caso de la nave del jugador, el mismo le va a poder dar órdenes en cierto momento, lo cual va a cambiar su comportamiento desde ese momento, pero eso no significa que no tenga un comportamiento definido en el tiempo.
- Es el escenario el encargado de que todos los objetos actúen en sincronía, diciéndoles que actúen.
- Cada objeto tiene forma circular, caracterizada por un radio, que llamaremos tamaño.
- Cuando en el enunciado se habla de arma, la suponemos como tal (no como un proyectil), y sólo ella puede disparar (una nave en sí no sabe disparar, sino operar el arma).
- Respecto a los vuelos circulares, los suponemos como que constan de una entrada en línea recta, luego un movimiento circular, y finalmente otra salida también en línea recta.
- Al vuelo en V de los cazas lo suponemos simplemente configurado al principio en formación, es decir que cada uno no tiene noción de cómo se mueve el otro.
- Cuando la nave guía es destruída, las naves huirán en línea recta hacia arriba, entendiendo por arriba a las “y” positivas.
- En ese momento en que la nave guía es destruída y las demás huyen, el juego debe encargarse de crear una nueva flota que entre al campo de batalla, hasta que el jugador logre acumular los 1000 puntos necesarios para pasar de nivel.
- Vamos a considerar que existen dos equipos: uno es el Algo42 y el otro las demás naves. Las armas y los proyectiles tendrán asignados los equipos correspondientes a la nave a la que pertenezcan. Consideramos que si dos objetos del mismo equipo chocan, no les pasa nada y se atraviesan sin afectarse.
- Los objetos pueden atravesarse entre sí sin cambiar su trayectoria, y cuando se encuentren superpuestos, se afectarán entre sí de la manera que corresponda (recibir daño, destruirse, etc.). Además, cada objeto no sabe qué le ocurrirá a otro al chocar, sólo conoce cómo se verá afectado él mismo.
- Vamos a suponer a las flotas como que tienen como nave guía a cualquier nave militar. Va a ser responsabilidad de dicha nave el saber qué hacer y cómo comunicarse con la flota, pero no es necesario que lo sepa (en dicho caso, se supone que los demás miembros de la flota no van a recibir órdenes del guía). En el caso de las flotas enemigas, estas sí tendrán una nave guía que mande a todos a retirarse cuando muera.

Elección del modelo entre los disponibles del TP1

Si bien los tres modelos del TP1 comprendian el modelo del dominio en su totalidad, se decidió optar por aquel que poseia una visión más amplia del dominio del problema. Con esto nos referimos a elegir uno que no necesitara demasiados ajustes de diseño para poder adaptarlo a un patrón MVC (el cual es la estructura central de como estará comprendida la división de responsabilidades en el desarrollo del juego).

De todas maneras, se utilizaron conceptos de diseño de los demás trabajos para poder potenciar aún mas el diseño del modelo central, permitiendo una estructura mas robusta y de mucha mas maniobrabilidad a la hora de ser utilizada.

Particularidades encontradas al transformar el modelo del TP1

En el proceso de migración del código implementado en Smalltalk a Java surgieron ciertas particularidades, que a pesar de no haber presentado dificultades, merecen ser resaltadas.

La primera particularidad, y la más obvia, es que Java es un lenguaje *tipado*, cosa que en Smalltalk no ocurría. Por lo tanto, se han tenido que definir los tipos de los atributos, de los parámetros que reciben los métodos de las clases y de los valores que estos devuelven. De esta manera, queda explícito en el código (en tiempo de compilación) el comportamiento que se pretende que tenga cada instancia, su polimorfismo y el grado de generalidad que puede adoptar en partes más integradoras del sistema (por ejemplo, cuando vemos al algo42 como un simple ObjetoEspacial actuando en el Escenario junto a los demás objetos). Esto último también nos fue útil para ocultar aún más la implementación interna de ciertos objetos (como por ejemplo colecciones), y separar completamente del exterior posibles cambios en su diseño.

Otro de los constrastes que se denotan al trabajar en un lenguaje interpretado a diferencia de uno compilado, como es Java, es la forma en la que se debugguea. Si en Smalltalk alguna clase no funcionaba como debiera en las unit test, existía la posibilidad de emular el código a mano en el workspace., de manera de encontrar la fuente del error. En cambio, en Java, dado que la compilación se produce en un momento completamente separado del tiempo de ejecución, no se pueden modificar los objetos sobre la marcha ni utilizar manualmente los objetos como en el workspace de Smalltalk. Esto último sumo importancia al planteo de las pruebas unitarias y al uso de herramientas como el *debugger*.

Además, se ha tenido que hacer uso de *generics* para definir los tipos soportados por las colecciones de datos, lo que hizo mas sencillo el uso de los objetos almacenados ya que nos asegura de que tipo es el contenido de esta, y a diferencia de Smalltalk, evitamos problemas en tiempo de ejecución. Por el contrario, al intentar ingresar algún objeto de tipo inválido el compilador nos avisa, siendo más facil detectar la fuente de error.

Por último, nos encontramos con el uso de *interfaces*, lo cual nos evitó estar atados a un tipo particular en ciertos casos, tales como en el uso de listas. Esto nos aportó una gran ventaja que es el permitirnos realizar futuras modificaciones de implementación. En el caso de las listas (interfaz *List*), por ejemplo, nos permitió cambiar el tipo de una colección que implementa esa interfaz por otra que también la implementa, reduciendo así el número de modificaciones a realizar en un posible cambio de diseño.

Modelo de dominio

Para el desarrollo del juego, como se mencionó anteriormente, se optó por aplicar el patrón MVC para llevar a cabo la solución del problema. Es por esto que se decidió crear tres paquetes distintos: *modelo*, *control* y *vista*. Además se le sumó a estos tres un último paquete, *persistencia*, el cual va a contener todo el manejo de persistencia del juego.

En este informe parcial nos centraremos en el *modelo*, que es la motor central de toda la aplicación ya que será el encargado de los procesamientos de datos los cuales serán utilizados por las otras partes que componen el software. Como el modelo ya se encontraba implementado en otro lenguaje, se comenzó por crear las distintas clases con sus respectivos métodos y atributos acompañadas todas por la debida documentación. Luego se migraron las pruebas para así tener una forma de testeo a lo largo de la migración del código de las distintas clases. Cada una de estas tiene como finalidad focalizarse en las funcionalidades más fuertes de cada clase, de manera de asegurar el correcto funcionamiento a la hora de la integración parcial o total.

La razón de haber mantenido esta forma de trabajo es lograr un diseño y desarrollo guiado por pruebas (TDD) de manera de optimizar el proceso de implementación y de ajustes.

Poniendo mas énfasis en como se logró modelar el dominio, viendo los supuestos, es evidente que fue necesario diseñar un comportamiento polimórfico para los objetos del juego que viven e interactuan en el escenario, de manera tal que cada uno actúe de una manera particular y propia ante un mensaje global enviado por el escenario.

A razón de que cada objeto tiene una posición determinada y una manera particular de moverse es que creamos las clases *ObjetoEspacial* y *Movil*, que definen un estado y comportamiento en común entre los objetos. Además, el resto de los objetos, salvo las armas (que están siempre dentro de naves o como parte de bonos), tienen una trayectoria de vuelo en el escenario por lo que los pudimos abstraer en una clase *ObjetoVolador* que tenga una referencia a un objeto de una clase *Vuelo* que defina la trayectoria de los objetos.

A partir de estas cuatro clases abstractas logramos definir cada uno de las entidades manteniendo una generalidad en el estado de los mismos. Se definieron también otras clases que se encarguen de crear los objetos con el estado correspondiente descripto anteriormente en el enunciado.

En cuanto a los choques, se definieron clases de comportamiento que serán explicadas con mayor detalle en el apartado que sigue.

Finalmente, para lograr un acoplamiento entre el modelo y las otras partes del patrón, se creó una clase *Partida*, la cual tiene la responsabilidad de crear una partida para los *jugadores* como así también la de permitir cargar una partida guardada anteriormente. Esta clase se relaciona directamente con una clase *Misión.* Esta última representa cada nivel de juego y es la encargada de crear el panorama de juego en el que interactuará el jugador.

Detalles de implementación

Para lograr un buen diseño se trabajo arduamente en reducir las responsabilidades de las clases como así también de sus métodos. Como consecuencia se obtuvieron, en general, métodos con implementaciones sencillas, logrando que cada una se encargara de realizar una acción básica en particular.

En este apartado decidimos limitarnos a profundizar sobre los *choques*, el punto por ahi menos claro de todo el modelo pero uno de los más importantes en el juego con respecto a funcionalidad.

Los choques estan diseñados con un comportamiento polimórfico, el cual puede resumirse de la siguiente manera. La clase más abstracta, *ObjetoEspacial,* define una clase con comportamiento por defecto (que no ocurra nada al chocar). Luego, cuando un objeto se ve afectado de manera diferente a los demás, se crea una nueva clase de comportamiento ante el choque, y cada vez que un objeto afecta de manera diferente a otro, se define un método a todos los comportamientos (que pueden heredarse entre sí). Finalmente, al chocar, la nave que afecta pide el comportamiento de la afectada y llama al método correspondiente con que la afecta. El choque es recíproco entre ambas partes.

Excepciones

Las excepciones creadas se realizaron con el fin de poder detectar de forma fácil y rápida el lugar en donde se ha provocado un error, o sea, una acción o situación inesperada. Para lograr esto, se crearon excepciones relacionadas a las clases que han de requerirlas de manera de que al obtener un error podamos captar de forma facil el lugar en donde se ha lanzado la excepción.

Cada paquete central del proyecto, tal como *modelo*, tendrá dentro un paquete *excepciones* donde se albergarán las distintas excepciones que componen ese paquete central y que, obviamente, serán utilizadas por las clases de ese mismo paquete.

Diagramas UML

Para una mejor visión y un entendimiento más profundo de como se representó el modelo del dominio se utilizaron diagramas UML. Todos estos se encuentran en el archivo *TP2 – Diagramas UML.asta* que acompaña al presente informe.

Checklist de corrección

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?

Diagramas de estado

•¿Está completo?

•¿Está bien utilizada la notación?

Código

Generalidades

•¿Respeta estándares de codificación?

•¿Está correctamente documentado?