**FIUBA - 75.07**

**Algoritmos y programación III**

*Trabajo práctico 2: AlgoCraft*

1er cuatrimestre, 2015

(trabajo grupal)

Alumnos:

|  |  |  |
| --- | --- | --- |
| **Nombre** | **Padrón** | **PuertoEstelarTerranMail** |
|  |  |  |
|  |  |  |
|  |  |  |

***Fecha de entrega final***: Miércoles 24/06/2015 - Jueves 25/06/2015

***Tutor***:

***Nota Final***:

# 

**Informe**

**Supuestos**

A la hora de realizar el Trabajo Práctico, se establecieron ciertos supuestos respecto a las reglas del juego AlgoCraft. En principio, se determinó que cada jugador tendría asociada una única Base, la cual se encargaría de administrar sus recursos. A su vez, para que uno de ellos quede eliminado de la partida, su base tendría que ser destruida. También se fijó que cuando a un jugador le destruyen algún centro recolector de suministros, ya sea un Centro de Mineral o una Refinería, el mismo no perderá los recursos que hayan sido recolectados por dicho edificio. Otra idea implementada fue que todas las unidades tuvieran el mismo rango de movimiento y que cada jugador tuviese un rango alrededor de la base, dentro del cual puede construir sus edificios; dicho rango irá aumentando por cada edificio que construya. El último supuesto adoptado es que el avance de turno de todos los jugadores se produce cuando termina una ronda, es decir después de que todos hayan realizado su acción.

## Modelo de dominio

Para poder determinar las entidades que pasarían a formar parte del modelo generado se observó el funcionamiento del StarCraft original. En este, cuando se hace un click sobre una unidad o un edificio de un jugador determinado, podemos ver todos sus atributos, siendo algunos de estos la vida, el jugador al que pertenece, su alcance y el daño de ataque (si es una unidad). A su vez, el jugador puede conocer los recursos que recolectó y las unidades que le pertenecen. Es por esto que se decidió modelar las clases Jugador, Edificio, Unidad, Recurso, Mapa y Juego, siendo estas las clases principales.

La clase Jugador tiene como atributos su nombre (un String), una lista con sus edificios (objetos de la clase Edificio), otra con sus unidades (objetos de la clase Unidad) y su base (objeto de la clase Base). Esta clase tiene métodos que le permiten construir los distintos tipos de unidades y edificios, así como otros que le permiten saber si el mismo ha sido derrotado. El almacenamiento de los recursos y la verificación de los mismos se encuentran delegados a la clase Base (que hereda de Edificio), la cual también se encarga de comprobar la población actual y máxima del Jugador.

La clase Edificio se encarga de generalizar el comportamiento y la estructura básica de los distintos edificios, teniendo como principales atributos su vida (objeto de la clase vida), su jugador y los turnos que tarda en construirse (un int). De esta clase heredan los demás tipos de edificios, siendo estos EdificioConstructor (clase que se encarga de producir las unidades), EdificioRecolector (objetos que recolectan los recursos), DepositoDeSuministros (permiten aumentar la población máxima de un jugador) y la mencionada Base.

La clase Recurso es aquella que representa los dos tipos de recursos que aparecen en el juego, siendo estos los simbolizados con las clases Refineria y Mineral. Su principal atributo es la cantidad de unidades de recurso restantes (un int).

Todas las clases antes mencionadas implementan la interfaz Atacable, la cual hereda de otra interfaz denominada Posicionable (salvo la clase Recurso que solo implementa Posicionable); estas interfaces le permiten poder ubicarse en el mapa, avanzar un turno (métodos definidos en Posicinable) y poder recibir daño (gracias a Atacable).

El Mapa tiene como atributo principal un hash de Posicionables, asociados a un objeto de la clase Posicion, que permite representa la posición en la que se encuentra un objeto (esta tiene como atributos x, y y si es o no terrestre). Dicha clase es utilizada por todas las demás, ya que todas las clases mencionadas, salvo Jugador, son ubicadas en el mapa.

Finalmente se encuentra la clase Juego, que es la que se encarga del inicio del juego, del manejo de los turnos y de hacer que todos los elementos que se encuentra ubicados en el mapa avancen su turno. Esta clase posee como principal atributo una lista con los Jugadores. También se encarga de definir cuando hay un ganador.

**Diagramas de clases**

Se encuentran adjuntos al Informe.

**Diagramas de secuencia**

Se encuentran adjuntos al Informe.

**Diagramas de estado**

Se encuentran adjuntos al Informe.

**Detalles de implementación**

Un detalle de implementación importante es que la clase Mapa fue implementada utilizando el patrón de diseño Singleton ya que todas las entidades del programa necesitan acceder al mismo (por ejemplo, cuando se crea una unidad, la misma es agregada al mapa en el Constructor por lo que se necesita tener una referencia al mapa). Se implementaron los métodos reiniciarInstanciaParaTest y limpiarMapa con el objetivo de poder borrar los elementos del mismo; el primero es invocado al comienzo de los tests que utilizan el mapa para poder garantizar que el mismo se encuentra vacío, mientras que el segundo se lo usa cuando se desea comenzar una nueva partida.

Otro detalle para destacar es la utilización del patrón de diseño Abstract Factory en la clase Jugador, para poder crear los disntintos edificios y unidades sin repetir código. Fue necesario crear dos Abtracts Factories, una para los edificios, llamada y otra para las unidades (ya que la forma de crearlas dentro de la clase Jugador es diferente), las cuales se llaman CreadorDeEdificios y CreadorDeUnidades respectivamente; de estas clases heredan las Factories que crean las diferentes unidades y edificios (como se puede observar en los diagramas de clase).

A la hora de programar la interfaz gráfica, se creó una clase madre vista de la cual heredan todas las demás vistas, habiendo una para cada clase de objeto que tenga que aparecer en la interfaz.

Excepciones

Todas las excepciones creadas para este Trabajo Práctico heredan de una clase madre AlgoCraftException, la cual a su vez es una subclase de RuntimeException. Se decidió que heredasen de esta última para evitar así los checkeos estáticos de las distintas excepciones, obteniendo así un código más legible.

Las excepciones pueden ser catalogadas según los paquetes en las que se encuentran:

**Excepciones de Edificios:**

ElEdificioEstaEnConstruccion: es lanzada cuando se trata de crear una nueva unidad con un edificio que se está construyendo.

**Excepciones del Juego/Jugador:**

ConstruccionFueraDeRangoException: es lanzada cuando se quiere construir un edificio fuera del rango permitido que tiene un jugador.

ElEdificioPerteneceAOtroJugador: se lanza cuando un jugador trata de utilizar un edificio que no le pertenece para construir una unidad.

ElJugadorNoEstaActivoException: se utiliza cuando se trata de hacer algo en un turno con un jugador que no se encuentra activo (no es su turno).

NombreYaExistenteException: utilizada cuando se trata de ingresar un nombre repetido cuando se inicia el juego.

NoSePuedeConstruirElEdificioException: lanzada cuando se trata de construir una Fábrica sin tener una Barraca, es decir, cuando no se cumplen las dependencias.

NoSePuedeConstrurLaUnidadPorSobrepoblacion: se lanza cuando se trata de construir una unidad y al construirla se supera la población máxima del jugador.

NoSeTienenLosRecursosSuficientes: esta excepción se usa para impedir que un jugador cree un edificio o unidad cuando no tiene los recursos para hacerlo.

**Excepciones de Mapa:**

PosicionInvalidaException: utilizada cuando se ingresa una posición con coordenadas invalidas.

PosicionOcupadaException: utilizada cuando se trata de agregar un elemento al mapa en una posición que ya se encuentra ocupada.

PosicionVaciaException: lanzada cuando se trata de obtener del mapa una unidad pasando por parámetro una posición que se encuentra vacía.

**Excepciones de Unidades:**

LaUnidadPerteneceAUnJugadorInactivo: es utilizada cuando un jugador trata de usar una unidad que no le pertenece; como el jugador a la cual le pertenece dicha unidad está inactivo (ya que no es el que está jugando el turno) se lanza dicha excepción.

LaUnidadYaSeMovioEnEsteTurno: lanzada cuando se trata de mover una unidad más de una vez en un mismo turno.

NoPuedeAtacar: lanzada cuando se trata de atacar más de una vez en un mismo turno.

RangoDeMovimientoInvalido: es utilizada cuando se trata de mover una unidad más del rango determinado por la misma.

# 

# **Checklist de corrección**

Esta sección es para uso exclusivo de los docentes, 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?