FACULTAD DE INGENIERIA
Universidad de Buenos Aires

Trabajo Práctico 2 — Java

[7507/9502] Algoritmos y Programación III
Curso 1
Segundo cuatrimestre de 2018

Alumnos:	Allo, Tomás
	Ponce, Antonella
	Cugliari, Pablo
	González, Andrés
Números de padrón:	101057
	100635
	100703
	95694
Emails:	tomasmanuelallo@gmail.com
	antonellaponce28@gmail.com
	pacugliari@hotmail.com
	andresgonzalez.fba@gmail.com

Índice

1. Introducción

El presente informe reune la documentación de la solución del segundo trabajo práctico de la materia Algoritmos y Programación III que consiste en desarrollar una aplicación de manera grupal utilizando los conceptos del paradigma de la orientación a objetos vistos hasta ahora en el curso y las téctinas de TDD e Integración Continua en el lenguaje Java.

2. Supuestos

Para este trabajo práctico se tuvo en cuenta que las posiciones del mapa sólo pueden ser ocupadas por una sola entidad. Es decir, en cada posición puede haber una única unidad o una parte de un edificio. Por lo tanto, ningún elemento del juego comparte posición.

Además, la incorporación de una unidad sólo depende de que el jugador involucrado tenga la cantidad de oro suficiente, no es necesario esperar un determinado tiempo ya que se crea de inmediato.

Teniendo en cuenta funcionalidades, al iniciar el juego los castillos de ambos jugadores se ubicarán en extremos opuestos del mapa, junto con los aldeanos correspondientes al lado del mismo y la plaza central.

Por último, se consideró que los aldeanos sólo pueden construir un edificio en caso de que la posición donde se lo desee crear sea aledaña a él.

3. Diagramas de clases

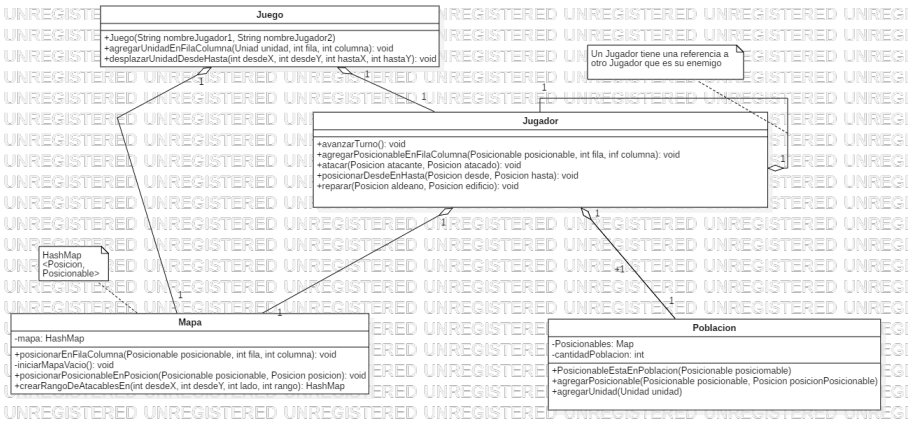

Figura 1: Diagrama que representa parte de la solución del trabajo práctico

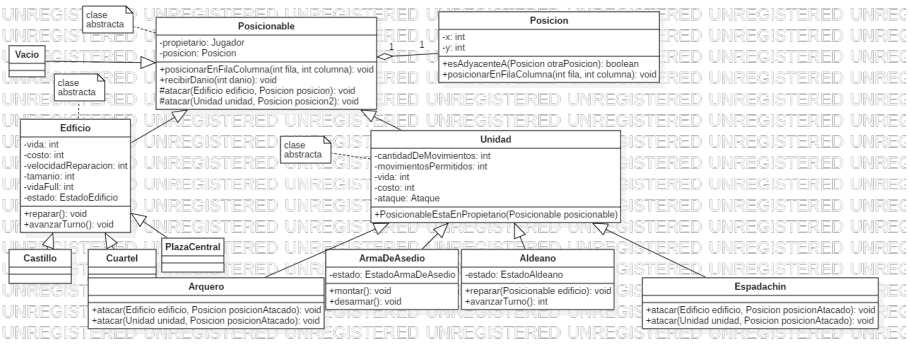

Figura 2: Diagrama que representa parte de la solución del trabajo práctico

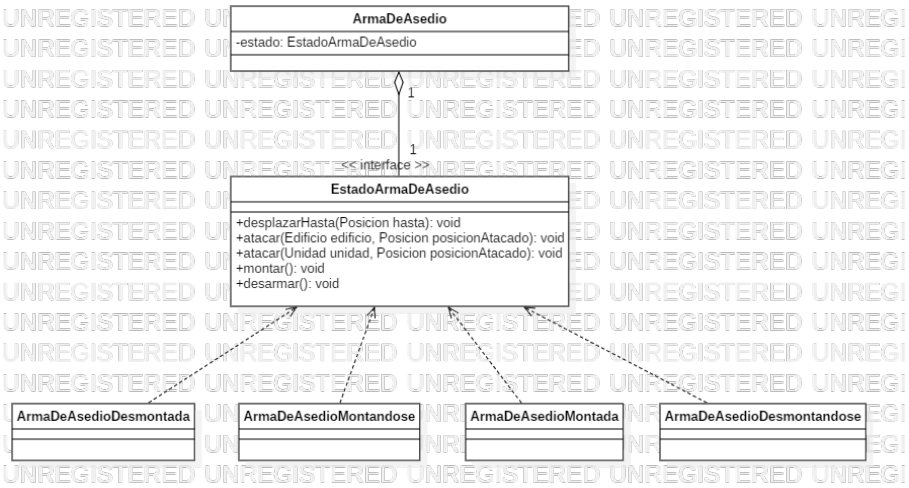

Figura 3: Diagrama que representa parte de la solución del trabajo práctico

4. Diagramas de secuencia

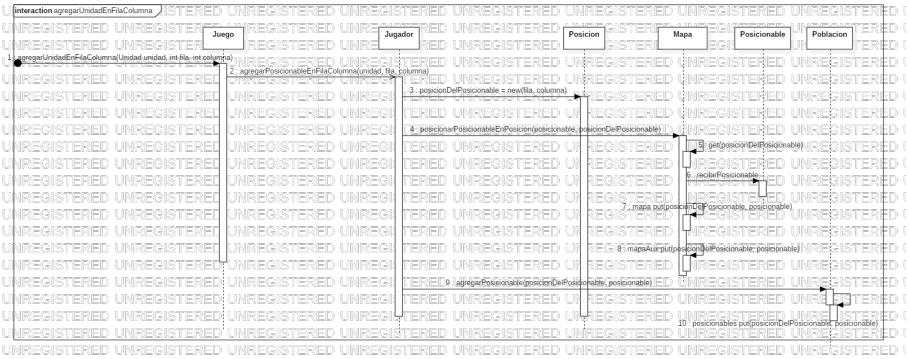

Figura 4: Diagrama de secuencias agregarUnidadEnFilaColumna

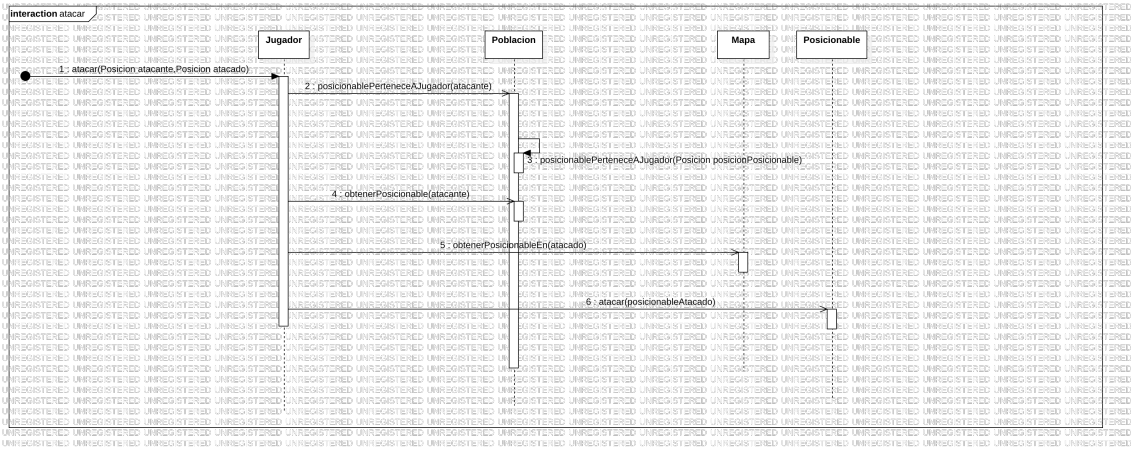

Figura 5: Diagrama de secuencias atacar.

5. Diagramas de paquetes

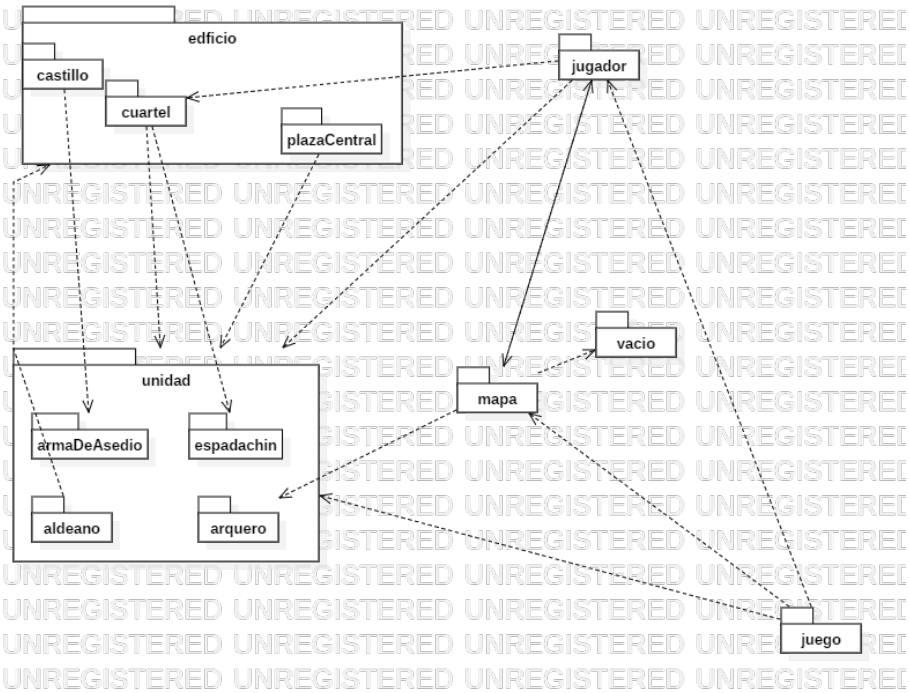

Figura 6: Diagrama de paquetes del trabajo práctico

6. Diagramas de estado

Figura 7: Diagrama de estado para un aldeano

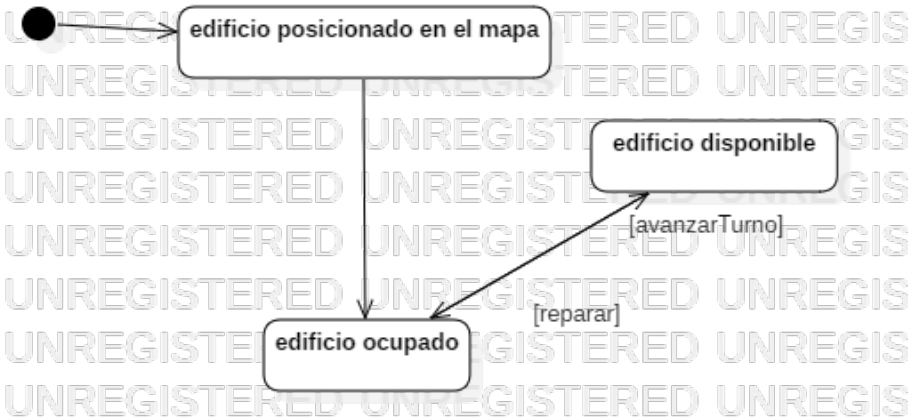

Figura 8: Diagrama de estado para un edificio

7. Detalles de implementación

7.1. Mapa

La clase Mapa tiene un atributo mapa que es un diccionario del tipo *Map <Posicion, Posicionable>* siendo Posicion la clave y Posicionable el valor. Almacena todas las posiciones disponibles en el terreno de juego como clave, asignándole como valor una clase del tipo Unidad, Edificio o Vacio. Estas clases heredan de Posicionable.

El constructor del mapa

```
this.mapa = new HashMap <Posicion, Posicionable>();
this.mapaAux = new HashMap <Posicion, Posicionable>();
this.filas = 30;
```

```
this.columnas = 30;
this.iniciarMapaVacio();
```

inicializa todas las posiciones del mapa con la clase Vacio. Además se crea un *mapaAux* que será utilizado para guardar las posiciones que se agregan al mapa cuando se agrega un Posicionable y luego agregar esas posiciones en cada Jugador.

7.2. Juego

Los atributos de esta clase son

```
private Mapa mapa;
private Jugador jugadorActual;
```

se tiene una referencia al mapa ya que representa al terreno de juego y además el *jugadorActual* representa al Jugador que se encuentra jugando en cierto momento. Este atributo se va actualizando a medida que se avanza el turno ya que cambia de Jugador.

Para esta clase se utilizó el *patrón Fachada*: a través de Juego se da la interacción con el usuario y ésta le delega todas las responsabilidades a la clase correspondiente.

El constructor

```
this.mapa = new Mapa ();
this.jugadorActual = new Jugador (this.mapa, nombreJugador1, nombreJugador2);
this.jugadorActual.iniciarPosicionables();
```

inicializa el Juego creando una instancia de Jugador (que a su vez instancia a otro Jugador). Delega al *jugadorActual* la tarea de inicializar los posicionables: esto es crear un Castillo, una PlazaCentral y los Aldeanos correspondientes para cada jugador.

7.3. Jugador

Esta clase posee dos constructores
el primero:

```
public Jugador(Mapa mapa, String nombreJugador, String nombreEnemigo) {

        this.mapa = mapa;
        this.nombre = nombreJugador;
        String colorEnemigo = "lightblue";
        this.iniciarAtributos();
        this.enemigo = new Jugador (mapa,nombreEnemigo, this);

}
```

el segundo:

```
public Jugador (Mapa mapa, String nombre, Jugador jugador, String color) {

        this.mapa = mapa;
        this.nombre = nombre;
```

```
            this.color = color;
            this.iniciarAtributos();
            this.enemigo = jugador;

}
```

Desde el primer constructor se llama al segundo y de esa forma es que quedan inicializados tanto el Jugador como su *enemigo*. Los atributos del Jugador son

```
private Mapa mapa;
private String nombre;
private Jugador enemigo;
private Poblacion poblacion;
private Castillo castillo;
private String color;
```

Conoce al *mapa* para poder actualizarlo, a su *enemigo* para poder devolverlo cuando se avanza un turno y a su *castillo* para que éste ataque a sus enemigos al alcance cuando se avanza el turno. Además, tiene un atributo *poblacion* que se encargará de cuestiones detalladas en la subsección *Población*. Vale aclarar que el atributo *color* implica no respetar estrictamente el patrón MVC. No obstante, es el único caso en el que esto sucede.

7.4. Población

Sus atributos:

```
private int oro;
private int produccionDeOro;
private int topeDePoblacion;
private int cantidadPoblacion;
private Map<Posicion, Posicionable> posicionables;
```

Esta clase almacena *posicionables* en un *Map*. Estos posicionables representan a aquellos que forman parte de la misma población y por lo tanto del mismo Jugador, ya que cada Jugador tiene una Poblacion asociada. Esta clase, principalmente, se encarga del manejo del oro y de las cuestiones poblacionales (aumentar o decrecer población) de cada Jugador.

7.5. Ataque

En esta clase se utilizó *sobrecarga de métodos* para solucionar el daño causado a Unidades y Edificios ya que no es el mismo para ambas clases. Firma de los métodos:

```
public void atacar (Unidad recibeAtaque);
public void atacar (Edificio recibeAtaque);
public void atacar (Posicionable recibeAtaque);
public void atacar (Unidad unidadAtacada, Posicion posicionAtacado, Posicion posicionAtac
public void atacar (Edificio edificioAtacado, Posicion posicionAtacado, Posicion posicion
```

7.6. Edificio

Los edificios pueden repararse, lo cual lleva cierta cantidad de turnos. Cuando éstos se encuentran en reparación no pueden realizar ninguna acción. Para solucionar esto se utilizó el *patrón state*: `EstadoEdificioOcupado` o `EstadoEdificioDisponible`. Cada Edificio se almacena su estado, .ºcupado"si se encuentra en reparación.

7.7. Castillo

Los atributos de esta clase son

```
private Ataque ataque;
private int alcance = 3;
private int danio = 20;
```

El atributo *danio* se refiere a la vida que el castillo le resta tanto a unidades como edificios cuando los ataca al pasar cada turno. Se utiliza como argumento del constructor de su *ataque*.

7.8. Unidad

Sus atributos son

```
protected int cantidadDeMovimientos;
protected int movimientosPermitidos;
protected int ataquesPermitidos = 1;
protected int cantidadDeAtaques = 0;
protected Ataque ataque;
protected int alcance;
protected Jugador propietario;
```

La *cantidadDeMovimientos* y los *movimientosPermitidos* se utilizan para corroborar que una Unidad avance sólo una vez por turno.

De la misma manera, *ataquesPermitidos* y *cantidadDeAtaques* se utilizan para verificar que una Unidad solo pueda atacar una vez por turno.

El atributo *propietario* representa al Jugador que posee a esa Unidad. Esto será utilizado a la hora de chequear si un Jugador ataca a un enemigo o a un aliado.

Se utiliza sobrecarga de métodos para recibir daño de otras Unidades o del Castillo:

```
public void recibirDanio (int danio);
public void recibirDanioDe (Posicionable posicionable);
public void recibirDanioDe (Unidad unidad);
public void recibirDanioDe (Edificio edificio);
```

7.9. Aldeano

Se utilizó el *patrón state* para solucionar que cuando un Aldeano repara o construye tarda cierta cantidad de turnos y durante esos turnos no puede realizar otra acción ni sumar oro. Los distintos estados son: `EstadoAldeanoOcupado` y `estadoAldeanoDisponible`.

7.10. ArmaDeAsedio

Se utilizó el *patrón state* para los distintos posibles estados del arma de asedio: `ArmaDeAsedioMontada`, `ArmaDeAsedioMontandose`, `ArmaDeAsedioDesmontada` y `ArmaDeAsedioDesmontandose`. El arma puede atacar únicamente si se encuentra montada lo cual tarda un turno (por eso el estado ArmaDeAsedioMontandose). También, puede desplazarse pero únicamente si está desarmada o desmontada lo cual también tarda un turno (ArmaDeAsedioDesmontandose).

7.11. Vacio

Esta clase fue incluída por la utilización del `patrón Null Object`. Se encuentra en las posiciones del mapa que no poseen niguna Unidad o Edificio todavía. Cualquier acción indicada a esta clase lanzará una excepción o simplemente no hará nada ya que no contiene información.

8. Excepciones

EdificioOcupadoException Un edificio no puede funcionar normalmente en caso de encontrarse en construcción o en reparación.

PosicionDesocupadaError No se puede desplazar una Unidad si ésta no se encuentra en la posición indicada. Es decir, no se puede realizar ningún tipo de desplazamiento si la posición indicada se encuentra vacía. Tampoco se puede atacar tanto a una Unidad como a un Edificio si la posición desde donde se quiere realizar el ataque se encuentra vacía, por lo que se lanzará esta excepción.

PosicionOcupadaError Esta excepción se lanzará en caso de querer agregar un elemento en una posición que ya se encuentra ocupada por otro actualmente.

MovimientoPorTurnoExcedidosError Cada Unidad puede realizar una cierta cantidad de movimientos en cada turno. En este trabajo práctico esa cantidad es uno. Por lo tanto, si se quiere realizar más de un desplazamiento con la misma unidad se lanzará esta excepción.

PosicionFueraDelMapaError Tanto los desplazamientos como la inclusión de objetos al juego sólo pueden ser en posiciones pertenecientes al mapa. Por lo tanto, en caso de violar esto se lanzará esta excepción.

AldeanoOcupadoException Un Aldeano no puede realizar una tarea en caso de estar Ocupado.

ArmaDeAsedioDesmontadaException El ArmaDeAsedio no puede realizar un ataque en caso de estar desmontada.

ArmaDeAsedioMontandoseException El ArmaDeAsedio no puede atacar ni desplazarse en caso de estar montándose, es decir, en caso de que no hayan pasado los turnos necesarios para que el arma termine de montarse.

ArmaDeAsedioDesmontandoseException El ArmaDeAsedio no puede atacar ni desplazarse cuando aún no pasaron los turnos necesarios para que se desarme.

ArmaDeAsedioMontadaException Un ArmaDeAsedio no puede desplazarse mientras se encuentre montada.

AldeanoNoPuedeAtacarError Un Aldeano no puede realizar ataques por lo que en caso de querer indicarle a un aldeano que realice un ataque se lanzará esta excepción.

CrearUnidadException Las únicas clases que pueden crear una Unidad son Cuartel, PlazaCentral y Castillo. Por lo tanto, si se le indica a un objeto de otra clase que se cree una unidad se lanzará esta excepción.

AtacandoAUnAliadoError Esta excepción será lanzada en caso de querer atacar a una Unidad o a un Edificio que pertenece al mismo jugador de quien ataca.

DesplazarAPosicionOcupadaError Esta excepción será lanzada en caso de querer desplazar a una unidad hacia una posición que se encuentra ocupada previamente tanto por una Unidad o un Edificio.

ConstruccionEdificioException Esta excepción será lanzada en caso de querer crear una Unidad, o reparar un Edificio que sea encuentra en construcción.

PosicionNoAdyacenteError Esta excepción será lanzada en caso de querer realizar un desplazamiento hacia una posición que no es adyacente.

JugadorSuperaTopePoblacionalException Esta excepción será lanzada en caso de querer aumentar la población de un jugador y que ésta ya haya llegado a su tope máximo y no admita más población.

JugadorSinOroException Esta excepción será lanzada en caso de querer descontarle la cantidad de oro a un Posicionable y éste no tenga la cantidad de oro suficiente para descontarl la cantidad indicada.

PlazaCentralNoPuedeAtacarError Esta excepción será lanzada en caso de querer indicarle a una PlazaCentral que realiace un ataque ya que ésta no puede atacar.

CuartelNoPuedeAtacarError el Cuartel no puede realizar un ataque entonces se lanzará esta excepción en caso de intentar atacar con un cuartel.

EdificiosNoSePuedenDesplazarError Los edificios no se pueden desplazar por lo tanto en caso de querer desplazar uno se lanza esta excepción.

AtacandoEsPosicionFueraDelAlcanceError cada Unidad o Castillo tiene cierto alcance para atacar. Cuando se quiere atacar a un Posicionable que se encuentra fuera de su alcance se lanza esta excepción.