Trabajo Práctico 2 — ALTEGO

[7507/9502] Algoritmos y Programación III
Curso 1
Primer cuatrimestre de 2021
Grupo 10

Alumna	Padrón	Email
PONT, María Fernanda	104229	mpont@fi.uba.ar
DI NARDO, Chiara Anabella	105295	cdinardo@fi.uba.ar
GADDI, María Pilar	105682	mgaddi@fi.uba.ar

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 basándose en el juego T.E.G. El trabajo se desarrollará en Java utilizando los conceptos del paradigma de la orientación a objetos vistos hasta ahora en el curso.

2. Supuestos

Debido a que en algunos casos la resolución de ciertos aspectos (no contemplados en la especificación del trabajo) quedaban a criterio del alumno, se tuvieron en cuenta los siguientes supuestos: En esta primera entrega se siguieron las reglas del juego establecidas por la cátedra. Entre ellas están:

- El atacante no podrá atacar con la misma cantidad de fichas que tiene el mismo en su país ni con una cantidad mayor.
- El defensor atacará con la cantidad de fichas que tiene el mismo en el país que posee y que es atacado.
- Se podrá jugar con un máximo de tres dados por ataque, y el atacante va a poder elegir cuantos dados utilizar, siempre y cuando esa cantidad no sea mayor o igual a la cantidad de fichas (ver supuesto 1).
- El jugador debe saber cuáles países son válidos. El mismo no puede inventar países nuevos.

3. Diagramas de clase

En las siguientes imagenes se tiene un diagrama de clases, en estos se muestran las clases utilizadas en el trabajo y las formas de las mismas.

El primer diagrama de clase muestra como se planteo la idea de la segunda entrega del juego T.E.G. Se tiene principalmente a la clase Juego, y la misma tiene varias fases que implementan una misma interfaz IFase.

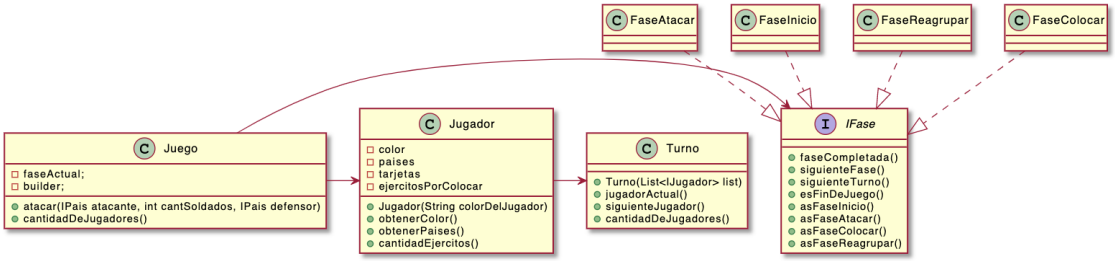

Figura 1: Diagrama de Juego (ALTEGO).

En el próximo diagrama se puede observar la idea de cómo inicializar la clase Mapa mediante una clase Factory, guiado por el patrón de diseño. La entidad Mapa tendría la responsabilidad de informar los continentes conquistados por un mismo jugador mediante un método.

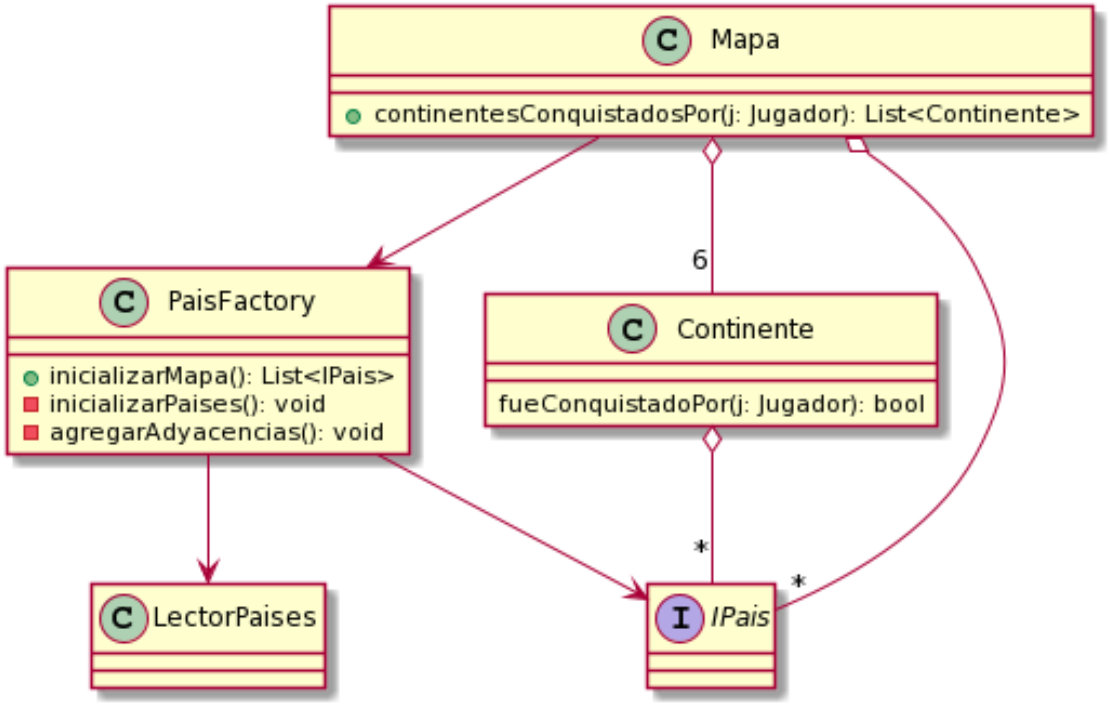

Figura 2: Diagrama de Mapa.

En este último, se observa la interfaz IPais implementada por la clase Pais y PaisMock, utilizada para implementar las pruebas.

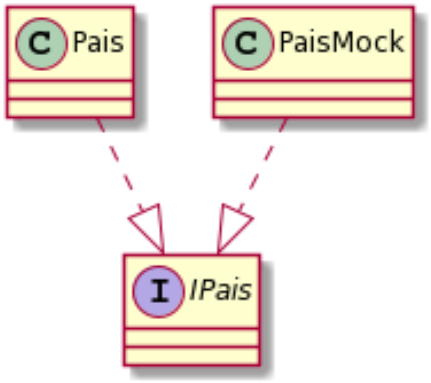

Figura 3: Diagrama de interfaz IPais.

4. Diagramas de secuencia

En la siguiente imagen se muestra la secuencia de atacar un pais y que como resultado el pais defensor salga victorioso.

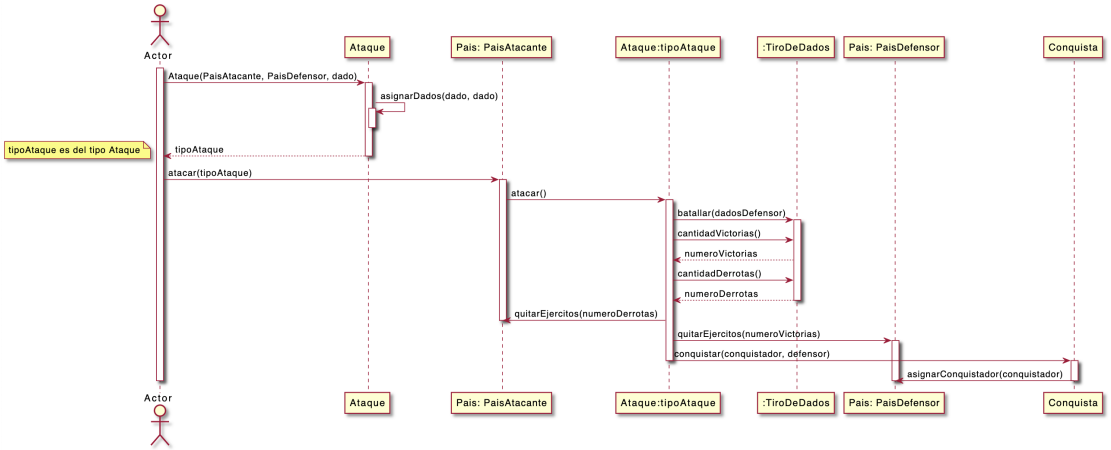

Figura 4: Atacar pais, gana defensor.

En la siguiente imagen se muestra la secuencia de asignar ejércitos en un país.

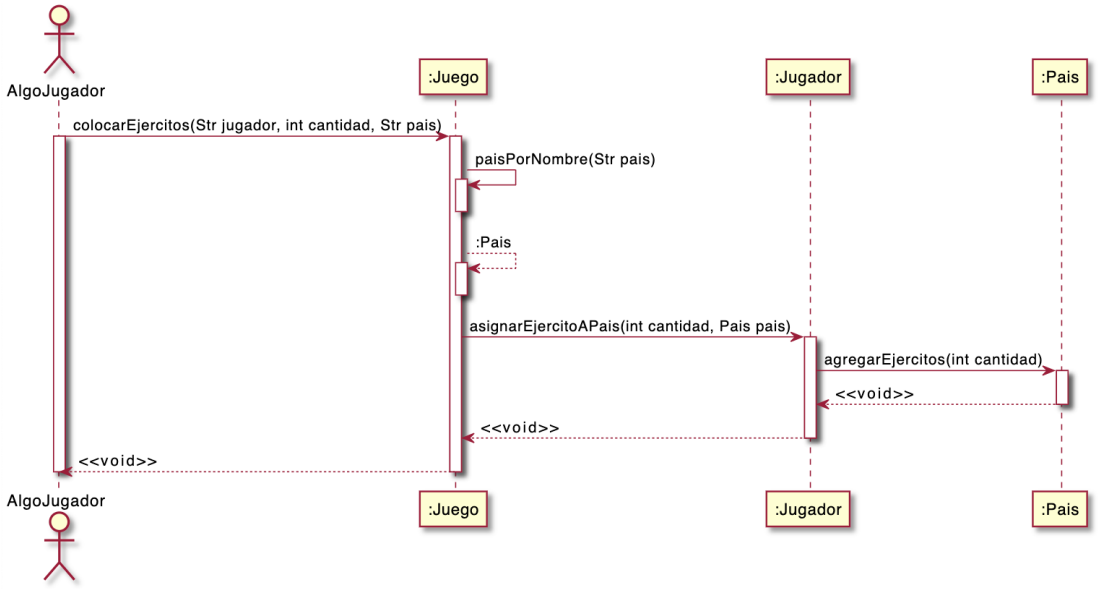

Figura 5: Jugador coloca ejércitos en un país.

En la siguiente imagen se muestra la secuencia de cambio de fases en el juego.

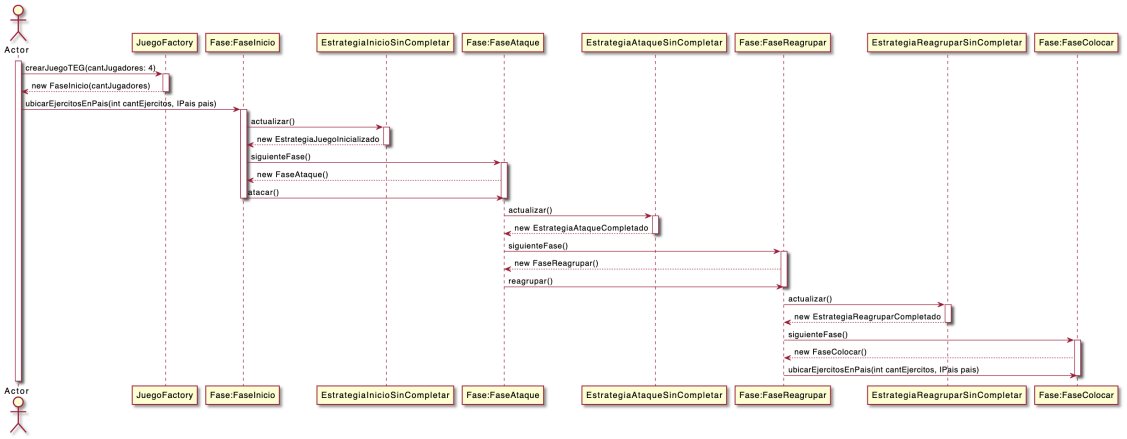

Figura 6: Cambios de fases en el juego.

5. Diagrama de Paquetes

En la siguiente imagen se muestra el diagrama de paquetes UML:

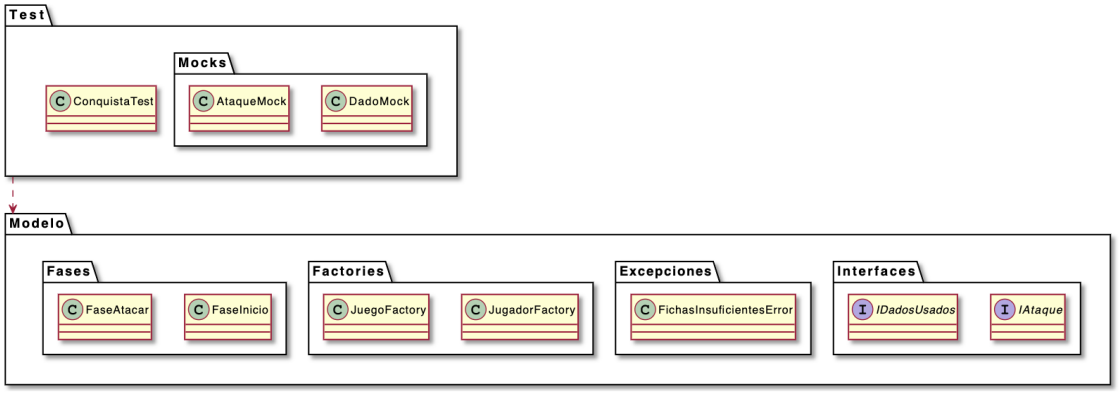

Figura 7: Diagrama de Paquetes UML.

6. Diagramas de Estado

En la siguiente imagen se muestra el diagrama de estado UML, en relación a cuando el Jugador se inicializa y luego se le agregan ejercitos y paises:

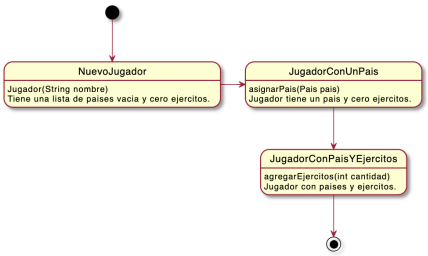

Figura 8: Diagrama de Estado Jugador Nuevo.

En la siguiente imagen se muestra el diagrama de estado UML, en relación a cuando el Pais se inicializa y luego se le agregan ejercitos:

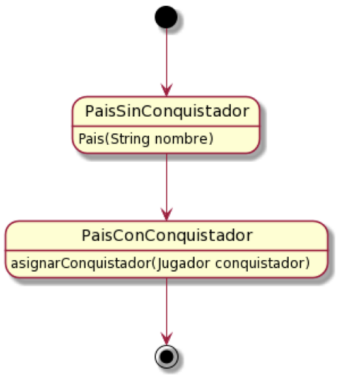

Figura 9: Diagrama de Estado Pais Nuevo.

En la siguiente imagen se muestra el diagrama de estado UML, en relación a cuando el juego va pasando por las diferentes fases:

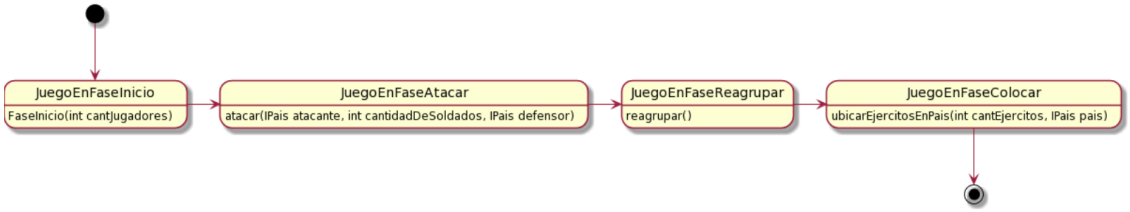

Figura 10: Diagrama de Estado Juego.

7. Detalles de implementación

Las partes esenciales del modelo son TEGEngine (Juego) y ALTEGO. TEGEngine es el 'engine' del juego y no sabe nada sobre interacción del usuario, sino que recibe mensajes de juego como 'colocar ejércitos' o 'terminar fase'. Luego, ALTEGO es una clase que utiliza el engine de acuerdo a eventos de UI, es decir es el único que sabe de JavaFX y los eventos de JavaFX.

TEG está compuesto de 2 fases:fase de inicio y fase de juego. Además, la fase de juego es un ciclo de 3 etapas: atacar, reagrupar y colocar ejércitos.

Cada iteración es un turno de un jugador. TEGEngine es quien define las reglas del juego. Durante la fase de inicio TEGEngine no acepta mensajes como atacar(), durante la fase de juego no acepta repartirPaises(). Además no se puede agregar más jugadores después de repartir los paises, en ese sentido existen 'etapas' dentro de la fase de inicio.

A continuación se detallará la implementación interna de algunas clases interesantes para lograr un mejor entendimiento del código.

7.1. Juego

Todo el juego será creado a través de la clase JuegoFactory. A partir de este factory también se van a construir todos los jugadores que sean necesarios para el juego. Los mismos serán creados a través de la clase JugadorFactory, se verá mas en detalle en la proxima sección. Cuando se crea el juego, se verifica que la cantidad de jugadores sea la correcta, se crea a todos los jugadores con un sistema de turnos establecido y un sistema de canjes. Se repartieron las responsabilidades del juego a través de las diferentes fases que tiene el mismo. Para esto se tienen las siguientes fases, que implementan la interfaz IFase:

```
FaseInicio
FaseAtacar
FaseReagrupar
FaseColocar
```

En cada una de estas se pueden realizar acciones específicas, y hasta que las acciones indispensables de las mismas no estén completadas, no se podrá seguir con las siguientes.

7.2. Jugador

La clase Jugador tiene como atributos un color, una lista de paises, una lista de tarjetas (tarjetas beneficio) y una cantidad de ejercitos (que serían las fichas totales que posee). Todos los jugadores serán creados al inicio del juego, ya que son indispensables para el mismo. De esta forma, se utiliza un JugadorFactory que crea todo lo necesario para los jugadores: le asigna un color, paises y una cantidad de ejercitos inicial.

```
public List<IJugador> construirJugadores(List<String> colores, int cantidad)
throws EjercitosException {
    List<IJugador> jugadores = jugadoresDeColores(colores.subList(0, cantidad));
    asignarPaisesAJugadores(paises, jugadores);
    asignarEjercitosAJugadores(jugadores);
    return jugadores;
}
```

7.3. Pais

La clase Pais es la responsable de atacar a otros países. tiene un jugador que lo controla, y una cantidad de soldados. Cuando hay un ataque el pais NO recibe soldados que no sean de su jugador. Cuando es conquistado por otro jugador B, B tiene que establecer al menos uno de sus soldados en el país.

Tiene como atributos un nombre y una determinada cantidad de ejercitos (que serían las fichas totales que tiene el país). También tiene un dueño, que sería su conquistador (un jugador).

El pais podrá atacar estableciendo un tipo de ataque personalizado (para que distintas circunstancias puedan ser testeadas) de la siguiente manera:

```
public void atacar(IAtaque ataque) {
    ataque.atacar();
}
```

También se podrá optar por un ataque predeterminado, sin necesidad de establecer un ataque específico:

```
public void atacar(Pais defensor, int numeroEjercitos) throws Exception {
    IAtaque ataque = new Ataque(this, defensor, numeroEjercitos);
    atacar(ataque);
}
```

7.4. DadosUsados

La clase TiroDeDados implementa la interfaz IDadosUsados. Tiene como atributos una lista de valores que son los valores de los dados que se van tirando durante la partida, y una lista de resultados que son aquellos que van resultando de la comparación de valores de los dados, entre el atacante y el defensor, para luego poder determinar al ganador. Además de la clase TiroDeDados, se implementó un mock llamado DadoMock, y este es utilizado para poder realizar las pruebas correspondientes a la clase Ataque, descripta a continuación.

7.5. Ataque

La clase Ataque implementa la interfaz IAtaque. Conoce a un pais atacante y a un pais defensor, como así también a los dados de ambos. Se podrán realizar dos tipos de ataques, un 'Ataque Predeterminado' y un 'Ataque Falso'. Este último fue utilizado principalemnte para que las pruebas unitarias puedan ser consistentes y para que se pueda asegurar que no cambien de resultado.

Para lograr un Ataque Predeterminado se utiliza el siguiente constructor:

```
public Ataque(IPais atacante, IPais defensor, int cantEjercitos) throws Exception {
    this.atacante = atacante;
    this.defensor = defensor;
    if(atacante.cantidadEjercitos() <= cantEjercitos
    || cantEjercitos > maxDados)
    throw new FichasInsuficientesError("El jugador sólo puede atacar con"
    + (atacante.cantidadEjercitos() - 1) + "ejércitos.");
```

```
    asignarDados(new DadosUsados(cantEjercitos),
    new DadosUsados(Math.min(defensor.cantidadEjercitos(), maxDados)));
}
```

Para lograr un AtaqueFalso se utiliza el siguiente constructor:

```
public Ataque(IPais atacante, IPais defensor, IDadosUsados dado) throws Exception{
    this.atacante = atacante;
    this.defensor = defensor;
    asignarDados(dado, dado);
}
```

Para este último se debe usar el mock DadoMock mencionado anteriormente.

7.6. Conquista

La clase Conquista presenta únicamente al método conquistar y, en caso de que se lo llame, se encarga de cambiar al dueño del país que se le pasa por parámetro. Esto lo hace de la siguiente manera:

```
public void conquistar(IJugador iJugador, IPais pais) {
    (pais.obtenerConquistador()).quitarPais(pais);
    pais.asignarConquistador(iJugador);
}
```

8. Excepciones

FichasInsuficientesError Es lanzada cuando el jugador intenta agregar ejércitos a un país pero no tiene las fichas suficientes y, también, cuando se quiere atacar un país con más ejércitos que en el país atacante.

PaisNoExistenteError Es lanzada cuando se el jugador quiere agregar ejercitos en un pais que no tiene.

FaseIncompletaException Es lanzada cuando se quiere acceder a la siguiente fase del juego, pero la actual está todavía incompleta.

FaseErroneaException Es lanzada cuando se quiere realizar alguna acción que no es responsabilidad de una fase determinada.

EjercitosException Es lanzada cuando se quiere agregar a un jugador una cantidad de ejercitos menor o igual a cero.

CantidadDeJugadoresError Es lanzada cuando se quiere agregar al juego una cantidad no válida de jugadores (menor a dos o mayor a seis).