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.

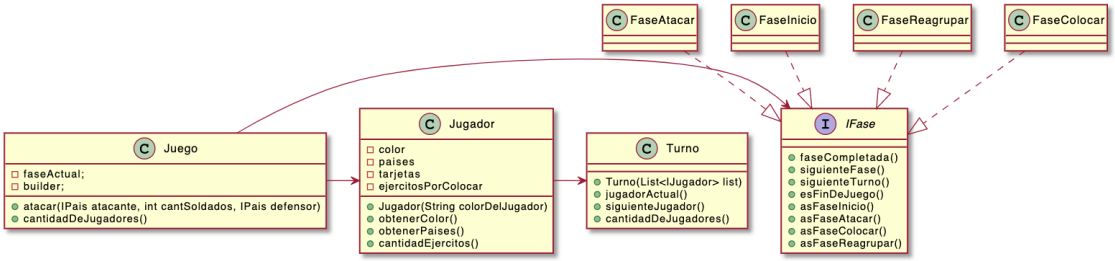

Figura 1: Diagrama de Juego (ALTEGO).

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

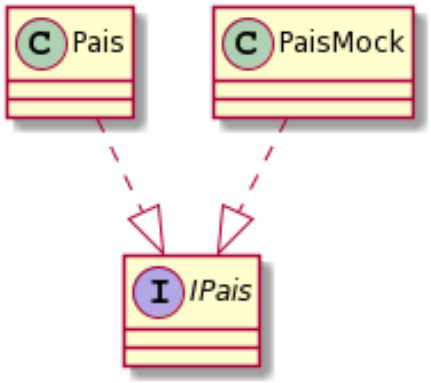

Figura 2: Diagrama de interfaz IPais.

En el siguiente diagrama se puede observar la interfaz ICanje implementada por las clases CanjeParaAgregadoDeEjercitosEnGeneral y CanjeParaAgregadoDeEjercitosEnUnPais. Estos canjes son diferentes pero se activan de igual forma por el jugador según las tarjetas correspondientes.

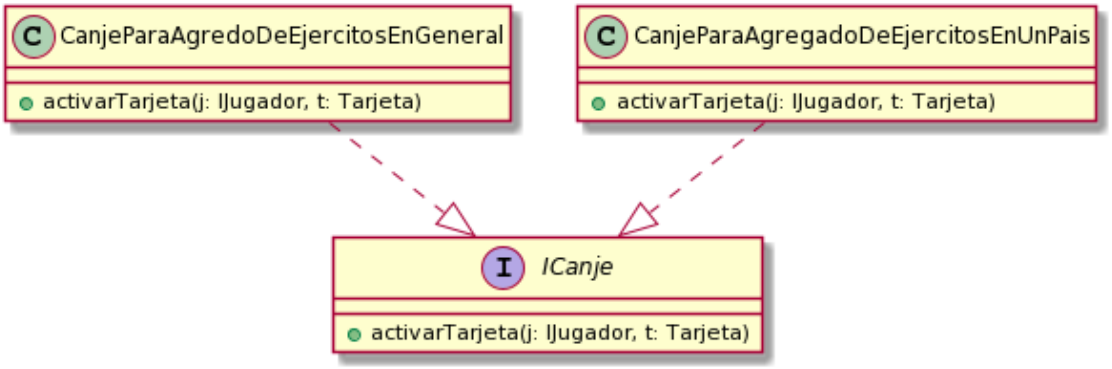

Figura 3: Diagrama de clases de ICanje.

El Jugador tiene como atributo el número de canjes realizados. Esto varía porque según la cantidad de canjes hechos, es distinta la cantidad de fichas a agregar.

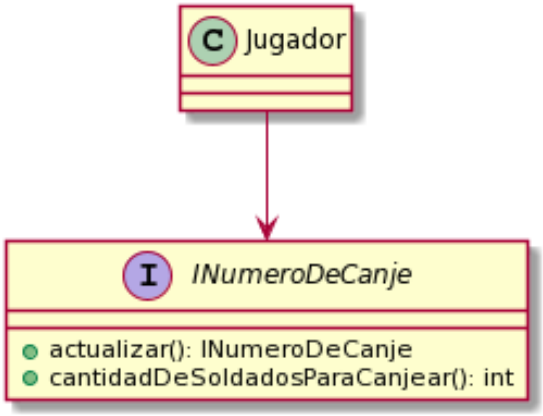

Figura 4: Diagrama de clases de Jugador y INumeroCanje.

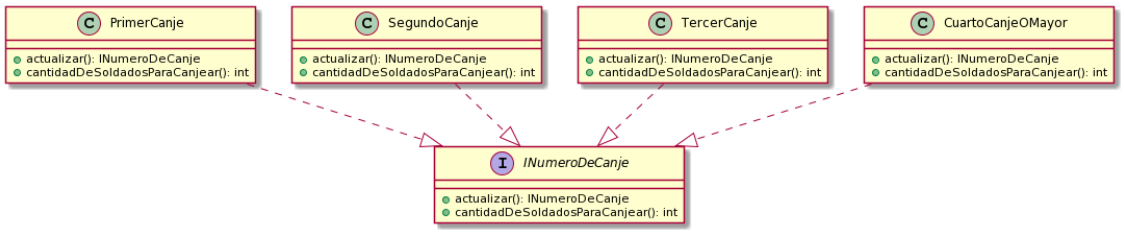

Figura 5: Diagrama de clases de interfaz ICanje y clases que lo implementan.

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.

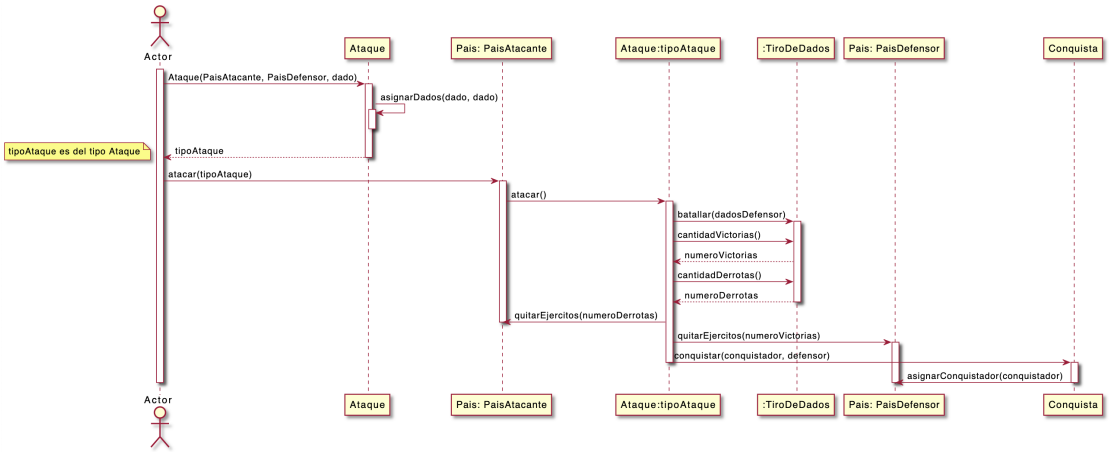

Figura 6: Atacar pais, gana defensor.

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

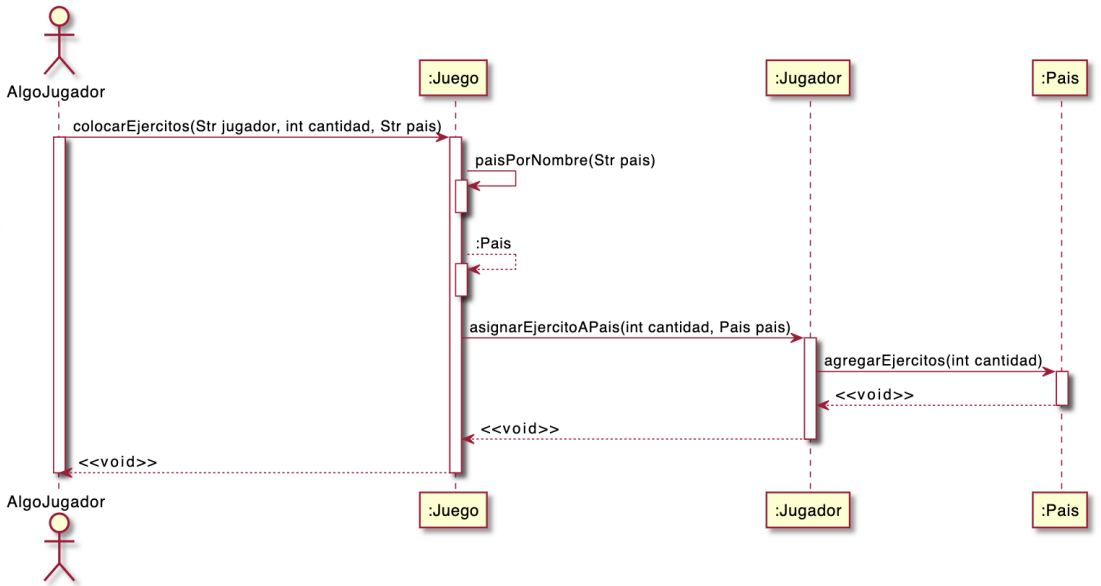

Figura 7: Jugador coloca ejércitos en un país.

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

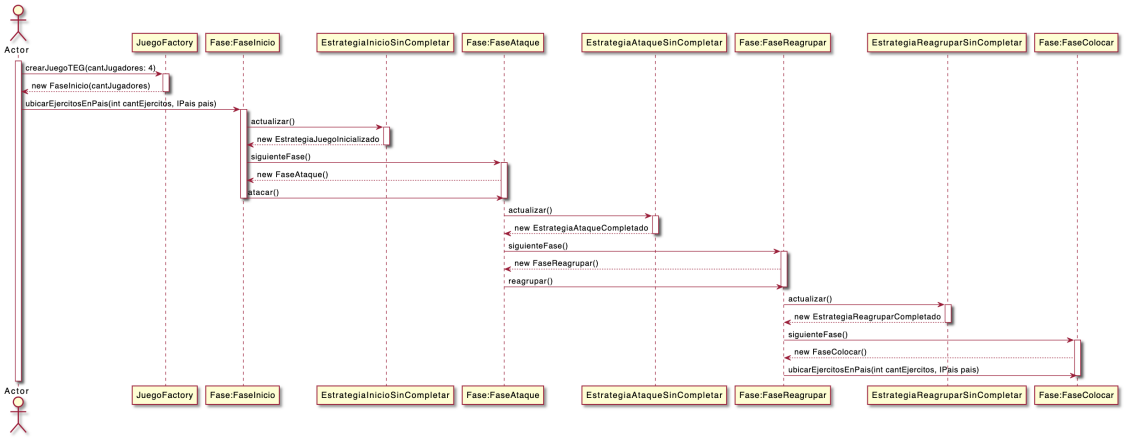

Figura 8: Cambios de fases en el juego.

5. Diagrama de Paquetes

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

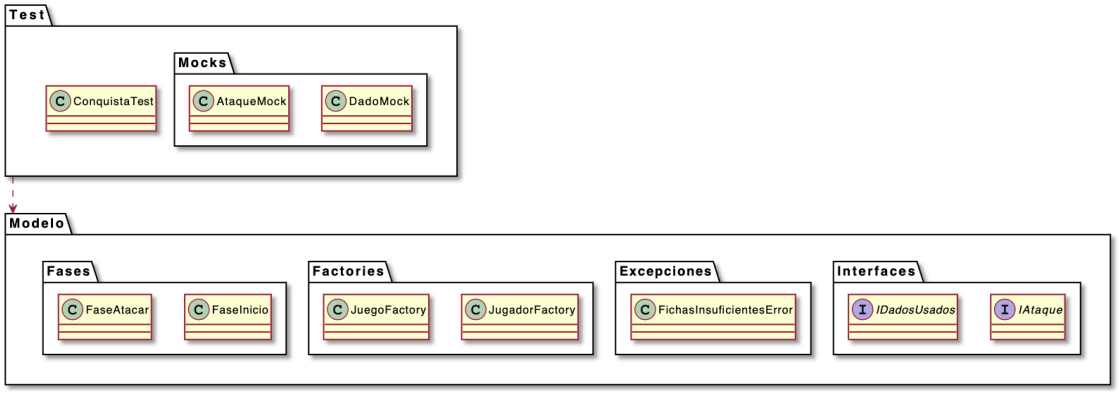

Figura 9: Diagrama de Paquetes UML.

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:

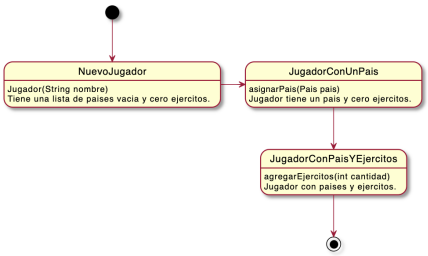

Figura 10: Diagrama de Estado Jugador Nuevo.

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:

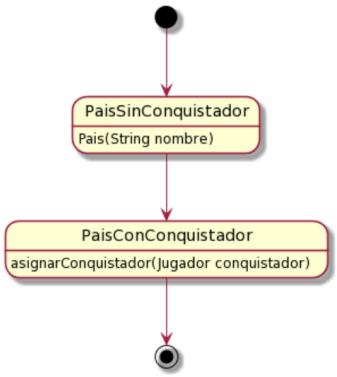

Figura 11: Diagrama de Estado Pais Nuevo.

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

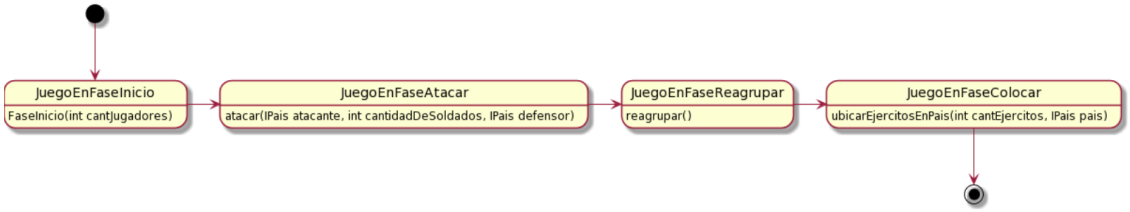

Figura 12: Diagrama de Estado Juego.

7. Detalles de implementación

Las partes esenciales del modelo son Juego y las fases. El juego está compuesto por fases ordenadas (inicio, atacar, reagrupar y colocar). Cada fase fue implementada en base del patrón de diseño Strategy, donde según sea la estrategia de la fase (por ejemplo, fase atacar con conquista o sin conquista) es el método que se usa. Esto es posible gracias a que se implementó una interfaz IFase donde todas las clases entienden la misma firma de métodos.

Por otro lado, el controlador interactúa con la clase Juego y esta delega a las fases la funcionalidad a ejecutar.

7.1. Juego

Todo el juego se inicializa con los objetos necesarios a utilizar a lo largo de todo el juego. Esta clase es con la cual interactúa el controlador o, en el caso que no exista vista, el usuario podrá jugar con los métodos de una instancia de juego. 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 método de la clase Turno 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) {
    mapa.asignarPaises(jugadores, new OrdenadorAleatorio());
    asignarSistemaDeCanje(mazo,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 AlgoTegException {
        this.atacante = atacante;
        this.defensor = defensor;
        if(atacante.cantidadEjercitos() <= cantEjercitos || cantEjercitos > maxDados)
            throw new FichasInsuficientesError("El jugador sólo puede atacar con" + (atacante.can
        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         AlgoTegExcept
    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);
}
```

7.7. Mapa

Es la clase contenedora de países creados por la clase MapaFachada (inicaliza los países de manera válida mediante un archivo) y es la responsable de asignarle los países a una lista de jugadores. Para que sea posible asignarles los países a los jugadores con un orden, se implementó inyectándole una interfaz ordenadora. Por otro lado, la clase Mapa devuelve los continentes conquistados por un jugador.

```
public Mapa() {
    MapaFachada mapaFachada = new MapaFachada();
    paises = mapaFachada.obtenerPaises();
    continentes = mapaFachada.obtenerContinentes();
}

public void asignarPaises(List<IJugador> jugadores, IOrdenador ordenador) {
    ordenador.ordenar(paises);
    for (int i = 0; i < paises.size(); i++) {
        IPais actual = paises.get(i);
        jugadores.get(i % jugadores.size()).inicializarPais(actual);
    }
}
```

7.8. Mazo

Similar al Mapa, Mazo utiliza una clasa MazoFachada para obtener las tarjetas en estado válido con su país y símbolo. Estos conseguidos mediante un archivo JSON. Tiene un método para devolver una tarjeta aleatoria del mazo para poder asignarsela al jugador cuando conquista un país en un ataque.

```
public Mazo(IMapa mapa) {
    MazoFachada mazoFachada = new MazoFachada(mapa);
    mazo = mazoFachada.obtenerTarjetas();
}
```

7.9. Canje

Canje es la clase responsable de realizar un canje de tarjetas. Con una clase auxiliar Tarjetas, valida si las tarjetas envíadas por parámetro son válidas y luego le asigna al jugador los ejércitos correspondientes. Existe una instancia de Canje por jugador y ésta es la que guarda la información de la cantidad de canjes que este último realizó. Dependiendo de la cantidad de canjes realizados, varía la cantidad de ejércitos a colocar.

```
public Canje() {
    numero = new PrimerCanje();
}
```

8. Excepciones

Todas las excepciones heredan de una excepción madre llamada AlgotegException.

FichasInsuficientesException 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.

PaisNoExistenteException 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.

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

NoSePuedeProducirCanjeException Es lanzada cuando se intenta realizar un canje con tarjetas inválidas.

NoExisteTarjetaException Es lanzada cuando se quiere canjear o activar una tarjeta que no existe.