FACULTAD DE INGENIERIA
Universidad de Buenos Aires

Trabajo Práctico 2 – A.L.T.E.G.O

[7507/9502] Algoritmos y Programación III

Curso 1

Primer cuatrimestre de 2021

Integrantes		
Alumno	**Padrón**	**Email**
Felipe De Luca Andrea	105646	fdeluca@fi.uba.ar
Marcelo Ariel Rondán	105703	mrondan@fi.uba.ar
Nicolás Ezequiel Zulaica Rivera	105774	nzulaica@fi.uba.ar
Ariana Magalí Salese D'Assaro	105558	asalese@fi.uba.ar
Gabriel Semoreli	105681	gsemorile@fi.uba.ar

Índice

Introducción **2**

Supuestos **2**

Diagramas de clases **3**

Diagramas de secuencia **6**

Diagrama de paquetes **8**

Diagramas de estado **9**

Detalles de implementación **10**

Excepciones **11**

1. Introducción

El presente informe reúne la documentación de la solución del segundo trabajo práctico de la materia Algoritmos y Programación III que consiste en desarrollar una versión del juego T.E.G. que plantea un conflicto bélico que ocurre sobre un planisferio dividido en 50 países.

2. Supuestos

Tarjetas

- Siempre que un jugador tenga las tarjetas necesarias para **realizar un canje** (tres tarjetas con el mismo símbolo o tres tarjetas con distinto símbolo) este se realizará **sin consultar** al jugador. Debido a este supuesto el jugador nunca tendrá más de 5 tarjetas, al igual que en el juego original.
- Si un jugador recibe una carta de un país que posee, esta se activará (se pondrán dos fichas en el país) **automáticamente**.
- Cuando un jugador invade al menos un país en la ronda de ataque, la tarjeta se le agrega automáticamente sin necesidad de que este la solicite.
- Las tarjetas “comodín” se reemplazaron por otros tipos de tarjetas con símbolos comunes.

Objetivos

- En caso de que un jugador tenga como objetivo destruir a otro jugador, **no importa quien lo destruya**, si este se queda sin países ganará cuyo objetivo era destruirlo.
- Si varios jugadores tienen como objetivo destruir al mismo jugador, si este es destruido por cualquier jugador de la partida, se dará un **empate** entre quienes comparten este objetivo.

Reagrupación

- A diferencia del juego original, luego de atacar los jugadores podrán **reagrupar la cantidad de fichas que desee**, siempre y cuando deje una en el país de origen.

3. Diagramas de clases

A continuación, a partir de ciertos diagramas se plasmará el diseño del modelo adoptado y la relación entre las diferentes clases creadas.

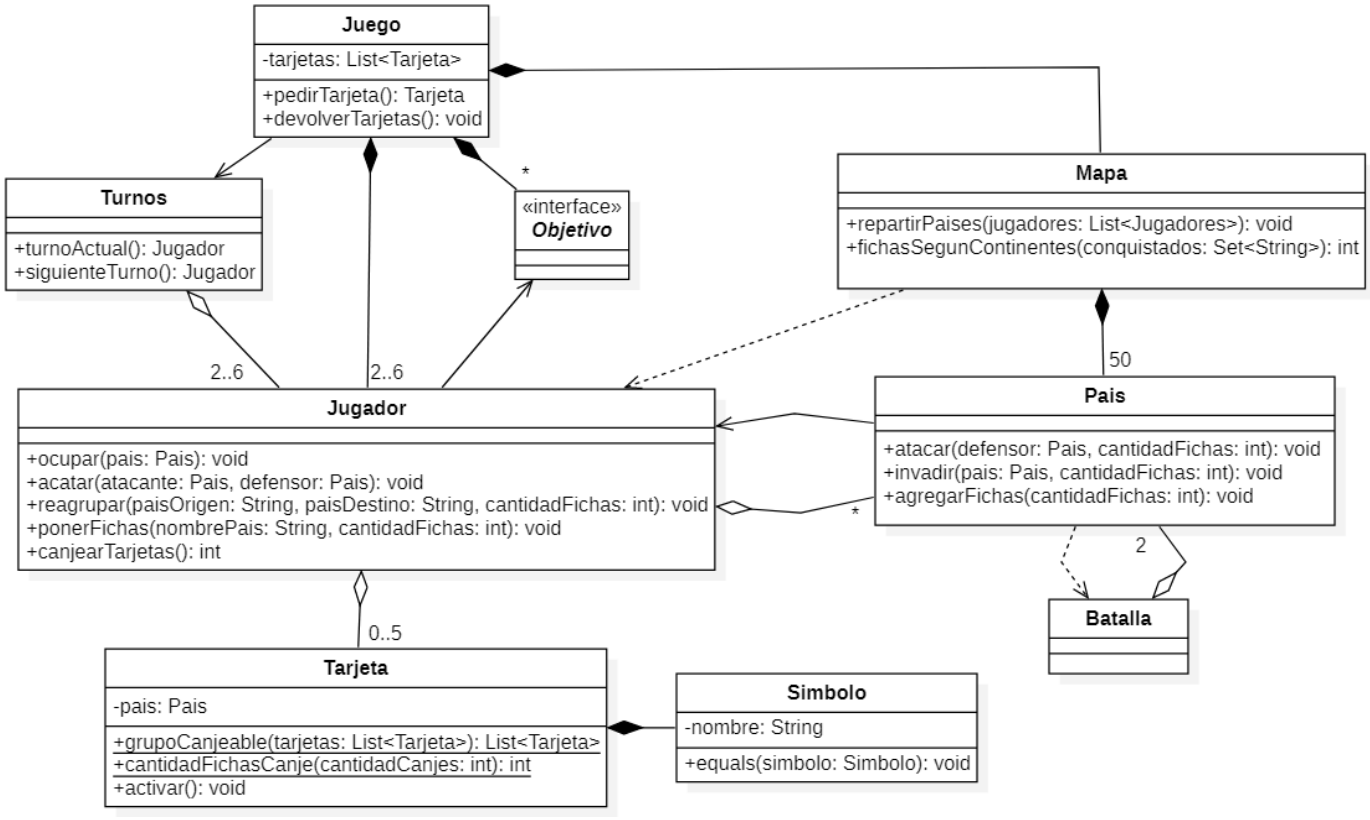


Figura 1: diagrama general de clases

Por otro lado, la interfaz objetivo y las clases que la implementan son:

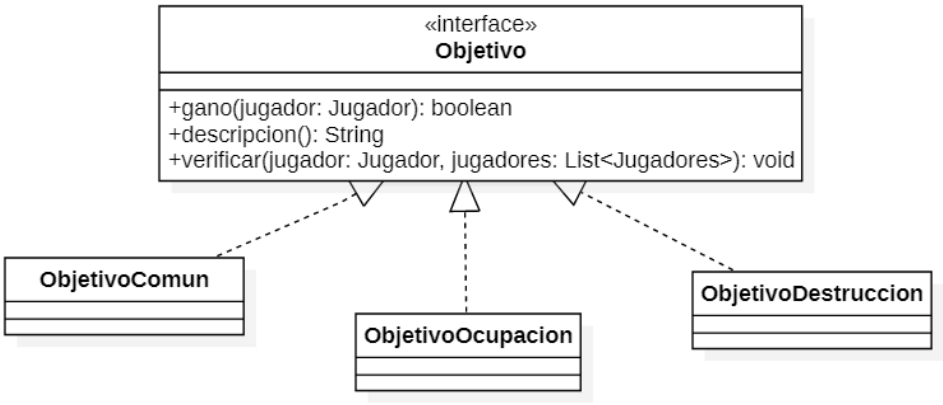


Figura 2: interfaz objetivo

También, se implementaron diferentes clases para mantener constancia de las fases y estado actual del juego:

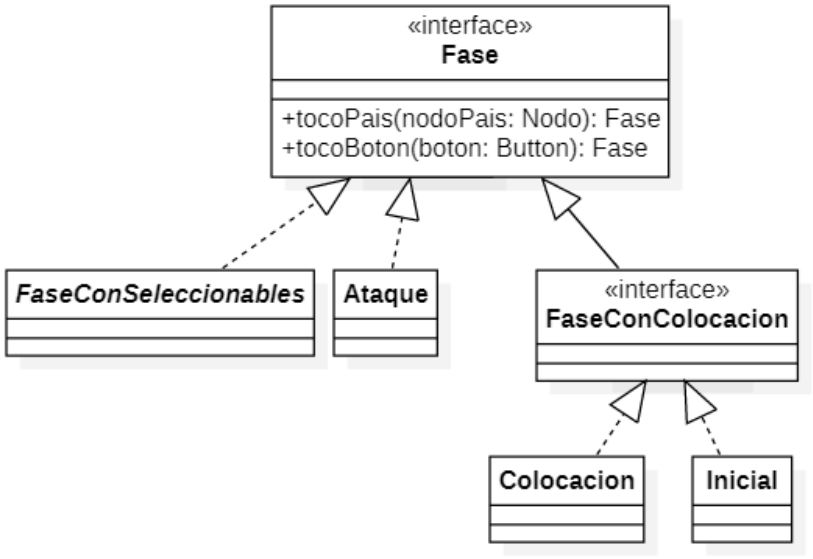


figura 3: interfaz fase

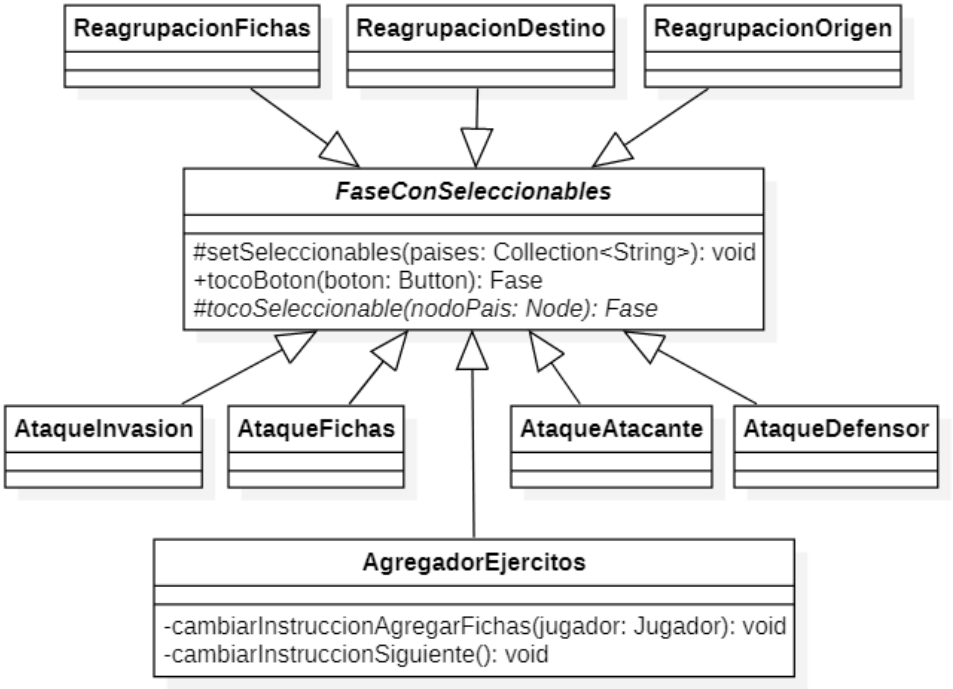


figura 4: interfaz FaseConSeleccionables

Por último, en el siguiente diagrama se presentan las vistas y controladores creados para completar la parte gráfica del trabajo práctico:

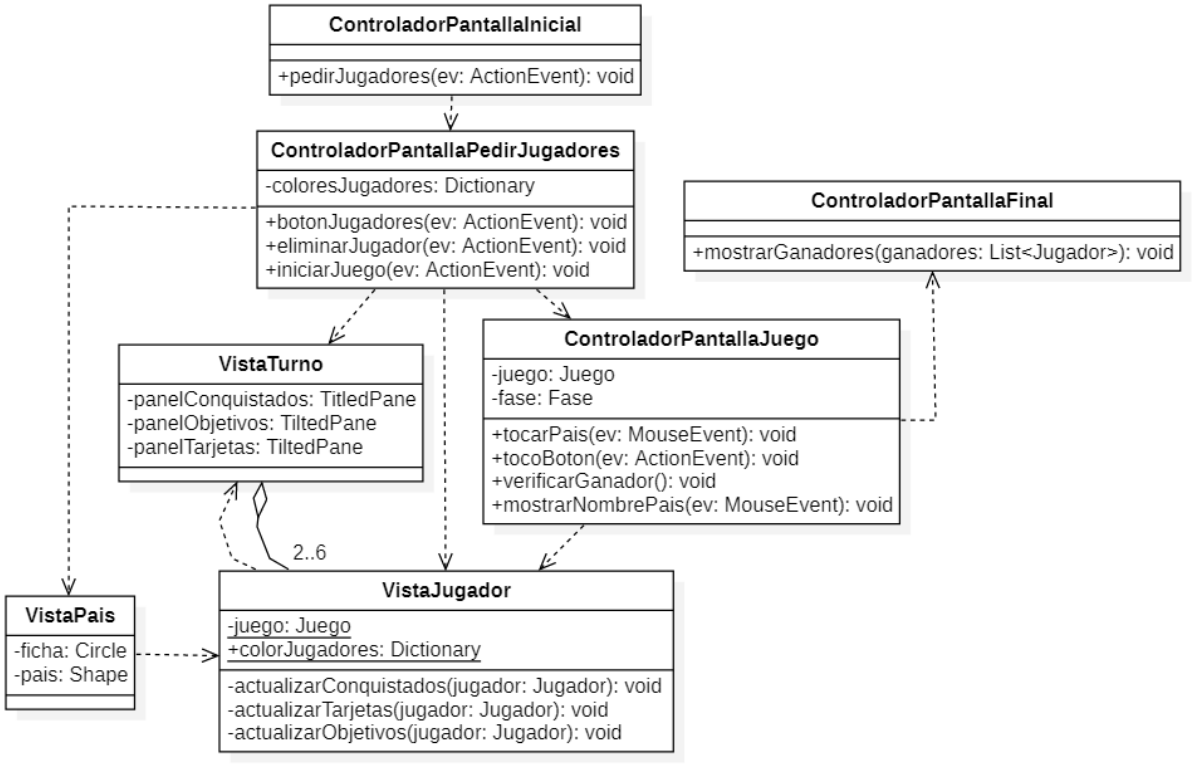


figura 5: vistas y controladores

4. Diagramas de secuencia

Inicialización

El próximo diagrama pretende explicar que sucede una vez creado el juego, como cada uno de los objetos necesarios se crean a partir de la información brindada por una serie de archivos.

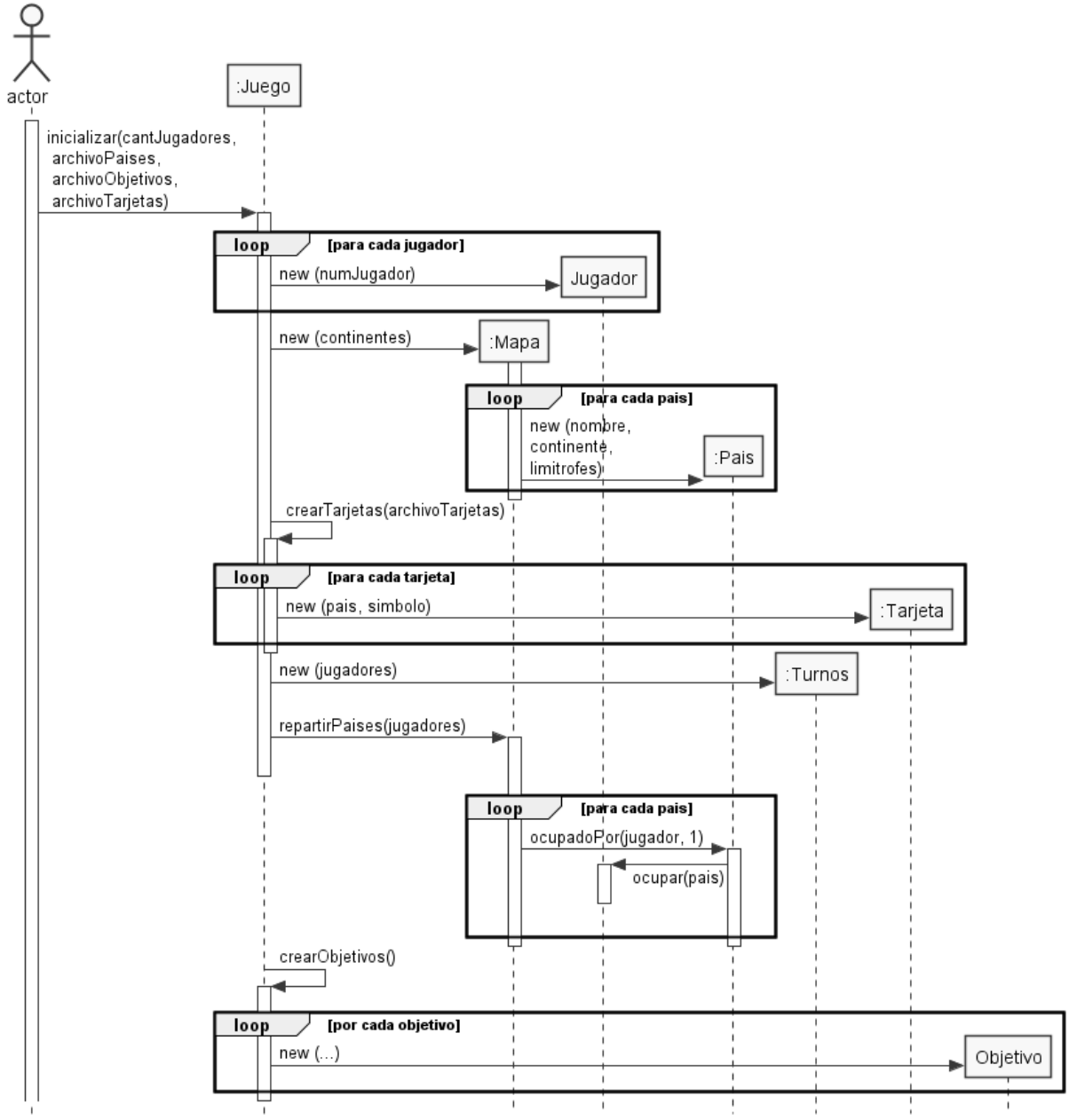


figura 6: inicialización

Ronda de ataque: atacante gana la batalla y obtiene tarjeta

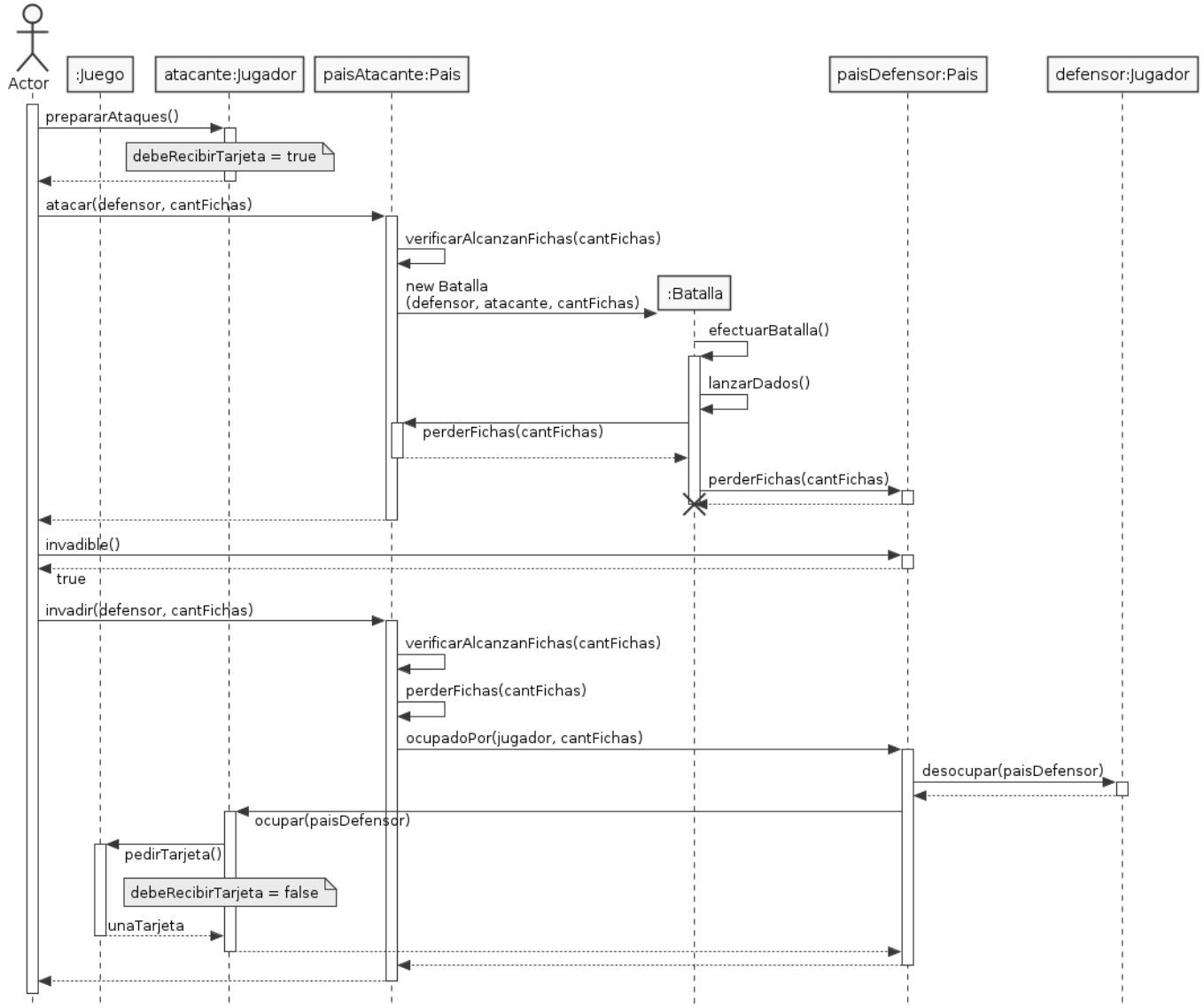


figura 7: ronda de ataque, gana atacante

Ronda de Colocación: Jugador canjea tarjetas

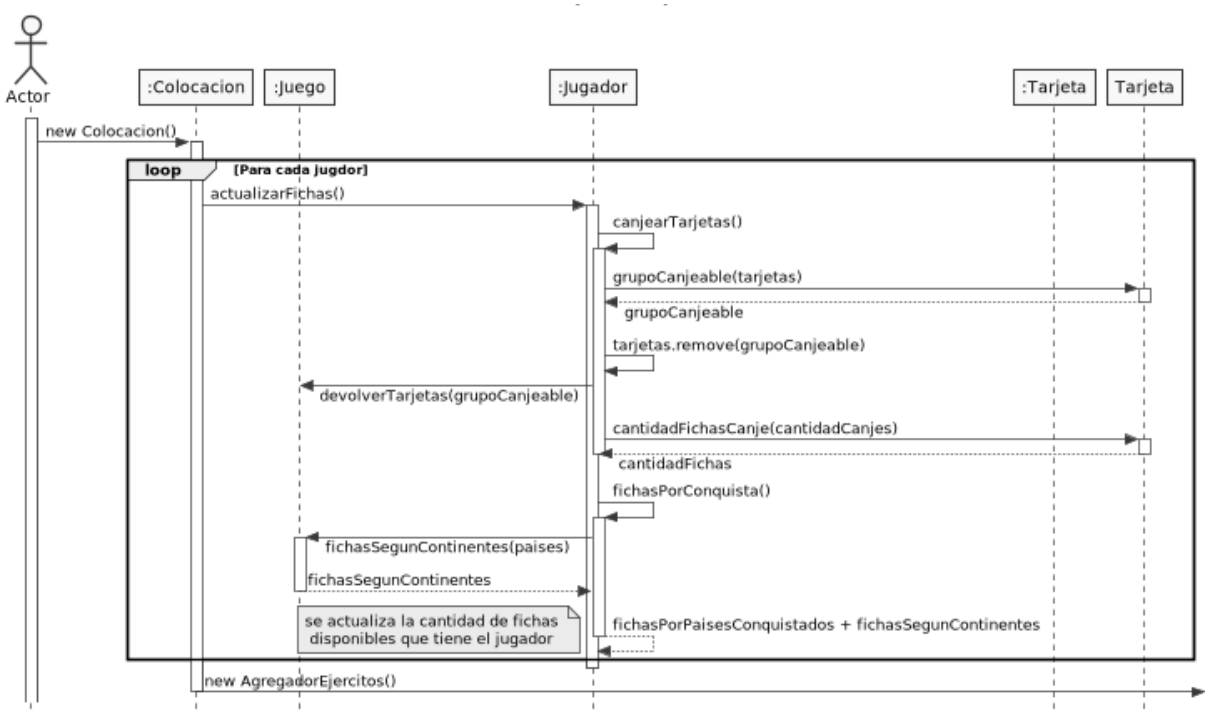
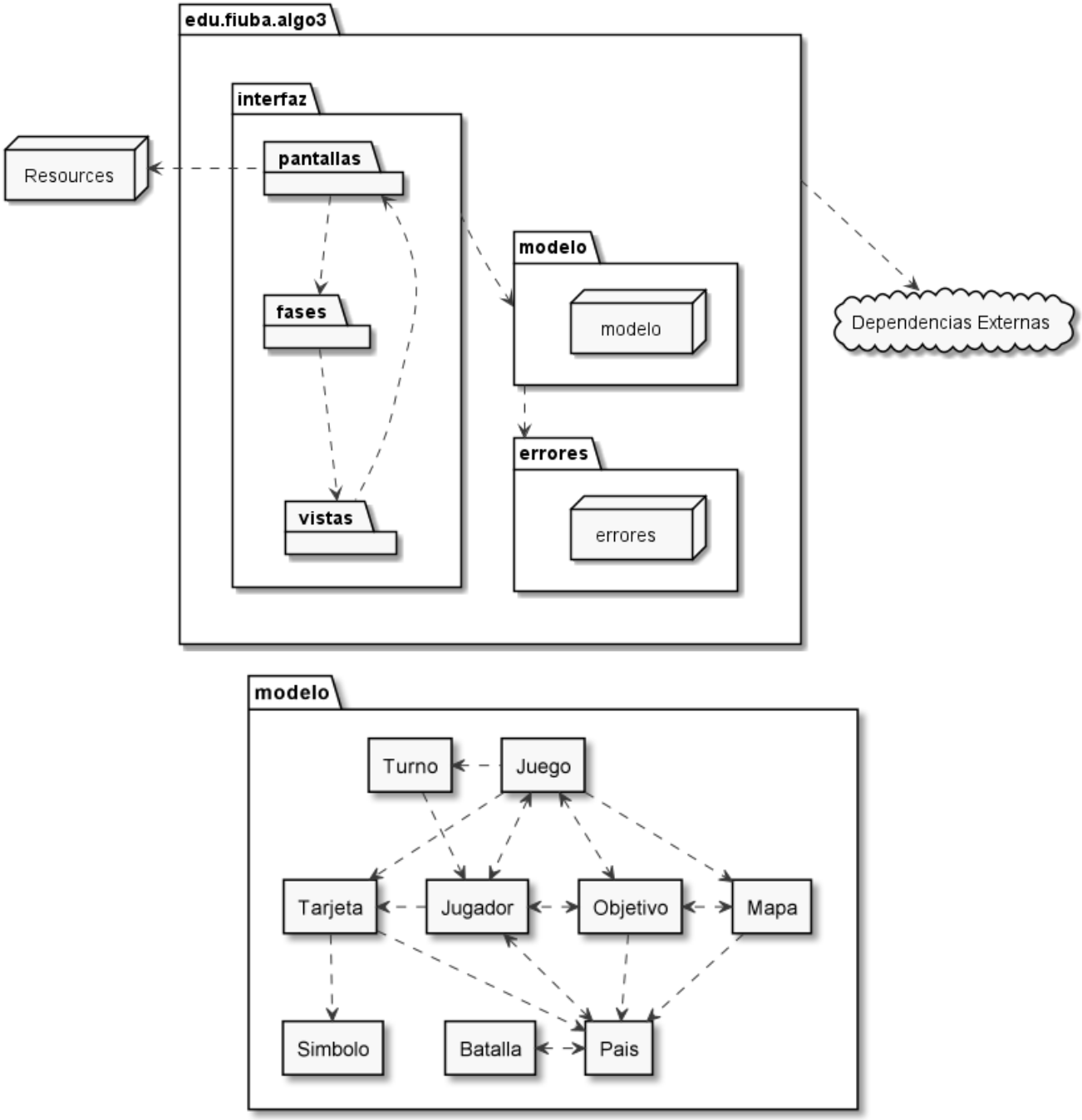
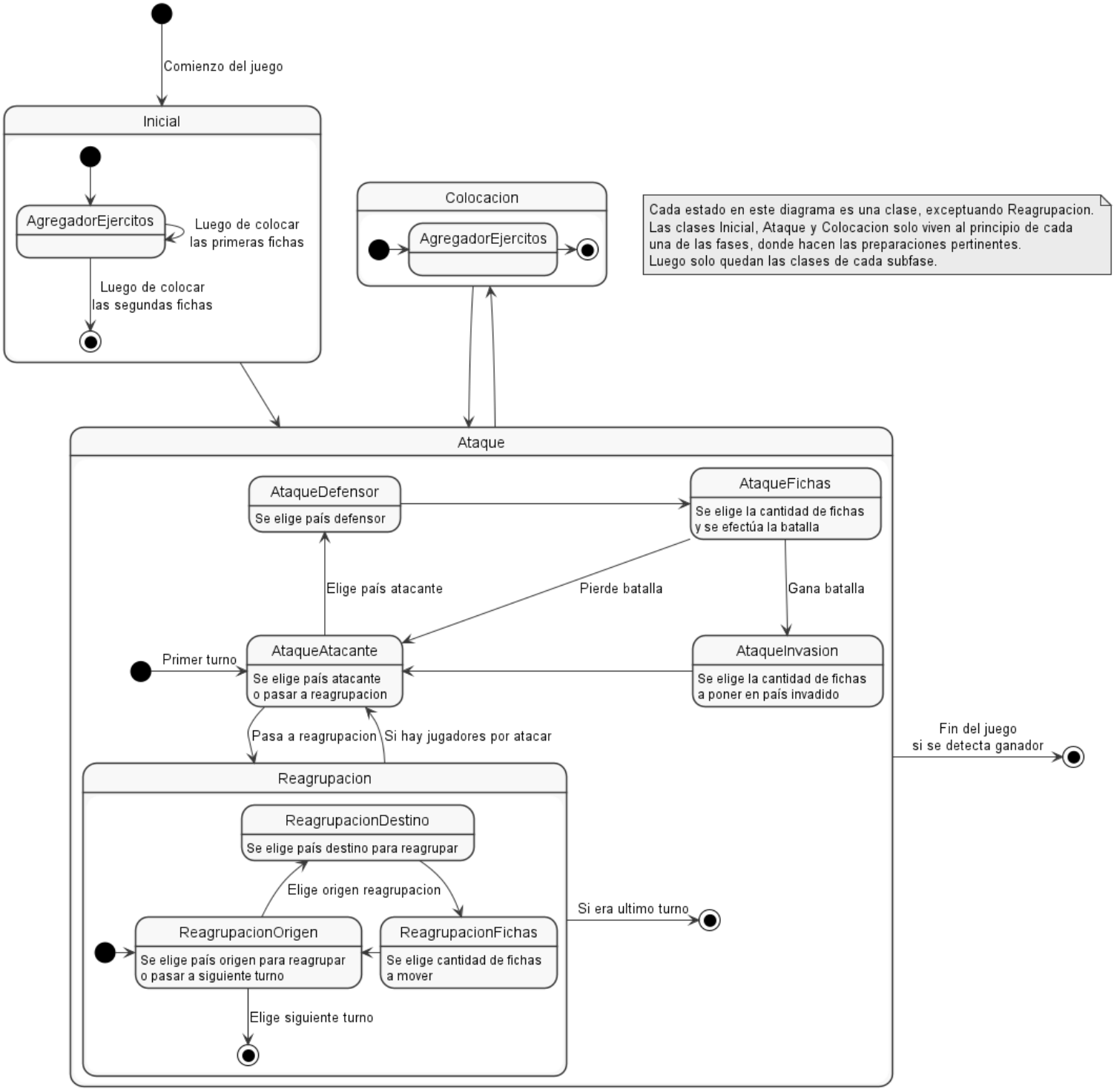


figura 7: ronda de colocación con canje

5. Diagrama de paquetes



6. Diagramas de estado



7. Detalles de implementación

Fases de juego

Las distintas fases del juego mostradas en el diagrama de estados están implementadas por medio de la interfaz *Fase*. Por medio de esta, el controlador de la pantalla principal (*ControladorPantallaJuego*) aplica un patrón State abstrayéndose de la fase actual del juego. Cada vez que se clickea un botón o un país, este controlador le delega el comportamiento a cada una de las fases para que realicen las gestiones necesarias mediante polimorfismo.

Los métodos públicos de *Fase* devuelven otra *Fase*, de manera tal que cuando cada vez que se usa uno de sus métodos el controlador sobreescribe su fase actual, de manera tal que las fases tienen libertad para permanecer en la actual o avanzar a la fase siguiente correspondiente.

Otra particularidad de estas fases es la clase abstracta *FaseConSeleccion*ables. Esta clase implementa el comportamiento de remarcar en el mapa los países que pueden ser clickeables en ese momento, ignorando todos los demás. Como se utiliza en varias fases, todas clases hermanas, decidimos hacer uso de herencia para no repetir este código. Hay algunas fases que no tienen esto, como *Ataque* o *Inicial* que solo se encargan de preparar a las demás subfases al principio del ataque o inicio del juego.

Patrón Observer

Para la mayor parte de la vista se utiliza el patrón observer. La única dependencia que tiene el modelo respecto a la vista es con la clase Observer, y mediante diversos métodos de la clase Juego desde la interfaz agregamos observadores a varias partes del modelo. De esta manera cuando el modelo se actualiza podemos actualizar también su vista. El ejemplo más claro vendría a ser con cosas como el dueño de un país y su cantidad de fichas, que al actualizar uno de esos datos país notifica a todos sus observadores. Las clases observables heredan también de la clase *Observable*.

Polimorfismo en Objetivos

Las clases que implementan la Interfaz *Objetivo* responden a *gano()* verificando que se cumpla el objetivo correspondiente a cada una de ellas y a *descripcion()* con un texto que describe el objetivo a cumplir.

El principal uso de *verificar()* se da en la clase *ObjetivoDestruccion,* el cual no puede contener como objetivo a destruir al mismo jugador al cual se asignó como objetivo en el juego (ya que en ese caso un jugador debería destruirse a sí mismo), por lo tanto debe verificarse la validez del objetivo una vez asignado a un jugador, en caso de ser inválido se cambia el objetivo por el jugador siguiente; las clases *ObjetivoComun* y *ObjetivoOcupacion* no cumplen ninguna función al recibir este método.

Modelo-Vista-Controlador

Se aplica el patrón MVC para separar responsabilidades relacionadas con la interacción con

el usuario, en el proyecto los 3 componentes están separados e interactúan entre si del siguiente modo:

Modelo: Se encuentra en el paquete modelo e interactúa con la vista mediante el patrón observer.

Vista: Se encuentra en en el paquete interfaz.vista y en los Recursos del programa (Resources). Las vistas son los objetos observadores del modelo, que actualizan la interfaz al detectar un cambio. En los recursos podemos encontrar los archivos .fxml que definen la estructura de la interfaz y los archivos .css que definen su estilo. El empleo de estos diferentes archivos nos permite separar la estructura, el estilo y la lógica de la interfaz.

Controlador: Se encuentra en los paquetes interfaz.pantallas e interfaz.fases. El primero se encarga de definir la API para capturar eventos disparados por la vista y delega el comportamiento de estos eventos al segundo, que maneja el flujo del programa.

Implementación del Mapa

Buscamos implementar la interacción el mapa del método más sencillo posible sin sacrificar la experiencia de usuario. Para ello optamos por crear el mapa como un archivo SVG que nos define las entidades mediante etiquetas y nos permite definir la interacción con ellas. Luego convertimos estas imagenes a archivos .fxml (conversor) para poder integrarlas al proyecto y definimos los eventos pertinentes en el controlador de vista del juego.

8. Excepciones

- Excepciones de la clase **País**
  - ***PaísDelMismoPropietarioNoPuedeSerAtacado***: Esta excepción se lanza en el método *atacar()* en caso de que el propietario del país atacante sea el mismo que el del país defensor.

  - ***PaísNoExiste***: Esta excepción se lanza en el metodo *obtenerPais()* de Mapa en caso de no encontrarse el país en él.

  - ***PaísNoPuedeInvadirAPaísNoVecino***: Esta excepción se lanza en el método *invadir()* de País cuando el país destino no es vecino a éste.

  - ***PaísNoPuedeReagruparAPaísNoVecino***: Esta excepción se lanza en el método *reagruparA()* de País cuando el país destino no es vecino a éste.

  - ***PaísNoTieneFichasSuficientes***: Esta excepción se lanza en los métodos *atacar()* y *moverEjercitos()* cuando el país no tiene suficientes fichas para realizar dicha acción.

  - ***PaísSoloPuedeAtacarVecinos***: Esta excepción se lanza cuando se ejecuta el método *atacar()* sobre un país defensor que no fue anteriormente informado como vecino del país atacante con el método *agregarVecino()*.

- Excepciones de la clase **Jugador**
  - ***JugadorNoTienePais****:* Esta excepción se lanza cuando un jugador quiere acceder (atacar con / agregar fichas) a un país que no ha conquistado.

  - ***JugadorNoTieneFichasSuficientes***: Esta excepción se lanza cuando un jugador quiere atacar a un país enemigo con una cantidad de fichas inválida.

- Excepciones de la clase **Turnos**
  - ***TurnoInválido****:* Esta Excepción se lanza en los métodos *turnoActual()* y *siguienteTurno()* en caso de que se pida un jugador antes de pedir el primer turno y en caso de que no haya más jugadores respectivamente.