FACULTAD DE INGENIERIA
Universidad de Buenos Aires

Trabajo Práctico 2 — Gwent

[7507/9502] Paradigmas De la Programacion
Primer cuatrimestre de 2025

Alumno	**Número de padrón**	**Email**
Sebastian Colazo	111737	scolazo@fi.uba.ar
Nahuel Giner	111884	nginer@fi.uba.ar
Aksel Mendoza	108171	aemendoza@fi.uba.ar
Miguel Zorrilla	110619	mzorrilla@fi.uba.ar
Iñaki Vydra	111505	ivydra@fi.uba.ar

Índice

1. Introducción

El presente informe reune la documentación de la solución del segundo trabajo práctico de la materia Paradigmas De la Programacion que consiste en desarrollar una aplicacion completa del juego de cartas **Gwent**, aplicando todos los conceptos vistos en el curso, utilizando el lenguaje Java con un diseño del modelo orientado a objetos y trabajando con las tecnicas de TDD e Integración Continua.

2. Detalles de implementación

2.1. Supuestos

- **Médico:** El modificador *Médico* permite agarrar una carta de la pila de descarte y jugarla en el momento. En nuestra implementación, se asume que la carta que se recupera es siempre la última unidad descartada. Esta elección simplifica la interacción y mantiene la dinámica del juego fluida. En la interfaz gráfica, se muestra la pila de descarte de cada jugador, lo que permite visualizar claramente cuál será la carta que será revivida.

- **Nombres:** Las cartas, tanto especiales como de unidad, tienen un nombre asociado. Para poder crear las vistas de las cartas, asumimos que todas las cartas con el mismo nombre deben representarse de la misma manera, es decir, con la misma descripción e imagen. Esta suposición nos permitió asociar un estilo visual único a cada nombre, simplificando la construcción de vistas carta.

2.2. Puntos conflictivos

Uno de los aspectos más complejos del trabajo práctico fue la programación de las secciones del tablero de cada jugador y la lógica de las unidades, ya que el puntaje de una sección depende de las unidades que contiene, pero a su vez, el puntaje de cada unidad puede depender del estado actual de la sección en la que está colocada. Por ejemplo, el puntaje de una unidad puede verse modificado si en la misma sección hay una unidad animadora, o si la sección está afectada por cartas especiales como *climas* o *morale boost*. Para resolver este problema, decidimos que sea la sección calcule su puntaje como la suma de los puntajes de sus unidades colocadas y dejar que cada unidad calcule su puntaje pasandole las unidades y efectos en la seccion en ese momento. De esta forma, la sección no necesita conocer los tipos específicos de unidades ni los efectos que la afectan; simplemente almacena sus componentes y delega el cálculo a quienes los conocen, lo cual es fundamental para que el diseño sea extensible.

2.3. Patrones de diseño

- **Observer:**
  Se utilizó el patrón *Observer* para actualizar las vistas correspondientes a la mano, las secciones del tablero y los mazos de los jugadores. Este patrón nos permitió suscribir las vistas a las clases del modelo, de modo que puedan ser notificadas automáticamente ante cambios relevantes, como por ejemplo cuando un jugador roba cartas de su mazo o cuando se agrega una unidad a una sección del tablero.

- **Singleton:**
  El patrón *Singleton* fue aplicado para implementar una especie de *cache* de constructores de vistas de cartas. Durante el proceso de parseo del GWENT.JSON con los datos de las cartas, se crea una instancia del modelo correspondiente a cada carta. Sin embargo, ciertos atributos como la descripción y el tipo de una carta especial o la imagen de las cartas son datos que se deben mostrar en la vista, pero que no deberían formar parte del modelo, ya que lo contaminarían con información propia de la interfaz. Para resolver este problema,

definimos estilos de vista para cada carta, que encapsulan esta información visual y saben construir una vista de carta a partir de una instancia del modelo. Utilizamos el nombre de la carta como identificador (ID), y al leer el JSON cargamos en `CacheEstilosVistaCarta` las configuraciones necesarias para representar visualmente cada carta según su nombre (ID). Era necesario que esta cache tuviera una única instancia accesible desde cualquier vista que necesitara mostrar cartas, por lo que la utilización del patrón *Singleton* resultó una solución adecuada.

- **Template + Decorator:**
  Crear las unidades y sus modificadores fue uno de los desafíos más complejos del trabajo, ya que constituyen uno de los aspectos centrales del juego. Un buen modelo debía permitir extender fácilmente la cantidad de modificadores que se pueden aplicar a las cartas. Inicialmente probamos enfoques como el patrón *Strategy*, permitiendo que cada modificador implemente su propia lógica sobre cómo debe jugarse la carta. Sin embargo, optamos por utilizar el patrón *Decorator*, ya que nos permitía tomar una unidad base y agregarle funcionalidades adicionales a sus métodos, lo cual encajaba perfectamente con la idea de los modificadores. Como ventaja adicional, nos permitió combinar múltiples modificadores de forma muy sencilla, simplemente decorando una unidad ya decorada. Complementamos esta solución con el patrón *Template Method*, que define una estructura base para el comportamiento general de una unidad al ser jugada. Cada modificador puede intervenir decorando sólo las partes relevantes de esa estructura, permitiendo un código limpio y modular, donde cada clase de modificador declara únicamente lo que difiere del comportamiento base.

Listing 1: Ejemplo de patrón Template + Decorator en una carta con modificador Espia

```
// En la interfaz Unidad
@Override
default void jugarCarta(Jugador jugador, Jugador oponente, Posicion posicionElegi
    if (!sePuedeColocar(posicionElegida)) {
        throw new UnidadNoPuedeSerJugadaEnEsaPosicion("");
    }
    Atril atrilDestino = atrilDestino(jugador, oponente);
    atrilDestino.colocarUnidad(this, posicionElegida);
    realizarAccionAdicional(jugador, oponente, atrilDestino, posicionElegida);
}

// En el decorador Espia
@Override
public Atril atrilDestino(Jugador jugador, Jugador oponente) {
    return oponente.getAtril(); // Coloca la unidad en el campo del oponente
}

@Override
public void realizarAccionAdicional(Jugador jugador, Jugador oponente,
                                    Atril atril, Posicion posicionElegida) {
    jugador.robarCartasDelMazo(CANTIDAD_DE_CARTAS_PARA_ROBAR);
    super.unidad.realizarAccionAdicional(jugador, oponente, atril, posicionElegid
}
```

El ultimo llamado a super.unidad.realizarAccionAdicional(...) se utiliza para continuar la cadena en el caso de que la carta tenga mas modificadores.

- **Factory Method:**
  El patrón *Factory Method* fue utilizado para proporcionar una interfaz clara y flexible a la hora de crear unidades del juego. Debido a la posibilidad de combinar múltiples modificadores, la construcción manual de una unidad podía volverse confusa o tediosa.

Por ejemplo, si deseamos crear una unidad de fuerza 5, ubicada en la posición Cuerpo a Cuerpo, con los modificadores de Ágil (en posición Asedio), Espía y Legendaria, sin una fábrica tendríamos que escribir algo como lo siguiente:

Listing 2: Ejemplo de creación de unidad sin usar UnidadFactory

```
Unidad nuevaUnidad = new UnidadBasica("nombre", 5, new CuerpoACuerpo());
Unidad unidadAgil = new Agil(nuevaUnidad, new Asedio());
Unidad unidadAgilYEspia = new Espia(unidadAgil);
Unidad unidadAgilEspiaYLegendaria = new Legendaria(unidadAgilYEspia);
```

Para simplificar este proceso, implementamos una clase `UnidadFactory` que permite construir unidades utilizando listas de modificadores y posiciones. De esta forma, el mismo ejemplo se puede expresar de manera mucho más legible:

Listing 3: Ejemplo de creación de unidad usando UnidadFactory

```
modificadores = new ArrayList<>(List.of("Agil", "Espia", "Legendaria"));
posiciones = new ArrayList<>(List.of("cuerpo a cuerpo", "asedio"));
Unidad unidad = UnidadFactory.crear("nombre", 5, modificadores, posiciones);
```

3. Diagramas

3.1. Diagramas de clase

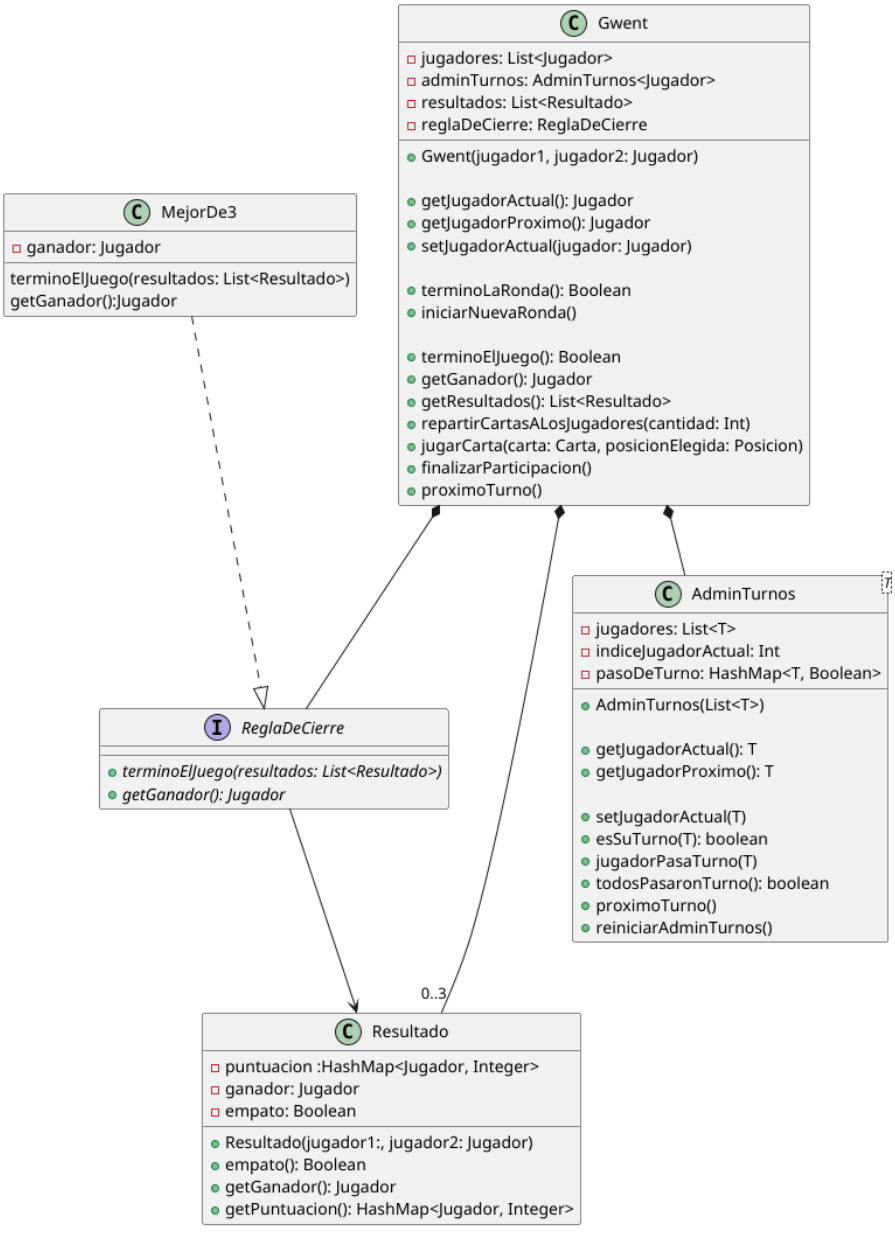

Figura 1: Juego `Gwent`.

En este diagrama se muestran las clases utilizadas para administrar los turnos y las rondas del juego. El `AdministradorDeTurnos` permite detectar si los jugadores han pasado su turno, y cuando ambos lo hacen, se genera un objeto de tipo `Resultado`, que contiene el estado actual de los jugadores, sus puntajes en ese momento y quién ganó la ronda. Luego, esa información se pasa a una `ReglaDeCierre`, que se encarga de determinar si el juego debe finalizar o continuar con una nueva ronda.

Figura 2: Jugador `Gwent`.

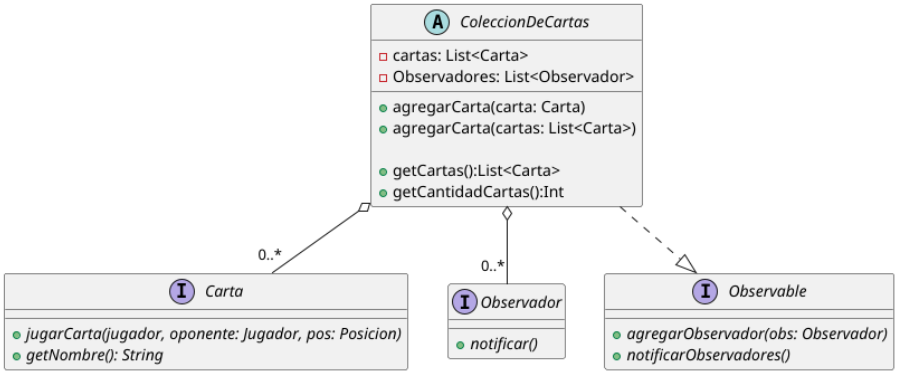

Figura 3: ColeccionDeCartas `Gwent`.

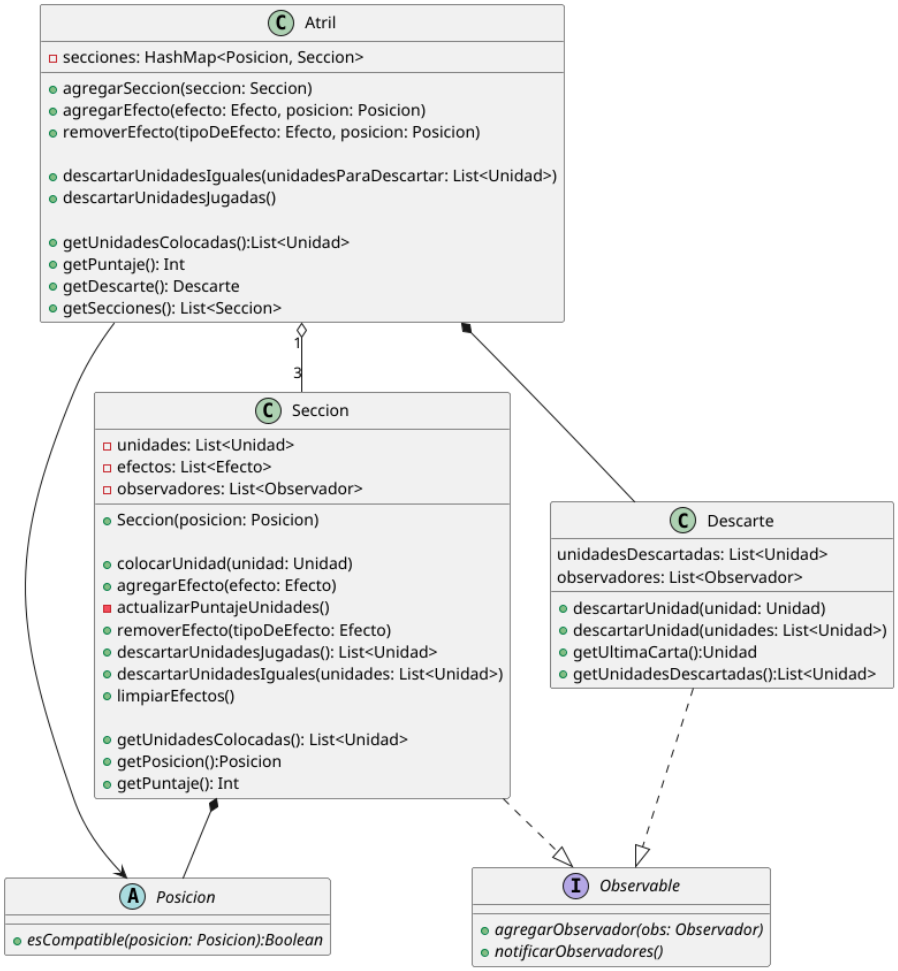

Figura 4: `atril`.

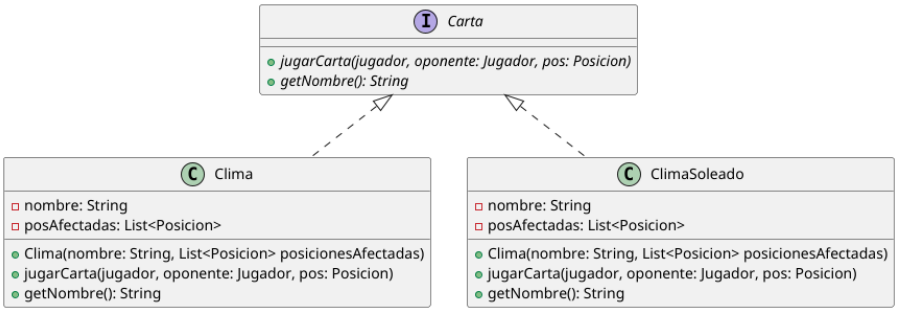

Figura 5: Cartas `Especiales`.

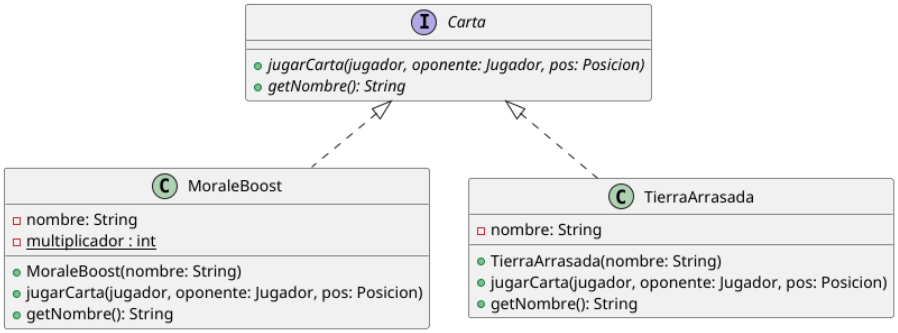

Figura 6: Cartas `Especiales`.

Figura 7: Cartas unidad `UnidadBasica`.

La clase `UnidadBasica` representa una unidad sin modificadores, y funciona como la base sobre la cual pueden aplicarse distintos modificadores. Por otro lado, la clase `UnidadModificada` implementa las interfaces `Unidad` y `Carta`, y contiene una referencia a otra unidad. Esto le permite delegar sus mensajes a la unidad que envuelve y modificar su comportamiento, agregando lógica adicional antes o después de invocar los métodos originales.

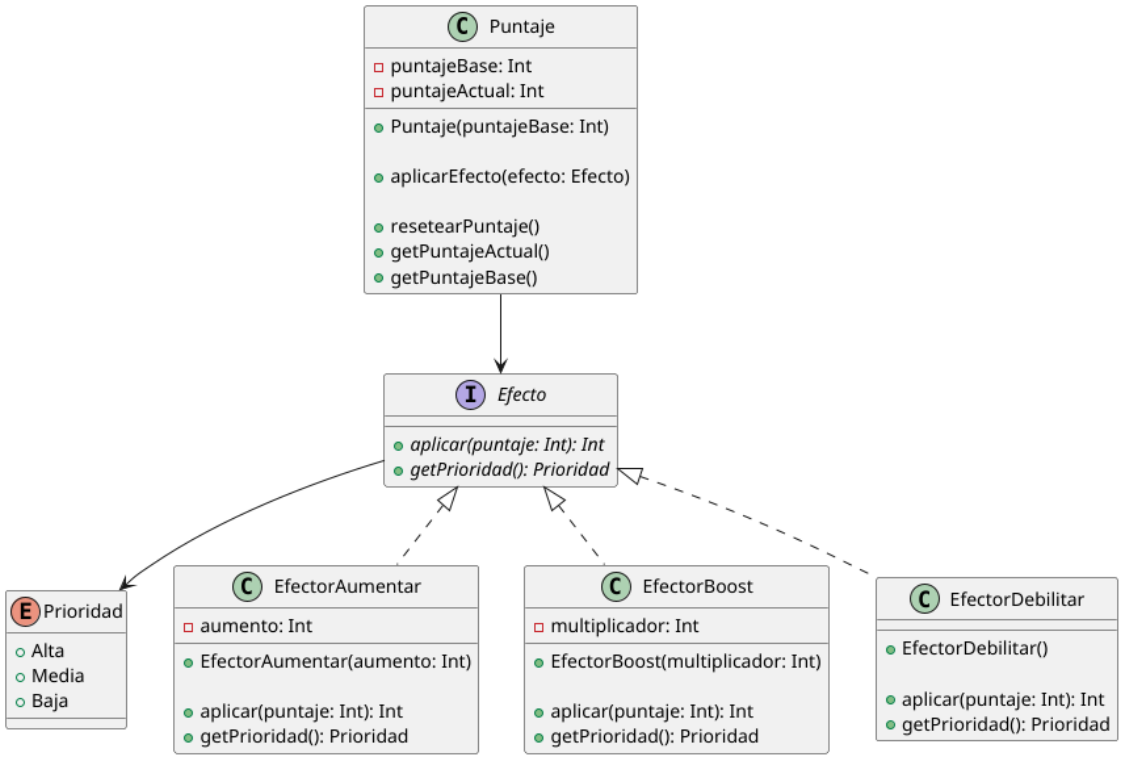

Figura 8: `Puntaje`.

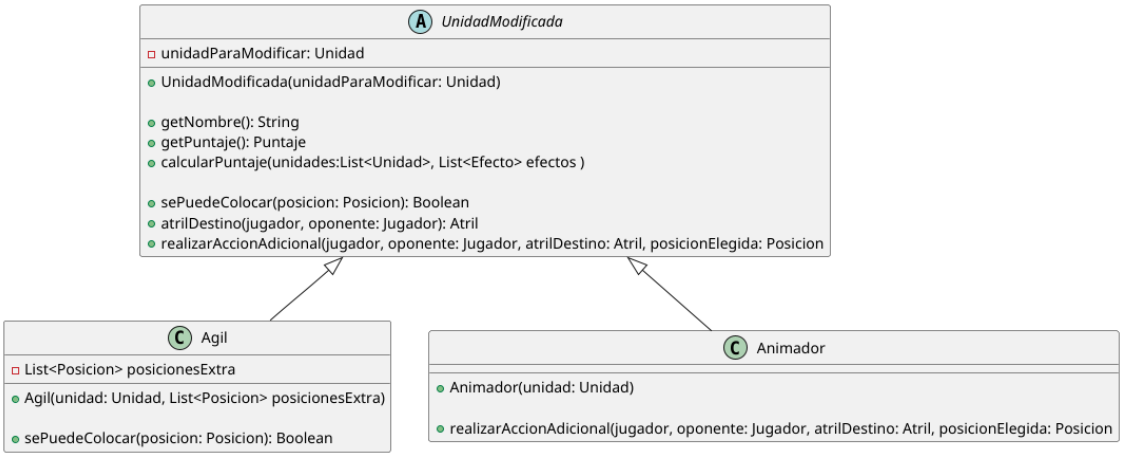

Figura 9: Cartas unidad `Modificadores`.

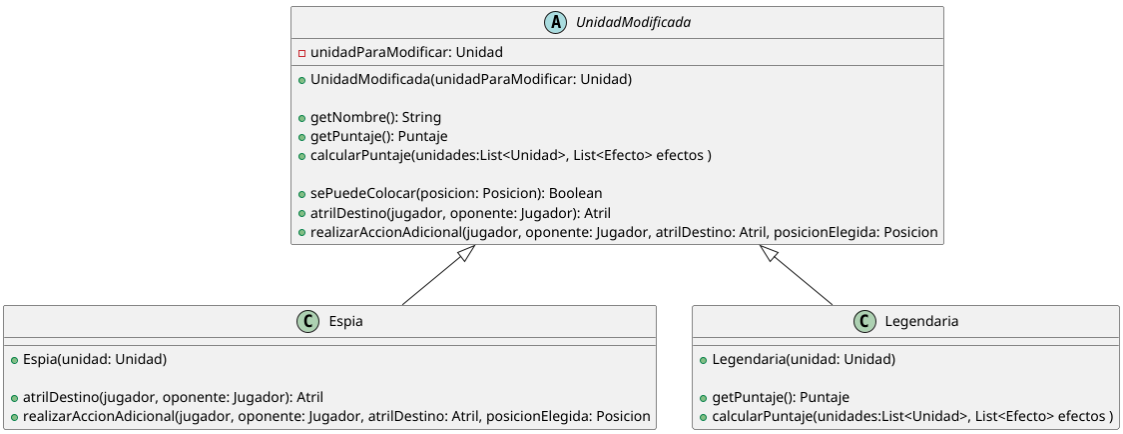

Figura 10: Cartas unidad `Modificadores`.

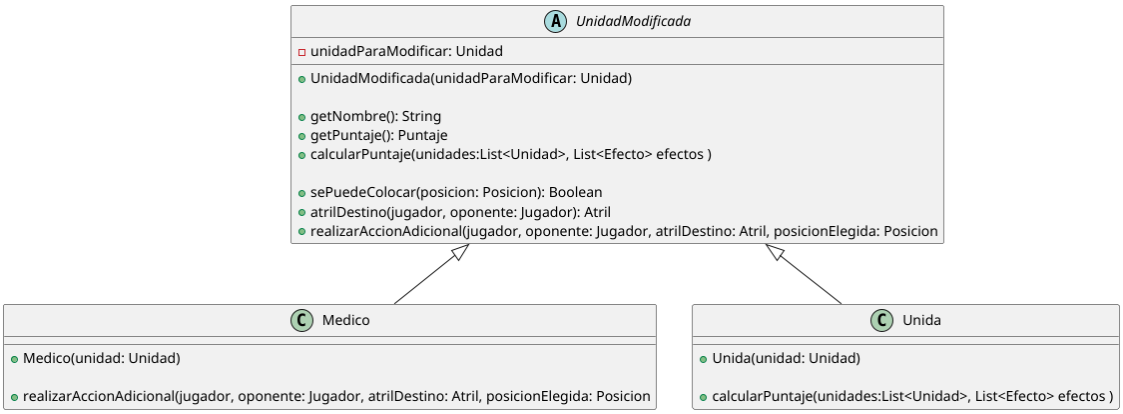

Figura 11: Cartas unidad `Modificadores`.

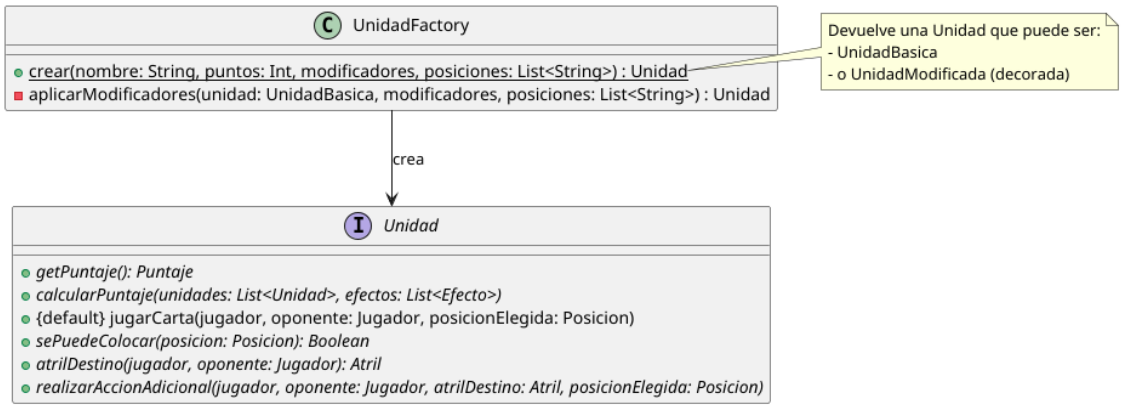

Figura 12: `UnidadFactory`.

3.2. Diagramas de secuencia