

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	2
2. Detalles de implementación	2
2.1. Supuestos	2
2.2. Patrones de diseño	2
3. Diagramas	3
3.1. Diagramas de clase	3
3.2. Diagramas de secuencia	3
3.3. Diagramas de paquetes	3

1. Introducción

El presente informe reúne la documentación de la solución del segundo trabajo práctico de la materia Paradigmas De la Programacion que consiste en desarrollar una aplicación 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 técnicas de TDD e Integración Continua.

2. Detalles de implementación

2.1. Supuestos

2.2. 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 posicionElegida)
```

```

        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, posicionElegida);
    }

```

■ 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

3.2. Diagramas de secuencia

3.3. Diagramas de paquetes