FACULTAD DE INGENIERIA
Universidad de Buenos Aires

Gwent - T.P. 2

[TB025] Paradigmas de Programación
Primer cuatrimestre de 2025

Grupo 07 - Estudiantes:

Nombre	Padrón	Mail
Matias Domine Folco	112268	mdomine@fi.uba.ar
Enrique Heller	111605	eheller@fi.uba.ar
Andy Mayuri	111492	amayuri@fi.uba.ar
Franco Montanelli	111521	fmontanelli@fi.uba.ar
Andrés Moyano	110017	ammoyano@fi.uba.ar

Tutor: Joaquín Pandolfi

Índice

1. Supuestos

A continuación se detallan algunos supuestos y decisiones de diseño relevantes para la implementación:

- **Cartas Especiales como modificadores:** Las Cartas Especiales tienen la capacidad de modificar aquellas cartas que se encuentren distribuidas por el tablero. Los modificadores que pueden ser atribuidos a las Cartas Unidad también pueden afectar el tablero en general. Por eso, las cartas especiales implementan la interfaz de modificador, definiendo cada una cómo aplica su efecto.
- **Lógica de modificador Médico:** El modificador Médico puede tomar la última carta de la pila de descarte y jugarla en la primera sección válida del jugador.
- **Carta especial Tierra Arrasada:** Al activarse, afecta a las secciones de ambos jugadores.
- **Lógica modificador Espía:** si se juega espía cuando termine la ronda va al descarte del jugador donde se colocó (el contrario al que la jugó).
- **Carta Moral Boost:** La carta Moral Boost, será la única carta especial que excepcionalmente podrá jugarse en una sección. Esto se debe a que su efecto tendrá que ser aplicado en una sección en particular.
- **Flujo de juego:** Se asume que los jugadores pueden jugar tantas cartas como deseen en su turno, hasta decidir pasar. Luego, el siguiente jugador hace lo mismo. Cuando ambos pasan, finaliza la ronda.

Ejemplo de flujo de ronda:

- **Jugador 1:** coloca una carta A en la sección Cuerpo a Cuerpo
- **Jugador 1:** coloca una carta B en la sección Rango
- **Jugador 1:** decide terminar su turno
- **Jugador 2:** coloca una carta C en la sección Cuerpo a Cuerpo
- **Jugador 2:** coloca una carta D en la sección Rango
- **Jugador 2:** coloca una carta E en la sección Asedio
- **Jugador 2:** coloca una carta F en la sección Rango
- **Jugador 2:** decide terminar su turno
- Fin de la ronda

2. Diagramas de clase

Para un entendimiento completo de cómo se intercomunican las distintas entidades definidas como objetos en nuestro modelo, a continuación se presentan los diagramas de clase y sus descripciones:

2.1. Diagrama general

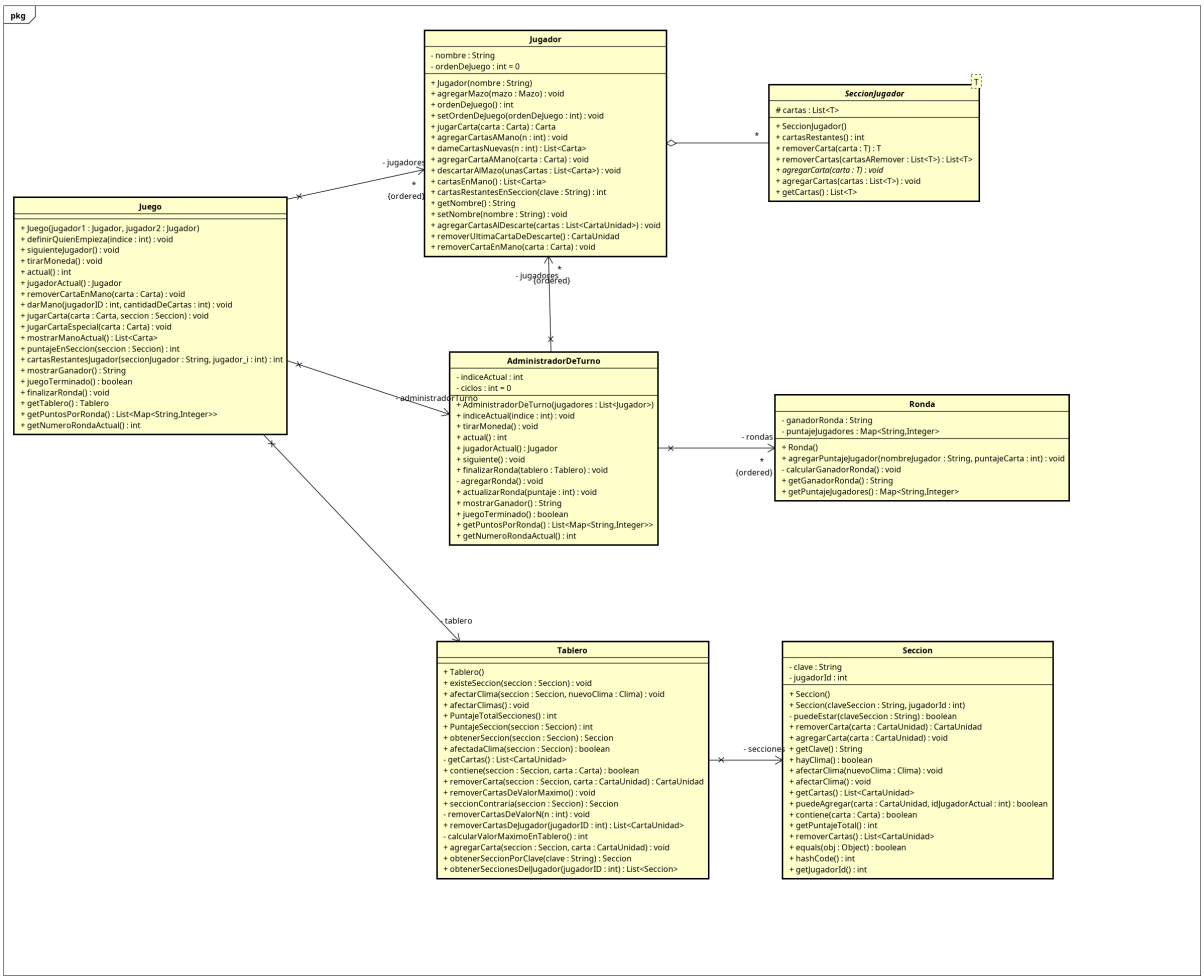

Figura 1: Diagrama general del modelo.

En este diagrama general podemos ver una primera aproximación de como **Juego** actúa como fachada en nuestro modelo. En este caso notamos la relación con sus tres "hijos"que son los que se harán cargo de abstraer de lo que se ocupa cada uno dentro del juego en sí: **AdministradorDeTurno**, **Jugador** y **Tablero**.

- **AdministradorDeTurno:** Es el encargado de coordinar el flujo de la partida, gestionando los turnos de los jugadores, el avance de las rondas y determinando cuándo finaliza el juego. Su responsabilidad es asegurar que las reglas de alternancia y finalización se cumplan correctamente.
- **Jugador:** Representa a cada participante de la partida. Se ocupa de administrar su mano de cartas, su mazo y pila de descarte. Esto involucran las acciones que puede realizar en su turno, como jugar cartas, pasar o descartar. Además, mantiene el estado individual de cada jugador a lo largo de la partida.
- **Tablero:** Es el espacio común donde se desarrollan las acciones del juego. Se encarga de organizar las secciones en las que los jugadores pueden colocar sus cartas y de mantener el estado global de las cartas jugadas, modificadores y efectos activos sobre el campo de juego.

2.2. Cartas

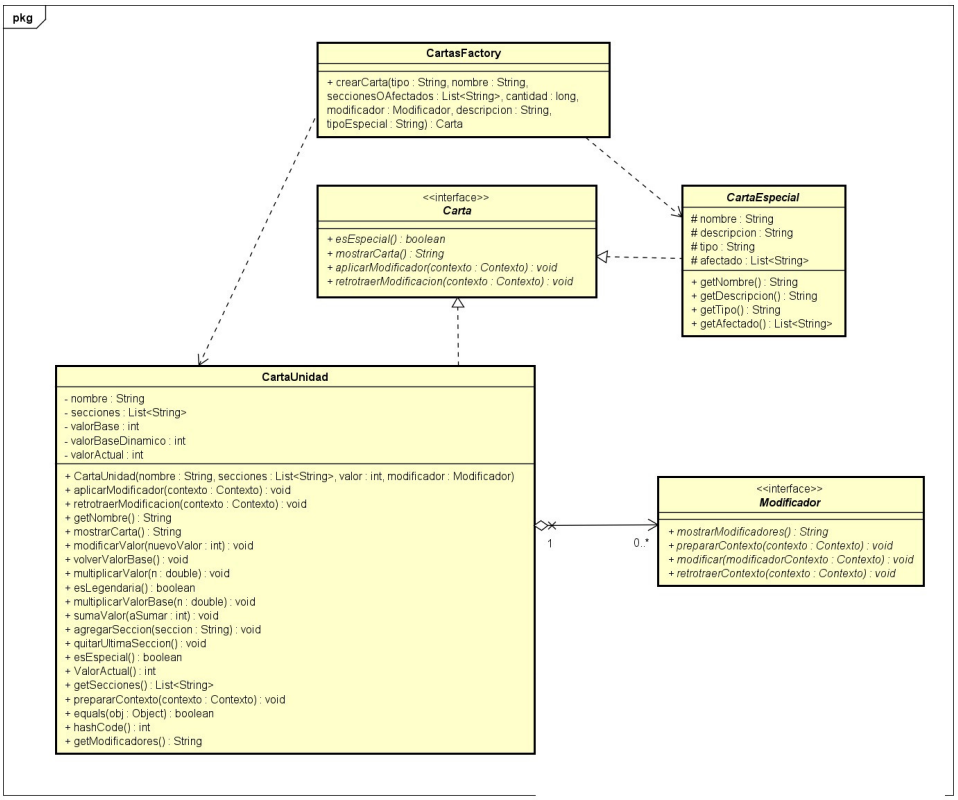

Figura 2: Jerarquía y relaciones entre las distintas clases de cartas.

En este diagrama se aprecia cómo el modelo organiza y abstrae los distintos tipos de cartas y su interacción con los modificadores:

- **CartasFactory:** Es la clase encargada de la creación de instancias de cartas a través del método `crearCarta()`. Permite parametrizar el tipo de carta, nombre, secciones afectadas, cantidad, modificador, descripción y tipo especial, centralizando la lógica de construcción de cartas en el sistema.
- **Interfaz Carta:** Define el contrato común para todas las cartas del juego. Entre sus métodos principales se encuentran `EsEspecial()` (para distinguir cartas especiales), `mostrarCarta()` (representación textual), y los métodos para aplicar o revertir modificadores en un contexto dado.
- **CartasEspecial:** Implementa la interfaz Carta y representa aquellas cartas con efectos globales o especiales, como las de clima o habilidades particulares. Sus atributos permiten identificar el nombre, descripción, tipo y las secciones afectadas. Provee métodos de acceso y lógica para aplicar sus efectos.
- **CartaUnidad:** También implementa la interfaz Carta y modela las cartas de unidad, que poseen estadísticas modificables. Gestiona atributos como nombre, secciones válidas, valor base y valores dinámicos, así como métodos para modificar estos valores y gestionar su interacción con modificadores y secciones del tablero.
- **Interfaz Modificador:** Abstrae los efectos que pueden alterar cartas o el tablero. Define métodos para describir el efecto, aplicarlo o revertirlo en un contexto, y preparar el entorno para la modificación.

Esta estructura valida los supuestos del modelo: por ejemplo, las CartasEspecial implementan Modificador, permitiendo que cartas como Tierra Arrasada afecten múltiples secciones, y CartaUnidad incorpora lógica para modificadores específicos como Moral Boost, que se aplica en una sección concreta.

2.3. Modificadores

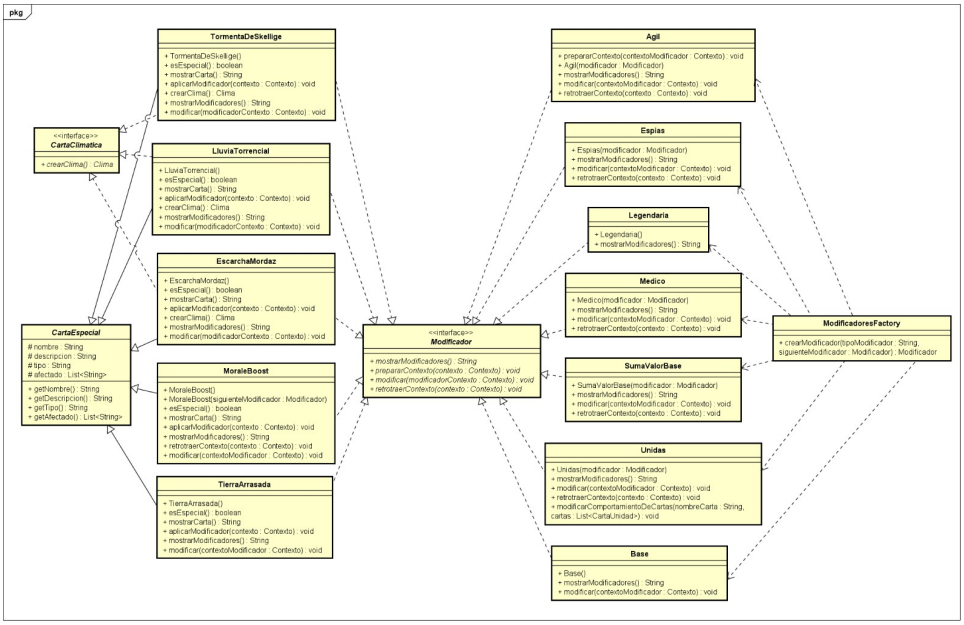

Figura 3: Estructura de los modificadores y su relación con otras entidades.

En este diagrama se observa cómo todos los modificadores que se aplican a unidades implementan la interfaz **Modificador** y son construidos a través de la clase `ModificadoresFactory`. Esto permite centralizar la lógica de creación y aplicar polimorfismo sobre los distintos efectos posibles en el juego.

Por otro lado, se muestra que las cartas especiales (cada una en particular) también implementan la interfaz **Modificador** y heredan de la clase abstracta `CartaEspecial`. Aquellas cartas especiales que son de tipo climático implementan, a su vez, la interfaz `CartaClimatica`, lo que permite agrupar comportamientos y atributos comunes a este tipo de efectos globales.

Entre los métodos más relevantes que se utilizan en este contexto se encuentran:

- `modificar(Contexto contexto)`: Aplica el efecto del modificador sobre el contexto de juego actual (por ejemplo, cambiar el valor de una carta o afectar una sección del tablero).
- `retrotraerContexto(Contexto contexto)`: Revierte el efecto aplicado por el modificador, restaurando el estado anterior del contexto.
- `mostrarModificadores()`: Devuelve una descripción textual del efecto del modificador, útil para mostrar en la interfaz o en logs.
- `prepararContexto(Contexto contexto)`: Permite preparar el entorno antes de aplicar el modificador, por ejemplo, validando condiciones o ajustando referencias.

Esta estructura facilita la extensión del sistema con nuevos modificadores y efectos especiales, manteniendo la coherencia y la reutilización de código.

2.4. Secciones y Tablero

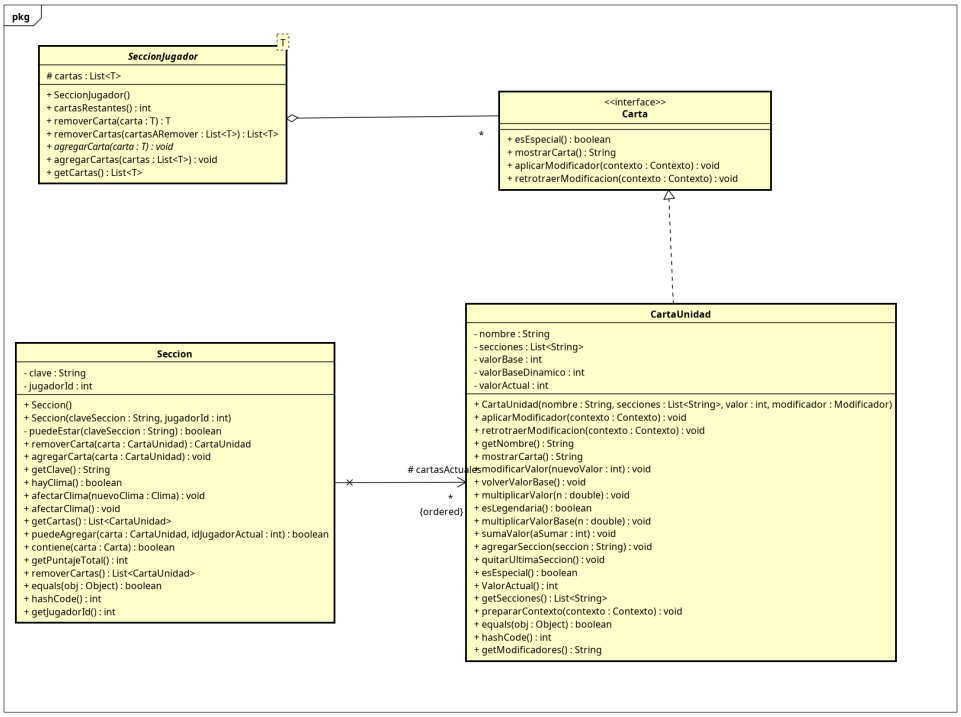

Figura 4: Organización de las secciones del tablero y su relación con cartas y jugadores.

En este diagrama se detallan las responsabilidades y relaciones de las secciones del tablero y su interacción con las cartas:

- **Clase Seccion:** Modela una sección del tablero (por ejemplo, Cuerpo a Cuerpo, Rango o Asedio) y mantiene atributos como la clave identificadora y el jugador dueño de la sección. Sus principales responsabilidades incluyen la gestión de cartas de unidad presentes en la sección (`agregarCarta()`, `removerCarta()`, `getCartas()`, `removerCartas()`), la validación de si una carta puede ser jugada en esa sección según su tipo y el jugador (`puedeAgregar()`), el manejo de efectos de clima (`hayClima()`, `afectarClima()`) y el cálculo del puntaje total de la sección (`getPuntajeTotal()`). Además, provee métodos de utilidad como `contiene()`, `getClave()` y `getJugadorId()`.
- **Contenedores de cartas (SeccionJugador):** Clases como `Mano`, `Mazo` y `Descarte` heredan de una clase abstracta genérica que actúa como contenedor para gestionar colecciones de cartas, permitiendo operaciones como `cartasRestantes()`, `agregarCartas()`, `removerCarta()` y `removerCartas()` para eliminar una o varias cartas.
- **CartaUnidad:** Además de lo presentado en el diagrama de cartas, acá se observa la integración con las secciones: el método `agregarSeccion()` vincula la carta a secciones válidas dinámicamente (por ejemplo, para cartas ágiles) mediante una palabra clave (respetando el formate en el que vienen los JSON), y `ValorActual()` que refleja los cambios dinámicos de puntaje durante el juego, considerando modificadores y efectos de clima.

Esta estructura implementa varios de los supuestos del modelo: la clase Section permite que las cartas se jueguen en secciones específicas (como requiere el supuesto de Moral Boost), el método `afectarClima()` soporta efectos globales como Escarcha o Niebla (alineado con el supuesto de Tierra Arrasada), y `puedeAgregarCartas()` asegura que las cartas sólo puedan jugarse en secciones válidas, reflejando el flujo de juego esperado.

2.5. Contexto

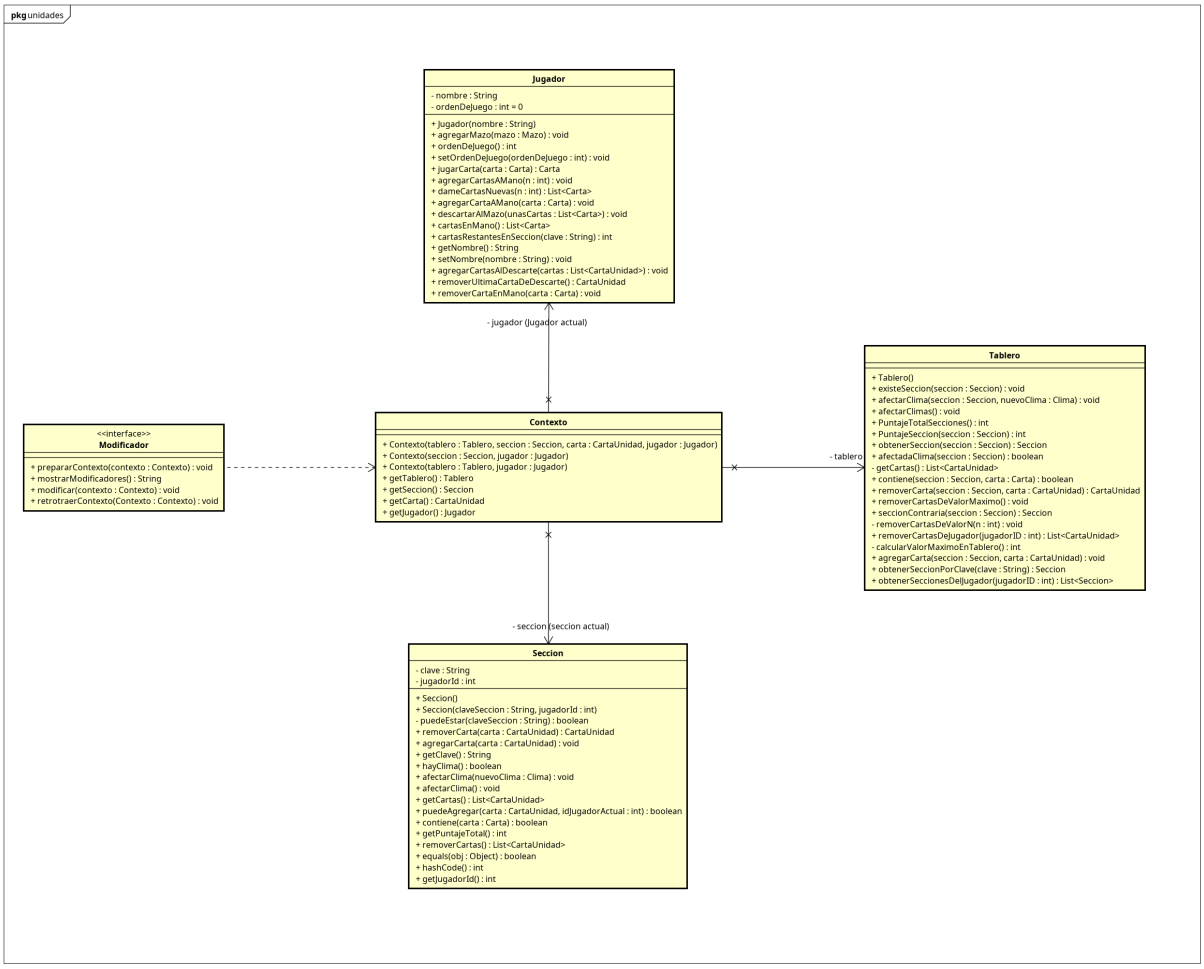

Figura 5: Diagrama de la clase Contexto y su rol en la aplicación de modificadores.

El diagrama ilustra el papel central de la clase **Contexto** en el modelo. Esta clase actúa como un contenedor que provee acceso común y centralizado a los elementos clave del juego: el `Tablero`, la `Seccion` y el `Jugador` actual (cuando corresponde). Su principal utilidad es facilitar la aplicación de modificadores, ya sean cartas especiales o modificadores ligados a una `CartaUnidad`, permitiendo que estos puedan afectar distintos lugares del juego de manera flexible y desacoplada.

Por ejemplo, al invocar `modificar(Contexto contexto)` o `aplicarModificador(Contexto contexto)`, los modificadores reciben toda la información relevante del estado actual del juego.
De este modo, pueden modificar cartas, secciones o el tablero completo según sea necesario. Esto permite que efectos como climas globales, moral boost o modificadores de cartas ágiles se implementen de forma coherente y reutilizable, sin acoplar la lógica a una única clase.

Entre los métodos más relevantes que aprovechan el uso de `Contexto` se encuentran:

- `modificar(Contexto contexto)`: Aplica el efecto del modificador sobre el elemento adecuado (carta, sección o tablero) según el estado provisto por el contexto.
- `prepararContexto(Contexto contexto)`: Permite ajustar o validar el entorno antes de aplicar el efecto, por ejemplo, seleccionando la sección o carta objetivo.
- `aplicarModificador(Contexto contexto)`: Método común en cartas especiales y de unidad para ejecutar el efecto correspondiente usando la información centralizada.

Esta estructura favorece la extensibilidad y el desacoplamiento, permitiendo que nuevos modificadores o efectos especiales puedan interactuar con el modelo sin depender de detalles internos de otras clases.

3. Diagramas de secuencia

3.1. Cálculo de puntaje total

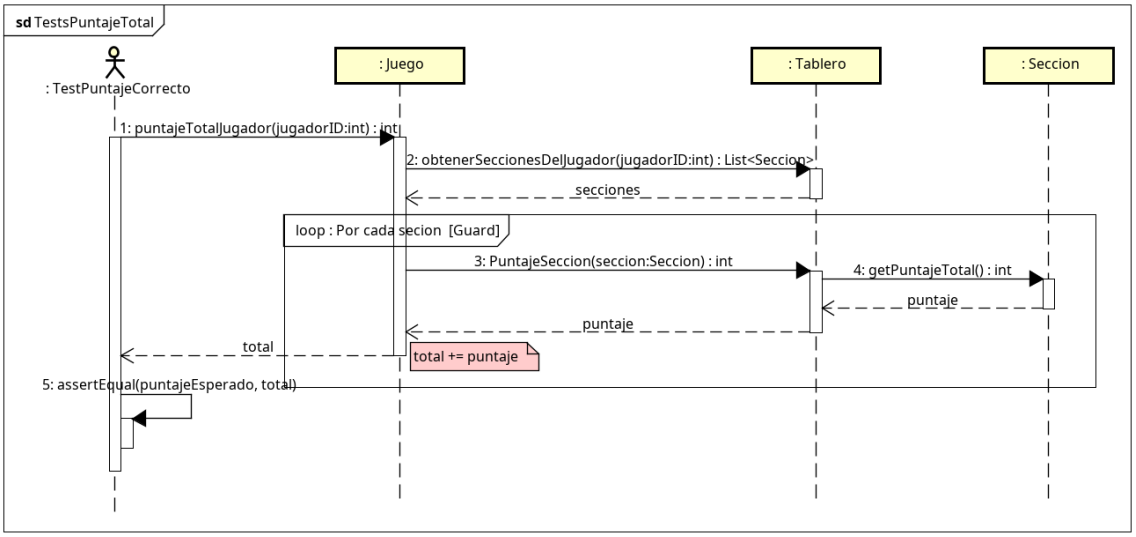

Figura 6: Diagrama de secuencia: cálculo del puntaje total de una sección.

En este diagrama se ilustra el flujo completo para el cálculo del puntaje total de una sección, comenzando desde un test automatizado. El test invoca al modelo a través del método correspondiente en la clase `Juego`, que a su vez delega la consulta al `Tablero`. El tablero solicita el puntaje a cada una de sus secciones, y cada sección calcula y devuelve su puntaje considerando las cartas y modificadores presentes. Este diagrama refleja el recorrido de la información y la colaboración entre objetos.

3.2. Test de Cartas Suficientes - Semana 1

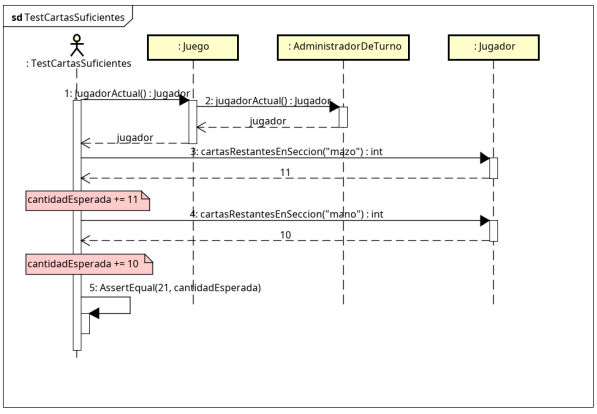

Figura 7: Diagrama de secuencia - Test de cartas suficientes

Este diagrama de secuencia muestra el comportamiento del test `Test01JugadorConCartasSuficientes`, que verifica si unjugador tiene la cantidad necesaria de cartas (21 en total) distribuidas entre su mazo y su mano al inicio de una partida. El flujo del test comienza cuando se solicita al objeto `Juego` el jugador actual mediante `jugadorActual()`, consulta que es delegada al `AdministradorDeTurno`. Posteriormente, se consulta la cantidad de cartas en dos secciones específicas: primero se invoca `cartasRestantesEnSeccion("mazo")` delvolviendo 11 cartas, y luego `cartasRestantesEnSeccion("mano")` obteniendo 10 cartas. La suma de estos valores de pasan como `cantidadEsperada` a `AssertEqual(21, cantidadEsperada)` para validar que el total corresponde exactamente a 21 cartas. Este test garantiza que la distribución inicial de cartas sea correcta.

3.3. Test Jugador juega una carta y tiene un puntaje parcial - Semana 1

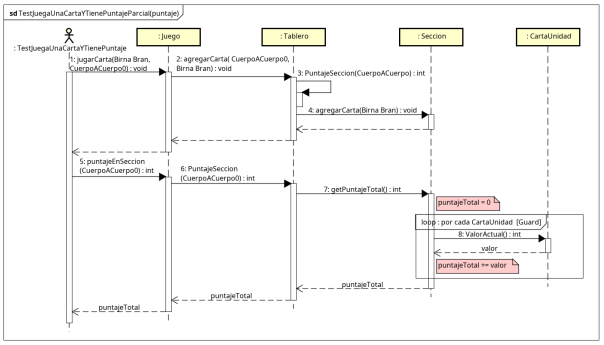

Figura 8: Diagrama de secuencia - Test jugador juega una carta

Este diagrama de secuencia representa el flujo natural del programa cuando un jugador coloca una carta en el tablero y posteriormente consulta cuántos puntos ha acumulado en esa sección. El test simula una situación típica del juego: primero se juega la carta "Birna Bran.en la sección ÇuerpoACuerpo0"del tablero. Una vez que la carta está en juego, el sistema necesita calcular el nuevo puntaje de la sección, por lo que se solicita esta información al tablero. La sección revisa todas las cartas que tiene y sumar los valores de cada una, considerando que estos valores pueden haber sido modificados por efectos especiales o habilidades de otras cartas.

4. Detalles de implementación

A continuación se detallan las principales decisiones y estrategias adoptadas para resolver los puntos más conflictivos del trabajo práctico, justificando el uso de herencia, delegación, principios y patrones de diseño según el modelo implementado:

- **Separación de responsabilidades y uso de fachada:** Se utilizó el patrón de *fachada* en la clase `Juego`, que centraliza la interacción con el modelo y expone una interfaz simple para el controlador y la vista. Esto permite desacoplar la lógica interna del modelo de la interfaz gráfica y facilita el mantenimiento.
- **Herencia vs. delegación en cartas y modificadores:** Se empleó herencia para modelar la jerarquía de cartas (`CartaUnidad`, `CartaEspecial`) y modificadores, permitiendo compartir comportamiento común y especializar según el tipo de carta o efecto. La delegación se utilizó en los modificadores, donde cada carta delega la aplicación de efectos a su modificador concreto, favoreciendo la composición y la extensión de nuevos efectos sin modificar las clases base.
- **Principio de abierto/cerrado y polimorfismo:** El modelo de modificadores y cartas especiales sigue el principio de abierto/cerrado, permitiendo agregar nuevos efectos o tipos de cartas sin modificar el código existente. Esto se logra mediante interfaces y métodos polimórficos como `modificar(Contexto contexto)` y `aplicarModificador(Contexto contexto)`.
- **Uso de contexto para desacoplar efectos:** La introducción de la clase `Contexto` permite que los modificadores operen sobre diferentes partes del modelo (tablero, sección, jugador) sin acoplarse a detalles internos. Esto facilita la extensión y el testeo de efectos complejos.
- **Colecciones genéricas y reutilización:** Para gestionar las distintas colecciones de cartas (mano, mazo, descarte), se utilizó una clase abstracta genérica (`SeccionJugador<T>`) que permite reutilizar lógica y evitar duplicación de código, aplicando el principio DRY.
- **Validaciones y control de errores:** Se implementaron excepciones específicas para manejar errores de dominio (por ejemplo, `TipoDeSeccionInvalidaError`), asegurando que las reglas del juego se respeten y facilitando la detección de errores en tiempo de desarrollo y ejecución.

- **Controladores y vistas desacoplados del modelo:** Se adoptó una arquitectura inspirada en el patrón *MVC* (Modelo-Vista-Controlador), separando la lógica de negocio (modelo), la presentación (vistas JavaFX) y la gestión de eventos (controladores). Sin embargo, a diferencia del patrón clásico, en nuestra implementación el controlador asume la responsabilidad de actualizar y refrescar las vistas tras los cambios en el modelo. De este modo, el modelo permanece completamente ajeno a la interfaz gráfica y no notifica directamente a la vista, lo que facilita el desacoplamiento y simplifica la evolución del sistema. Los controladores actúan como intermediarios: traducen las acciones del usuario en operaciones sobre el modelo y, posteriormente, actualizan la vista según corresponda. Esta decisión resultó conveniente para mantener una clara separación de responsabilidades y reducir el acoplamiento entre capas.

Estas estrategias permitieron construir un modelo flexible, extensible y fácil de mantener, facilitando la incorporación de nuevas reglas, efectos y tipos de cartas sin comprometer la robustez del sistema.

5. Excepciones

El modelo implementa un manejo robusto de errores mediante el uso de excepciones específicas para los distintos casos de dominio, asegurando la integridad de las reglas del juego y facilitando la detección temprana de situaciones inválidas. A continuación se describen las principales excepciones utilizadas y su tratamiento:

- **TipoDeSeccionInvalidaError:** Se lanza cuando se intenta crear o acceder a una sección del tablero con una clave inválida (por ejemplo, una sección inexistente o mal escrita). Esto previene la manipulación de secciones fuera de las permitidas (`CuerpoACuerpo`, `Rango`, `Asedio`) y asegura la consistencia del tablero. Su uso es recurrente en la construcción de secciones y en métodos que requieren validar la existencia de una sección antes de operar sobre ella.
- **CartaNoJugable:** Se utiliza para señalar que una carta no puede ser jugada en la sección o contexto actual, ya sea por restricciones de tipo, reglas especiales (por ejemplo, intentar jugar una carta especial en una sección no permitida) o por violar condiciones del juego. Esta excepción es capturada en los controladores y permite informar al usuario sobre jugadas inválidas sin comprometer el estado del modelo.
- **NoSePuedeEliminarClimaSiNoHayClima:** Específica para el manejo de cartas de clima, se lanza cuando se intenta eliminar un efecto climático en una sección o tablero que no tiene clima activo. Esto evita inconsistencias y asegura que los efectos de cartas como `TiempoDespejado` sólo se apliquen cuando corresponda.
- **NoSePuedeCumplirSolicitudDeCartas:** Se arroja cuando no es posible cumplir una solicitud de cartas, por ejemplo, al intentar robar más cartas de las que hay disponibles en el mazo o la pila de descarte. Permite controlar el flujo del juego y evitar estados inválidos en las colecciones de cartas.
- **UnoDeLosMazosNoCumpleRequitos:** Indica que alguno de los mazos iniciales no cumple con los requisitos mínimos para iniciar la partida (por ejemplo, cantidad insuficiente de cartas). Esta validación se realiza al comienzo del juego y previene partidas con configuraciones inválidas.
- **PilaDescarteNula:** Se utiliza para señalar intentos de acceder a una pila de descarte inexistente o nula, lo que puede ocurrir en situaciones límite o de error en la inicialización de las colecciones de cartas.
- **NoEsLaMismaUnidad:** Se lanza al intentar aplicar efectos de cartas que requieren operar sobre unidades equivalentes, pero se detecta que las cartas involucradas no cumplen con los criterios de igualdad (por ejemplo, nombre o modificador distinto en cartas unidas).

En todos los casos, las excepciones son propagadas hasta los controladores, donde se capturan y se informa al usuario mediante mensajes claros en la interfaz. Esto permite mantener la robustez del sistema, evitar corrupciones de estado y facilitar el diagnóstico de errores tanto en desarrollo como en ejecución.