Trabajo Práctico 2 — Gwent

[75.07 / 95.02 / TB025] Algoritmos y Programación III Primer cuatrimestre de 2025

Nombre	**Padrón**	**Mail**
Ames Berrospi, Andy Bruno	105366	aames@fi.uba.ar
González Lago, Mercedes	110796	mmgonzalez@fi.uba.ar
Hirak, Jonathan Julian	104184	jhirak@fi.uba.ar
Moore, Juan Ignacio	112479	jmoore@fi.uba.ar
Narváez Yaguana, Gabriel Alejandro	111432	gnarvaez@fi.uba.ar

Tutor: Diego Sánchez

Nota Final: _____

Índice

1. Supuestos
2. Diagramas de clase
3. Diagramas de secuencia
4. Diagramas de paquetes
5. Detalles de implementación
6. Excepciones

Supuestos

Durante el desarrollo del trabajo práctico se tomaron diversos supuestos a partir de ambigüedades o aspectos que no fueron especificados en el enunciado original.
Primero, se asumió que, debido a que la implementación no contempla un entorno multijugador, se decidió mostrar los mazos de ambos jugadores en la interfaz gráfica, escondiendo el que en ese momento no tenía el turno activo. Esto implica una confianza en que el usuario no observará el mazo del oponente cuando no es su turno. Lo consideramos una simplificación adecuada para el alcance del trabajo.
Luego, a diferencia del juego original de Gwent, se decidió no implementar facciones, ya que estas no se encuentran especificadas en el enunciado ni forman parte de los requisitos mínimos del modelo.
Por último, al utilizar una carta con modificador Médico, el jugador recupera automáticamente la última carta de la pila de descarte correspondiente a la fila donde fue colocada, descartando la idea de brindarle al usuario la posibilidad de elección entre diversas cartas descartadas.

Diagramas de clases

En esta sección incluímos los diagramas que muestran las relaciones estáticas entre las clases de nuestro proyecto. Decidimos separarlo en secciones para lograr reflejar la claridad esperada.
Primero presentamos el apartado de “Turn Management” donde se muestran las clases encargadas del flujo de cada partida.

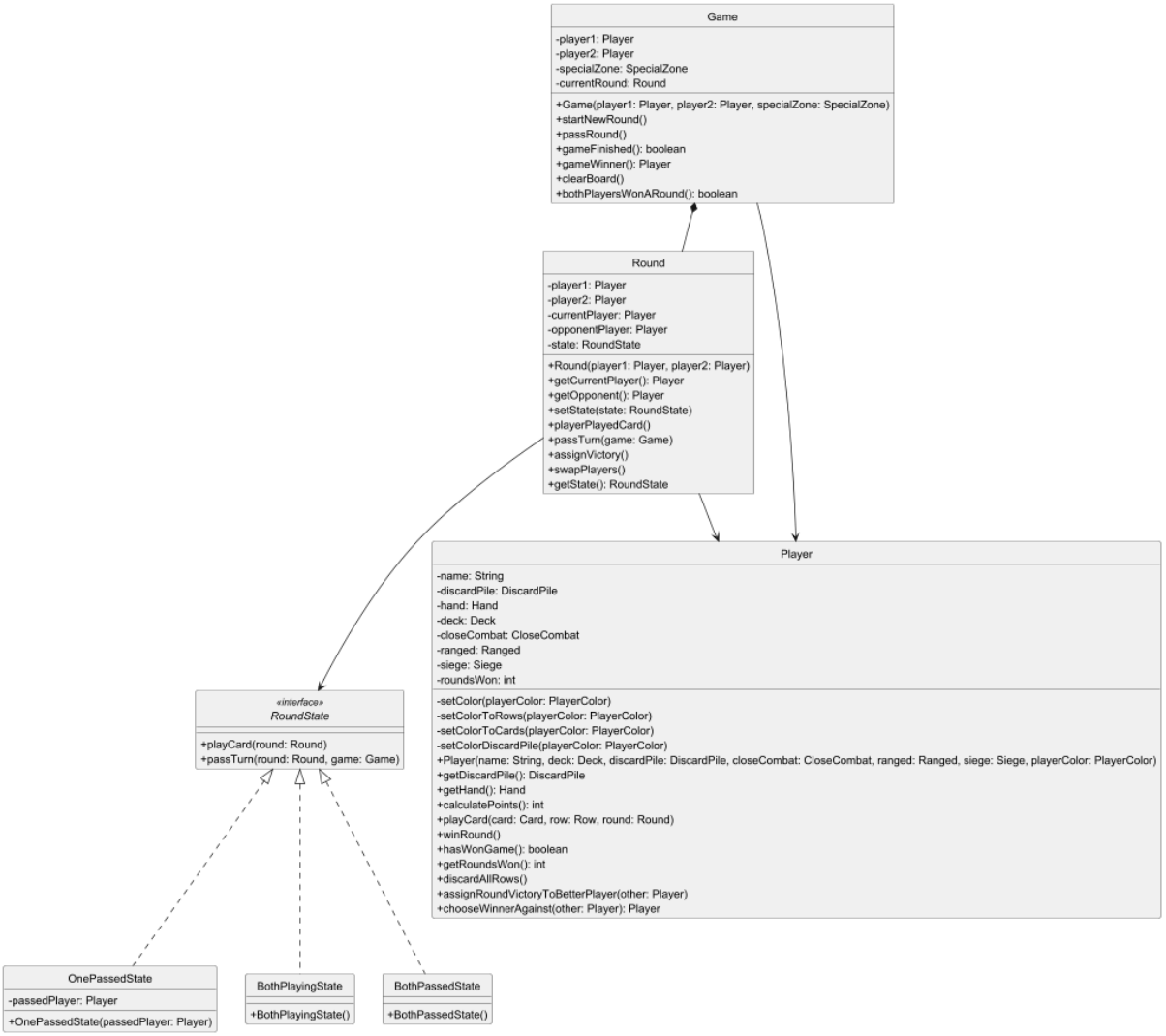

Diagrama de Turn Management

Siguiendo con el próximo paquete, presentamos en varias partes las clases que tienen que ver con el modelado de las cartas.

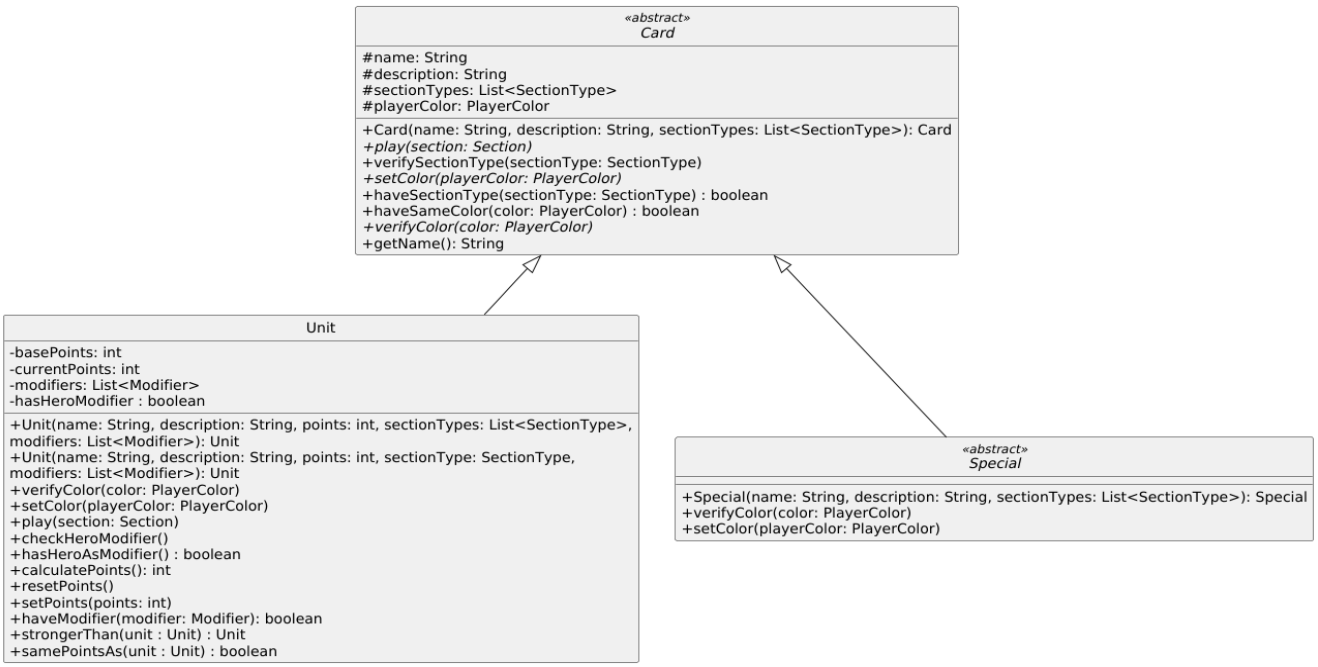

Diagrama de Card

Ahora desarrollamos el lado de Special con las clases hijas:

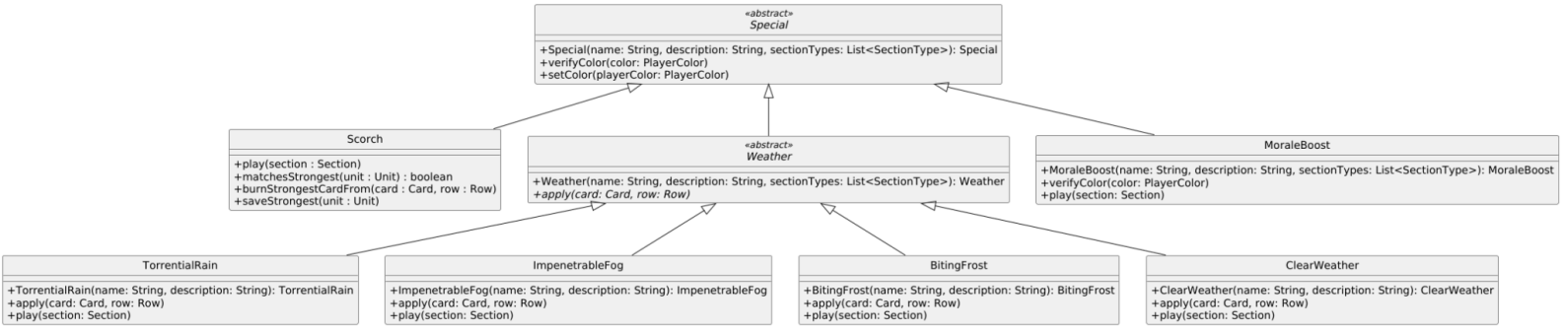

Diagrama de Special

Y por último los modificadores que usa Unit:

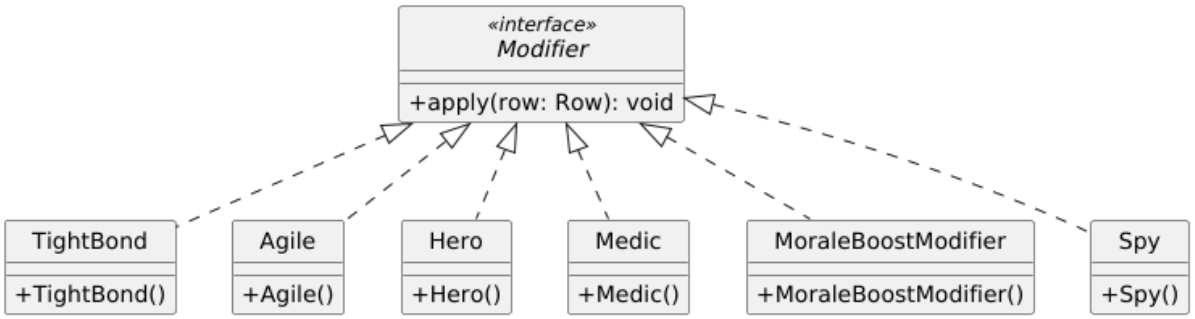

Diagrama de Modifier

Adjuntamos también el diagrama de CardCollection, con subclases como Deck, donde se muestra cómo manejamos la validación de las cartas mínimas que necesita un mazo para poder usarse.

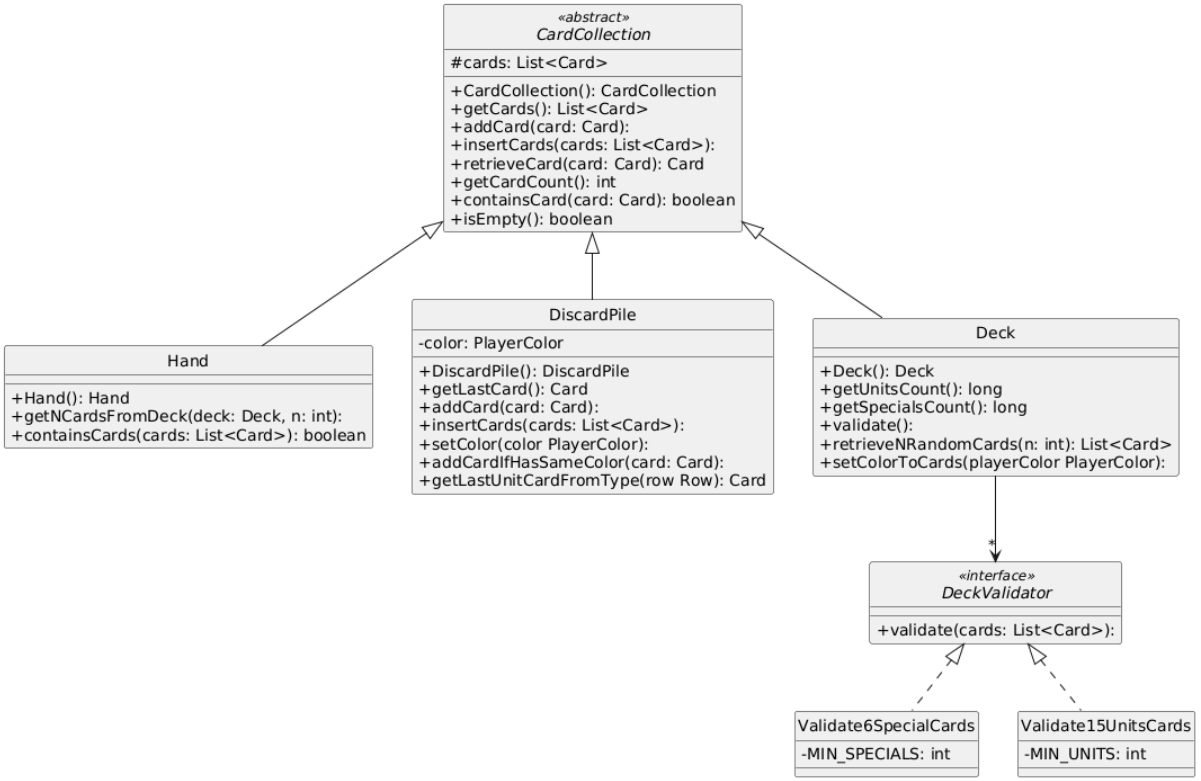

Diagrama de CardCollection

También, con lo comentado anteriormente en los supuestos, creamos la clase abstracta PlayerColor para lograr manejar la colocación de las cartas de una manera correcta, y no sea problema ampliar requisitos como ágil o espía.

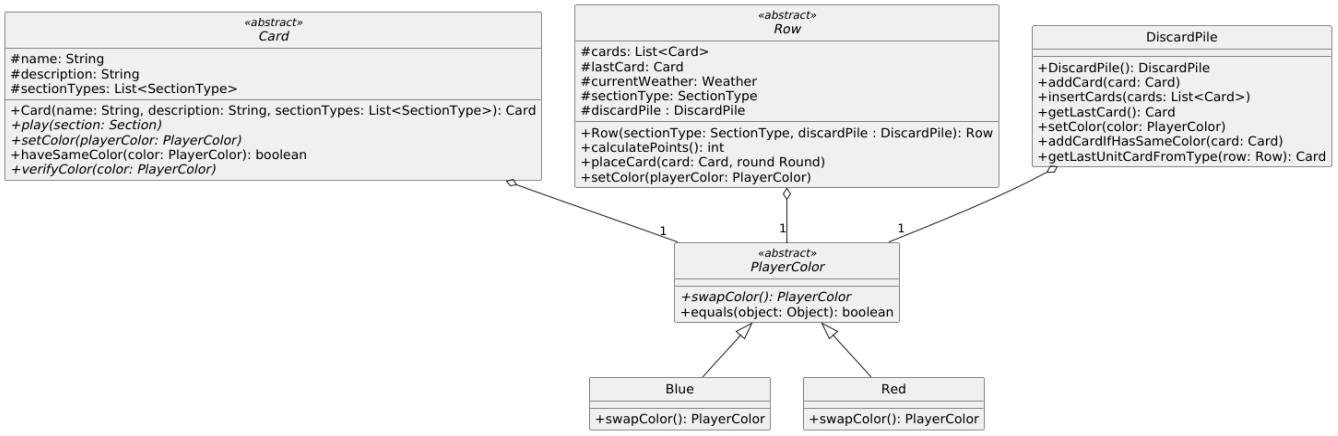

Diagrama PlayerColor

Continuando, hicimos el diagrama de Sections, separado en varias partes para mayor legibilidad, comenzando por Row y sus clases hijas:

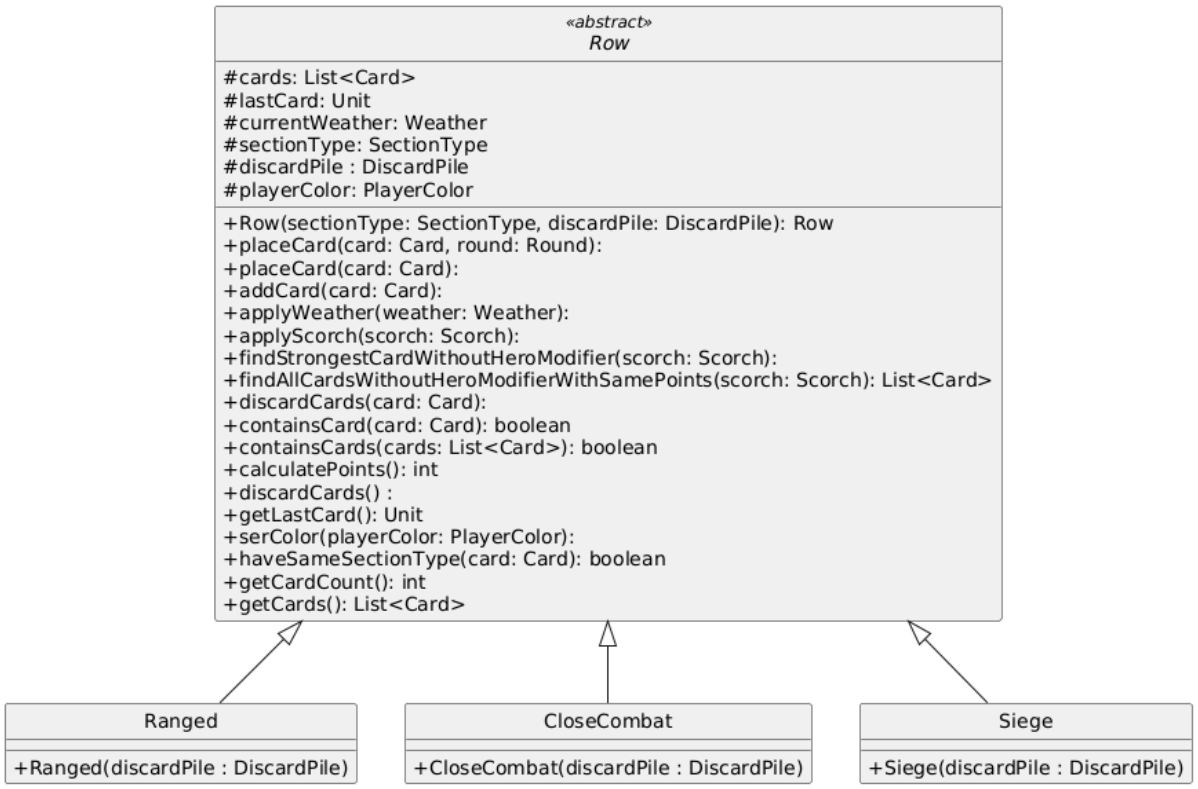

Diagrama de Sections (Row)

Siguiendo por como implementamos la interfaz Section, de la cual implementan tanto Row como SpecialZone:

Diagrama Sections (SpecialZone)

Y también viendo como SpecialZone usa SectionType:

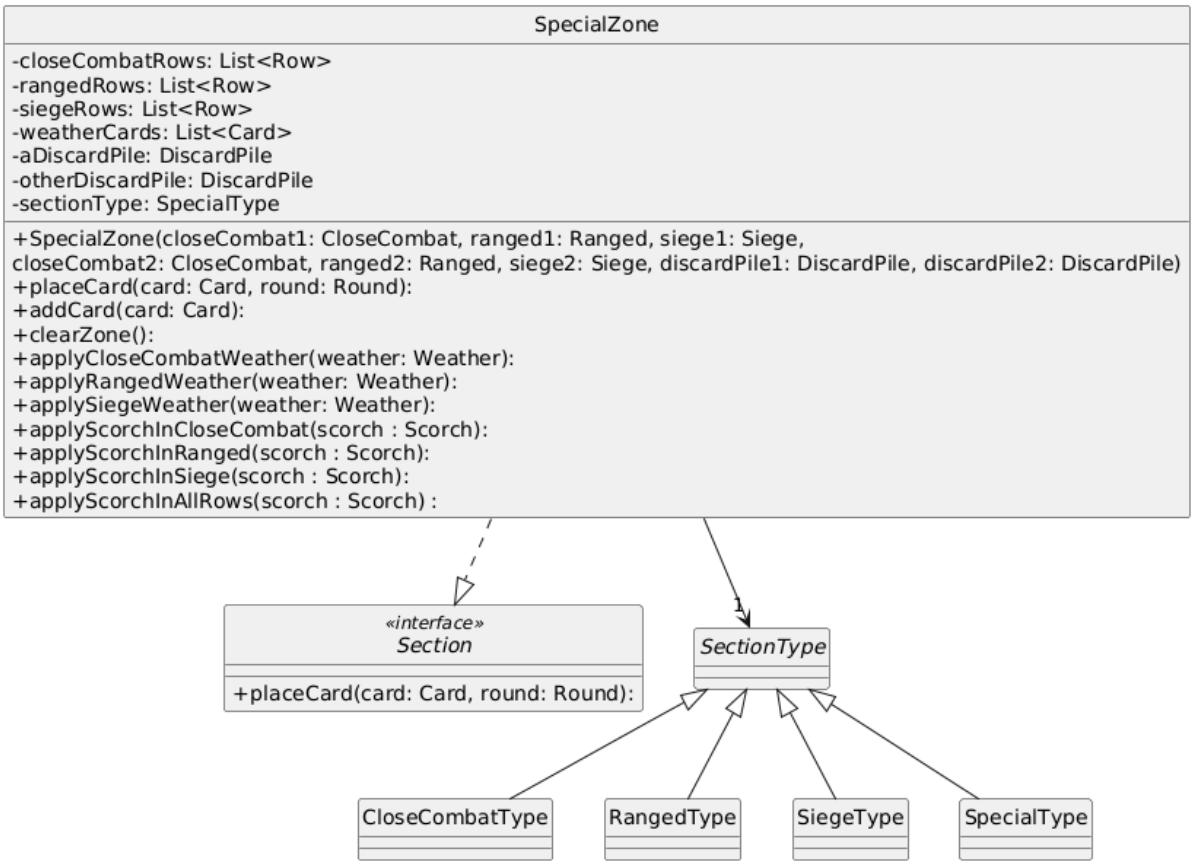

Diagrama Sections (SectionType)

Por último, hicimos el diagrama de las excepciones que atrapamos durante el flujo del programa, subdividiendo el diagrama en dos para mayor visibilidad.

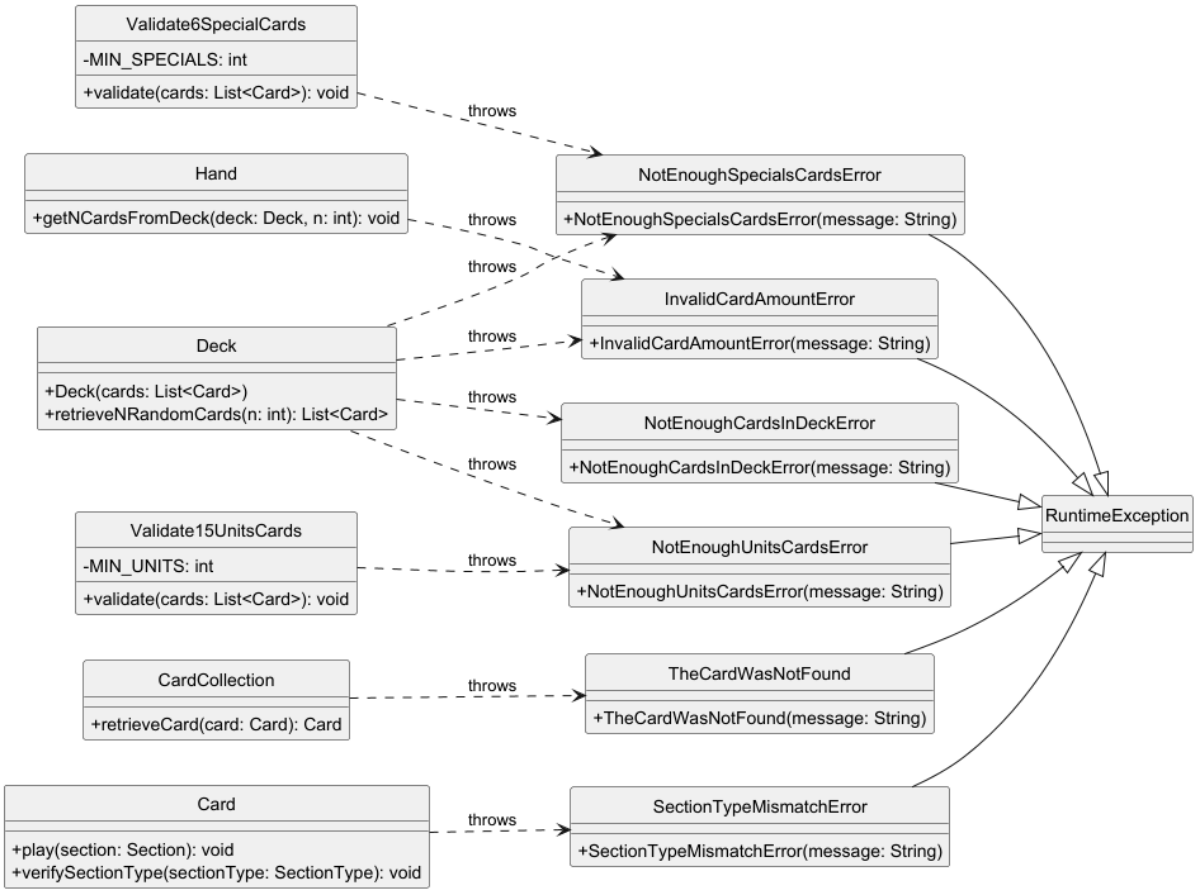

Diagrama excepciones 1

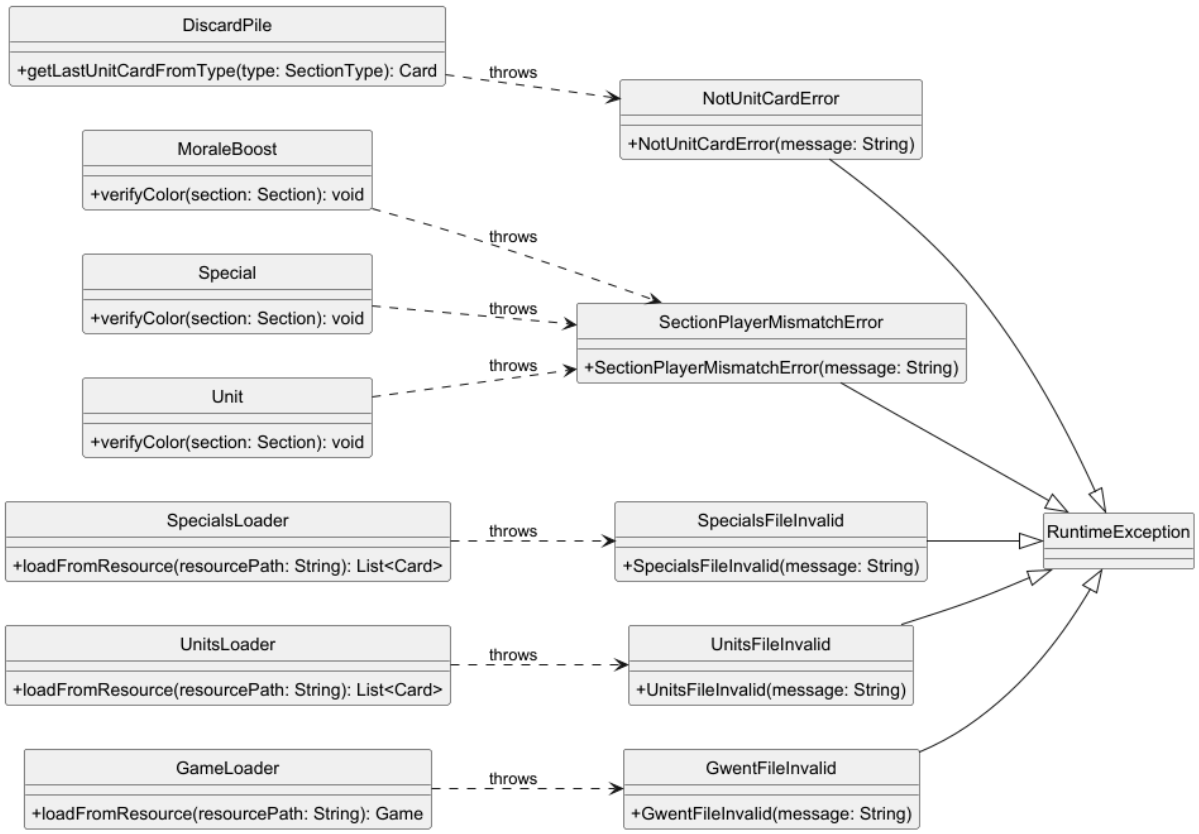

Diagrama excepciones 2

Diagramas de secuencia

En este apartado mostramos distintos escenarios interesantes de nuestro trabajo práctico, explayando cómo los objetos se comunican entre sí para lograr lo planteado.
Empezamos con la secuencia de cómo se aplica un clima, en especial, cómo se aplica Biting Frost a las filas de combate cercano:

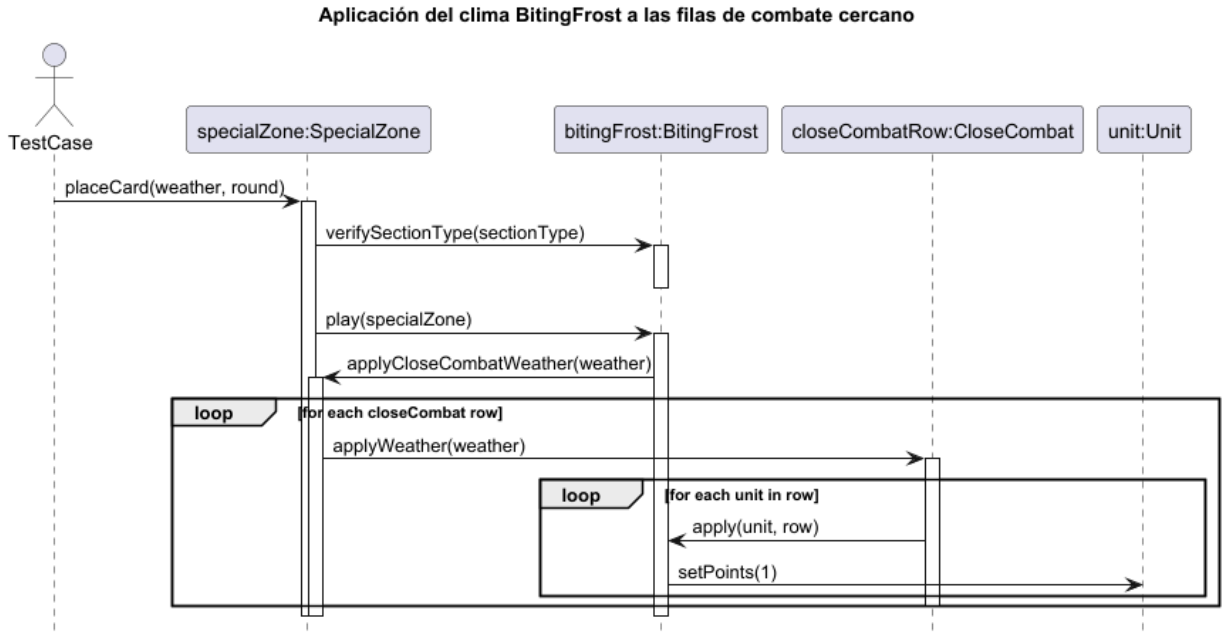

Diagrama secuencia Biting Frost

El próximo diagrama a mostrar es cómo se calcula puntaje de un jugador, delegando a las filas:

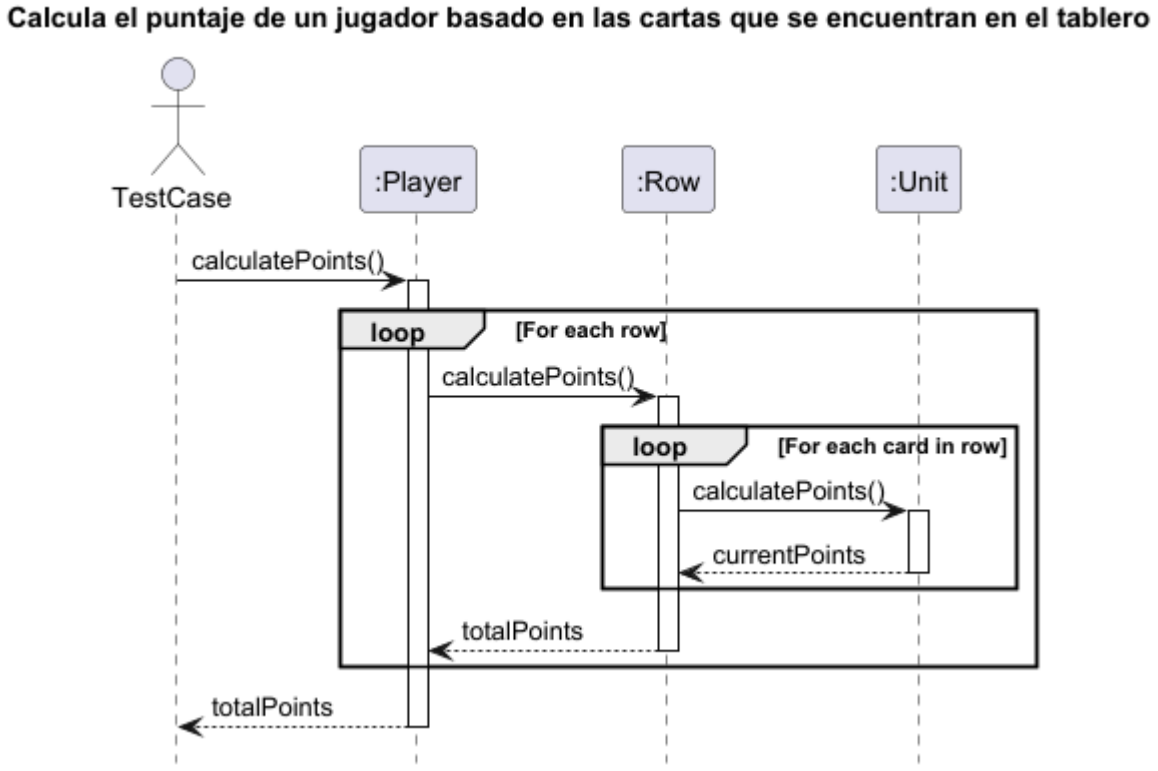

Diagrama de calcular puntaje

Luego, decidimos mostrar el caso en donde un jugador pone una carta en una sección del tablero:

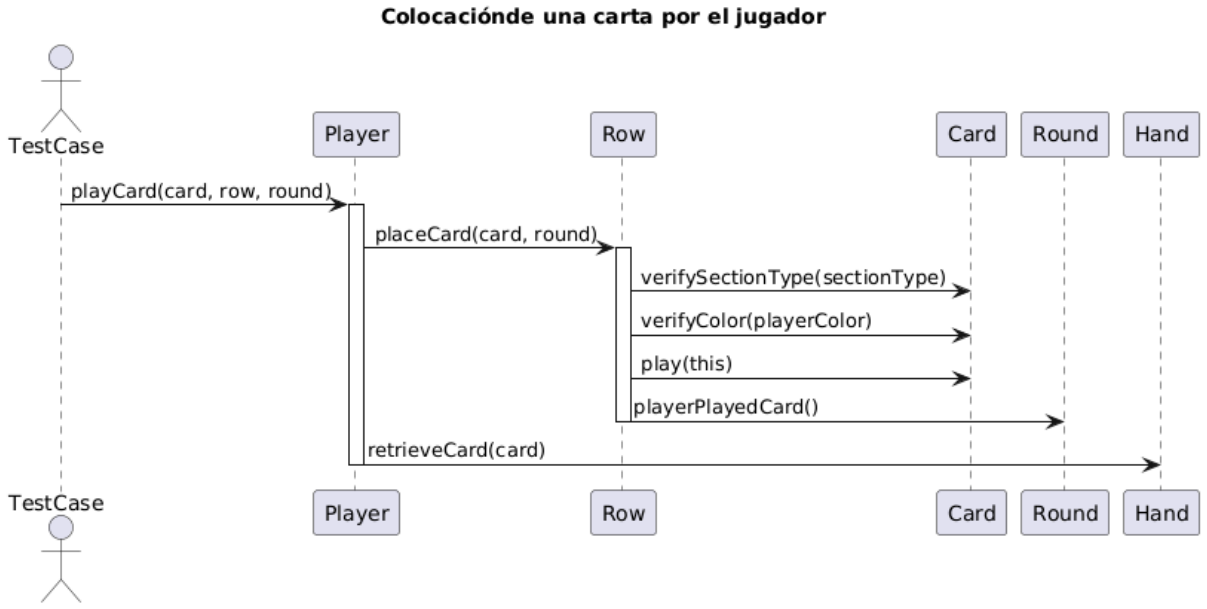

Diagrama de paquetes

Acompañado a lo anterior, incluímos un diagrama de paquetes para que se logre ver el acoplamiento de nuestro programa final:

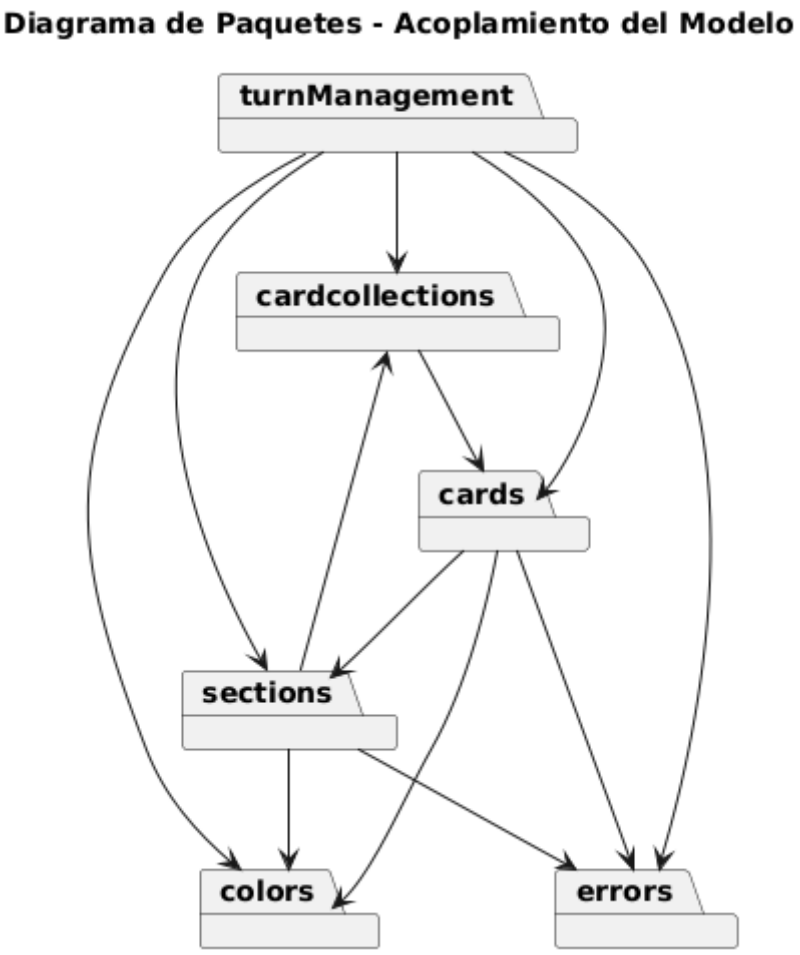

Diagrama de paquetes

Se puede observar que el paquete turnManagement centraliza gran parte de la lógica de control, lo que refleja su rol como coordinador en el flujo del juego. A su vez, se muestra cómo cards tiene dependencias que indican una interacción con la estructura y reglas del tablero.

Detalles de implementación

En esta sección contamos cómo resolvimos las partes más conflictivas de nuestro proyecto.
Uno de los problemas más grandes que tuvimos fue como validar que una carta se ponga en la fila adecuada. Al principio esto no parecía una tarea ardua, hasta que nos topamos con los modificadores Ágil y Espía, que te permiten poner una carta en más de una fila o en una fila enemiga, respectivamente. Para esto definimos una clase abstracta Color, de la cual heredan Red y Blue, permitiendo distinguir a qué jugador pertenece cada carta. Cada una contiene un atributo color, que se establece al momento de repartir las cartas. Además, dentro de la clase Card agregamos una lista sectionTypes (de la también implementada clase SectionType) que indica en qué secciones puede colocarse una carta. Para validar esto se implementó el método verifySectionType. Esta solución nos permitió generalizar la validación sin tener que hacer lógica separada para cada tipo de modificador.
Luego, para modelar los estados de una ronda y manejar la transición de turnos entre jugadores implementamos el patrón de diseño State. Creamos la interfaz RoundState que define los métodos que implementan las clases concretas: BothPlayingState (estado inicial), OnePassedState (cuando un jugador pasa, se transiciona a este estado) y BothPassedState (ambos jugadores pasaron). Esto lo usamos en el objeto Round, que contiene el estado actual y delega todas las acciones al objeto de estado correspondiente. Además, para manejar la alternancia de turnos, Round mantiene referencias a currentPlayer y opponentPlayer, intercambiandolos con el método swapPlayers(). Toda esta implementación se integró en el objeto Game, el cual detecta el fin del juego y se encarga de iniciar nuevas rondas o finalizar la partida.
También, uno de los puntos clave del diseño fue cómo modelar los distintos tipos de cartas del juego, como las de unidad, especiales y subtipos como clima. En esto vimos un comportamiento común, el cual fue abstraído en la clase Card mediante el uso de herencia. Esto fue justificado porque, principalmente, todas las cartas, independientemente de su tipo, comparten atributos y comportamientos. Dentro de la clase Special, se definió una nueva abstracción Weather, ya que todas estas comparten la idea de afectar unidades sobre el

tablero, pero con implementaciones distintas según la fila que modifican. Esto nos ayuda a usar una lista o colección de Card que contenga tanto unidades como especiales y a invocar métodos sin importar qué subtipo sea y que el comportamiento sea polimórfico.
Además, aplicamos delegación en los casos donde un objeto no es responsable directo de realizar una acción. Un ejemplo de esto se da en el método playCard de Player, donde no se realiza directamente la lógica de la colocación de una carta en el tablero, sino que delegamos esta responsabilidad al objeto Row.

Excepciones

Se diseñaron diversas excepciones personalizadas con el objetivo de atrapar correctamente estados inválidos o errores de flujo e informar de manera correcta al usuario. Cada una fue definida como una clase específica que extiende de RunTimeExcepction.

InvalidCardAmountError

Se lanza cuando se solicita un número inválido de cartas (cero o negativo), ya sea desde un mazo o hacia una mano. Mensaje: Invalid number of cards requested, must be greater than zero.

NotEnoughCardsInDeckError

Se utiliza para indicar que un mazo no contiene suficientes cartas para cumplir con una operación solicitada. Mensaje: Deck without enough cards.

NotEnoughSpecialsCardsError
Se lanza durante la validación del mazo inicial si este no contiene la cantidad mínima de 6 cartas especiales, como exige la consigna. Mensaje: Instance of Deck without enough special cards.

NotEnoughUnitsCardsError
Similar a la anterior, se valida que el mazo tenga al menos 15 cartas de unidad. Mensaje: Instance of Deck without enough unit cards.

SectionTypeMismatchError
Se lanza cuando se intenta jugar una carta en una fila no permitida, es decir, su SectionType no coincide con el especificado al momento de jugarla. Mensaje: SectionType does not match for this card.

TheCardWasNotFound
Sucede cuando se intenta remover una carta de una colección y la carta no está presente. Mensaje: The card is not in the deck.

GwentFileInvalid
Esta excepción ocurre cuando hay un error leyendo o parseando el archivo en la clase GameLoader. Mensaje: Error reading or parsing file.

NotUnitCardError
Esto ocurre en la clase DiscardPile. La excepción que informa es que no hay ninguna carta de tipo unidad en la pila de descarte. Se usa para cartas con el modificador Médico. Mensaje: No unit card found in discard pile.

SectionPlayerMismatchError
Ocurre cuando la carta quiere ser colocada en la parte enemiga del tablero (en nuestro caso, cuando tiene el color opuesto). Mensaje: Side does not match for this card.

SpecialsFileInvalid
Se implementó con el fin de que falle cuando se lea o parsee un archivo de especiales de forma errónea. Mesaje: Error reading or parsing file.

UnitsFileInvalid

Misma idea que el anterior pero en este caso con los archivos de cartas unidad.
Mensaje: Error reading or parsing file.