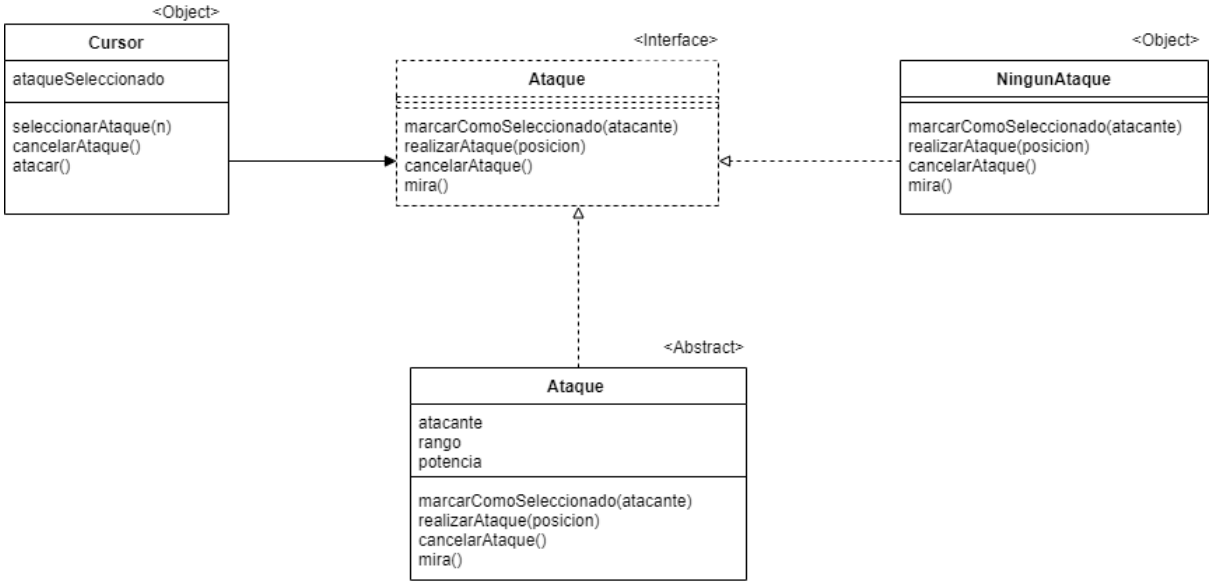
Explicación y aplicación de conceptos teóricos

- **Polimorfismo**: ¿Cuál es el mensaje polimórfico? ¿Qué objetos lo implementan? ¿Qué objeto se aprovecha de ello?

El cursor usa polimórficamente a cualquier objeto con la interfaz Ataque: es decir los mensajes “marcarComoSeleccionado”, “realizarAtaque”, “mira”, y “cancelarAtaque”. Cuando seleccionamos un ataque, se guarda en el cursor una referencia a este (en la variable ataqueSeleccionado), y el cursor luego manda distintos mensajes a dicho ataque. Al realizar el ataque, se resetea el valor de ataqueSeleccionado a ningunAtaque.
La interfaz Ataque es implementada por cualquier objeto que herede la clase Ataque y también por el objeto ningunAtaque, que define ciertos comportamientos que debería tener el cursor cuando no hay ningún ataque seleccionado.
La clase Ataque es una clase abstracta que es heredada por distintos ataques, que causan distintos efectos en sus víctimas.



- **Colecciones**: ¿Qué operaciones de colecciones se utilizan? ¿Usaron mensajes con y sin efecto? ¿Para qué?

Utilizamos mensajes sin efecto de colecciones en los métodos posiciones() de los rangos de movimiento de los personajes. Por ejemplo en este caso utilizamos map y flatten para obtener las posiciones de los casilleros pertenecientes a las columnas pedidas.

```
class RangoColumnas inherits Rango{
    const columnas
    override method posiciones() = columnas.map({n => tablero.columna(n)}).flatten().map({casillero => casillero.position()})
}
```

Otra de las operaciones con colecciones es la de crear casilleros. Dada una colección de casillas, se utilizan los métodos de *crearFila* y *crearCasillas*, en forma de iteración para poder configurar el tablero de juego, siendo estos mensajes con efecto.

```
method crearFila(n) { tamanioHorizontal.times({i => casillas.add(new Casillero(coordenadas = new Coordenadas(x = i, y = n)))}) }
method crearCasillas() { tamanioVertical.times({i => self.crearFila(i)}) }
```

Una de las operaciones con colecciones, es la de agregar personajes. Cada jugador entiende el mensaje *agregarPersonajes(listaPersonajes)* que recibe una colección de personajes. De esta manera, podemos agregar a cada uno de los elementos de la *listaPersonajes*, mediante un forEach, a una colección *personajes*. A su vez, obteniendo el personaje podemos identificar a qué jugador pertenece.

```
method agregarPersonajes(listaPersonajes){
    listaPersonajes.forEach({personaje => self.agregarPersonaje(personaje)})
}

  jugador1.agregarPersonajes([soldadoNaziJp1, healerJp1])
  jugador2.agregarPersonajes([soldadoNaziJp2, healerJp2])


method agregarPersonaje(personaje) {
    personajes.add(personaje)
    personaje.jugador(self)
    personaje.rango(self.rangoDeDespliegueDeUnidades())
}
```

- **Clases**: ¿Usan clases? ¿Por qué? ¿Dónde o cuándo se instancian los objetos?

Sí, se usan clases porque se quieren modelar varias instancias de distintos conceptos que aparecen en el juego, como tropas para los jugadores, además de los ataques que pueden tener los personajes. Por ejemplo, se usa la clase Personaje porque un Jugador puede tener varias instancias de un personaje que tengan el mismo comportamiento, como varios soldados o healers. Estos objetos se instancian al comenzar el juego donde cada jugador va a tener una cantidad de tropas inicial.

- **Herencia**: ¿Entre quiénes y por qué? ¿Qué comportamiento es común y cuál distinto?

Podemos ver herencia entre los objetos *jugador1* y *jugador2*, que heredan el comportamiento de la clase jugador. Su comportamiento difiere en la imagen que cada uno posee para indicar su victoria mediante *pantallaGanadora*, y la imagen de su cursor.
Otra diferencia es que el jugador 1, en el juego, cumple el rol de defensor. Tiene edificios para defender, y el jugador 2, el atacante, debe destruirlos. Esto se refleja en los métodos de cada jugador. El jugador 1 tiene distintos métodos para manejar sus edificios, y su método perdió() tiene una implementación distinta, ya que además de perder cuando mueren sus tropas, pierde cuando se queda sin edificios.
También cambia el rango de despliegue de unidades, ya que cada jugador tiene una zona de influencia donde puede poner sus tropas al inicio del juego.

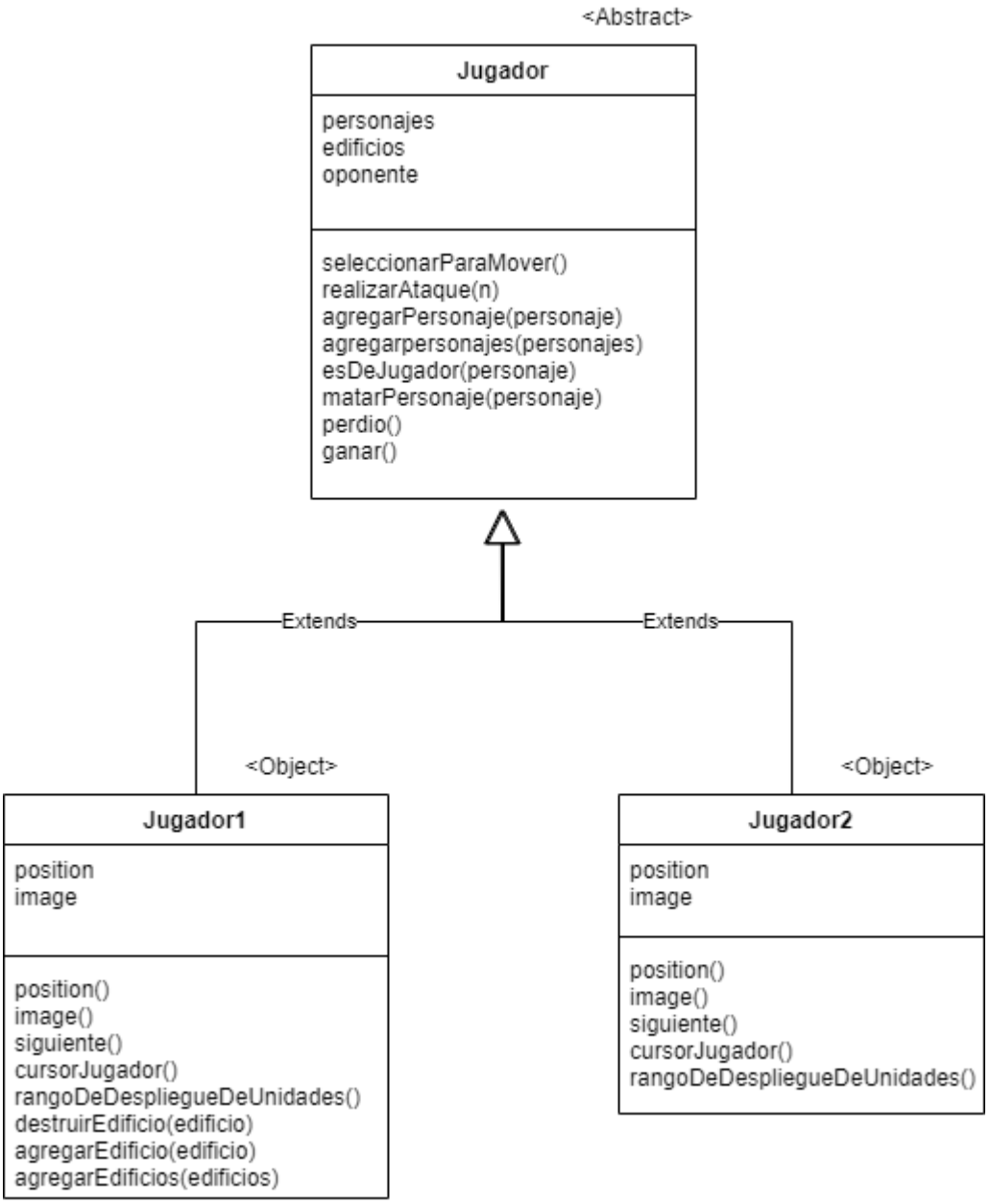


Diagrama de clases mostrando la herencia en los jugadores

- **Composición**: ¿Qué objetos interactúan? ¿Dónde se delega? ¿Por qué no herencia?

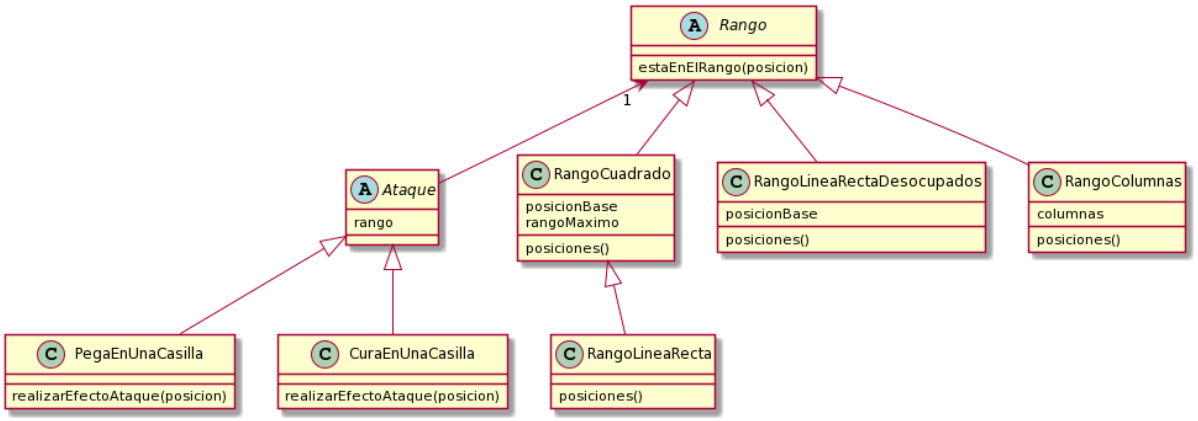
Tenemos composición con los rangos de movimiento de los personajes y los rangos de alcance de los ataques. Elegimos hacerlo con composición porque ya usábamos la herencia para otras cosas tanto en los ataques como en los personajes. Además, la lógica entre los rangos de ataque y de movimiento es compartida, por lo que podemos usar las mismas clases Rango.
Cada personaje y ataque tiene una variable ”rango”, que guarda los casilleros a los que se puede mover o atacar, según corresponda.
Los ataques y personajes mandan el mensaje .estaEnElRango(posición) a su variable “rango” para delegar la tarea de chequear, en el caso de los personajes, si es posible moverse a una posición, y en el caso de los ataques, si es posible atacar a una posición.

```
class Rango{
    method posiciones()

    method estaEnElRango(posicion) = self.posiciones().contains(posicion)
}
```

Rango es una clase abstracta que es heredada por los distintos tipos de rango (en los que definimos el método posiciones()).

Además, el Ataque ya está jerarquizado desde el punto de vista del daño o curación que hace en una casilla, entonces no se podría volver a aplicar la herencia para clasificarla por otro criterio, como el tipo de rango, a la misma clase Ataque. Al gastarse la herencia, se pasa a usar composición que nos permite modelar de otra forma la relación entre los objetos, delegando los distintos tipos de rango: rango cuadrado, rango línea recta, rango línea recta desocupados y rango columna. De esta forma, el Ataque conoce a varios tipos de objetos que son los distintos rangos de un Ataque.



Link del Diagrama de Clases:
www.plantuml.com/plantuml/png/dL71IWCn5Bo_hmWzrKCFtdjeNVUc80K_OCm-Io2RrBm4nEg_cpPTD6O5CRUycJTlcDb8H8XfjBHzaXYWezf6l2HMHr8gmEszeW_wGdUdKGOpodctTxqzZvO7BqORxwvET6qXKhXTmX0m-BFWHBg1yAH_ZrSpUlg6iYn_gjmPnzYnZb37lp9lMRnEXtnRskEjlEtQspGwb4Lz-VpkyO7twDsZGmSnrgAanx3c3Q5_pcvyYNKkfYw5_onFHQtVLwl6G8Ks5uWkFIwk5-kC5-uqBJjLULHz2wCffdgjrdmnSymfVDgm6z9eFm40