FACULTAD DE INGENIERIA
Universidad de Buenos Aires

Trabajo Práctico 2 — Java

[7507/9502] Algoritmos y Programación III
Grupo 06
Primer cuatrimestre de 2024

Alumno	Padron	Email
MARTURET, Valentín	104526	vmarturet@fi.uba.ar
DAVILA, Esteban	109815	edavila@fi.uba.ar
IBAR, Patricio	109569	pibar@fi.uba.ar
SOLES, Nahuel	109638	dsoles@fi.uba.ar
FONTANA, Maria Agustina	108090	mafontana@fi.uba.ar

Índice

1. Introducción

El presente informe reune la documentación de la solución del segundo trabajo práctico de la materia Algoritmos y Programación III que consiste en desarrollar una aplicación de un juego por turnos, de dos o más jugadores conformado de un panel en el cual se mostrarán preguntas con múltiples opciones de respuesta. Utilizando los conceptos vistos en el curso y un lenguaje de tipado estático (Java).

2. Supuestos

1. Suponemos que el puntaje de cualquier jugador inicialmente es 0.
2. Suponemos que el límite de preguntas por default es 25 (cantidad del json dado).
3. Suponemos el límite de puntos es 100 por default

3. Modelo de Dominio

3.1. Visión general del modelo

El diseño de la solución propuesta para el programa del trabajo práctico se enfocó en seguir de manera sencilla la lógica de negocio que define el funcionamiento del juego.

Durante una **RondaDePreguntas**, cada jugador visualiza una pantalla que muestra la pregunta actual, los *poderes* (modificadores de puntaje) disponibles para utilizar, y las opciones para responder la pregunta. El jugador prepara su jugada y presiona "jugar"para enviarla y pasar al siguiente turno.

Al registrar la jugada de un jugador (respuesta a la pregunta y poderes utilizados), la **RondaDePreguntas** recibe un mensaje con la jugada y el jugador correspondiente. La respuesta del jugador es evaluada por un objeto **Pregunta** específico de esa ronda, junto con los modificadores de puntaje aplicados.

3.2. Paquete AlgoHoot

- La clase **AlgoHoot** está diseñada para gestionar las rondas de preguntas, la tabla de jugadores y el gestor de preguntas. Actúa como punto de entrada al modelo, permitiendo que otros paquetes externos, como la vista o el controlador, interactúen con el modelo de manera controlada. **AlgoHoot** se asocia con las clases **RondaDePreguntas**, **GestorDePreguntas** y **TablaDeJugadores**.

- La clase **RondaDePreguntas** se instancia con la pregunta que se va a jugar. Es responsable de recibir las respuestas de cada jugador a través de objetos **Respuesta** y enviarlas a la **Pregunta** para su puntuación. Además, guarda los puntajes parciales de cada jugador, maneja y aplica los modificadores utilizados por los jugadores, y asigna los puntos a cada jugador al final de cada ronda.

- La clase **Jugador** es responsable de identificar y distinguir a los usuarios, llevar un registro de sus puntajes durante el juego y manejar la disponibilidad de modificadores de puntaje. Para gestionar el puntaje a lo largo de toda la partida, esta clase está asociada con la clase **PuntajeTotal**. La entidad **TablaDeJugadores** contiene todas las instancias de **Jugador** que participan en el juego.

- La clase **TablaDeJugadores** es responsable de registrar todos los jugadores con sus puntajes y de identificar un ganador, si lo hay.

- La clase **GestorDePreguntas** es responsable de proporcionar las preguntas para cada **RondaDePreguntas**. Las preguntas se devuelven en un orden aleatorio, asegurando que no haya

preguntas del mismo tipo de manera consecutiva. Para obtener el listado de preguntas, el gestor delega esta tarea a la clase **Lector**, que luego las ordena aleatoriamente.

3.3. Paquete Puntuación

- La clase **PuntajeTotal** representa el puntaje acumulado de un jugador a lo largo del juego. Es responsable de registrar cada **PuntajeParcial** por separado y de devolver la suma de estos puntajes.
- La clase **PuntajeParcial** representa el puntaje obtenido por un jugador en una ronda de preguntas. Consta de un puntaje base, que depende de la precisión de la respuesta del jugador, y una lista de modificadores individuales.
- **ModificadorIndividual** es una interfaz implementada, entre otras, por las clases **Duplicador** y **Triplicador**, que representan dos de los *poderes* que puede utilizar el jugador. Las clases que implementan esta interfaz son utilizadas por los objetos **PuntajeParcial** para modificar sus puntajes base.
- **ModificadorGlobal** es una interfaz implementada por las clases **Anulador** y **Exclusividad**, que representan los otros dos *poderes* que puede utilizar el jugador. Estas clases modifican una lista de puntajes parciales en su conjunto. En el modelo, las **RondaDePreguntas** operan con los modificadores globales elegidos por los jugadores para modificar todos los puntajes de la ronda.

3.4. Paquete Preguntas

En este paquete se encuentran todas las implementaciones de la interfaz **Pregunta**, así como la interfaz misma. Las implementaciones de la interfaz corresponden a diferentes tipos de preguntas (Verdadero-Falso, MultipleChoice, etc.) que manejan la respuesta y puntuación de manera distinta.

La decisión de agrupar estas implementaciones bajo una única interfaz facilita su gestión, permitiendo instanciarlas, utilizarlas en las rondas de preguntas y mostrarlas por pantalla de manera uniforme.

Además, en este paquete se encuentran las clases relacionadas con las respuestas para las preguntas, así como las formas de puntuar cada una de ellas (por ejemplo, **OpcionCorrecta**, **OpcionIncorrecta**, **Orden**, ...).

3.5. Paquete Fabricas

Este paquete contiene todas las implementaciones de la interfaz **Fabrica**, así como la propia interfaz. Las implementaciones de esta interfaz corresponden a las fábricas que instanciarán las distintas clases de preguntas utilizadas en el juego. Los datos utilizados para instanciar estas preguntas se obtienen de un archivo JSON y son manejados por la clase **Lector**.

La clase **Lector** es responsable de crear y devolver el listado de preguntas que se utilizarán durante el juego. Para lograrlo, obtiene los datos de las preguntas desde una archivo JSON y luego delega la creación de las preguntas a clases "Fábrica"(detalladas más adelante en la sección "Patrones de Diseño") y las almacena. **Lector** asigna a las diferentes "Fábricas"dependiendo del tipo de pregunta que se esté creando en ese momento.

4. Diagramas de paquetes

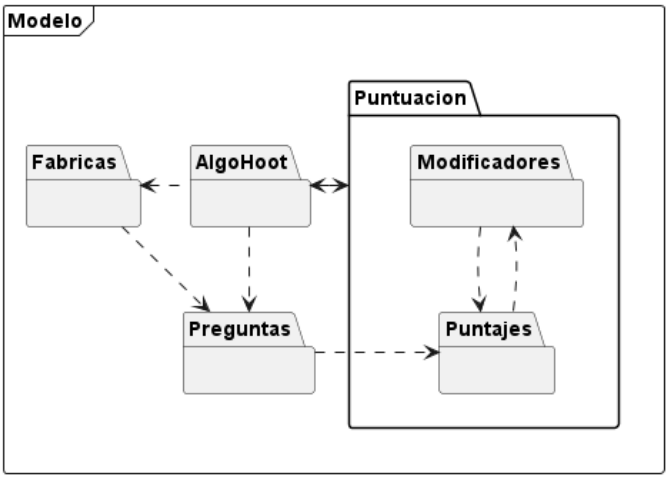

Figura 1: Diagrama de paquetes del Modelo.

5. Diagramas de clases

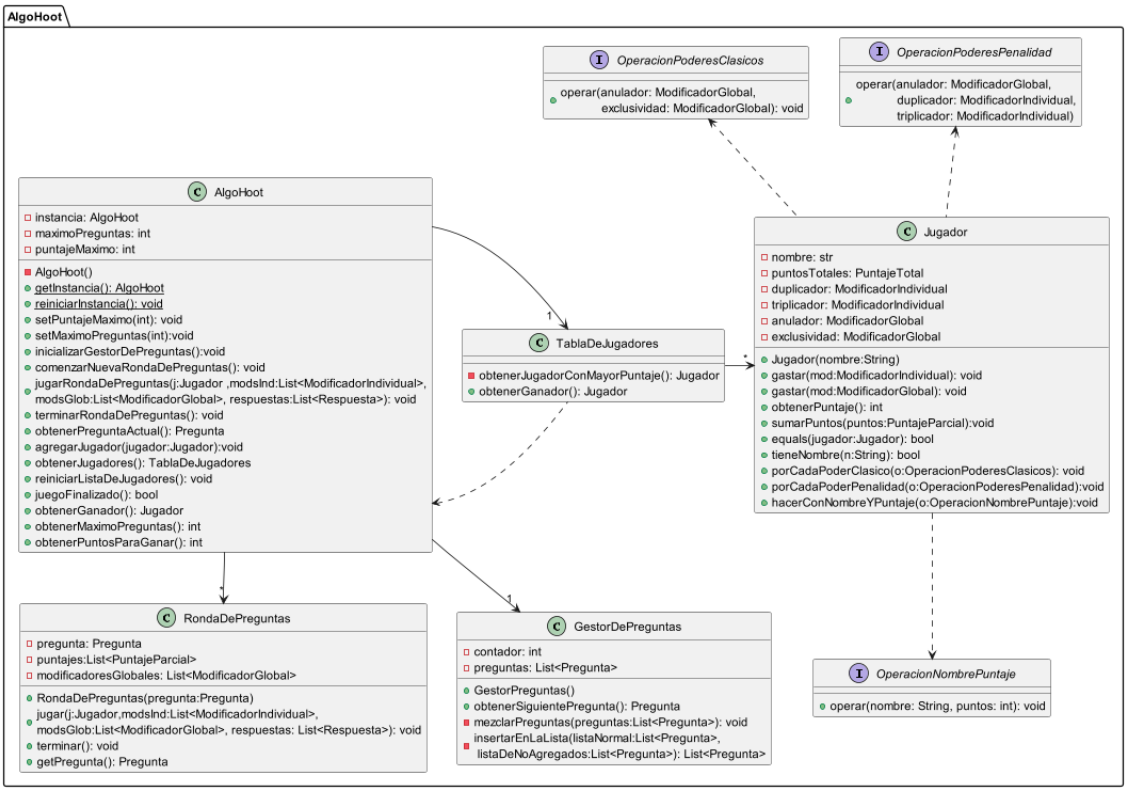

Figura 2: Diagrama de clases del paquete AlgoHoot.

Figura 3: Diagrama de clases del paquete Preguntas. Implementación de las preguntas.

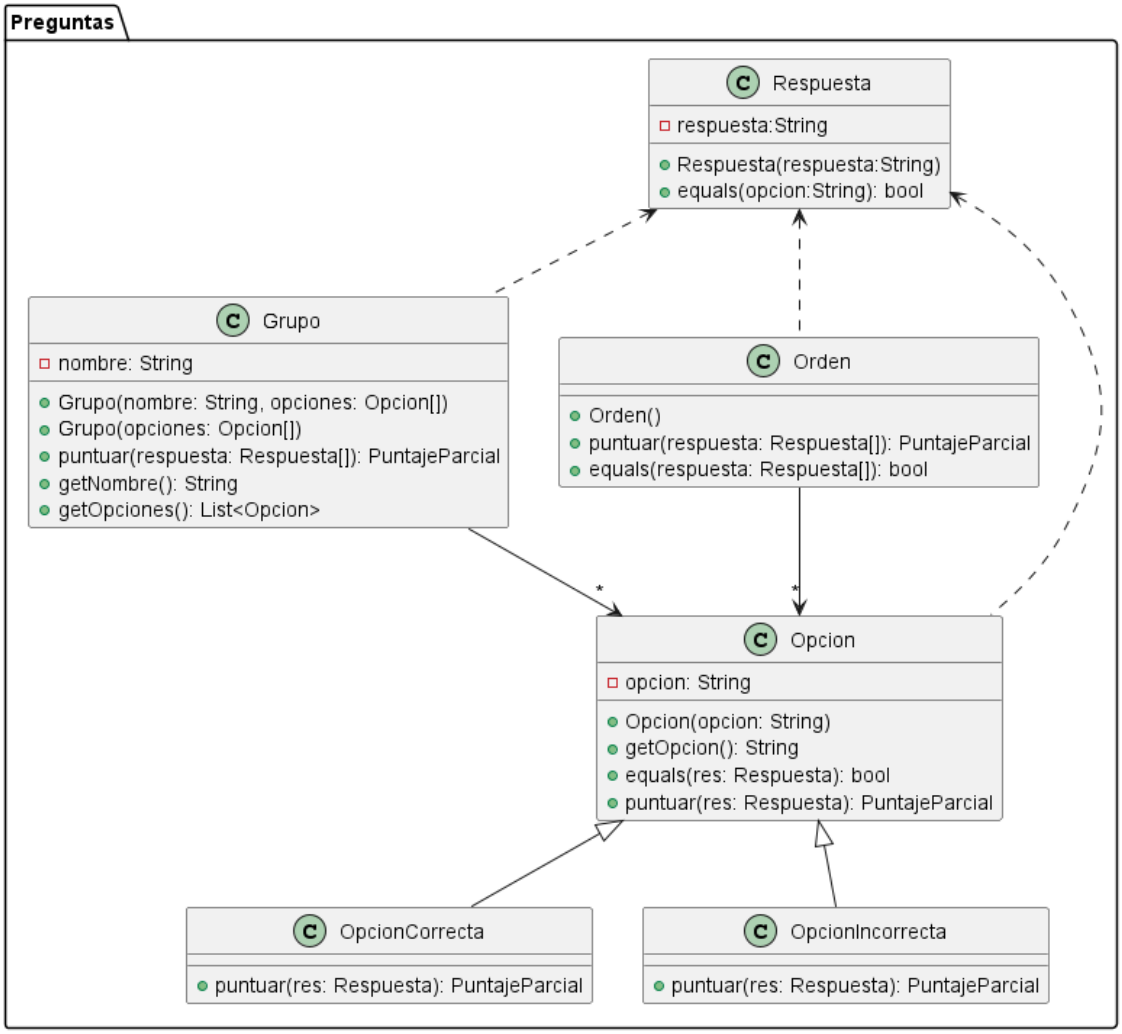

Figura 4: Diagrama de clases del paquete Preguntas. Implementación de los componentes Opcion y Respuesta

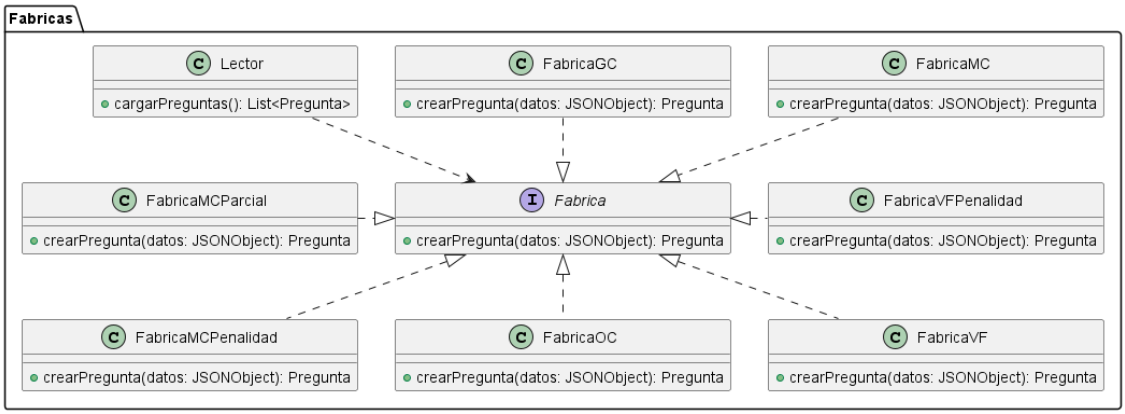

Figura 5: Diagrama de clases del paquete Fabricas.

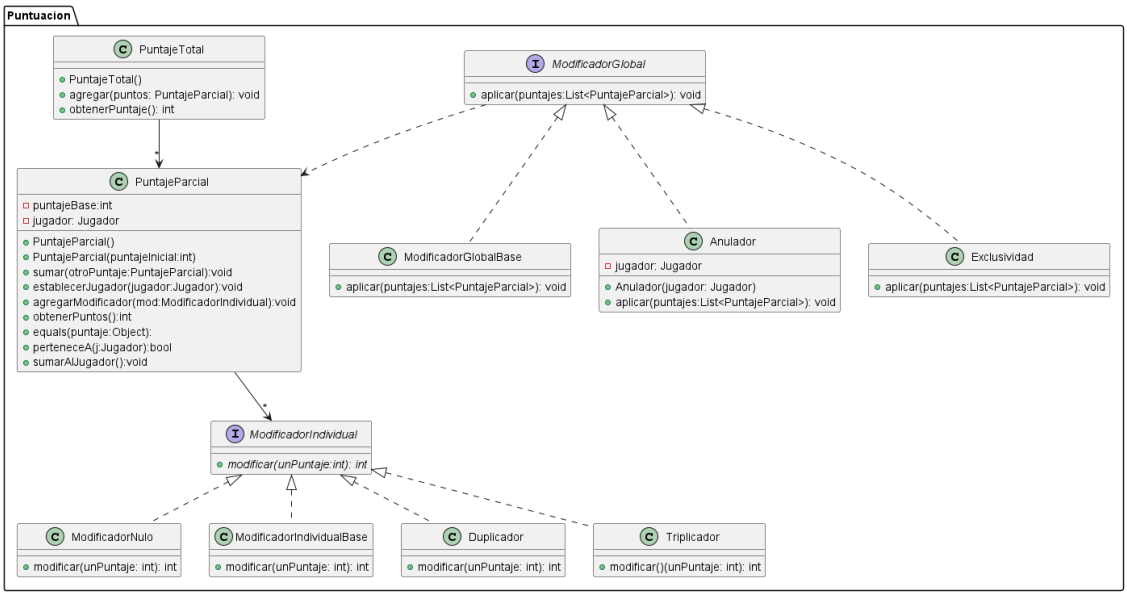

Figura 6: Diagrama de clases del paquete Puntuacion.

6. Diagramas de secuencia

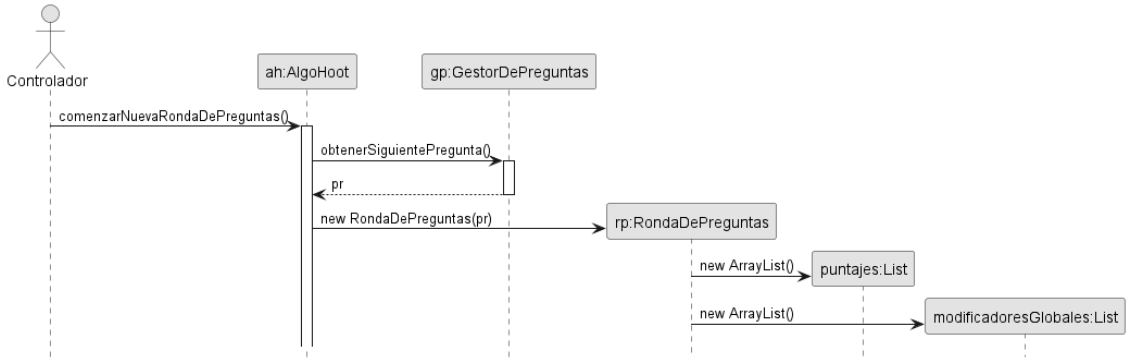

Figura 7: Secuencia de inicialización de una RondaDePreguntas.

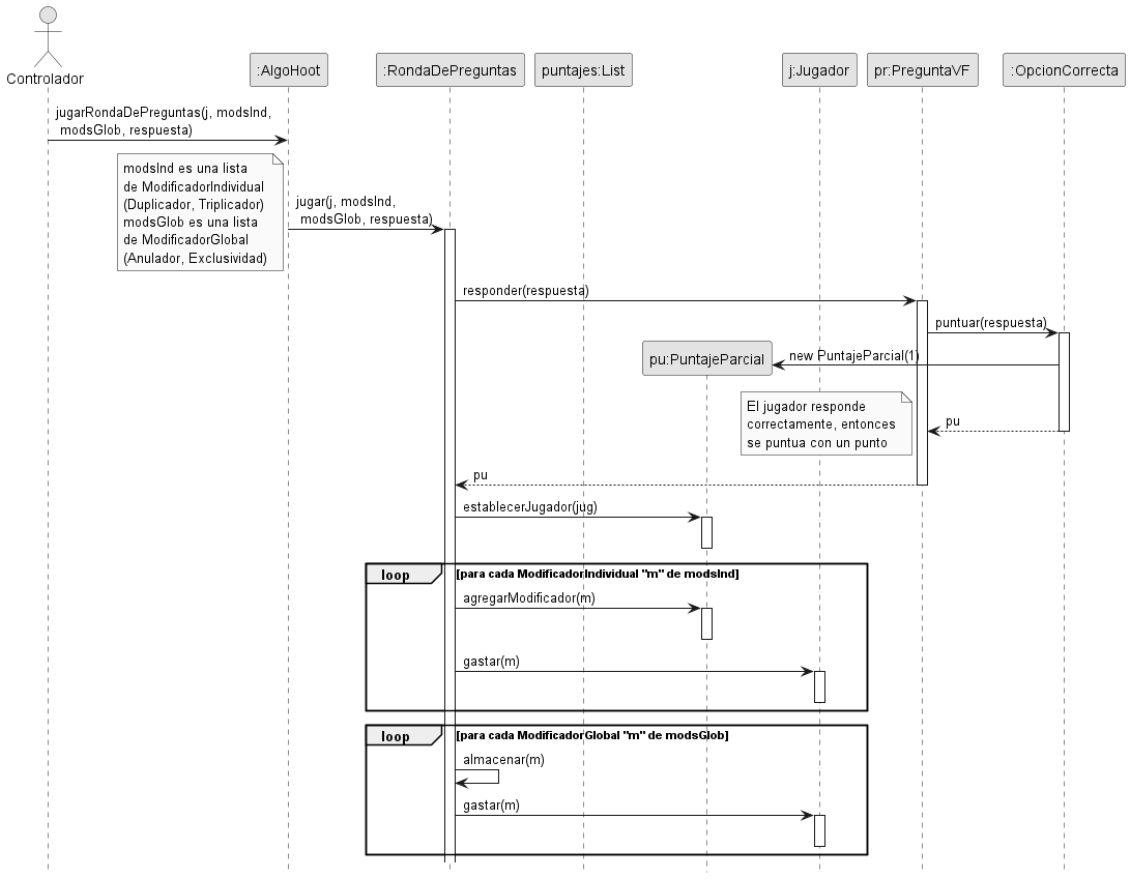

Figura 8: Secuencia de jugar una RondaDePreguntas.

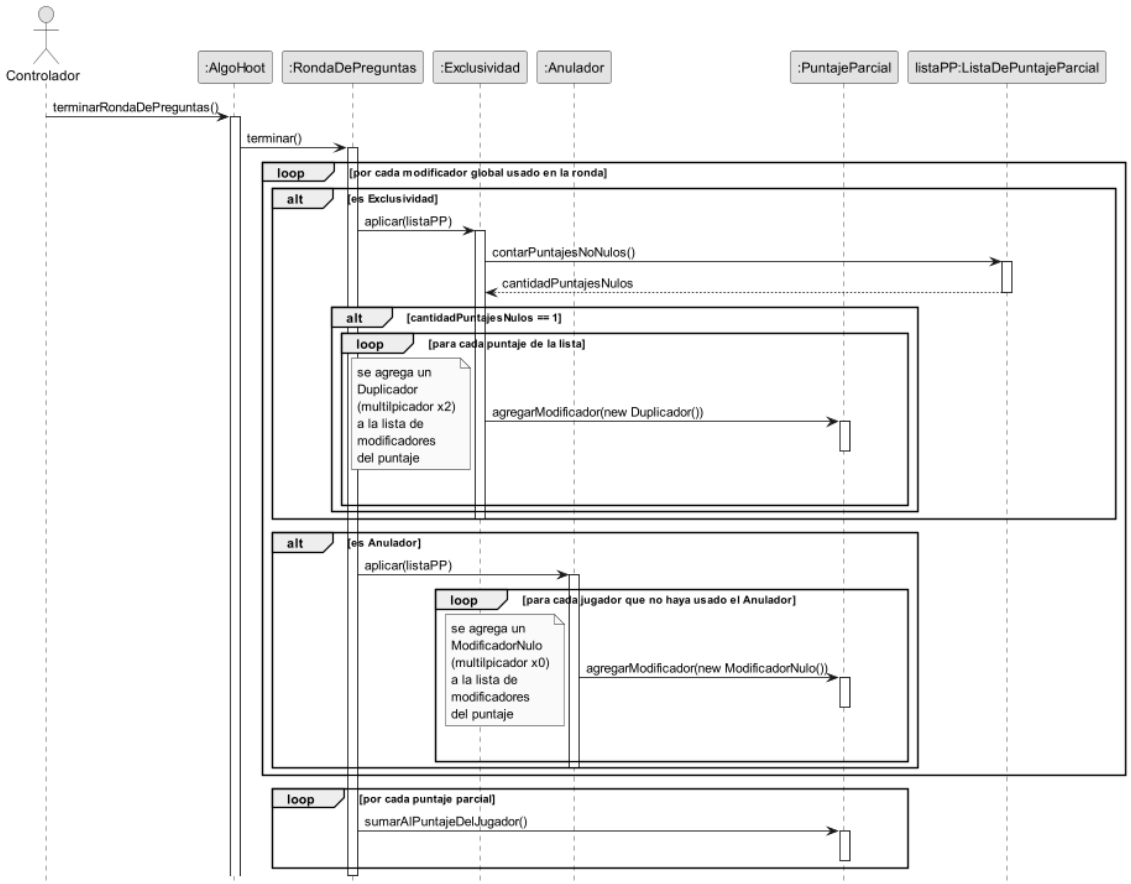

Figura 9: Secuencia de finalización de una RondaDePreguntas.

7. Detalles de implementación

7.1. Respuesta y Puntuación de Preguntas

Cada tipo de pregunta específica almacena la respuesta correcta de una manera particular y también está diseñada para evaluar la respuesta de un jugador de manera correspondiente. A continuación se presentan algunos ejemplos concretos.

7.1.1. Verdadero/Falso

Las preguntas del tipo Verdadero/Falso almacenan un objeto de la clase **OpcionCorrecta** y otro de la clase **OpcionIncorrecta**. Ambas clases están diseñadas para compararse con una **Respuesta**, devolviendo un **PuntajeParcial**. Si una OpcionCorrecta coinciden con la respuesta proporcionada, devuelven un puntaje de 1; de lo contrario, devuelven un valor nulo. Las opciones incorrectas devuelven un puntaje de -1 si coinciden, o un valor nulo en caso contrario.

Al puntuar una **Respuesta** en una pregunta Verdadero/Falso, se espera que devuelva un puntaje de 1 si la respuesta es correcta, o un puntaje de 0 si es incorrecta. Para lograr esto, la respuesta del jugador se compara únicamente con la **OpcionCorrecta**, y se devuelve el puntaje correspondiente.

7.1.2. Multiple Choice Penalidad

Las preguntas del tipo Multiple Choice con Penalidad almacenan una lista de opciones que pueden ser de las clases **OpcionCorrecta** o **OpcionIncorrecta**. Estas preguntas puntúan las respuestas de los jugadores sumando un punto por cada opción correcta seleccionada y restando un punto por cada opción incorrecta seleccionada. Para lograr esto, las respuestas seleccionadas por el jugador se comparan con la lista de opciones almacenadas. Como resultado de esta comparación, cada selección del jugador recibe un puntaje de 1 o -1. Se espera que no se obtenga un puntaje de 0, ya que el jugador no puede seleccionar opciones que no están disponibles en la interfaz gráfica. El puntaje total para la respuesta del jugador se calcula sumando los puntajes de cada una de sus selecciones.

7.1.3. Ordered Choice

Las preguntas del tipo Ordered Choice almacenan sus opciones en un orden específico en un objeto llamado **Orden** correcto. Los jugadores que respondan en ese mismo orden reciben un punto; de lo contrario, no obtienen ningún punto. Para puntuar, la pregunta compara la respuesta del jugador, que se envía en el orden seleccionado, con el Orden correcto almacenado. Si coinciden, se otorga un punto; de lo contrario, se puntúa con cero.

7.2. Puntajes

7.2.1. Puntajes Parciales y Modificadores Individiuales

El puntaje obtenido por un jugador al responder una pregunta puede ser afectado por múltiples modificadores. Por ejemplo, si el jugador utiliza los multiplicadores x2 y x3, su puntaje se multiplicará por x6. Sin embargo, si además otro jugador utiliza un anulador, el puntaje del primer jugador será cero.

Para abordar esta problemática, decidimos implementar los puntajes parciales mediante un sistema que consta de un puntaje base, dependiente de la precisión de la respuesta del jugador, y una lista de modificadores individuales. Esta lista comienza con un modificador base (multiplicador x1) al cual se le pueden agregar otros modificadores. De esta manera, al consultar el puntaje obtenido por el jugador en una pregunta, el objeto PuntajeParcial ajustará su puntaje base según los modificadores que se le hayan aplicado.

```
public class PuntajeParcial {
    private int puntajeBase;
    private List<ModificadorIndividual> modificadores;

    public PuntajeParcial() {
        this.puntajeBase = 0;
        this.modificadores = new ArrayList<ModificadorIndividual>();
        this.modificadores.add(new ModificadorIndividualBase());
    }

    [. . .]

    public int obtenerPuntos() {
        return this.modificadores.stream()
                .reduce(puntajeBase,
                        (puntaje, mod) -> mod.modificar(puntaje),
                        (puntaje1, puntaje2) -> puntaje2);
    }
}
```

7.2.2. Modificadores Globales

Los modificadores globales, como el anulador y la exclusividad, requieren acceso a la lista completa de puntajes parciales para poder ser aplicados. Debido a esta necesidad, estos modificadores no se aplican hasta que se finaliza la ronda de preguntas.

7.3. Disponibilidad de Poderes

Los jugadores mantienen como atributos objetos de los modificadores que tienen disponibles para usar. Cuando un jugador utiliza uno de sus modificadores al responder una pregunta, dicho modificador se *gasta* y ya no está disponible para próximas preguntas. Para crear los botones en la interfaz gráfica según la disponibilidad de poderes de un jugador, se utilizan métodos que reciben funciones anónimas: porCadaPoderClasico() y porCadaPoderPenalidad().

```
public class Jugador {
    private String nombre;
    private PuntajeTotal puntosTotales;
    private ModificadorIndividual duplicador;
    private ModificadorIndividual triplicador;
    private ModificadorGlobal anulador;
    private ModificadorGlobal exclusividad;
    [. . .]
    public void porCadaPoderClasico( OperacionPoderesClasicos operacion ) {
        operacion.operar(anulador,exclusividad);
    }
}

public class RondaDePreguntas {
    [. . .]
    public void jugar(Jugador j, List<ModificadorIndividual> modsInd,
    List<ModificadorGlobal> modsGlob, Respuesta... respuestas){
        [. . .]
        modsInd.forEach(j::gastar);
        modsGlob.forEach(j::gastar);
        [. . .]
    }
    [. . .]
}
```

Los botones en la interfaz gráfica se inicializan recibiendo objetos que implementen las interfaces ModificadorGlobal o ModificadorIndividual según corresponda. Si el objeto proporcionado no es de la clase esperada (por ejemplo, una instancia de Anulador para un BotonAnulador), el botón se inicializa deshabilitado.

```
public class ConjuntoPoderesClasicos extends ConjuntoPoderes{
    [. . .]
    public void reestablecer(Jugador jugador) {
        botonesPoderes.clear();
        jugador.porCadaPoderClasico(
            (a,e) -> {
                botonesPoderes.add(new BotonAnulador(a));
                botonesPoderes.add(new BotonExclusividad(e));
            }
        );
```

```
            actualizarBotones();
        }
    }

    public class BotonExclusividad extends CustomToggleButton implements BotonPoderGlobal {
        private final ModificadorGlobal modificador;
        public BotonExclusividad(ModificadorGlobal mod) {
            [. . .]
            this.modificador = mod;
            ModificadorGlobal tipoBoton = new Exclusividad();
            if (mod.getClass() != tipoBoton.getClass()) {this.disableProperty().set(true);}
        }

        public ModificadorGlobal obtenerModificador() {
            return this.isSelected() ? modificador : null;
        }
    }
```

7.4. Patrones De Diseño

7.4.1. Singleton

Se aplicó el patrón de diseño **Singleton** en la creación de la clase **AlgoHoot**. Este patrón garantiza que solo exista una única instancia de la clase **AlgoHoot**, la cual funciona además como el principal punto de acceso a la información de la partida y a la lógica del modelo. También proporciona un punto de acceso global a través de la clase **AlgoHoot** desde cualquier parte del programa, facilitando su uso al evitar tener que pasar una instancia única de **AlgoHoot** entre los distintos paquetes que componen la aplicación.

7.4.2. Factory Method

Se aplicó el patrón de diseño **Factory Method** para resolver el problema de cómo crear instancias de las distintas clases de preguntas utilizando los datos obtenidos del archivo **JSON** proporcionado por la cátedra.

Para crear las preguntas, este patrón se implementó en la clase **Lector**, la cual lee uno por uno los objetos que se encuentran dentro del archivo **JSON** y utiliza el valor de la clave 'tipo' de cada objeto para determinar que fabrica utilizar. Una vez asignada la fabrica apropiada se le pasan los datos del objeto como parámetros al método **crearPregunta()** de dicha fabrica, una vez creada la **Pregunta** se almacena en una lista para al finalizar la lectura del archivo retornar la lista.

7.4.3. Null Object

Se aplicó el patrón de diseño **Null Object** para los modificadores de puntajes. Dado que los modificadores pueden acumularse, se decidió almacenarlos en listas. Al momento de modificar un puntaje, se recorre la lista aplicando cada modificador. La implementación de modificadores nulos (ModificadorIndividualBase y ModificadorGlobalBase) agiliza este proceso. Si un jugador no ha utilizado ningún modificador, se envía un modificador "base", lo que evita modificar el comportamiento de las clases que manejan modificadores y elimina la necesidad de implementar casos especiales para cuando no se utilizan modificadores.

8. Excepciones

ArchivoInexiste Esta es lanzada cuando el archivo de preguntas no existe o no se encuentra en su carpeta correspondiente.