.UBAfiuba
FACULTAD DE INGENIERÍA

# Trabajo Práctico 2 — Java

[7507/9502] Algoritmos y Programación III
Curso 1
Primer cuatrimestre de 2020


| **Alumno** | **Padrón** | **Mail** |
|---|---|---|
| Gomez, Joaquin | 103735 | joagomez@fi.uba.ar |
| Grassano, Bruno | 103855 | bgrassano@fi.uba.ar |
| Roussillian, Juan Cruz | 104269 | jroussilian@fi.uba.ar |
| Stancanelli, Guillermo | 104244 | gstancanelli@fi.uba.ar |

# Índice

# 1. Introducción

El presente informe reúne la documentación de la solución del segundo trabajo práctico de la materia Algoritmos y Programación III que consiste en desarrollar un juego estilo "Kahoot" en Java utilizando los conceptos del paradigma de la orientación a objetos vistos en el curso.

# 2. Supuestos

El trabajo se realizo con los siguientes supuestos que fueron surgiendo a lo largo de la realización del trabajo, ya que estos casos no están contenidos en las especificaciones entregadas en el enunciado.

* Es posible tener una sola respuesta correcta en un multiple choice. No necesariamente tienen que ser como mínimo dos correctas, se puede llegar a tener una correcta y otra incorrecta.
* En una pregunta de puntaje parcial, no necesariamente tiene el jugador que responder todas las opciones correctas para que se aplique la exclusividad.
* Los jugadores para responder deben de seleccionar mínimo una opción en multiple choice, no sera valido no responder nada.
* Es valido tener grupos vacíos. Por ejemplo, un grupo de verduras, y otro de frutas, donde las opciones que hay para elegir pertenecen todas al de frutas.
* Se considera que el jugador puede usar el multiplicador x2 y x3 en una misma ronda, no se le limita a usar como máximo uno a la vez. De esta forma el jugador puede multiplicar x6 sus puntos en una ronda.

# 3. Modelo de dominio

A continuación se detallaran brevemente las responsabilidades y razones por las que fue hecha cada parte del modelo.

## Paquete "modelo"

En este paquete se encuentra el código correspondiente a todo el modelo del juego. las siguientes subsecciones están adentro de este paquete.

* **Clase Jugador** Representa a un jugador de "Algohoot". Tiene un nombre que es elegido por el usuario y modificadores para usarse a lo largo del juego. Al final del juego se determina un ganador entre los dos jugadores posibles en base a quien obtuvo el mayor puntaje.
* **Clase Jugada** Representa a una jugada del juego. Esta clase se encarga de delegar y manejar lo correspondiente a una ronda.
* **Clase AlgoHoot** Esta clase agrupa a todas las jugadas que tendrá el juego. Tiene una visión general del mismo al inicializar el juego. Para esta clase se utilizo un patrón Singleton.
* **Interfaz Ganador** Esta interfaz existe para manejar la parte final del juego. Estos casos pueden ser cuando hay un ganador o cuando ocurre un empate. Las clases que la implementan son *Jugador* y *Empate*.

## Paquete "preguntas"

Este paquete contiene toda las clases que se encargan de la representación de las preguntas. A continuación se nombran las que están en el paquete general.

* **Clase abstracta Pregunta** Esta clase garantiza que los diferentes tipos de preguntas posibles entiendan los mismos mensajes y se pueda manejar desde *Jugada* sin inconvenientes. Al principio se usaba una interfaz, pero con el desarrollo del trabajo se vio que se repetía código entre las distintas preguntas, por lo que se decidió convertirla en una clase abstracta.
* **Interfaz PreguntaAutoEvaluable** Esta interfaz esta para las preguntas de Multiple Choice y Verdadero Falso. Les agrega los métodos para que esperen una respuesta valida.
* **Interfaz PreguntaComparable** Similar a la anterior, solo que para las preguntas de grupo y orden.
* **Interfaz OpcionEvaluable** Es una interfaz que esta para las opciones que tienen las preguntas de verdadero falso y multiple choice. Estas saben si son correctas o no por su cuenta como se vera mas adelante.
* **Clase OpcionSimple** Esta clase esta para las preguntas de orden y grupo, cuyo resultado depende mas del conjunto y no de las respuestas individuales.
* **Clase FabricaDePreguntas** Es una clase que simplifica la creación de preguntas. No es del todo un patrón factory, esta clase es llamada desde el lector y las pruebas.

## Paquete "verdaderoFalso"

Se nombran las partes del modelo correspondientes al verdadero falso.

* **Clase VerdaderoFalso** Esta clase extiende de *Pregunta* y representa a una pregunta de verdadero o falso, la cual solo puede tener dos respuestas posibles.
* **Interfaz PuntajeVerdaderoFalso** Esta interfaz está para que solamente se le pueda aplicar a un verdadero falso los puntajes clásicos y penalizables.

* **Clases OpcionCorrectaVerdaderoFalso y OpcionIncorrectaVerdaderoFalso** Estas clases son las respuestas que tiene la pregunta de verdadero falso. Implementan la interfaz *OpcionEvaluable*. Estas opciones pueden tener como texto solo verdadero o solo falso.

## Paquete "multipleChoice"

Se nombran las partes del modelo correspondientes al multiple choice.

* **Clase MultipleChoice** Esta clase extiende de *Pregunta* y representa a una pregunta de multiple choice.

* **Interfaz PuntajeMultipleChoice** Esta interfaz esta para separar los puntajes que puede tener el multiple choice con respecto al verdadero falso. En este caso, acepta los tres tipos de puntaje que hay.

* **Clases OpcionCorrectaMultipleChoice y OpcionIncorrectaMultipleChoice** Estas clases son las respuestas que tiene la pregunta de multiple choice. Implementan la interfaz *OpcionEvaluable*, ya que saben por si mismas si son correctas o no.

## Paquete "groupChoice"

Se nombran las partes del modelo correspondientes a la pregunta de Group Choice.

* **Clase GroupChoice** Esta clase extiende de *Pregunta* y representa a una pregunta de grupos.

* **Clase Grupo** Esta clase representa al grupo de un conjunto de respuestas. Los grupos son los encargados de saber compararse entre si.

## Paquete "orderedChoice"

Se nombran las entidades de la pregunta de orden ahora.

* **Clase OrderedChoice** Esta clase implementa *Pregunta*. Su función es representar a la pregunta de orden.

* **Clase Orden** Representa al orden que tendrá la pregunta. Sabe comparar los ordenes también.

## Paquete "puntajes"

A continuación se detallan los puntajes que tiene el juego.

Estas clases de puntaje están actuando como las fabricas concretas de un patrón de 'abstract factory' en donde las fabricas abstractas son *PuntajeMultipleChoice* y *PuntajeVerdaderoFalso*. Los productos que instancian son los diferentes resultados posibles. Esto se vera mejor mas adelante.

* **Clase PuntajeClasico** Representa un puntaje normal, es decir que si respondiste bien recibís los puntos, caso contrario no se reciben puntos. Implementa las interfaces *PuntajeChoice* y *PuntajeVerdaderoFalso*.

* **Clase PuntajeParcial** Representa un puntaje parcial, en este tipo de puntaje solo se reciben puntos si no se respondió alguna respuesta incorrecta. Este tipo esta solamente para el método de choice. Implementa la interfaz *PuntajeChoice*.

* **Clase PuntajePenalizable** Representa un puntaje penalizable, da puntos por cada respuesta correcta elegida, y quita por cada incorrecta. Implementa las interfaces *PuntajeChoice* y *PuntajeVerdaderoFalso*.

## Paquete "resultado"

Se muestran las clases que se encuentran en el subpaquete de resultados. Estos resultados son instanciados por el tipo de puntaje que tenga la pregunta, resultando ser los productos concretos de las fabricas mencionadas antes.

* **Interfaz Resultado** Esta interfaz ayuda a un manejo general de los diferentes resultados que hay. Hace que las diferentes clases se independicen del resultado que se este usando.
* **Clase ResultadoClasico y ResultadoParcial** Implementa la interfaz *Resultado*. Internamente tiene un estado que representara si se respondió correctamente o no.
* **Clase ResultadoPenalizable** Implementa la interfaz también. Ira acumulando los puntos que le corresponda al usuario.
* **Interfaz EstadoResultadoParcial** Esta interfaz está para el resultado parcial, ya que este tiene dos estados posibles.
* **Interfaz EstadoResultadoClasico** Idéntico al caso anterior. Se hizo para separar y evitar que un resultado parcial tenga un estado perteneciente al clásico. Con estas dos interfaces se utilizo un patron State, ya que en caso de responder incorrectamente se cambia de estado.
* **Clase EstadoCorrectoClasico, EstadoCorrectoParcial y EstadoIncorrecto** Implementan sus correspondientes interfaces. Estas clases son las encargadas de acumular los puntos o no si es que corresponde.

## Paquete "puntos"

Se detallan las entidades que representan al sistema de puntos del modelo. Se utilizo un patrón Composite, el cual permitió reducir la cantidad de 'ifs' de las primeras versiones del trabajo. Al aplicar este patrón, se observó que aumento la cobertura del sistema, ya que se cubrían mas casos.

* **Interfaz Punto** Interfaz que asegura que los diferentes puntos puedan multiplicarse y mostrarse.
* **Clase Puntuacion** Implementa la interfaz *Punto*. Representa a un conjunto de puntos. Este puede contener cualquier objeto que implemente la interfaz.
* **Clase PuntoNulo, PuntoPositivo, PuntoNegativo** Implementan la interfaz *Punto*. Cada clase representa a un punto distinto con un valor unico.
* **Clase PuntuacionRepresentable** Esta clase esta para acumular los valores numéricos de los diferentes puntos. Esta clase es la que sabe representar su valor.

## Paquete "modificadores"

Se muestran las partes correspondientes al sistema de modificadores que presenta el juego. Se utiliza Double Dispatch para la asignación de un modificador a una pregunta, y para saber si aplica la exclusividad.

* **Interfaz Modificador** Interfaz que permite manejar de forma mas modular los modificadores del juego.
* **Interfaz Multiplicador** Interfaz para que los multiplicadores tengan el mismo tipo y entiendan el mismo mensaje.
* **Clase Exclusividad** Representa a la exclusividad del juego. Esta clase se encarga de delegar en un analizador el ver si alguien la gano.

* **Interfaz SituacionesExclusividad** Esta interfaz se crea para que se puedan juntar los estados y analizarlos al momento de querer ver si alguien gano la exclusividad.

* **Clase AnalizadorExclusividad** Analiza los resultados, viendo si ambos respondieron correctamente, si se equivoco uno, o se equivocaron ambos.

* **Clase UsuarioRespondioBien y UsuarioSeEquivoco** Representan un estado que almacena el analizador. Se crea en base al resultado que se tenga.

* **Interfaz EstadoAnalizador** Esta interfaz esta para que se pueda ir cambiando el estado que tiene el analizador, estos estados son:

    1. cuando tiene una sola situación
    2. cuando ya tiene las dos y puede determinar si aplica la exclusividad (tiene la información necesaria para funcionar)
    3. cuando ya se uso la exclusividad. Este ultimo estado se creo para hacer mas claro el proceso.

* **Clase MultiplicadorJugador** Es el multiplicador que tiene el jugador. Este puede ser el x2 o el x3 y puede usarlo en una pregunta penalizable.

* **Clase MultiplicadorExclusividad** Es un multiplicador único para la exclusividad. Este es siempre x2 y su uso es interno.

## Paquete "respuestas"

Se ponen las entidades del subpaquete respuestas. Estas entidades van a representar a las respuestas que haya hecho el jugador a una determinada pregunta.

* **Interfaz Respuesta** Esta interfaz hace que las preguntas no sepan los distintos tipos de respuestas que hay. Solamente conocen el que les pertenece. Esto se consigue con un Double Dispatch.

* **Interfaz RespuestaAutoEvaluable** Esta interfaz está para las respuestas que por si mismas ya saben si son correctas o no, ellas son las que se evalúan. Su uso es en las preguntas auto evaluables.

* **Interfaz RespuestaComparable** Esta interfaz esta para las respuestas que para saber si son correctas hay que evaluar el conjunto, comparándolos entre si.

* **Clase RespuestaVerdaderoFalso y RespuestaMultipleChoice** Implementa a *RespuestaAutoEvaluable*. Delega en las opciones que respondió el usuario el evaluarse.

* **Clase RespuestaOrderedChoice y RespuestaGroupChoice** Implementa a *RespuestaComparable*. Tienen sus formas de compararse con los grupos/ordenes.

## Paquete "turnos"

Este paquete tiene la lógica del sistema de turnos que tiene el juego. Con ella se aplica un sistema de patrón State.

* **Clase abstracta Turno** Reúne la información y métodos de lo que corresponde a un turno, por ejemplo el jugador. Las siguientes clases son las implementaciones.

* **Clase TurnoPrimerJugador, TurnoSegundoJugador y TerminoJuego** Son los estados que puede tener el juego. Este ira alternando entre los dos primeros hasta que se acaban las preguntas.

# 4. Diagramas de clase

En la figura 1 podemos observar las relaciones de las diferentes clases. En este diagrama no se incluyeron los métodos y atributos para dar mas claridad al mismo. Estos serán mostrados en otros diagramas.

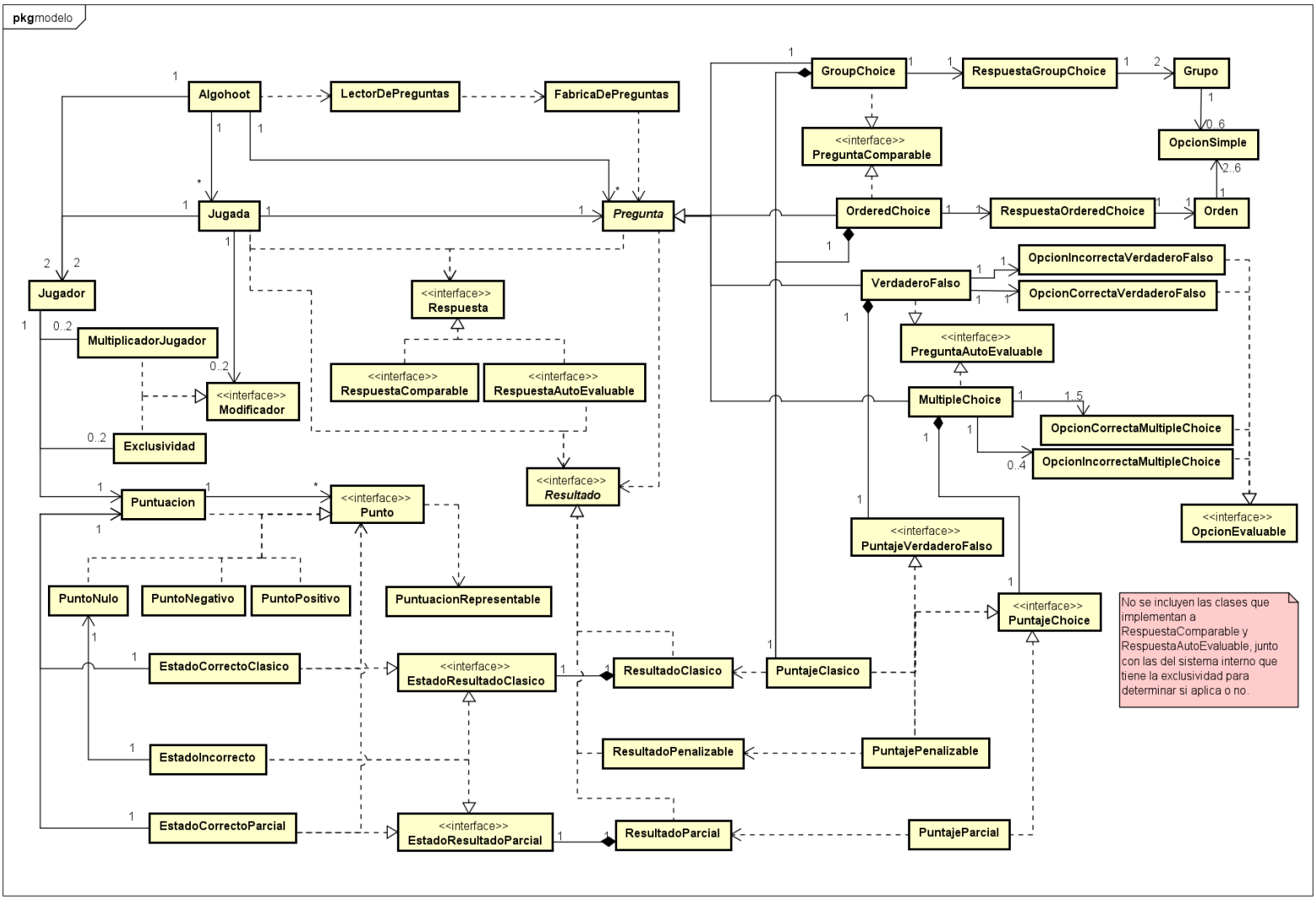


Figura 1: Diagrama general del trabajo

Se muestra ahora la parte que no se incluyo en el anterior diagrama. Los detalles internos de la exclusividad se ven en la figura 5.

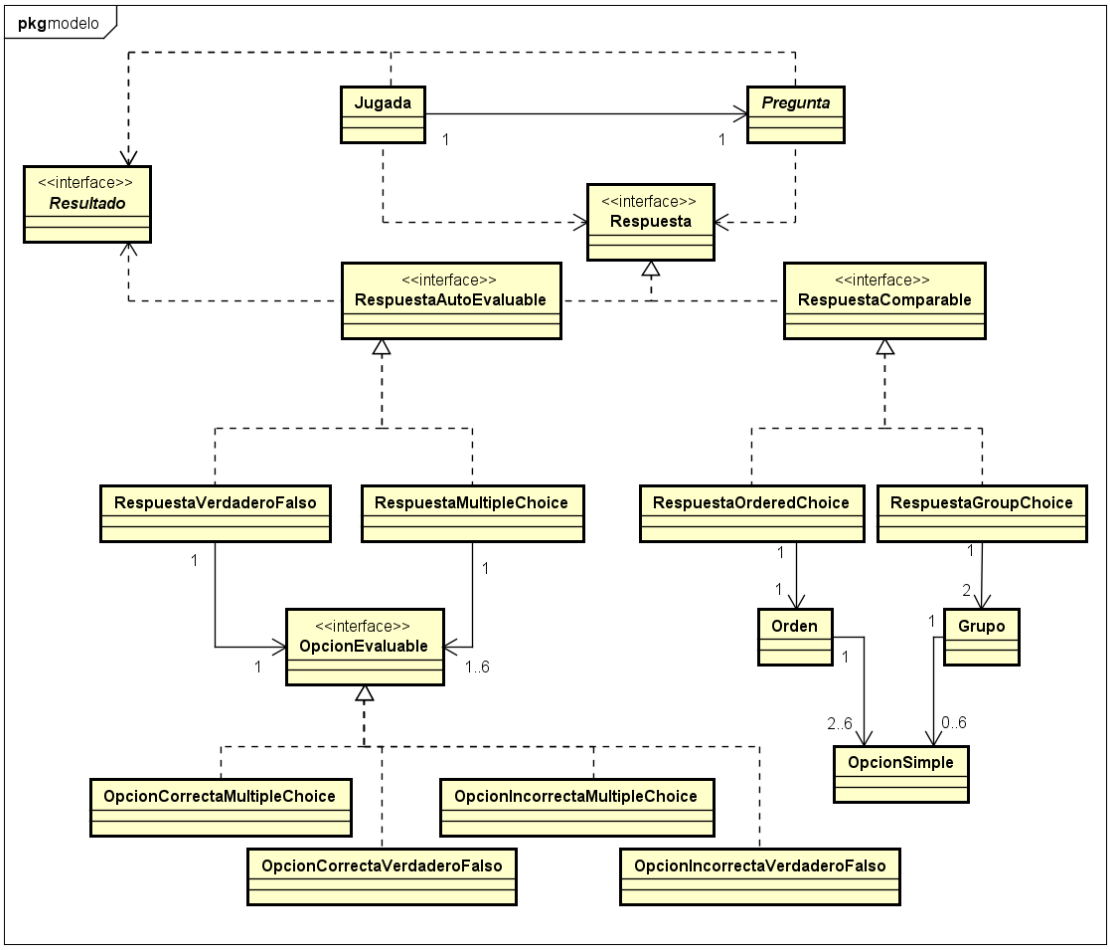


Figura 2: Diagrama mostrando las respuestas a las preguntas

La figura 3 muestra los detalles del sistema de puntajes del juego. La parte correspondiente a *obtenerResultado* cumple con un patrón de fabrica abstracta, mientras que la de *puedeUsarModificador* es parte de un Double Dispatch con los modificadores.

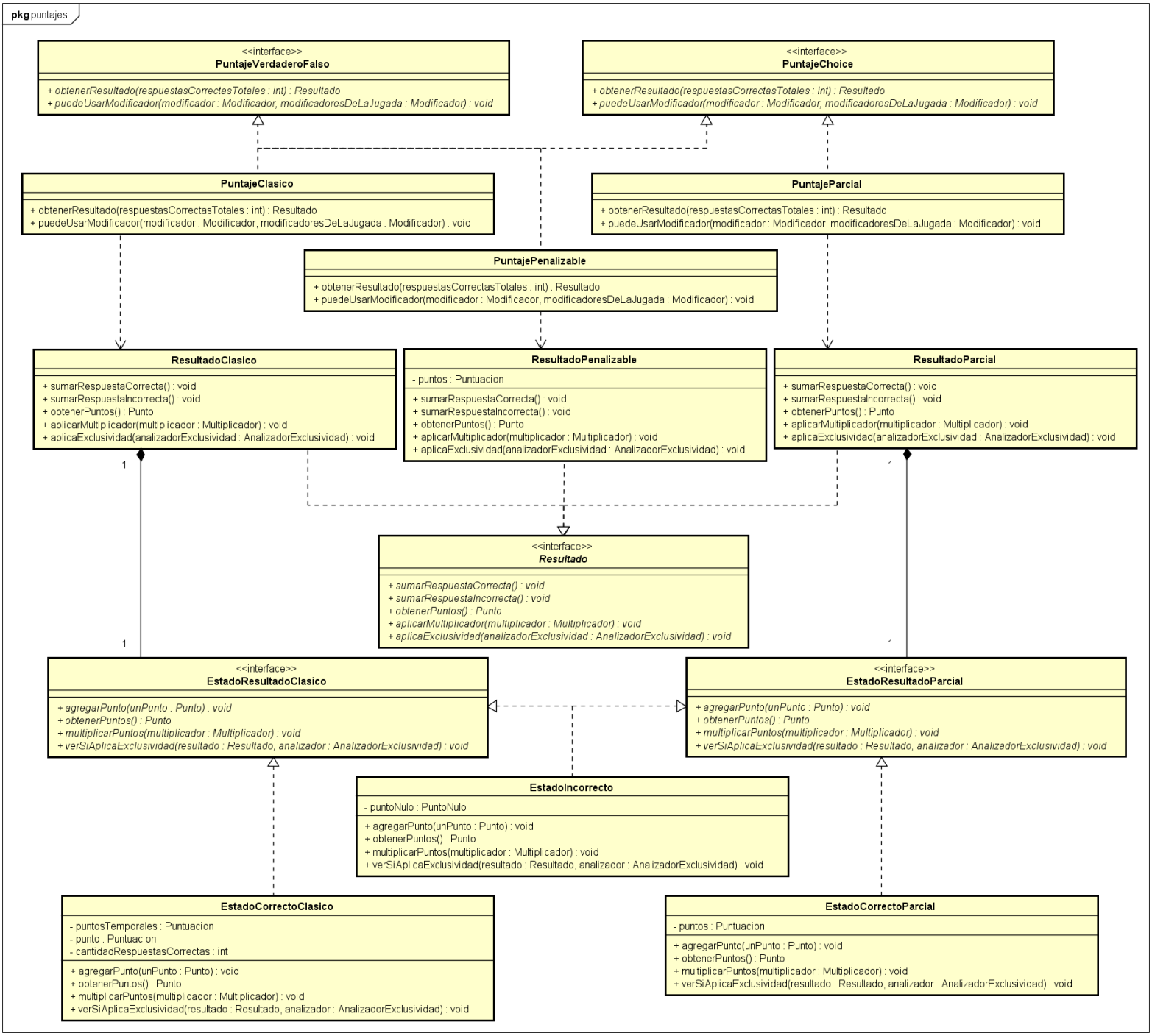


Figura 3: Diagrama del sistema de puntos

En este diagrama de clases (Figura 4) vemos las relaciones del sistema de puntos del juego. Como se puede ver, se utiliza un patrón Composite.

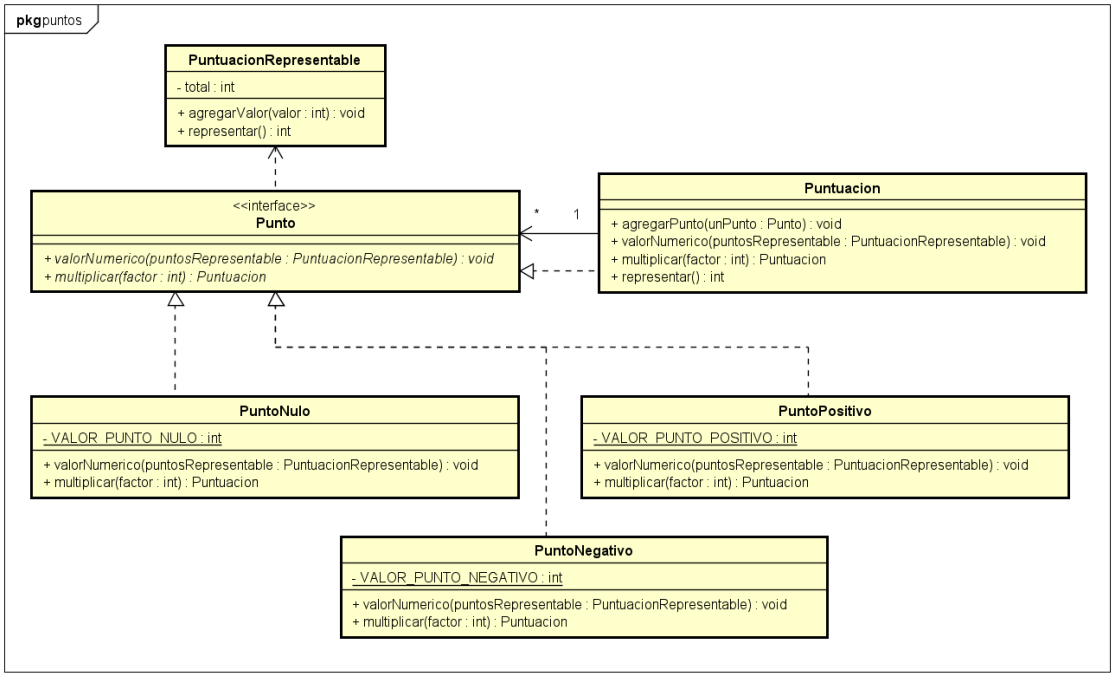


Figura 4: Diagrama del sistema de puntos

En este diagrama (Figura 5) se muestran las relaciones del sistema de modificadores del juego. En el caso del analizador de exclusividad se utiliza un Double Dispatch al momento de comparar situaciones. Este método de comparación se usa también cuando se quiere saber si se agrega un modificador o no a la jugada. No se incluyeron los estados internos que puede tener el resultado, estos son los que se encargan de agregar las situaciones al analizador como se pudo ver en el diagrama de la figura 3 con el método *verSiAplicaExclusividad*.

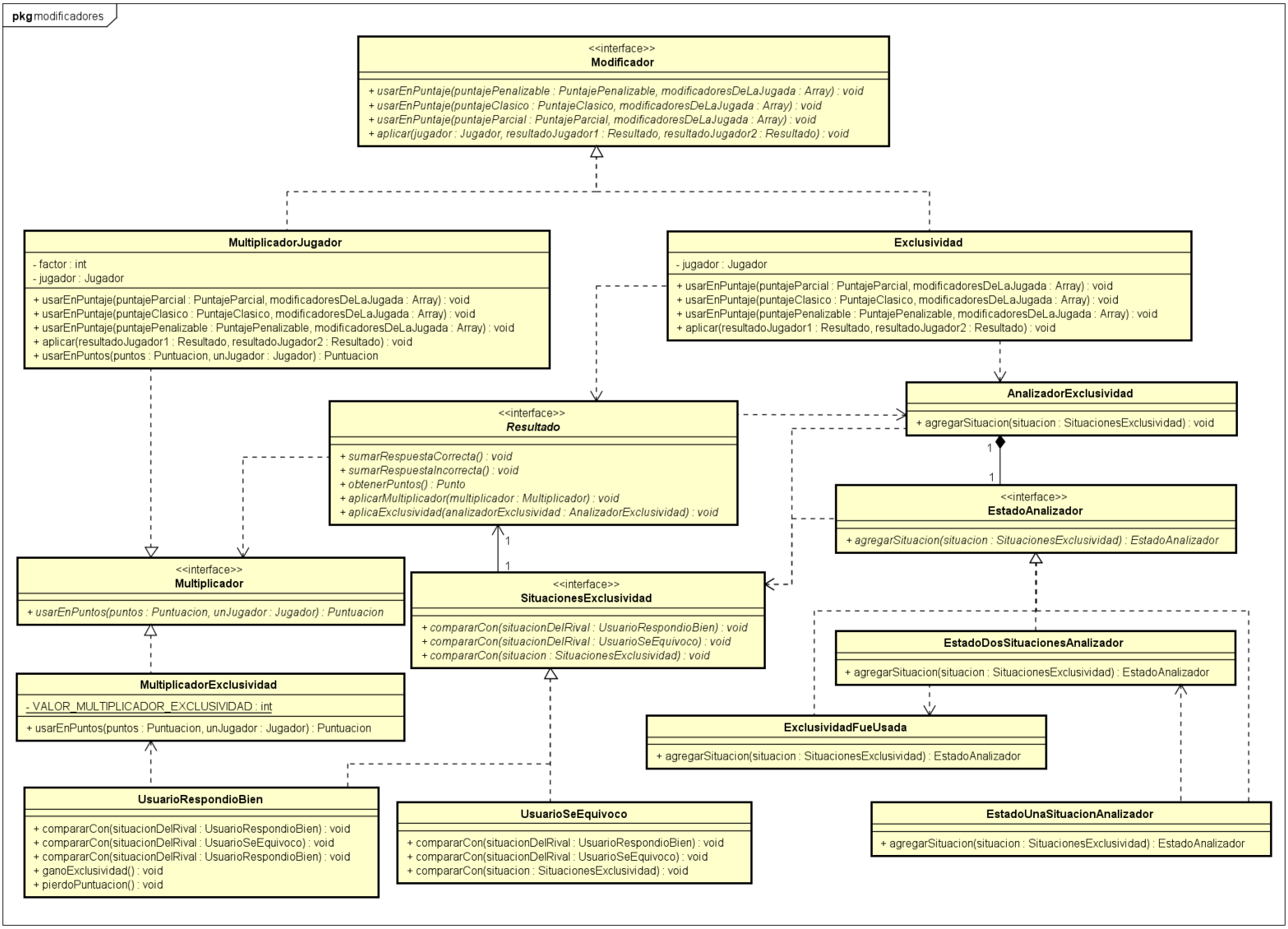


Figura 5: Diagrama del sistema de modificadores que presenta el juego.

En este diagrama se muestran las clases de las preguntas que tiene el juego.

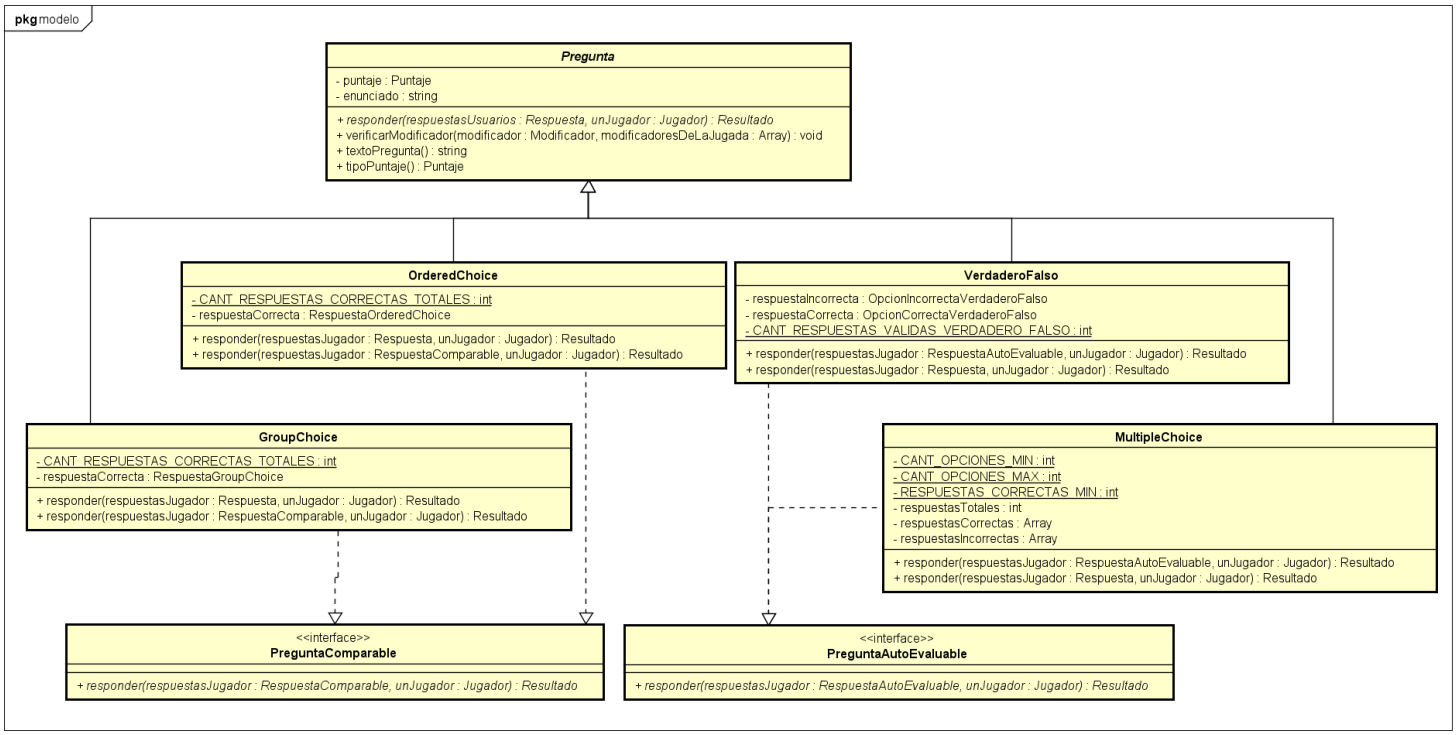


Figura 6: Diagrama mostrando en detalle las preguntas que tiene el juego, no se incluyeron algunos getters

# 5. Diagramas de Paquetes

Mostramos a continuación un diagrama de paquetes del paquete modelo. No se incluyeron en el diagrama las relaciones con las clases que no tienen un paquete en especifico. Tampoco se incluyeron las relaciones internas que hay en el paquete de preguntas.

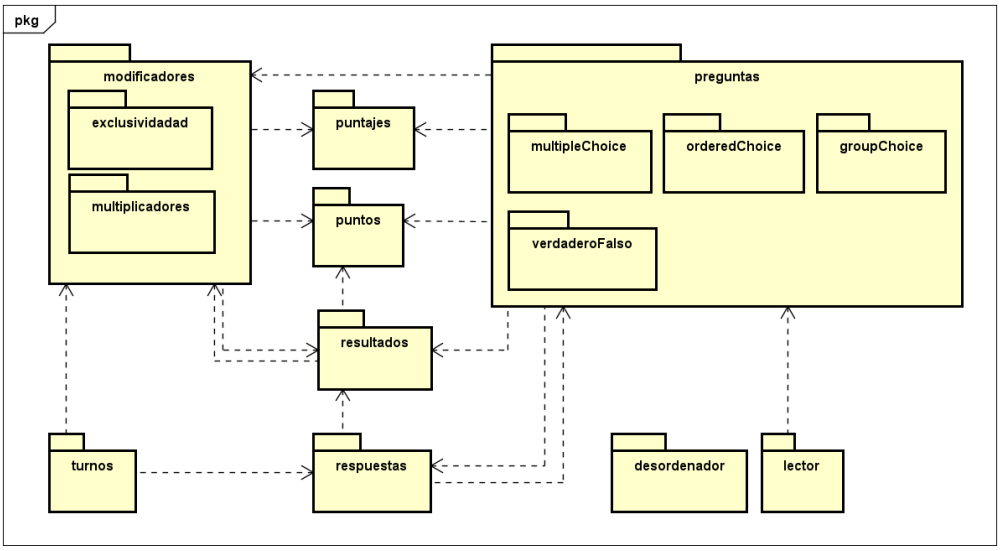


Figura 7: Diagrama de paquetes del modelo

# 6. Detalles de implementación

## 6.1. Clase Jugador

Esta clase fue creada para representar al jugador. Su estructura esta compuesta por su nombre, los puntos que tenga, y los modificadores con los que comienza el juego. Estos son 2 multiplicadores (un x2 y un x3), y otros 2 modificadores de exclusividad. Estos los podrá usar dependiendo del tipo de pregunta.

Los métodos que tiene son sumarPuntos(), obtenerPuntos(), pierdeModificador(), obtenerNombre(), compararYObtenerGanador(). Se muestran algunos de ellos.

Otra cosa que tiene esta clase es que implementa la interfaz *Ganador*.

```
public void sumarPuntos(Punto unosPuntos){
    puntos.agregarPunto(unosPuntos);
}

public Ganador compararYObtenerGanador(Jugador jugadorContrario){
    if(this.obtenerPuntos() < jugadorContrario.obtenerPuntos()){
        return jugadorContrario;
    }else if (this.obtenerPuntos() > jugadorContrario.obtenerPuntos()){
        return this;
    }else {
        return new Empate();
    }
}
```

## 6.2. Clase VerdaderoFalso y MultipleChoice

Las principal diferencia entre estas clases es la información que contienen, ya que la de verdadero o falso va a contener siempre dos opciones, mientras que la de multiple choice puede tener más. Se detallan los métodos principales que tienen. Extienden a la clase de Pregunta.

#### Método responder

Recibe la respuesta enviada por el usuario en una ronda del juego, y analizara si le corresponden puntos. Esto lo hace pidiéndole a esa respuesta que se analice junto con un resultado. El resultado este se lo pide a la fabrica concreta que tiene representada por su puntaje.

Inicialmente recibe una respuesta genérica, y a través de un Double Dispatch llega al método responder correcto que viene de la interfaz.

```
public Resultado responder(Respuesta respuestasUsuario, Jugador unJugador) {
    return respuestasUsuario.evaluarEnBaseAPregunta(this,unJugador);
}

@Override
public Resultado responder(RespuestaAutoEvaluable respuestasJugador, Jugador unJugador) {
    Resultado unResultado =
            puntaje.obtenerResultado(CANT_RESPUESTAS_VALIDAS_VERDADERO_FALSO,unJugador);
    respuestasJugador.evaluar(unResultado);
    return unResultado;
}
```

## 6.3. Clase OpcionCorrectaVerdaderoFalso y OpcionIncorrectaVerdaderoFalso

Se muestran los métodos importantes que tienen. Como se ve, le indican al resultado del jugador si se respondió bien o no. Lo mismo sucede con las opciones que tiene el multiple choice.

```
 //De la opcion correcta

    @Override
    public void evaluar(Resultado unResultado) {
        unResultado.sumarRespuestaCorrecta();
    }

//------//
//De la opcion incorrecta

    @Override
    public void evaluar(Resultado unResultado) {
        unResultado.sumarRespuestaIncorrecta();
    }
```

## 6.4. Interfaz Puntaje

Esta interfaz esta para poder tener un puntaje en la clase abstracta de pregunta independientemente de que tipo en especifico es. Surgió ya que al principio, al estar separados los puntajes posibles, se necesitaba una forma de poder juntarlos y evitar repetir el codigo en *Pregunta*. Se muestran los métodos que tiene.

```
void puedeUsarModificador(Modificador modificador,
                ArrayList<Modificador> modificadoresDeLaJugada);

Resultado obtenerResultado(int respuestasCorrectasTotales, Jugador unJugador);
```

## 6.5. Interfaz PuntajeVerdaderoFalso y PuntajeChoice

Estas interfaces extienden a Puntaje y permiten separar los tipos de puntajes que puede tener una pregunta verdadero-falso de una de multiple choice. Los puntajes que tiene disponibles son clásico y penalizable para la primera, y para la segunda los mismos mas parcial. El método que tienen es:

```
Resultado obtenerResultado(int respuestasCorrectasTotales, Jugador unJugador);
```

## 6.6. Clase PuntajeClasico, PuntajeParcial, y PuntajePenalizable

Estas son las implementaciones de las interfaces *PuntajeChoice* y *PuntajeVerdaderoFalso* dependiendo el caso. Esas dos interfaces actúan como fabricas abstractas, y estas ultimas clases como fabricas concretas, que envían como producto concreto un resultado. Se muestra el código correspondiente.

```
@Override
public Resultado obtenerResultado(int respuestasCorrectasTotales, Jugador unJugador) {
    return new ResultadoClasico(respuestasCorrectasTotales,unJugador);
}

@Override
public Resultado obtenerResultado(int respuestasCorrectasTotales, Jugador unJugador) {
    return new ResultadoPenalizable(unJugador);
}

@Override
public Resultado obtenerResultado(int respuestasCorrectasTotales, Jugador unJugador) {
    return new ResultadoParcial(unJugador);
}
```

Otra cosa que hacen es iniciar un Double Dispatch con los modificadores, para saber si se puede usar un modificador en una determinada pregunta. Se muestra el código.

```
@Override
public void puedeUsarModificador(Modificador modificador,
                    ArrayList<Modificador> modificadoresDeLaJugada) {
    modificador.usarEnPuntaje(this,modificadoresDeLaJugada);
}
```

## 6.7. Interfaz Punto

Esta interfaz se usa en el sistema de puntos del juego, con ella se logra el patrón Composite. Los métodos que tiene son:

```
Puntuacion multiplicar(int factor);

void valorNumerico(PuntuacionRepresentable puntuacionRepresentable);
```

## 6.8. Clase Puntuacion

Implementa la interfaz *Punto*. Esta clase guarda el conjunto de puntos que van teniendo los jugadores, y que van almacenando los resultados de cada jugada. Se detallan los métodos importantes de esta clase.

#### Método valorNumerico

Este método lo que hace es representar numéricamente el valor total de los puntos acumulados. Lo consigue delegándole a un objeto que se encarga de representarse. Este objeto acumula el valor total y se muestra

```
@Override
public void valorNumerico(PuntuacionRepresentable puntuacionRepresentable) {
    for (Punto punto:puntos) {
        punto.valorNumerico(puntuacionRepresentable);
    }
}

public int representar() {
    PuntuacionRepresentable puntuacionRepresentable = new PuntuacionRepresentable();
    for (Punto punto:puntos) {
        punto.valorNumerico(puntuacionRepresentable);
    }
    return puntuacionRepresentable.representar();
}
```

#### Método multiplicar

Este método multiplica los puntos por el factor que se le indica. Para eso crea una nueva puntuación e ira guardando ahi los puntos multiplicados.

```
public Puntuacion multiplicar(int factor) {
    Puntuacion puntuacionMultiplicada = new Puntuacion();
    for (Punto punto:puntos) {
        puntuacionMultiplicada.agregarPunto(punto.multiplicar(factor));
    }
    return puntuacionMultiplicada;
}
```

### 6.9. Clases PuntoNulo, PuntoPositivo, y PuntoNegativo

Implementan la interfaz *Punto*. Cada uno representa a un punto con un determinado valor numérico. Se muestran las implementaciones de los métodos que tienen de la interfaz. En el caso del de valor numérico esta el del *PuntoNulo* para no incluir todos.

```
@Override
public Puntuacion multiplicar(int factor) {
    Puntuacion puntosMultiplicados = new Puntuacion();
    for(int i = 0; i < factor ; i++){
        puntosMultiplicados.agregarPunto(this);
    }
    return puntosMultiplicados;
}

@Override
public void valorNumerico(PuntuacionRepresentable puntuacionRepresentable) {
    puntuacionRepresentable.agregarValor(VALOR_PUNTO_NULO);
}
```

## 6.10. Clases ResultadoClasico y ResultadoParcial

Estos resultados son similares en cuanto a que ambos tienen un estado interno, seria un patron State. Inician con un estado correcto, y en caso de que el jugador haya respondido incorrectamente se cambia el estado a uno incorrecto. Se muestra el caso del clásico, donde las operaciones son delegadas al estado.

```
public ResultadoClasico(int unaCantidadDeRespuesasCorrectasTotales, Jugador unJugador) {
    estado = new EstadoCorrectoClasico(unaCantidadDeRespuesasCorrectasTotales);
    jugador = unJugador;
}


@Override
public void sumarRespuestaCorrecta() {
    estado.agregarPunto(new PuntoPositivo());
}

@Override
public void sumarRespuestaIncorrecta() {
    estado = new EstadoIncorrecto();
}

@Override
public Punto obtenerPuntos() {
    return estado.obtenerPuntos();
}

@Override
public void aplicarMultiplicador(Multiplicador multiplicador) {
    estado.multiplicarPuntos(multiplicador,jugador);
}

@Override
public void aplicaExclusividad(AnalizadorExclusividad analizador) {
    estado.verSiAplicaExclusividad(this,analizador);
}
```

## 6.11. Clases EstadoCorrectoClasico, Estado Incorrecto, y EstadoCorrectoParcial

Estos son los estados que usan las clases mencionadas anteriormente. Se muestra el código, comparando los métodos de ambos estados. Arriba se encuentra el de uno correcto (se muestra el del parcial), y abajo el del incorrecto.

```
@Override
public void agregarPunto(Punto unPunto) {
    puntos.agregarPunto(unPunto);
}

@Override
public void agregarPunto(Punto unPunto) {
}

//----//
```

```
@Override
public Punto obtenerPuntos() {
    return puntos;
}

@Override
public Punto obtenerPuntos() {
    return puntoNulo;
}

//----//

@Override
public void multiplicarPuntos(Multiplicador multiplicador, Jugador unJugador) {
    puntos = multiplicador.usarEnPuntos(puntos,unJugador);
}
@Override
public void multiplicarPuntos(Multiplicador multiplicador, Jugador unJugador) {
}

//----//

@Override
public void verSiAplicaExclusividad(Resultado resultado, AnalizadorExclusividad analizador) {
    analizador.agregarSituacion(new UsuarioRespondioBien(resultado));
}

@Override
public void verSiAplicaExclusividad(Resultado resultado, AnalizadorExclusividad analizador) {
    analizador.agregarSituacion(new UsuarioSeEquivoco(resultado));
}
```

## 6.12. Clase ResultadoPenalizable

Lo que tiene esta clase es que no importa si el usuario respondió bien o no, lo que hace es agregar el punto que le indican a su puntuación. Se muestran esos dos métodos.

```
@Override
public void sumarRespuestaCorrecta() {
    puntos.agregarPunto(new PuntoPositivo());
}

@Override
public void sumarRespuestaIncorrecta() {
    puntos.agregarPunto(new PuntoNegativo());
}
```

## 6.13. Interfaz Respuesta

Esta interfaz es la genérica, evita que todo el sistema de preguntas conozca a todas las respuestas posibles, por lo que logra que solo conozca la de su tipo.

Los métodos que tiene son los indicados a continuación. Con ellos se hace Double Dispatch con la pregunta.

```
Resultado evaluarEnBaseAPregunta(PreguntaAutoEvaluable pregunta, Jugador unJugador);

Resultado evaluarEnBaseAPregunta(PreguntaComparable pregunta, Jugador unJugador);
```

## 6.14. Interfaz RespuestaAutoEvaluable y RespuestaComparable

Estas interfaces son las que tienen el método correspondiente a la respuesta. El primero corresponde a la auto evaluable y los otros dos a la comparable.

```
void evaluar(Resultado unResultado);

void compararContra(RespuestaGroupChoice unaRespuesta, Resultado unResultado);

void compararContra(RespuestaOrderedChoice unaRespuesta, Resultado unResultado);
```

## 6.15. Clase RespuestaGroupChoice y RespuestaOrderedChoice

Estas son las respuestas que se usan para las preguntas comparables. Lo que saben estas clases es compararse contra otra respuesta. En el caso del trabajo practico, se comparan contra la respuesta correcta, y en base a eso se determina si el usuario obtiene puntos o no.

Se muestra el caso del de grupos.

```
@Override
public void compararContra(RespuestaGroupChoice unaRespuesta, Resultado unResultado) {
    if(unaRespuesta.gruposSonIguales(grupo1Respuesta, grupo2Respuesta)){
        unResultado.sumarRespuestaCorrecta();
    }
}

private boolean gruposSonIguales(Grupo grupo1, Grupo grupo2) {
    return (grupo1.esIgual(grupo1Respuesta) && grupo2.esIgual(grupo2Respuesta));
}
```

## 6.16. Clase RespuestaMultipleChoice y RespuestaVerdaderoFalso

Los métodos interesantes que tienen estas clases son el de evaluar, y los de evaluarEnBaseAPregunta. El primero lo que hace es decirle a las opciones que selecciono el usuario que se evalúen. Mientras que el segundo es parte del Double Dispatch con la pregunta. En el caso del verdadero falso no tiene el *for*.

```
@Override
public void evaluar(Resultado unResultado){
    for (OpcionEvaluable opcion: opcionesJugador){
        opcion.evaluar(unResultado);
    }
}

@Override
public Resultado evaluarEnBaseAPregunta(PreguntaAutoEvaluable pregunta, Jugador unJugador){
    return pregunta.responder(this,unJugador);
}

@Override
public Resultado evaluarEnBaseAPregunta(PreguntaComparable pregunta, Jugador unJugador){
    throw new RespuestaNoAptaParaPreguntaException();
}
```

## 6.17. Clase OrderedChoice y GroupChoice

Son las preguntas de orden y grupos, se muestra la llamada que hacen para comparar la respuesta que reciben con la correcta. Internamente tienen una respuesta, la cuales a su vez tienen un orden o un grupo.

```
@Override
public Resultado responder(RespuestaComparable respuestaJugador, Jugador unJugador){
    Resultado unResultado = puntaje.obtenerResultado(CANT_RESPUESTAS_CORRECTAS_TOTALES,
                                                     unJugador);
    respuestaJugador.compararContra(respuestaCorrecta, unResultado);
    return unResultado;
}
```

## 6.18. Interfaz Modificador

Esta interfaz logra que se trabajen los modificadores que tiene el jugador de forma mucho mas genérica. Los métodos de usarEnPuntaje son los que con Double Dispatch determinan si el modificador es valido en una pregunta.

```
void usarEnPuntaje(PuntajePenalizable puntajePenalizable,
                   ArrayList<Modificador> modificadoresDeLaJugada);

void usarEnPuntaje(PuntajeClasico puntajeClasico,
                   ArrayList<Modificador> modificadoresDeLaJugada);

void usarEnPuntaje(PuntajeParcial puntajeParcial,
                   ArrayList<Modificador> modificadoresDeLaJugada);

void aplicar(Resultado resultadoJugador1, Resultado resultadoJugador2);
```

## 6.19. Interfaz Multiplicador

Esta interfaz surgió ya que en el modelo decidimos crear un multiplicador especial para la exclusividad, por lo que abstraíamos la idea del multiplicador, creando el ya mencionado y uno para el jugador. El método que tiene es el siguiente:

```
Puntuacion usarEnPuntos(Puntuacion puntos, Jugador unJugador);
```

## 6.20. Clase MultiplicadorJugador

Este multiplicador es el que tiene el jugador, internamente tiene el factor, el cual puede ser x2 o x3 y una referencia al jugador al cual pertenece. Solamente puede ser usado en una pregunta con puntaje penalizable. Implementa las interfaces *Modificador* y *Multiplicador*.

Se muestra el código correspondiente al Double Dispatch que hace y a cuando se aplica. Como se ve en el aplicar, se envía a si mismo a ambos resultados, pero solamente a uno le hará efecto.

```
@Override
public void usarEnPuntaje(PuntajePenalizable puntajePenalizable,
                          ArrayList<Modificador> modificadoresDeLaJugada) {
    modificadoresDeLaJugada.add(this);
    jugador.pierdeModificador(this);
}

@Override
```

```
    public void usarEnPuntaje(PuntajeClasico puntajeClasico,
                              ArrayList<Modificador> modificadoresDeLaJugada) {
    }

    @Override
    public void usarEnPuntaje(PuntajeParcial puntajeParcial,
                              ArrayList<Modificador> modificadoresDeLaJugada) {
    }

    @Override
    public void aplicar(Resultado resultadoJugador1, Resultado resultadoJugador2) {
        resultadoJugador1.aplicarMultiplicador(this);
        resultadoJugador2.aplicarMultiplicador(this);
    }

    public Puntuacion usarEnPuntos(Puntuacion puntos, Jugador unJugador) {
        if(jugador == unJugador) {
            return puntos.multiplicar(factor);
        }
        return puntos;
    }
```

## 6.21. Clases Exclusividad y MultiplicadorExclusividad

La exclusividad es un modificador que puede usar el jugador en las preguntas con puntaje clásico o parcial. Para saber si es posible usar la exclusividad en una pregunta se hace Double Dispatch con el puntaje. Esto determinara si se puede agregar o no. Al aplicarse, el efecto que hace es que si un solo jugador respondió bien, ese recibe el doble de puntos. Si ambos responden bien, ninguno recibe puntos. En el caso de que el otro jugador use la exclusividad, se acumula el modificador.

Internamente guardan el jugador al que pertenecen, esto esta para poder sacarle al jugador el modificador si es que lo aplica.

Se muestran los métodos que tiene. Los primeros 4 corresponden al Double Dispatch mencionado, mientras que el ultimo al momento en que se aplica. En la siguiente sección se explicara el uso del analizador.

```
@Override
public void usarEnPuntaje(PuntajePenalizable puntajePenalizable,
                          ArrayList<Modificador> modificadoresDeLaJugada) {
}

@Override
public void usarEnPuntaje(PuntajeClasico puntajeClasico,
                          ArrayList<Modificador> modificadoresDeLaJugada) {
    seUsoExclusividadEnPreguntaValida(modificadoresDeLaJugada);
}

@Override
public void usarEnPuntaje(PuntajeParcial puntajeParcial,
                          ArrayList<Modificador> modificadoresDeLaJugada) {
    seUsoExclusividadEnPreguntaValida(modificadoresDeLaJugada);
}

private void seUsoExclusividadEnPreguntaValida(ArrayList<Modificador> modificadoresDeLaJugada){
```

```
    modificadoresDeLaJugada.add(this);
    jugador.pierdeModificador(this);
}

@Override
public void aplicar(Resultado resultadoJugador1, Resultado resultadoJugador2) {
    AnalizadorExclusividad analizador = new AnalizadorExclusividad();
    resultadoJugador1.aplicaExclusividad(analizador);
    resultadoJugador2.aplicaExclusividad(analizador);
}
```

## 6.22. Clase AnalizadorExclusividad

El analizador de exclusividad es un objeto que recorre los resultados de los jugadores. Mientras que lo hace, recibe las situaciones de exclusividad. Estas indican si el usuario respondió bien o se equivoco.

Este objeto, lo que permite es evitar hacer *ifs* que terminen rompiendo el encapsulamiento de los resultados. Internamente tiene un estado, este inicia cuando no tiene situaciones, después pasa a otro estado en que tiene las dos situaciones y puede determinar quien gana la exclusividad. Una vez determinada, deja el estado en que ya se uso para dar claridad.

Se muestra el método en que se agregan las exclusividades.

```
public void agregarSituacion(SituacionesExclusividad situacion){
        estado = estado.agregarSituacion(situacion);
    }
```

## 6.23. Interfaz SituacionesExclusividad

Esta interfaz es la que representa finalmente los estados de si el jugador se equivoco o no. Se muestran los métodos que tiene. Con ellos se hace Double Dispatch y se llega al resultado de si se aplica o no.

```
    void compararCon(UsuarioRespondioBien situacionDelRival);

    void compararCon(UsuarioSeEquivoco situacionDelRival);

    void compararCon(SituacionesExclusividad situacion);
```

## 6.24. Clases UsuarioSeEquivoco y UsuarioRespondioBien

Estas clases son las clases que representan a los resultados que pueden tener los jugadores. Internamente se guardan una referencia al resultado al que pertenecen.

Se muestran primero los métodos en caso de que el jugador se haya equivocado. Como se ve en el primer método, como se recibe que el otro usuario respondió bien, y el objeto mismo representa que se equivoco, eso significa que el otro usuario gano la exclusividad. En el caso de que se recibe un *UsuarioSeEquivoco*, significaría que ambos se equivocaron, por lo que no hay que hacer nada.

```
    @Override
    public void compararCon(UsuarioRespondioBien situacionDelRival) {
        situacionDelRival.ganoExclusividad();
    }

    @Override
    public void compararCon(UsuarioSeEquivoco situacionDelRival) {
    }
```

```
@Override
public void compararCon(SituacionesExclusividad situacion) {
    situacion.compararCon(this);
}
```

Ahora los de si el jugador respondió bien. El método de *pierdoPuntuacion* llama a sumar respuesta incorrecta, ya que este haría que el resultado cambie de estado, haciendo que el punto que otorgue sea uno nulo.

```
@Override
public void compararCon(UsuarioRespondioBien situacionDelRival) {
    pierdoPuntuacion();
    situacionDelRival.pierdoPuntuacion();
}

@Override
public void compararCon(UsuarioSeEquivoco situacionDelRival) {
    ganoExclusividad();
}

@Override
public void compararCon(SituacionesExclusividad situacion) {
    situacion.compararCon(this);
}

public void pierdoPuntuacion(){
    resultadoDelJugador.sumarRespuestaIncorrecta();
}

public void ganoExclusividad() {
    resultadoDelJugador.aplicarMultiplicador(new MultiplicadorExclusividad());
}
```

## 6.25. Otras partes del trabajo

**Lector**

Para el lector de preguntas se decidió usar el patrón Facade. Se tiene una interfaz que es implementada por el lector JSON que decidimos utilizar. Este lector se encarga de leer las preguntas llamando a los diferentes parser que tiene. Estos parser luego llaman a la fabrica de preguntas que se implementó.

Se muestran los formatos que se tienen que seguir para poder agregar preguntas al juego. Cada una tiene su propio archivo. En todos los casos hay que escribir el enunciado correspondiente a la pregunta.

En el caso del verdadero falso, el penalizable indica el puntaje que tiene la pregunta, si se pone como false se tomará que la pregunta tiene puntaje clásico. El enunciadoEsCorrecto corresponde a la veracidad de la pregunta.

```
{
        "VoF": {
                "penalizable": true,
                "enunciado":"La primera computadora en Argentina se llamo Clementina",
                "enunciadoEsCorrecto": true
        }
}
```

En el caso del multiple choice, hay que escribir el puntaje, este puede ser ”clasico”, ”penalizable”, o ”parcial”. Después dentro de cada categoría se escriben las opciones que pueden tener.

```
{
        "MC": {
                "puntaje": "clasico",
                "enunciado" : "Seleccionar cuales marcas producen memoria ram",
                "respuestasCorrectas": [
                    "Kingston",
                    "G-skill"
                ],
                "respuestasIncorrectas": [
                    "Princeton"
                ]
        }
}
```

El caso de las preguntas de grupo es mas sencillo, solo hay que escribir las opciones de cada grupo junto con su nombre.

```
{
        "GROUP": {
            "enunciado" : "Agrupar en tipado estatico y dinamico",
                "labelGrupo1" : "Estatico",
                "labelGrupo2" : "Dinamico",
                "respuestasGrupo1": [
                        "Java",
                        "C"
                ],
                "respuestasGrupo2": [
                        "Python",
                        "Smalltalk"
                ]
        }
}
```

El caso de las preguntas de orden lo único que tiene de particular es que hay que escribir las opciones en el orden correspondiente, este sera usado en el juego como el correcto.

```
{
        "ORDER": {
                "enunciado" : "Colocar en orden descendiente las siguientes velocidades",
                "respuestasEnOrden": [
                        "256 KBps",
                        "256 Kbps",
                        "1024 b/s"
                ]
        }
}
```

### Vista

En el caso de la vista, se decidió utilizar el patrón Modelo Vista Controlador para hacer la interacción entre lo que ve el usuario y el modelo. Para simplificar algo el proceso de creación de las diferentes vistas que puede tener el juego con respecto a las preguntas, se hizo también una especie de fábrica de vistas (Muy similar al sistema de fabrica de preguntas), la cual devolverá la vista que corresponda.

**Desordenador**

Se hizo también una interfaz llamada *CriterioOrdenamiento*, la cual tiene como método desordenar. En este caso se podría decir que se utiliza una especie de patrón Strategy, ya que le decimos a *AlgoHoot* que criterio usar dependiendo del caso, si se esta usando en las pruebas, o en la aplicación. En el trabajo se usa una implementación que es propia de Java a través de la interfaz hecha, mientras que para las pruebas se cambia de estrategia a una que no desordena, garantizando que siempre se obtenga el mismo resultado.

# 7. Excepciones

**CantidadErroneaDeRespuestasParaPreguntaException** Esta excepción fue creada para evitar que se agregue una cantidad errónea al crear una pregunta o que se respondan una cantidad de opciones distinta a las permitidas, ya sea por tener menos o mas. Esta excepción puede ser lanzada por las clases MultipleChoice, GroupChoice, y OrderedChoice. Una vez lanzada es atrapada en lector que utilizamos, y lo que se realiza es evitar agregar esa pregunta a la lista de preguntas totales. De esta forma se puede seguir jugando el juego, solo que con menos preguntas de las que esperaría el usuario si es que edito el archivo de preguntas.

**RespuestaNoAptaParaPreguntaException** Esta excepción esta para indicar que ocurrió un error al mandar una respuesta que no corresponde con la pregunta. Por ejemplo, si se manda una respuesta perteneciente al verdadero falso a una pregunta de grupos.

**TipoDePuntajeEnArchivoNoValidoException** Esta excepción la tiran los parsers del lector para indicar que leyeron un puntaje que no existe. Al ser lanzada es atrapada en el lector y se imprime la historia de la pila del programa.

**ArchivoNoEncontradoException** Esta excepción es lanzada en el lector, y esta para indicar que no se encontraron los archivos de preguntas. Se utiliza esta excepción para no estar dependiendo de la propia de Java. Es atrapada en la vista, donde se le muestra una indicacion al usuario de lo sucedido.

# 8. Diagramas de secuencia

Mostramos a continuación un diagrama de secuencia de una prueba simple. No se incluyeron los detalles internos de lo que sucede entre *:Pregunta*,*:Resultado*, y *:Respuesta*.

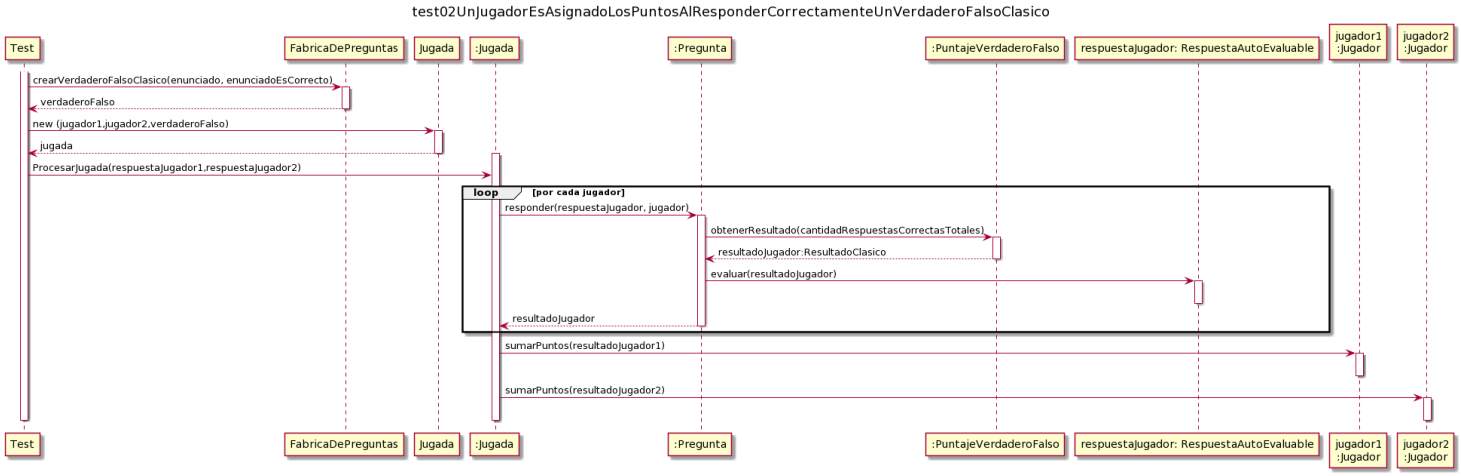


Figura 8: Diagrama de secuencia mostrando el análisis de una respuesta correcta para verdadero falso clásico

Se muestra ahora un diagrama del sistema de puntos en el caso de un resultado clásico.

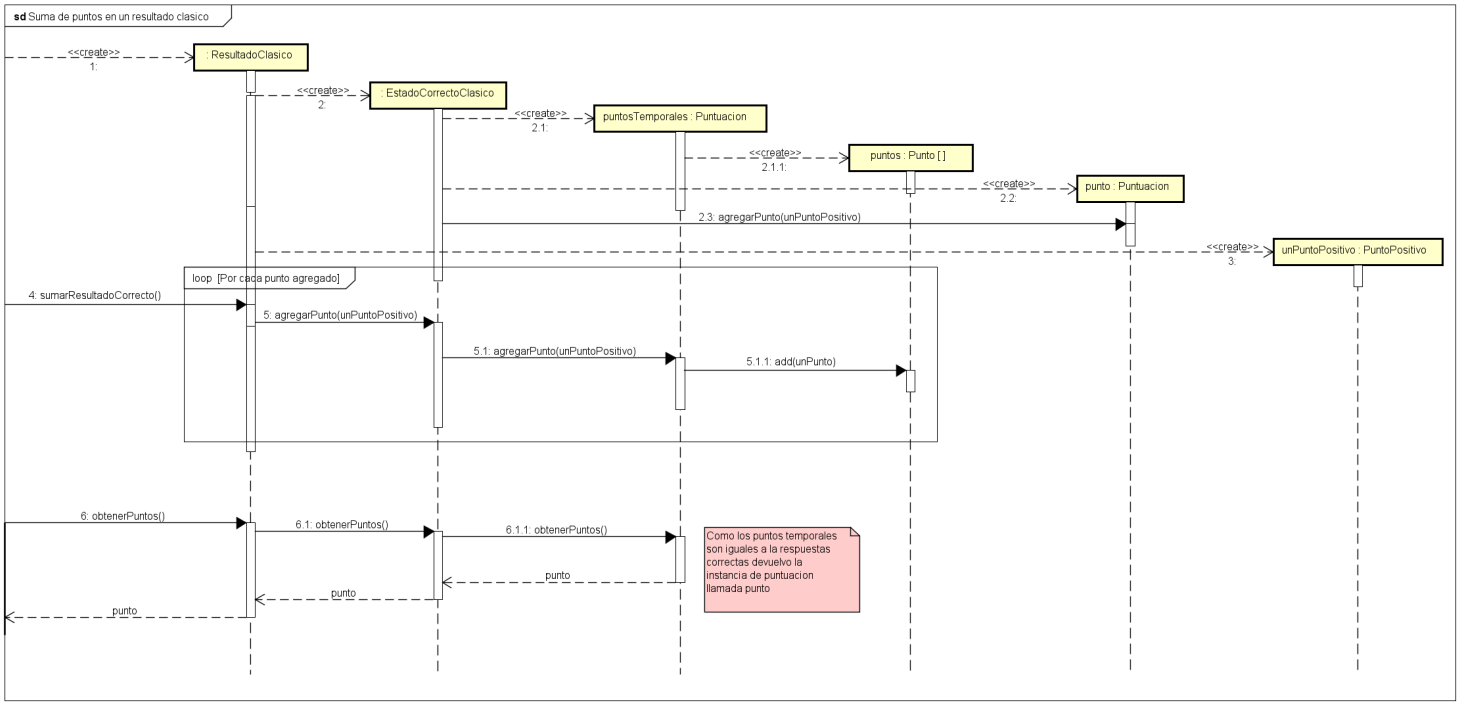


Figura 9: Agregado de puntos en un resultado clásico. En este caso se respondió la cantidad correcta de respuestas.

Se muestra un diagrama con el agregado de un modificador, en este caso un multiplicador a una jugada.

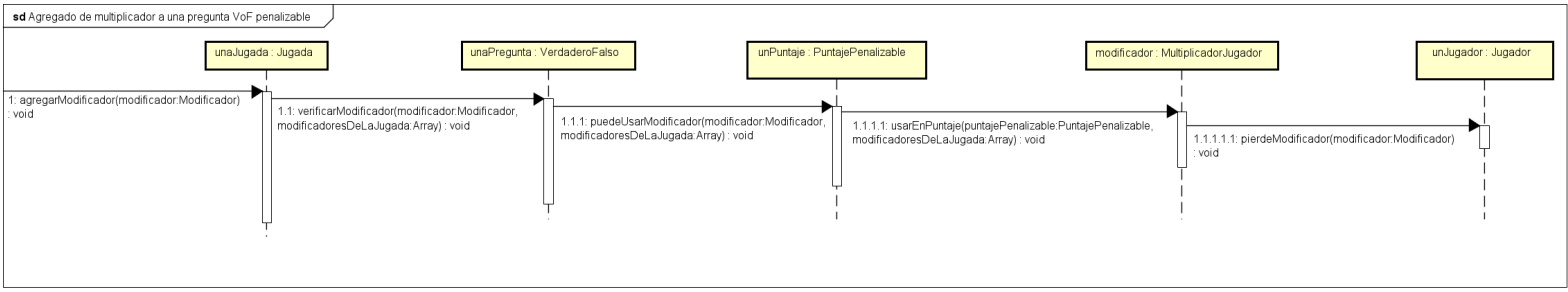


Figura 10: Sistema de agregado de un modificador, en este caso se guarda en la lista con los modificadores de la jugada.

En este diagrama se ve como se multiplican los puntos. Se muestra principalmente el proceso con una puntuación, para los otros puntos posibles es similar el proceso, siendo la diferencia que ellos mismos se agregan en la puntuación multiplicada.

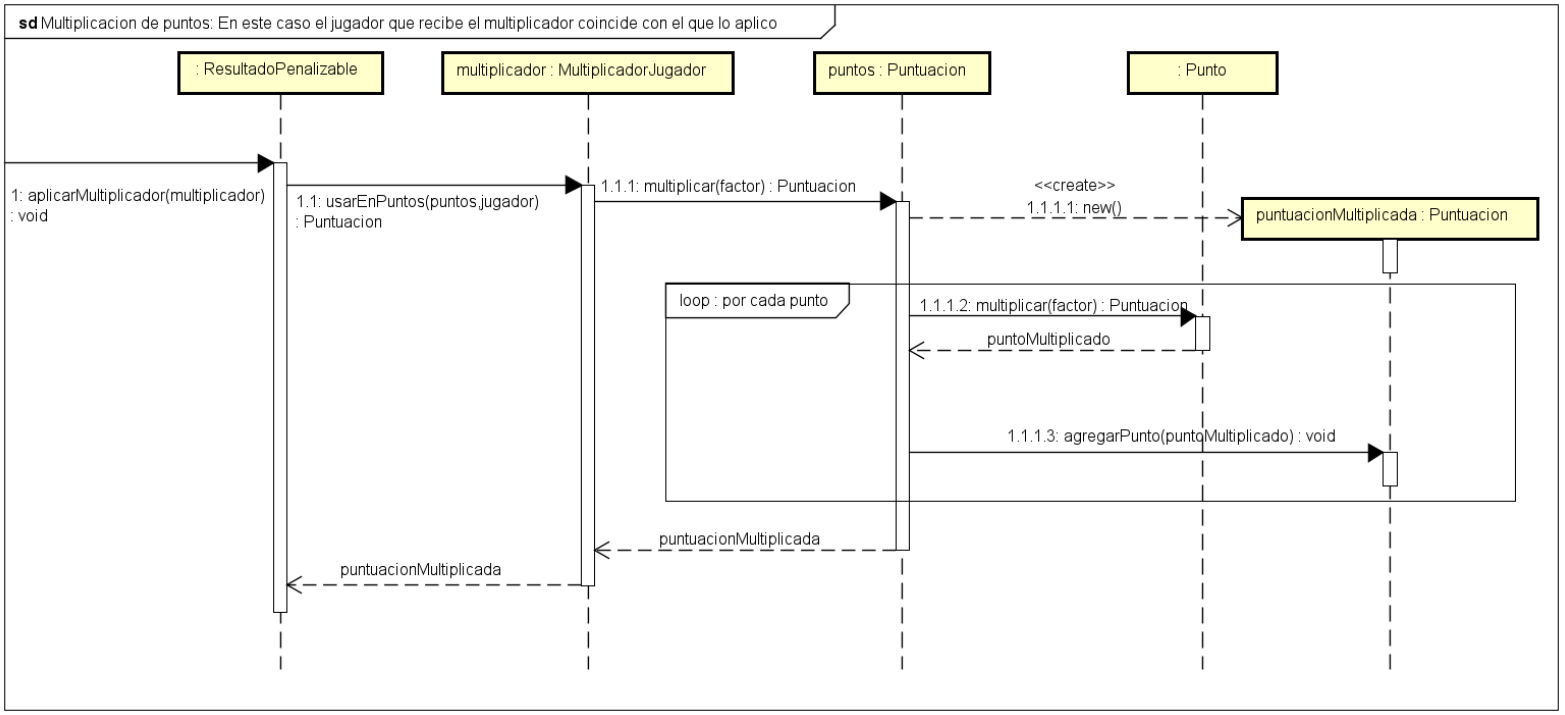


Figura 11: Diagrama mostrando como se multiplican los puntos

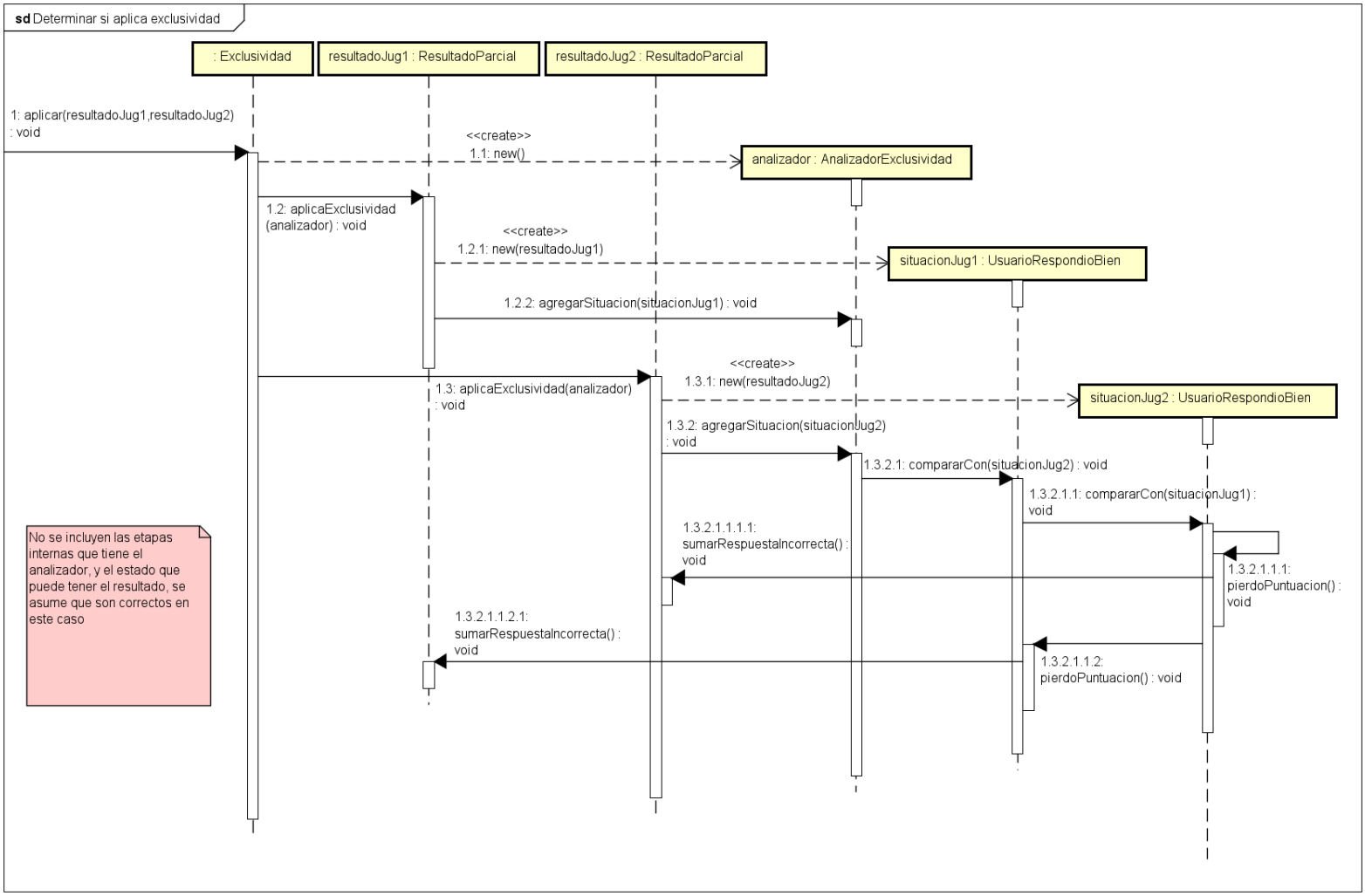


Figura 12: Proceso de aplicar la exclusividad

En el diagrama a continuación se muestra el proceso en que se determina un ganador, en este diagrama se asume que el juego ya termino (estado TerminoJuego) y que el jugador2 tiene mas puntos que el primer jugador.

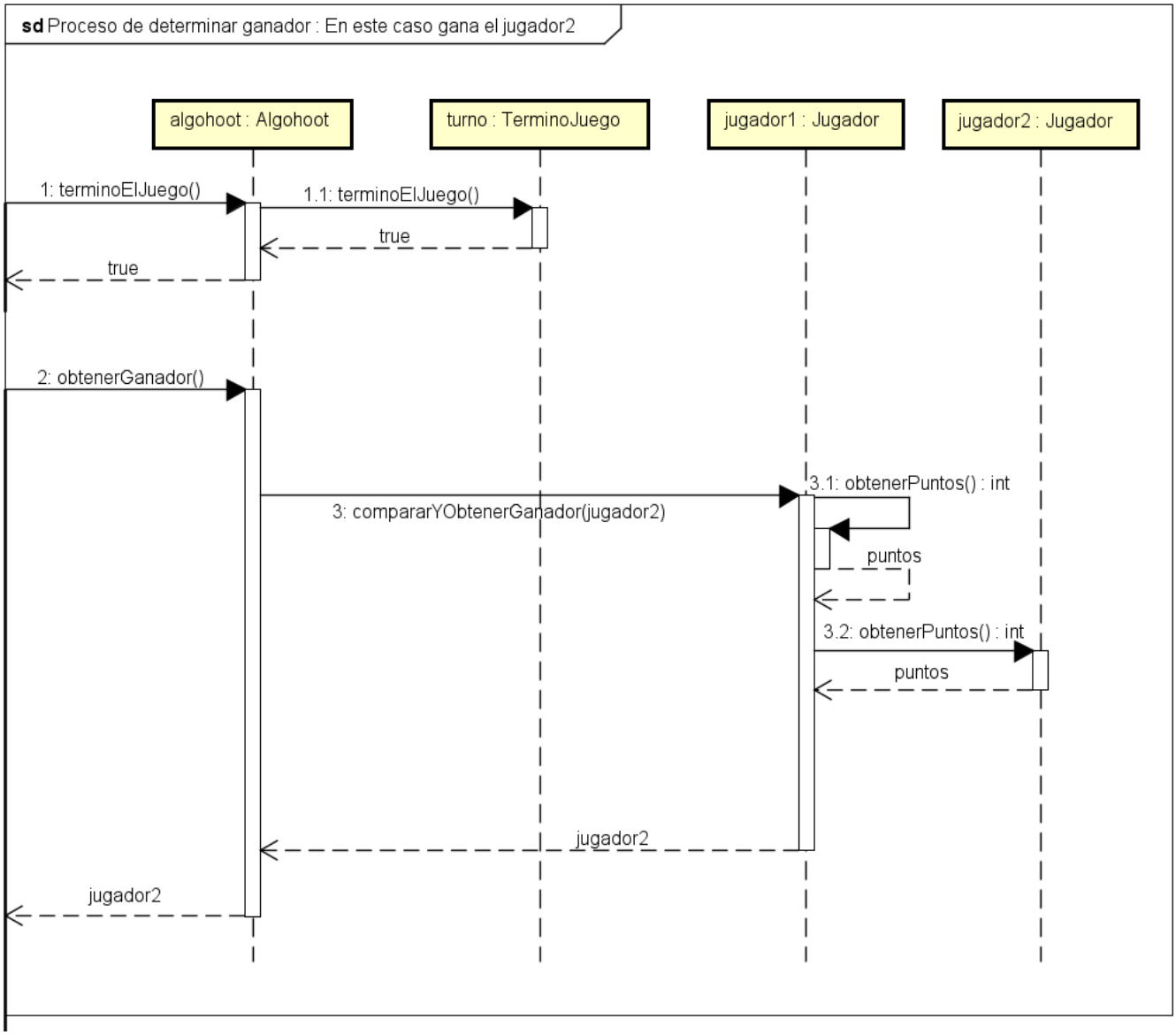


Figura 13: Proceso en que se determina el ganador del juego

# 9. Diagramas de Estados

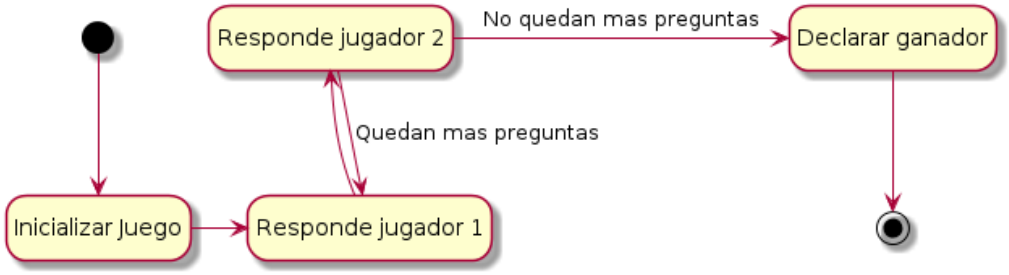


Figura 14: Diagrama de estados que muestra de forma simple el funcionamiento del juego

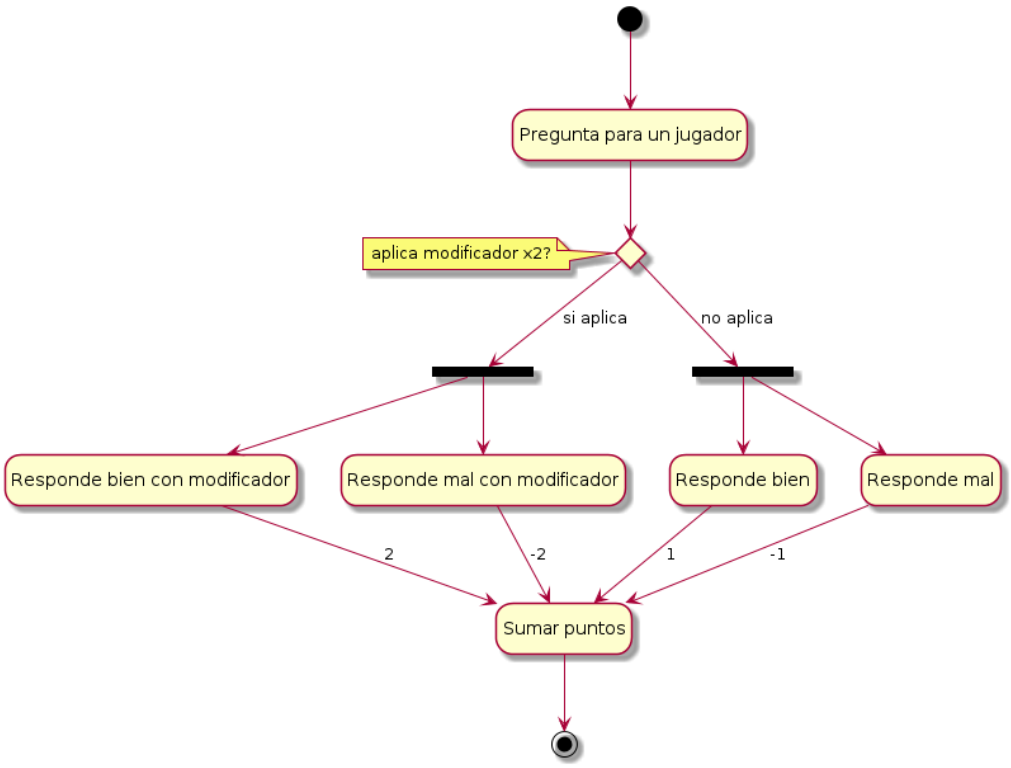


Figura 15: Diagrama mostrando la secuencia en caso de que el jugador responda con un multiplicador x2 una pregunta Verdadero Falso Penalizable