FACULTAD DE INGENIERIA
Universidad de Buenos Aires

Trabajo Práctico 2 — Java

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

Alumno	Número de padrón	Email
Rosario Szuplat	100798	rszuplat@fi.uba.ar
Aldana Cardoso Rastrelli	98408	arastrelli@fi.uba.ar
Josefina Iterman	104031	jiterman@fi.uba.ar
Ignacio Janeiro	103550	ijaneiro@fi.uba.ar

Índice

1. Introducción

En este trabajo práctico se desarrolló la aplicación *Cursando por un Sueño* usando *Java* y diseñando el modelo considerando el paradigma de la orientación a objetos. En primer lugar, se pensó el modelo teniendo en cuenta algunos patrones de diseño como *mvc*, *factory* y *strategy*. Luego, se lo implementó usando *Test-driven development* y se desarrolló la interfaz gráfica usando *JavaFX*.

Dicha aplicación es un juego en el que participan dos personas, cada una en su turno. Deben responder una serie de preguntas, las cuales pueden ser del tipo *multiple choice*, verdadero o falso, de agrupar las opciones en diferntes categorías o de ordenarlas a fin de ir acumulando puntos. El que obtiene un puntaje mayor al finalizar el juego gana.

Los puntos se asignan de diferente manera según el tipo de pregunta. Por ejemplo, si es *multiple choice* clásico y el jugador selecciona todas las respuestas correctas, obtiene un punto. En cambio, si es *multiple choice* con penalidad, el jugador obtiene un punto por cada respuesta correcta seleccionada y se le descuenta uno por cada pregunta incorrecta. Si la pregunta es del tipo *multiple choice* con puntaje parcial, se obtiene un punto por cada respuesta correcta seleccionada siempre y cuando no se haya seleccionado ninguna incorrecta. Análogamente, hay preguntas de verdadero o falso clásicas y con penalidad. En las primeras, se le asigna un punto a cada jugador que responda correctamente mientras que las segundas se le suma uno si responde bien y se le resta uno si lo hace mal. Finalmente, en las *ordered choice*, se asigna un punto si todas las opciones se colocan en el orden correcto y, en las *group choice*, se gana un punto si todas las opciones se asignan al grupo debido.

Por otro lado, también se cuenta con *boosts* como los multiplicadores x2 y x3 y la exclusividad de puntaje. Cada jugador puede usar una vez el x2, una el x3 y dos la exclusividad a lo largo del juego. Los primeros dos solo se pueden usar en las preguntas que tienen penalidad penalidad y la tercera solo en las que no. Los multiplicadores duplican y triplican respectivamente tanto los puntos *positivos* como los *negativos*. Es decir, tanto los obtenidos por responder bien como los correspondientes a la penalización por equivocarse. En el caso en el que un jugador use el *boost* de exclusividad y responda correctamente y el otro, erradamente, obtiene el doble del puntaje que obtendría si no hubiera usado el *boost*. Sin embargo, si ambos responden bien, ninguno sumará puntos. Cabe aclarar que ambos jugadores pueden decidir usarlo en la misma ronda. En ese caso, si uno sólo responde asertadamente, recibirá el cuádruple de puntos.

2. Supuestos

Dado que no se especificó lo que debería suceder en el caso en el que un jugador no seleccione una respuesta, se decidió que, si esto sucede, ese jugador no recibe puntos. Sin embargo, si el otro jugador elije usar el *boost* de exclusividad (y responde correctamente), el mismo recibirá el puntaje duplicado como si el primer jugador hubiera respondido erradamente en lugar de no responder.

Además, en el caso de que algún jugador no indique su nombre, se le asigna un nombre predeterminado. Por ejemplo, si el que no lo indica es el primero, se lo nombre *Jugador1*. Si el que no lo hace el segundo, *Jugador2*. De esta manera, se asegura que todos estén denominados de alguna manera pero sin imponerle al usuario que ingrese un nombre si no lo desea.

Con respecto a los *boosts*, se supuso que en cada turno sólo se debían mostrar los que tiene disponibles el jugador correspondiente según sea una pregunta con o sin penalidad y según los *boosts* usados anteriormente por ese participante. Esto se consideró mejor que que cada jugador tenga todos los *boosts* disponibles y, si elige uno que no puede usar, se le indique que debe seleccionar otro o no usar ninguno.

Por otro lado, se aclara que las preguntas con sus respectivas opciones y respuesta correcta se determinan a partir de un archivo *.json*. Nuevamente se adoptó esta modalidad porque en la consigna no se indicaba cómo conocía el juego las diferentes preguntas. Se consideró inicialmente que los participantes escriban las preguntas antes de empezar a jugar pero se lo descartó por considerarse aburrido un juego en el que antes de empezar deban inventarse las preguntas para

poder jugar. Además, basta con cambiar el archivo *.json* para poder cambiar las preguntas.

Finalmente, se supuso que en el flujo del programa debía haber una pantalla en la que se muestren los puntos ganados o perdidos por cada jugador en cada pregunta, una en la que se muestren los que lleva acumulados cada jugador al finalizar cada ronda y una pantalla final en la que se indique quién ganó o si se produjo un empate . En las especificaciones se indicaba únicamente la fase inicial en la que los jugadores introducen su nombre y la fase de juego en la que cada uno ve en pantalla la pregunta y elije la(s) opciones que considera correctas. Se prefirió indicar los puntajes de cada uno luego de que ambos respondan dado que, si se usa el *boost* de exclusividad, es necesario conocer ambas respuestas para poder determinar el resultado. Además, dado que ambos jugadores juegan en la misma pantalla, es necesario que el segundo no sepa si lo que respondió el primero es correcto o no. Por lo tanto, no puede saber cuántos puntos obtuvo.

3. Diagramas de clase

En las figuras 1 y 2 se muestran las relaciones estáticas entre las clases con las que se modeló la aplicación. En la 1 se priorizó una visión general del modelo a fin de mostrar cómo esta compuesto sin detallar en la determinación de la pregunta y su comportamiento. Por lo complejo que es esto, se lo muestra aislado del resto del modelo en la figura 2. En la misma se muestran también los patrones de diseño utilizados para poder modelar los diferentes tipos de preguntas (verdadero o falso, *multiple choice*, *group choice* y *ordered choice*) y su comportamiento (clásico, con penalización y puntaje parcial) respetando la orientación a objetos.

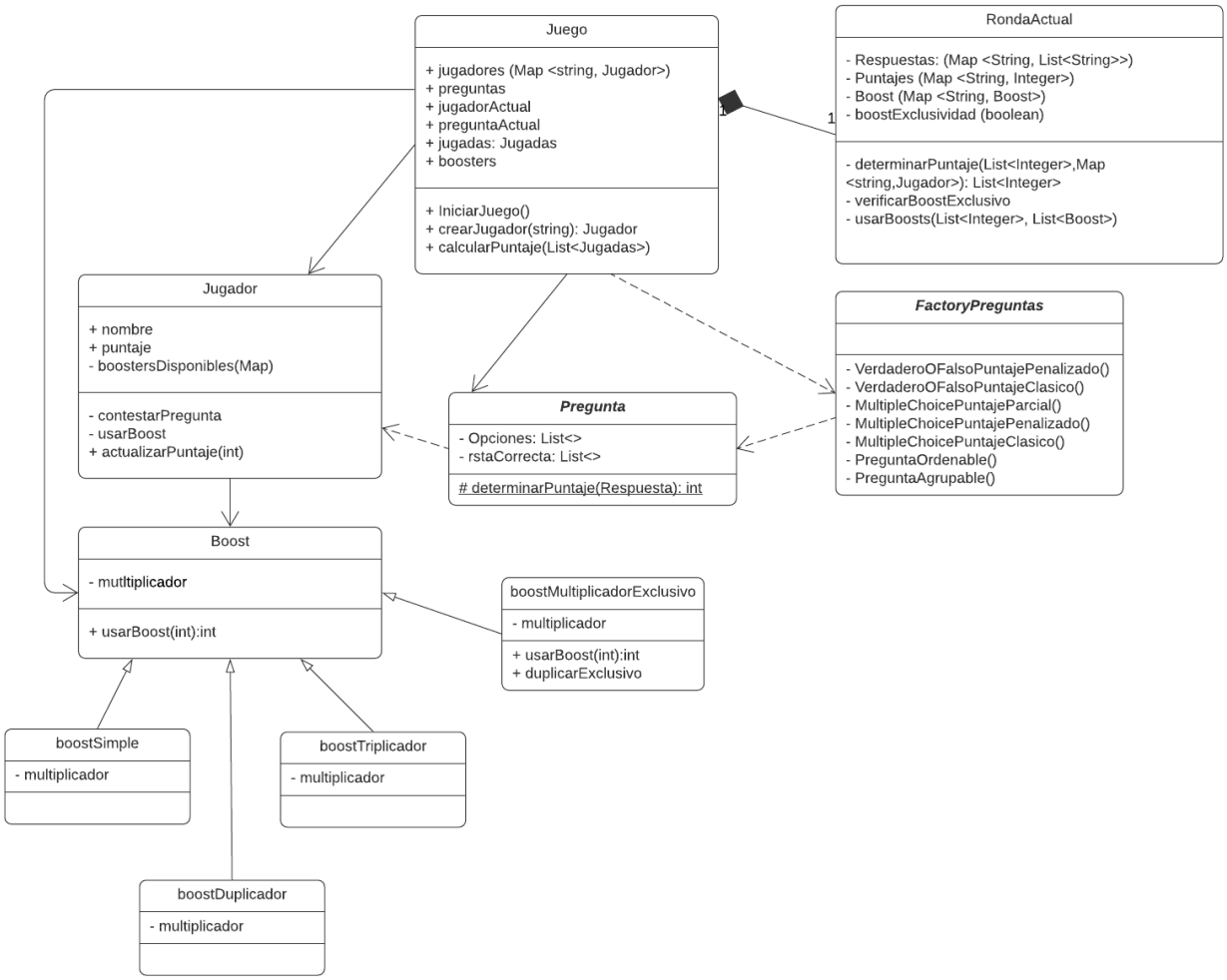

Figura 1: Diagrama de clases de la aplicación.

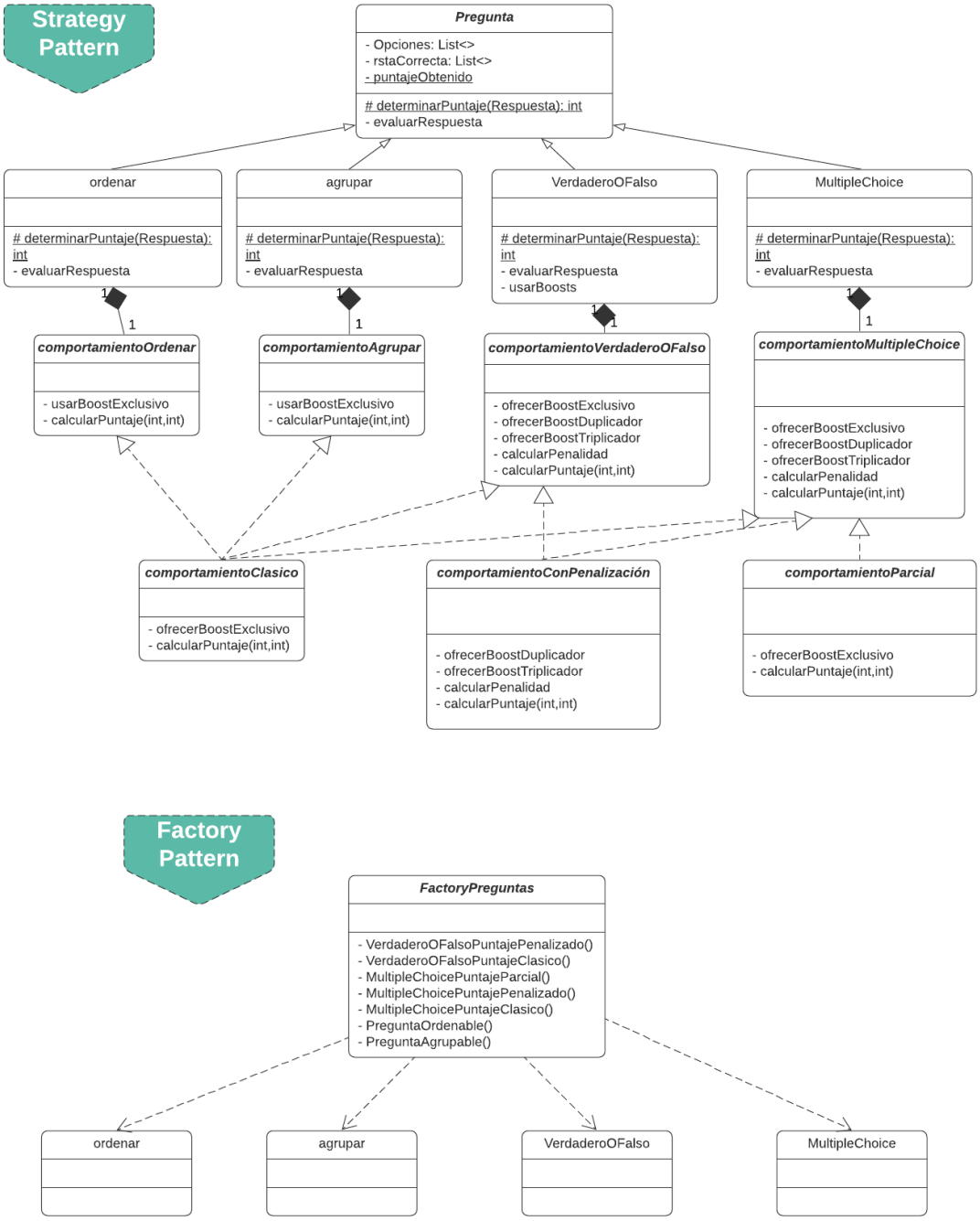

Figura 2: Diagrama de clases detallado del modelo de las preguntas.

4. Detalles de implementación

La parte crucial del modelo diseñado es la relativa a las preguntas ya que son las que más varían tanto en su comportamiento como en su estructura. Esto se debe a que, por un lado, hay diferentes tipos de preguntas, lo que implica que las opciones y la respuesta correcta se deban

modelar de manera diferente y, por otro, preguntas del mismo tipo puedan asignar puntajes de manera diferente (según si tienen o no penalidad o si es asigna puntaje parcial). Además, cambian los *boosts* que se pueden usar según el comportamiento de la pregunta y la interfaz gráfica de cada una es completamente diferente.

Es por esto que a continuación se detalla como se modelaron las preguntas a partir de diferentes abstracciones. También, se comentan la interfaz gráfica y su relación con el modelo y la abstracción *RondaActual*.

4.1. Tipo de Pregunta

A medida que fluye el programa, el tipo de pregunta (ordenar, agrupar, verdadero o falso y multiple choice) y su comportamiento asociado (con y sin penalidad y puntaje parcial) van cambiando pero no es posible saber cuál se tendrá a continuación. Esta es una de las razones por las que es conveniente poder crear las diferentes preguntas sin tener que especificar exactamente a qué clase pertenecen. Para ello, se usó el patrón de diseño *Factory*.

Tal como se observa en la figura 2, se cuenta con una clase *FactoryPreguntas* que cuenta con un método para crear cada tipo de pregunta, teniendo en cuenta incluso el modo de asignar puntos que debe tener. Por lo tanto, crear una pregunta es tan simple como llamar al método *crear*, que como parámetros recibe el tipo, el comportamiento, el título, las opciones y la respuesta correcta de la misma. Este método es el que llama a la función correspondiente segun el tipo y comportamiento que se tiene, que finalmente crea la pregunta necesaria.

De esta manera, se pueden crear todas las preguntas con el mismo método (*crear*), lo cual es fundamental en el paradigma de objetos ya que contribuye a desacoplar las clases: cuando *Juego* lee las preguntas del *.json* no necesita saber qué tipo de pregunta es ni su comportamiento.

En la figura 2 también se observa que los tipos de pregunta que usa *FactoryPreguntas* heredan de *Pregunta*. Esto simplifica el código de manera polimórfica y es posible gracias a este patrón. El *Factory* permite contar con clases hijas que deciden cómo se crea un objeto de la clase padre y qué tipo de objetos contiene como atributos.

4.2. Comportamiento de cada Tipo de Pregunta

Dado que cada tipo de pregunta puede tener diferente forma de asignar puntaje para un mismo tipo de respuesta, es decir, diferente *comportamiento*, se uso el patrón *Strategy*. El mismo permite elegir el comportamiento que se necesita e ir intercambiándolo dinámicamente. De esta manera, la clase Pregunta tiene clases hijas según el tipo (verdadero o falso, multiple choice, ordenar y agrupar) y cada tipo se compone con un comportamiento, que se usa a modo interfaz. Por lo tanto, cada interfaz de comportamiento podrá utilizar y definirse según sea necesario para cada tipo de pregunta, ya que no todos los tipos de pregunta comparten los mismos tipos de comportamiento.

Se delegó, entonces, la responsabilidad de calcular el puntaje en el *comportamiento*, mientras que la pregunta calcula la cantidad de opciones correctas o incorrectas que ha elegido el usuario. Además, este uso de las diferentes interfaces, mostradas en la figura 2, evita código repetido por su accionar polimórfico respecto del cálculo de los puntajes y/o penalidades. Se observa que dicho *comportamiento* esta determinado por un lado por el tipo de pregunta y por otro por la forma en la que debe asignarse el puntaje. Esta diferenciación fue necesaria para que no haya demasiadas clases herederas de *Pregunta*, ya que con 4 tipos de pregunta y 3 tipos de comportamiento -no todos en una relación 1 a 1- la cantidad de clases herederas de Pregunta en este caso hubiesen sido 7 en vez de sólo 4.

Cabe destacar que este modelo permitió incluir fácilmente el uso de los diferentes *boosts* y restringir su uso según correspondiera. Esto hubiera sido mucho más complicado si no se hubiera diferenciado entre el tipo de pregunta y su comportamiento.

4.3. Patrón *mvc*

Con respecto a la interfaz gráfica, era requisito usar el patrón *mvc* y así se hizo. De esta manera se logró desacoplar completamente el modelo de la aplicación de la implementación de su interfaz gráfica. Esto es necesario para permitir que, en un futuro, este modelo se pueda usar en otro contexto que implemente la intefaz gráfica con otras herramientas. Si el mismo dependiera en algún punto de *JavaFX*, esto no podría realizarse, limitando considerablemente el modelo. Además, es fundamental en el paradigma de objetos desacoplar al máximo las diferentes clases y paquetes. Por lo que una dependencia del modelo con la interfaz gráfica hubiera sido perjudicial.

Sin embargo, cabe destacar que al tener que elegir la vista a mostrar en pantalla usar este patrón fue perjudicial desde el punto de vista de la orientación a objetos. Dicha vista esta dada por el tipo de pregunta, por lo que una llamada polimórfica a un método que devolviera una determinada vista determianda según la clase que recibiera dicho llamado. Pero, como el modelo no podía contener las vistas por ser éstas parte de la interfaz gráfica, esto no fue posible y fue necesario implementar un *switch* en la clase *Sistema* que devuelva una vista según el tipo de pregunta que se esté considerando.

4.4. Clase *RondaActual*

La clase *RondaActual* es necesaria para guardar la información de cada jugador en, justamente, una ronda. Una ronda consta de dos turnos, uno para cada jugador, en el que pueden elegir un *boost* a utilizar y la respuesta que consideren correcta. Si bien delega la responsabilidad de calular puntaje en *Pregunta* y la de modificar dicho puntaje al *boost* si fuera necesario, es necesaria para controlar si debe corregirse el puntaje como consecuencia del use de un *boost* de exclusividad ya que el mismo está determinado por la respuesta de **ambos** boosts. También, se encarga de determinar si alguno de los jugadores se quedó sin *boosts* disponibles para usar y de asignarle a cada jugador su puntaje.

Como se ve, es una clase necesaria para las interrelaciones entre los dos jugadores y el *boost* y la respuesta que elije cada uno. Esto se diferencia de la clase *Juego*, que también está asociado a las clases *Boost*, *Jugador* y *Pregunta*, porque sólo considera los objetos de una ronda. Es decir, los relacionados con una misma pregunta, sacándole asi responsabilidades a *Juego*.

5. Excepciones

En el modelo diseñado se implementó la siguiente excepción:

NoTieneBoostDisponibleException Se lanza en el caso que un *bug* produzca que un jugador vea como disponible y seleccione un *boost* que en realidad no puede usar en esa pregunta, ya sea por el tipo de *boost* o porque ya usó todos los *boosts* de ese tipo que se permiten en el juego. No se espera que esta excepción se lance porque el modelo permite que los jugadores vean disponibles únicamente los *boosts* que pueden elegir usar sin problemas. Por lo tanto, no se la atrapa.

6. Diagramas de secuencia

En las figuras 3 a 8, se presentan los diagramas de secuencia implementados para mostrar distintas interacciones entre los objetos de la solución.

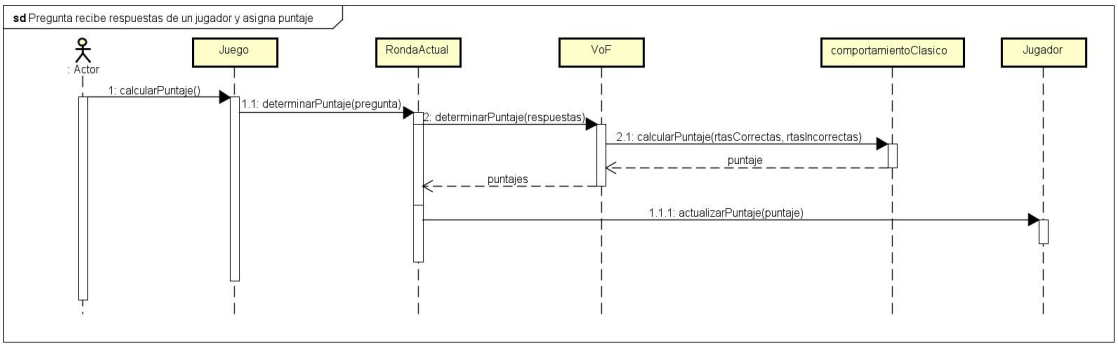

Figura 3: Asiganción de puntaje a un jugador.

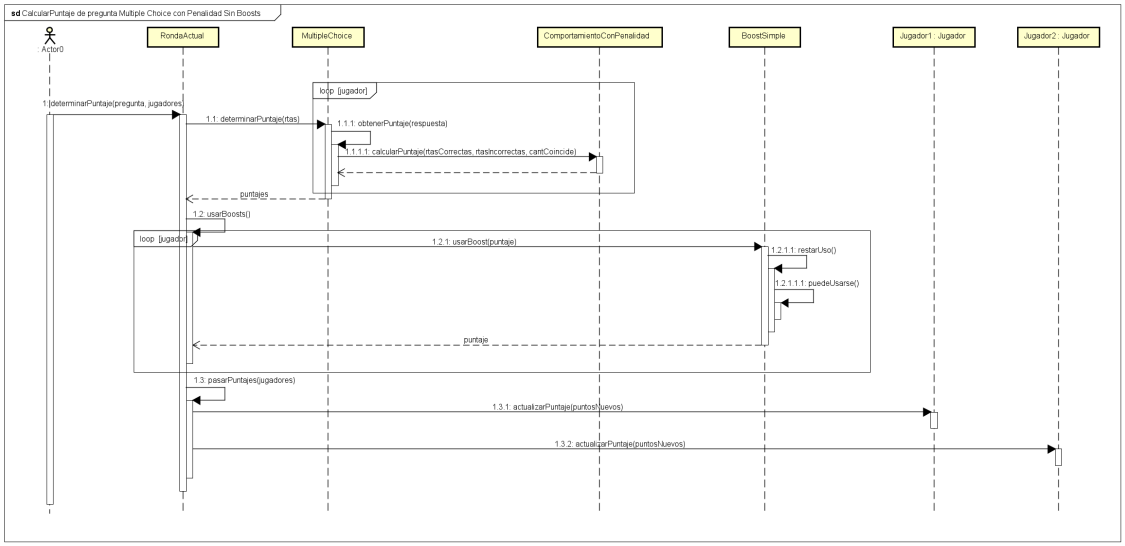

Figura 4: Cálculo del puntaje obtenido en una pregunta Multiple Choice con Penalidad Sin Boosts.

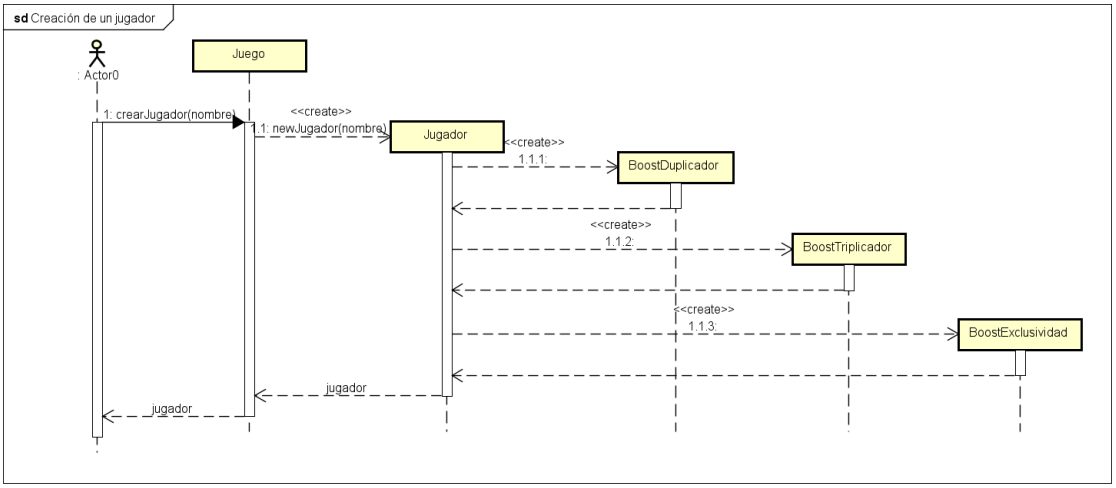

Figura 5: Creación de un jugador.

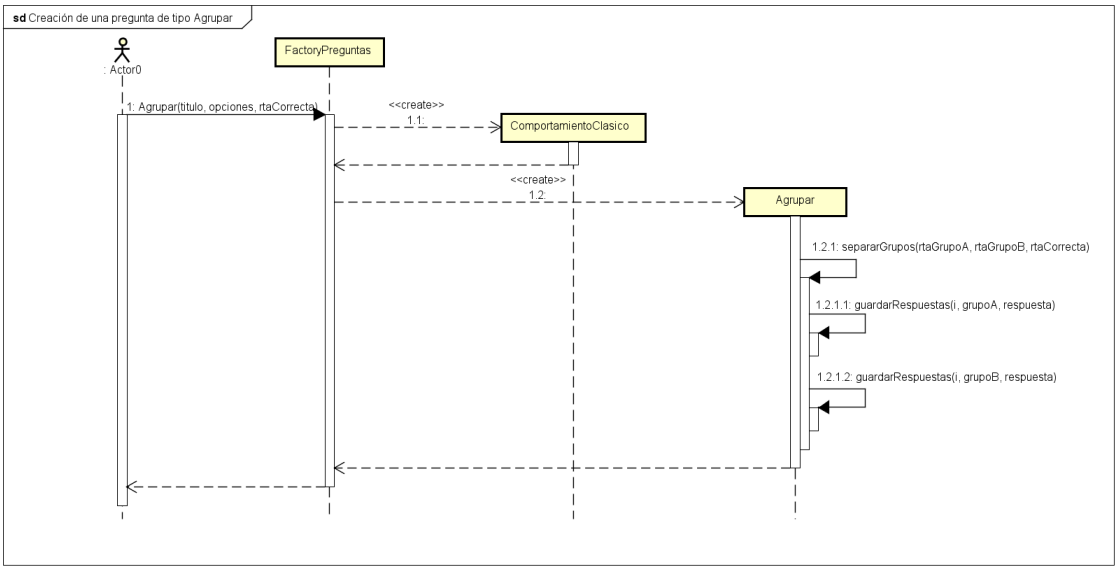

Figura 6: Creación de una pregunta de tipo Agrupar.

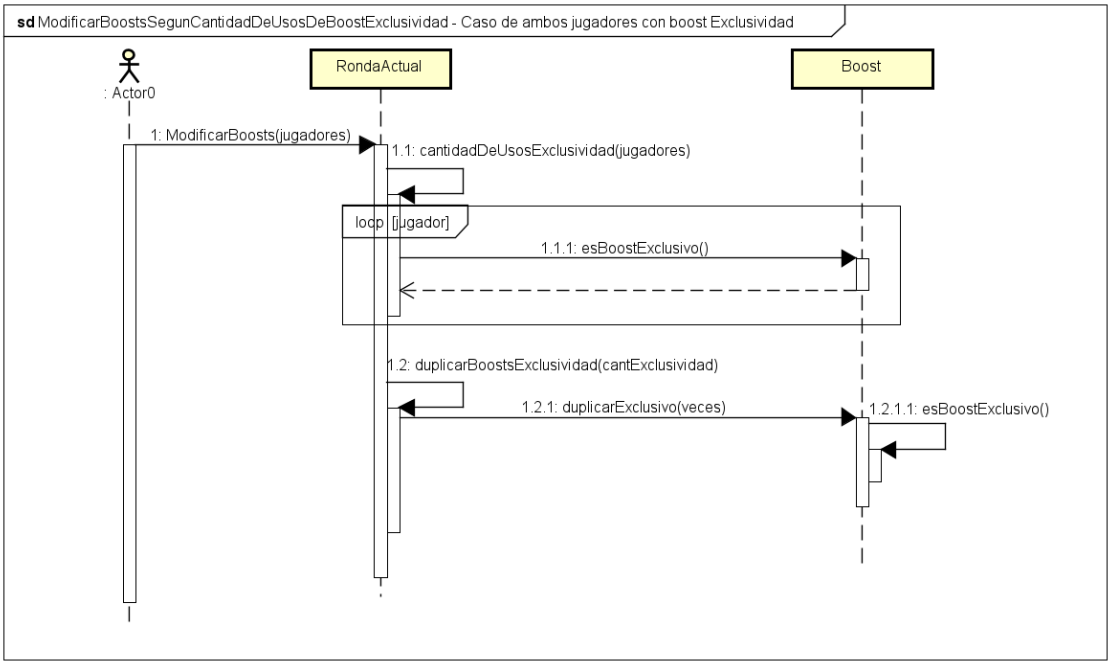

Figura 7: Modificación del *Boost* cuando ambos jugadores elijen exclusividad.

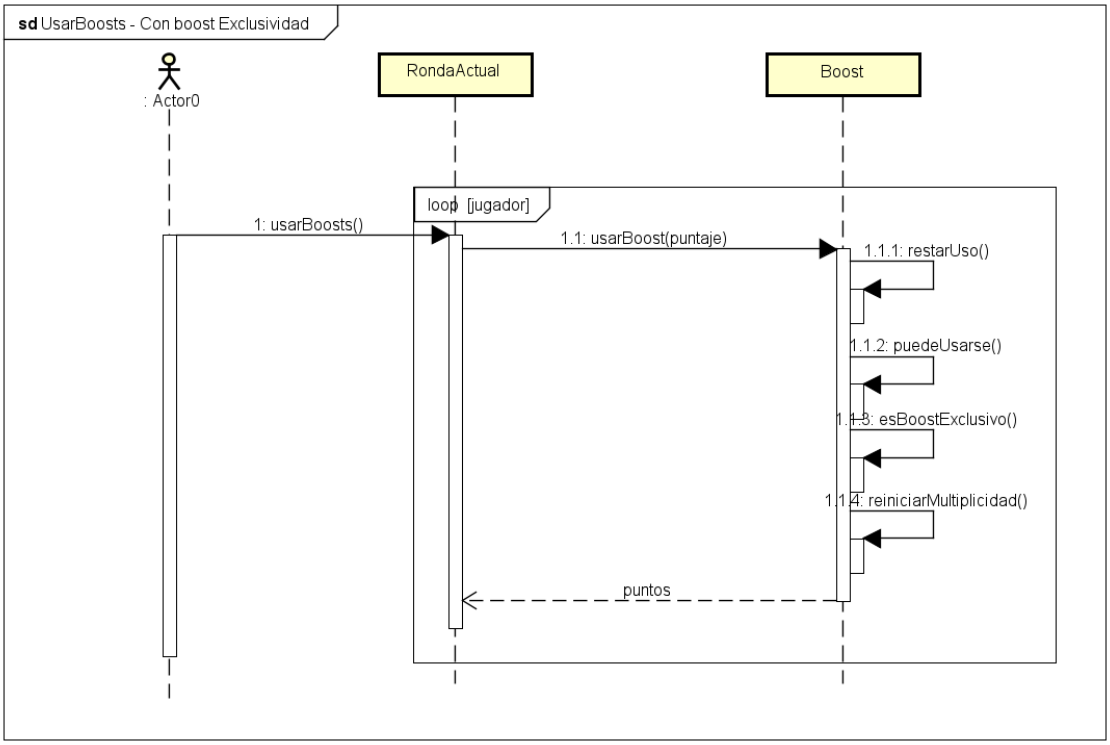

Figura 8: Uso *Boost* exclusivo.

7. Diagrama de Paquetes

En la figura 9 se incluye el diagrama de paquetes UML para mostrar el acoplamiento del modelo diseñado. Se observa que, si bien *Juego* se relaciona con la mayor parte de los paquetes (lo cual es lógico porque contiene la clase central *Juego*), cada paquete se relaciona con pocos paquetes. Por lo tanto, se puede afirmar que el acoplamiento es bajo.

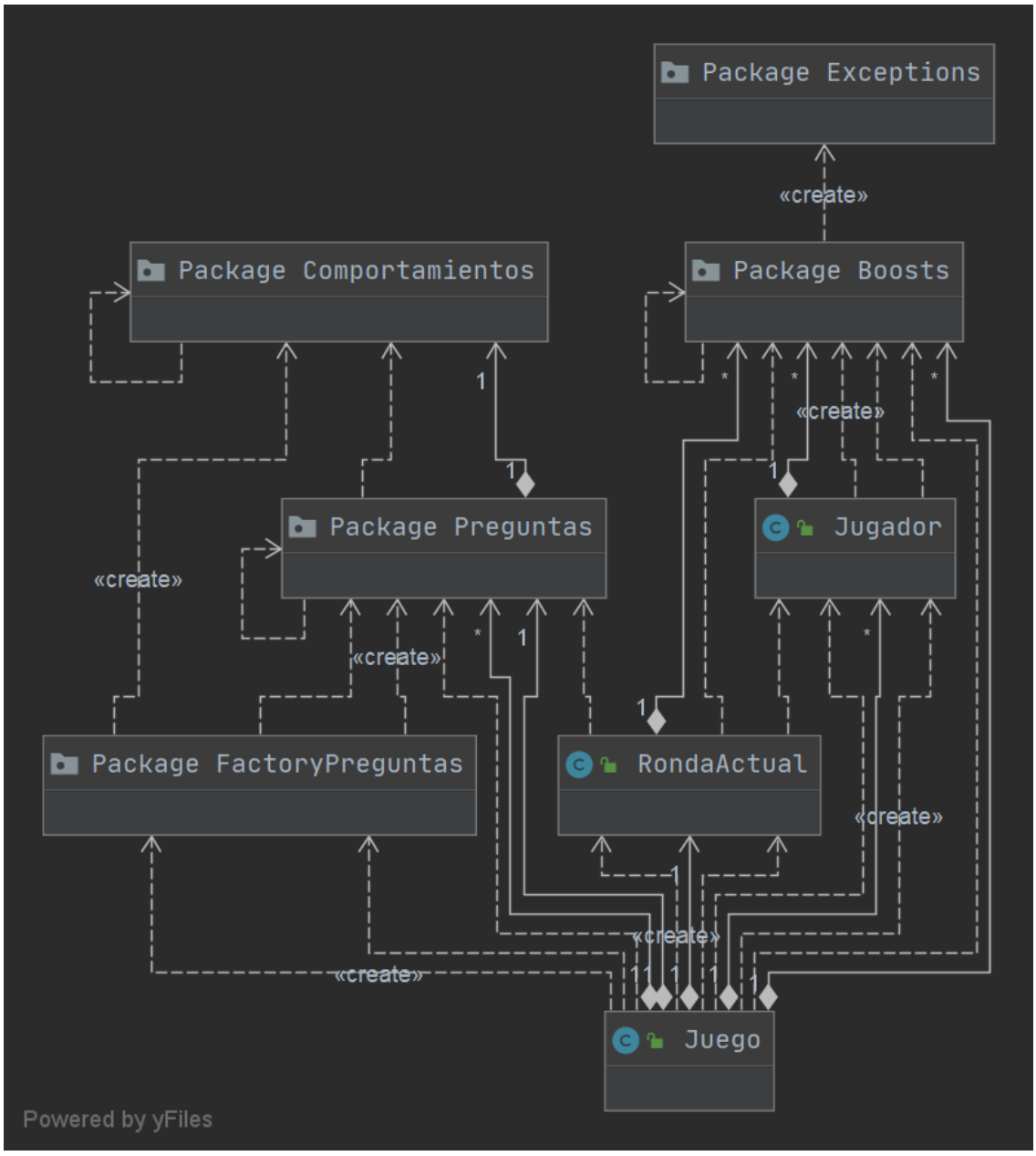

Figura 9: Diagrama de Paquetes

8. Diagrama de Estados

En la figuras 10 a 10 se presentan los diagramas de estados a fin de mostrar los estados y las distintas transiciones que suceden en los turnos.

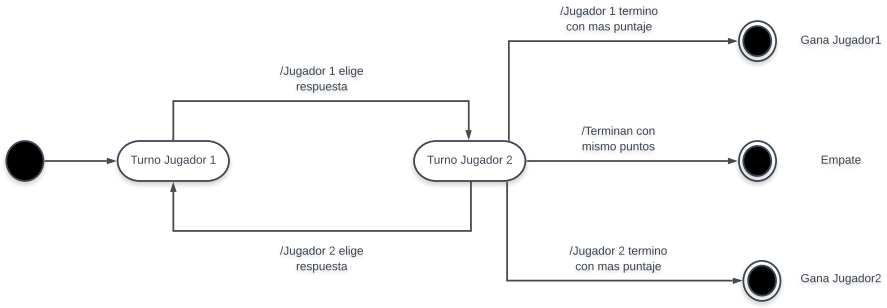

Figura 10: Diagrama de Estados