Universidad de Buenos Aires
Facultad De Ingeniería
Año 2020- 1er Cuatrimestre

Algoritmos y programación III (95.02)

(Curso 2)

Trabajo práctico N°2
Tema: Cajú
Fecha: 20 de agosto de 2020
Alumnos:

Chaparro, Raúl Antonio <rchaparro@fi.uba.ar>	- #96222
De Vita, Stefano Mauro <sdevita@fi.uba.ar>	- #104462
Grzegorczyk, Iván <igrzegorczyk@fi.uba.ar>	- #104084
Tarzia, Luciana Vanina < ltarzia@fi.uba.ar>	- #100662
Vazquez, Francisco Manuel <igrzegorczyk@fi.uba.ar>	- #104128

Índice

1. Introducción

El siguiente trabajo tiene como objetivo desarrollar en Java un juego tipo *quiz* para dos jugadores, con seis tipos de preguntas. El mismo deberá implementar una interfaz gráfica intuitiva que le permita a cada jugador seleccionar la opción u opciones que crea correctas, poder elegir el orden de las opciones o agruparlas en grupos según corresponda. El objetivo que se busca es poder aplicar los conceptos vistos sobre el paradigma de objetos, UX y calidad de *software*.

2. Supuestos

Para el desarrollo de la aplicación se tuvieron que analizar distintos casos borde y a partir de eso poder definir algunos supuestos en el flujo de la aplicación:

- En la clase Panel, para usar los métodos `usarTriplicador` y `usarDuplicador` se debe haber creado anteriormente una instancia de Jugador y una instancia de Pregunta.
- En el caso en que el jugador no responda una vez acabado el tiempo en el cual debía responder la pregunta, su respuesta será un conjunto vacío.
- En la pregunta del tipo `Multiple Choice con Puntaje Parcial` y en el caso que se utilice la exclusividad, se multiplicará el puntaje del jugador únicamente si uno de los jugadores responde mal. De lo contrario, si responde parcialmente bien o completamente bien, no se contarán los puntos para ninguno de los dos jugadores.
- En todos los tipos de pregunta debe existir por lo menos una respuesta correcta.
- En las preguntas del tipo *`Group choice`*, se deben agrupar todas las opciones, caso contrario la respuesta se considerará como incorrecta.

3. Diagramas de clases

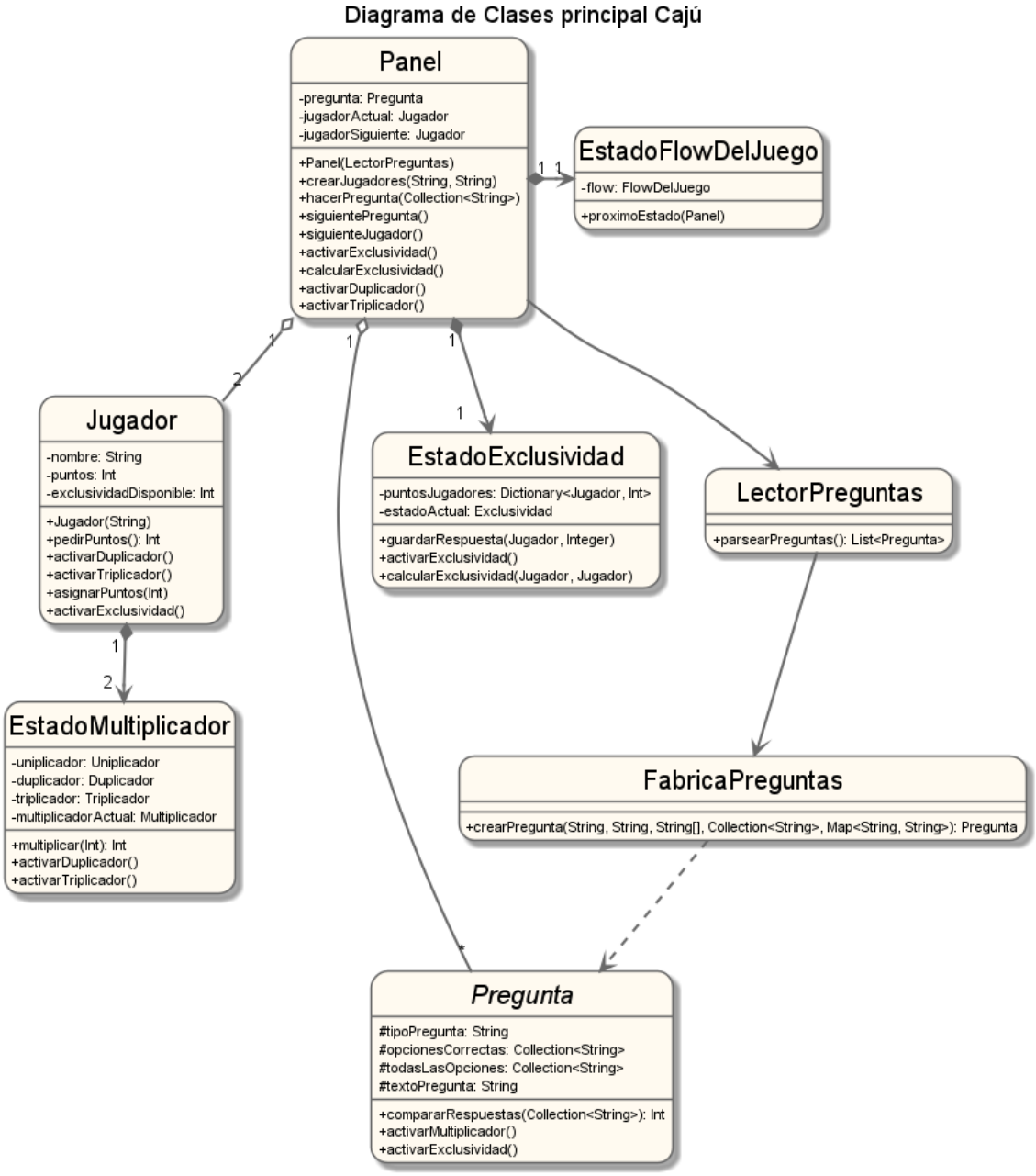

Figura 1: Diagrama de clase.

Como se puede ver en el diagrama de la figura 1, hay una clase donde se canaliza el flujo del programa. Esta clase es `Panel`. En ella se crean los dos jugadores y se mantiene una referencia al jugador actual y al siguiente que a su vez tienen una referencia del estado `EstadoMultiplicador` para saber si tiene disponible el multiplciador por dos y por tres. Se le indica a los jugadores contestar la pregunta y asignarse los puntos. Además tiene una lista de preguntas del tipo `Pregunta`

donde se guardan todas las preguntas del juego.

Las preguntas se crean por medio de la clase `LectorPreguntas`. La responsabilidad de esta clase es leer el archivo `JSON` donde se encuentran las preguntas a utilizar y parsearlas. Esto se muestra en la imagen de la figura 5.

Una vez leído el archivo con las preguntas, la clase `LectorPreguntas` utiliza a la clase `FabricaPreguntas` que de acuerdo al *`String`* que recibe como parámetro sabe que tipo de pregunta crear. De acuerdo al tipo de pregunta a crear, se llama a los distintos constructores de cada pregunta que heredan su comportamiento de la clase abstracta `Pregunta`. Al mismo tiempo, las preguntas implementan un comportamiento en específico, según que tipo de pregunta sea. Los comportamientos están implementados mediante la interfaz `Comportamiento`. Los mismos son: `ComportamientoClasico` (figura 2), `ComportamientoConPenalidad` (figura 3) y `ComportamientoConPuntajeParcial` (figura 4).

Por otra parte el `Panel` cuenta con una referencia al `EstadoExclusividad` para saber si el jugador eligió dicha opción. Esta funcionalidad tiene la peculiaridad de necesitar conocer a ambos jugadores, característica que el objeto `Jugador` no posee por si solo, pero si `Panel`.

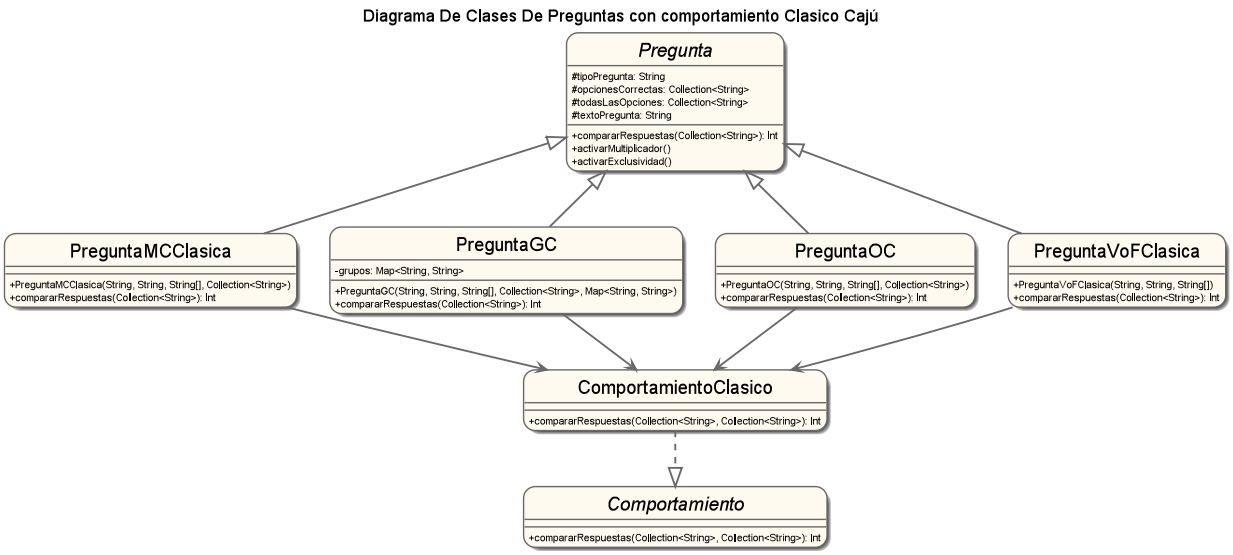

Figura 2: Diagrama de clases de Preguntas con comportamiento Clásico.

En la imagen de la figura 2 se muestra el diagrama para las preguntas que poseen un comportamiento clásico. Estas preguntas son: `PreguntaMCClasica`, `PreguntaGC`, `PreguntaOC` y `PreguntaVoFClasica`. Como se pueden ver, cada pregunta utilizan la forma de comparar las respuestas del tipo `ComportamientoClasico`. Es por este motivo que se decidió agruparlas juntas. Cabe aclarar que cada pregunta redefinió su constructor de a cuerdo a las necesidad de cada clase.

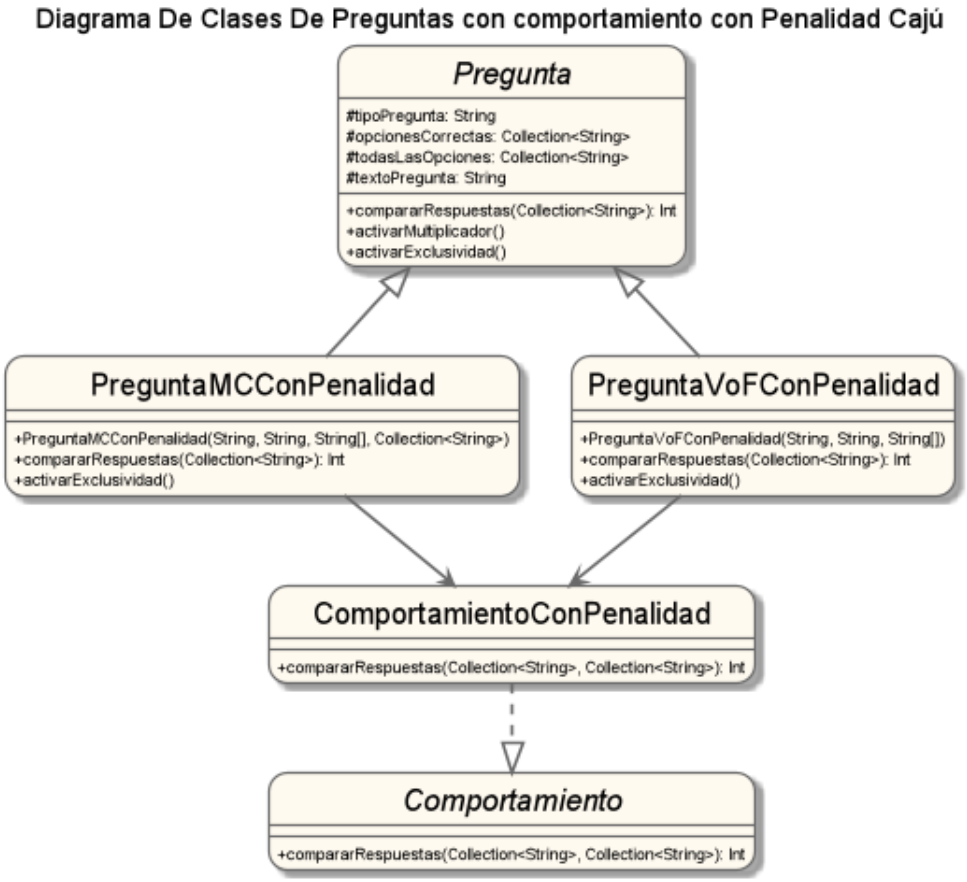

Figura 3: Diagrama de clases de Preguntas con comportamiento con Penalidad.

Algo similar ocurrió con las preguntas `PreguntasMCConPenalidad` y `PrecuntasVoFConPenalidad`. Como utilizan la misma lógica para la comparación entre las respuestas correctas y las elegidas por el jugador, comparten el mismo comportamiento.

Lo mismo se puede observar en el diagrama de la figura 4 para la pregunta `PreguntMCConPuntajeParcial`. Al haber una única pregunta de este tipo, se podría pensar como excesivo, pero esto deja abierta la posibilidad de que en un futuro se agregue un tipo nuevo de pregunta y se pueda incorporar sin grandes cambios, respetando el principio *open close*.

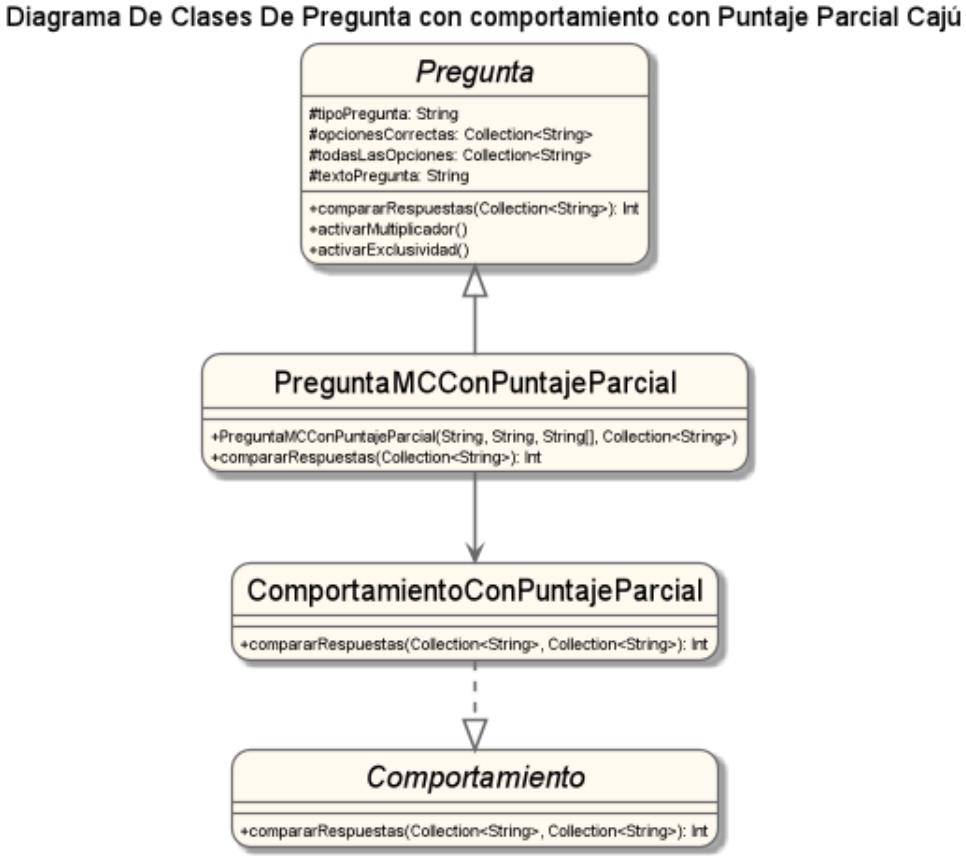

Figura 4: Diagrama de clases de Preguntas con comportamiento con Puntaje Parcial.

Es importante aclarar que la responsabilidad más importante de la clase `Pregunta` es la de saber como comparar la respuesta del jugador con la correcta. Es por eso que se crearon los comportamientos. De esta forma cada pregunta le delega a su comportamiento la comparación y esta le devuelve los puntos obtenidos por el jugador.

Anteriormente se explicó que las preguntas son leídas de un archivo `JSON`. En el diagrama de la figura 5 se muestra como la clase `LectorPreguntas` le delega la responsabilidad a las clases `Preguntas` y `Grupo` para saber como parsear el archivo `JSON` que lee. Es decir, la responsabilidad de la clase `LectorPregunta` es crear una lista de objetos `Preguntas` pero para eso se ayuda con las otras dos clases.

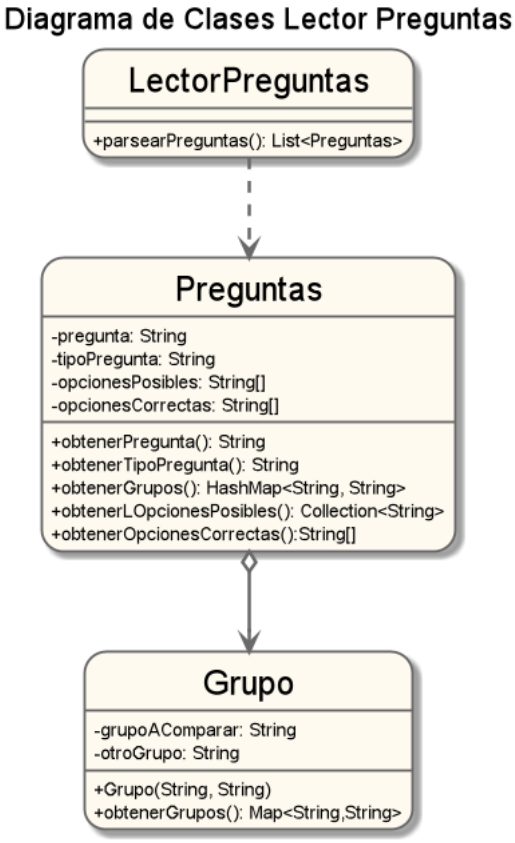

Figura 5: Diagrama de clases del lector de pregunta.

La clase `Jugador` tiene la responsabilidad de saber si tiene disponibles sus exclusividades y de asignarse los puntos calculados por `Pregunta` y `Comportamiento`.

Se implementaron tres máquinas de estados, una para las `exclusividades`, otra para los `multiplicadores` y una tercera para el `flujo del juego` (turnos y rondas). Por un lado, se creó la interfaz `Exclusividad` para administrar las "exclusividades" de los jugadores, por otro lado la interfaz `Multiplicador` y por último, la interfaz `FlowDelJuego` como se pueden ver en los diagramas de las figuras 6, 7 y 8 respectivamente.

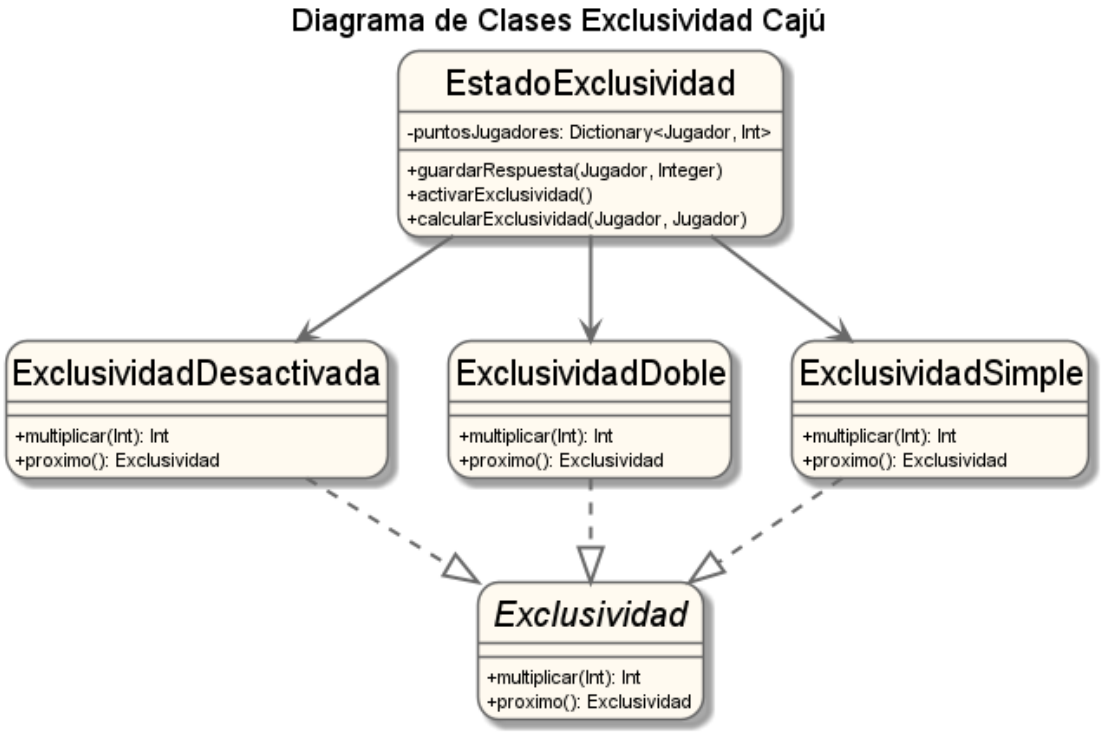

Figura 6: Diagrama de clases de la máquina de estados de las Exclusividades.

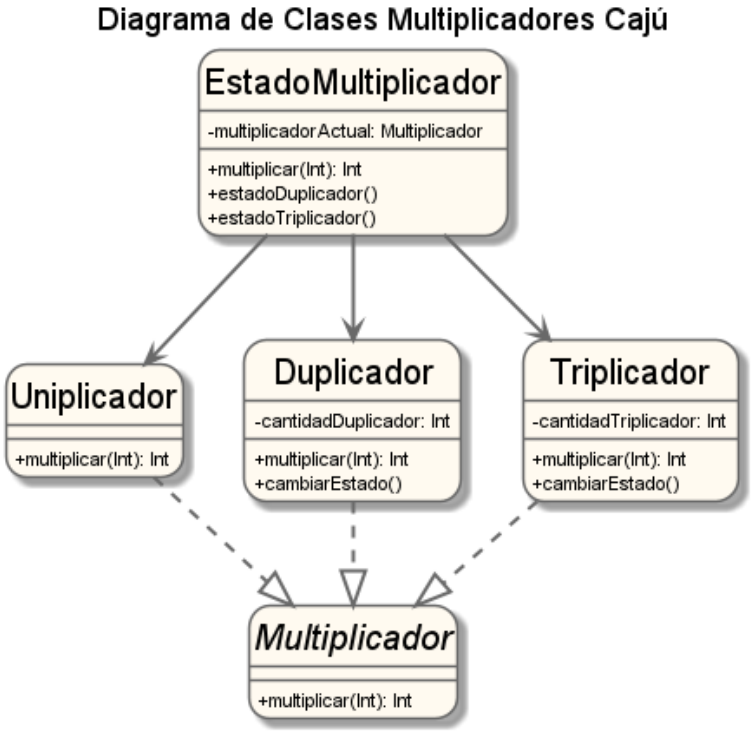

Figura 7: Diagrama de clases de la máquina de estados de los Multiplicadores.

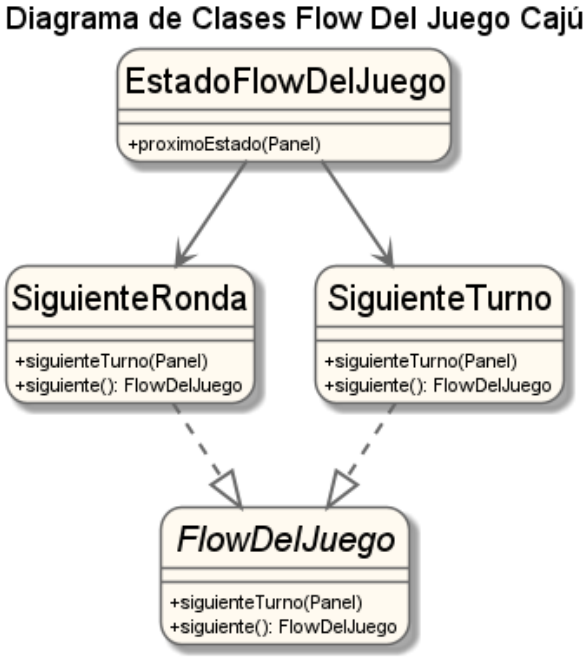

Figura 8: Diagrama de clases de la máquina de estados del flujo del juego.

4. Diagramas de secuencia

En la imagen de la figura 9 se muestra el diagrama de secuencia para la creación de las preguntas. Se puede ver como desde el lector de preguntas se crea la fabrica, el panel y se genera un *loop* donde el lector le manda un mensaje a la fábrica para crear una pregunta de acuerdo al tipo. La fábrica le manda el mensaje a la clase `Pregunta` y esta al `Comportamiento`. Este *loop* se repite por cada pregunta.

Por otro lado, en la imagen de la figura 10 se muestra la secuencia para la creación de los jugadores. En esta se puede ver como el `Panel` le manda los mensajes a la clase `Jugador` para que cree a cada jugador y como el panel interactúa con la clase `EstadoFlowDelJuego` para ir pasando de turnos y de rondas según sea el caso.

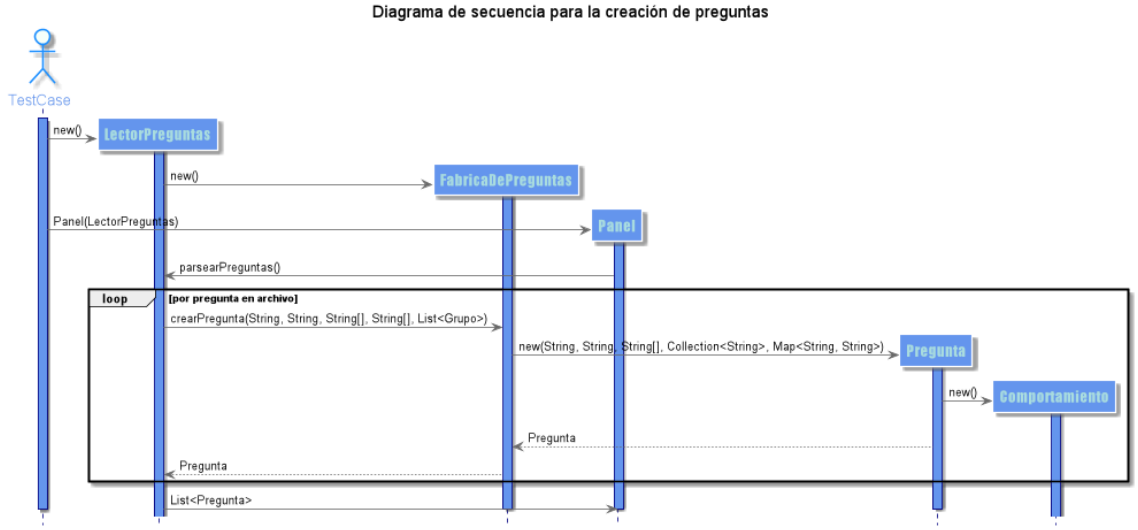

Figura 9: Diagrama de secuencia para la creación general de preguntas.

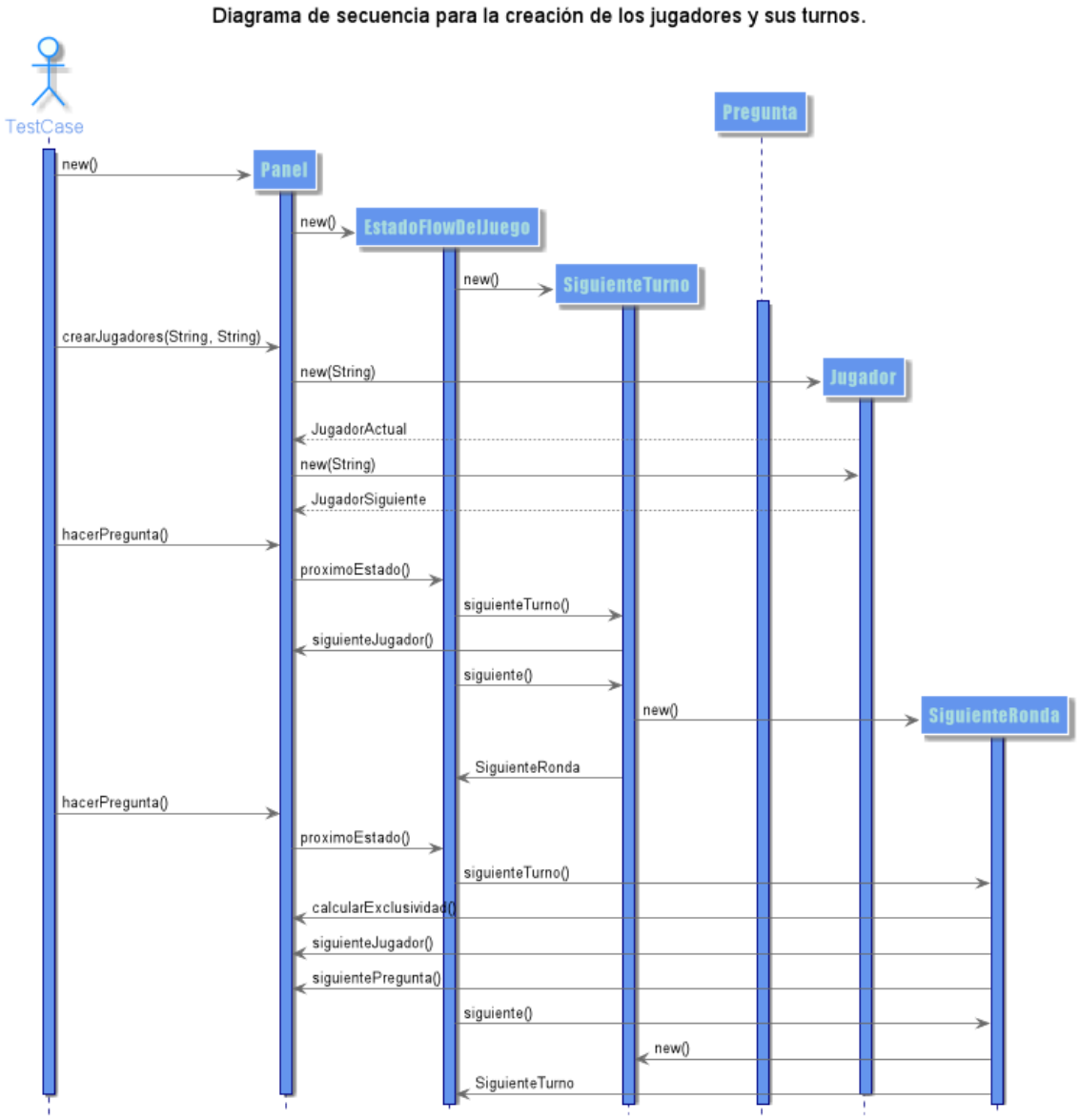

Figura 10: Diagrama de secuencia para la creación de Jugadores y la modalidad de turnos.

A continuación se muestran en la imagen de la figura 11 se muestra la interacción cuando se quiere utilizar la exclusividad con una pregunta con penalidad. Por otra parte, en la imagen de la figura 12 se muestra algo similar, pero para el caso de los multiplicadores.

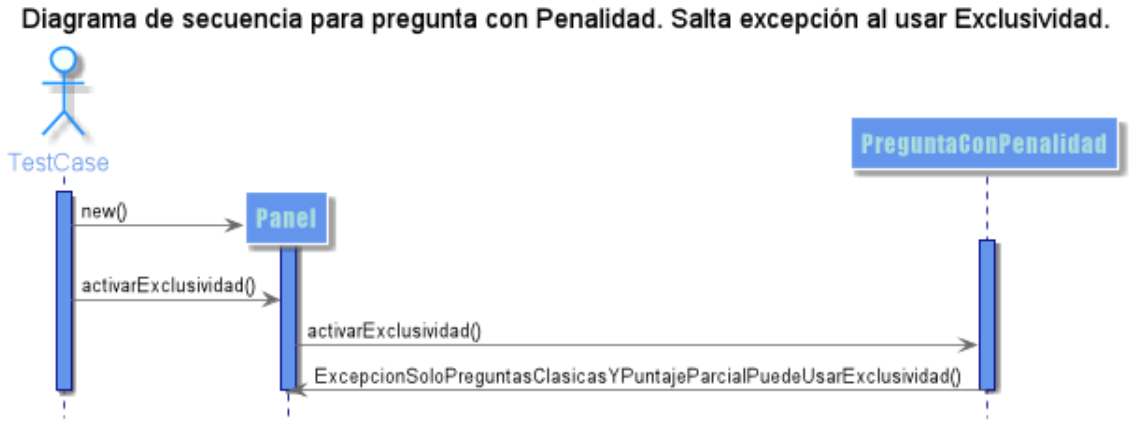

Figura 11: Diagrama de secuencia que lanza una excepción cuando se quiere utilizar Exclusividad en preguntas con Penalidad.

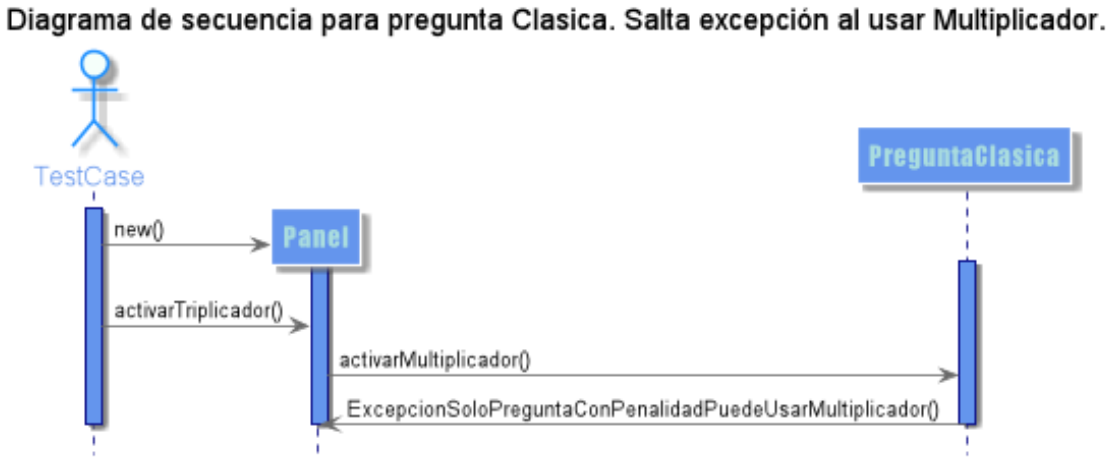

Figura 12: Diagrama de secuencia que lanza una excepción cuando se quiere utilizar Multiplicadores en preguntas sin Penalidad.

Además, en la imagen de la figura 13 se muestra el flujo del juego para una pregunta `Multiple Choice` utilizando `Exclusividad`. Por otro lado la imagen de la figura 14 muestra el flujo del juego para una pregunta `Verdadero o Falso con Penalidad` utilizando un `Duplicador`. Es interesante ver como en estas dos imágenes se muestran los *loops* para contestar y calcular y pedir los puntos de cada jugador.

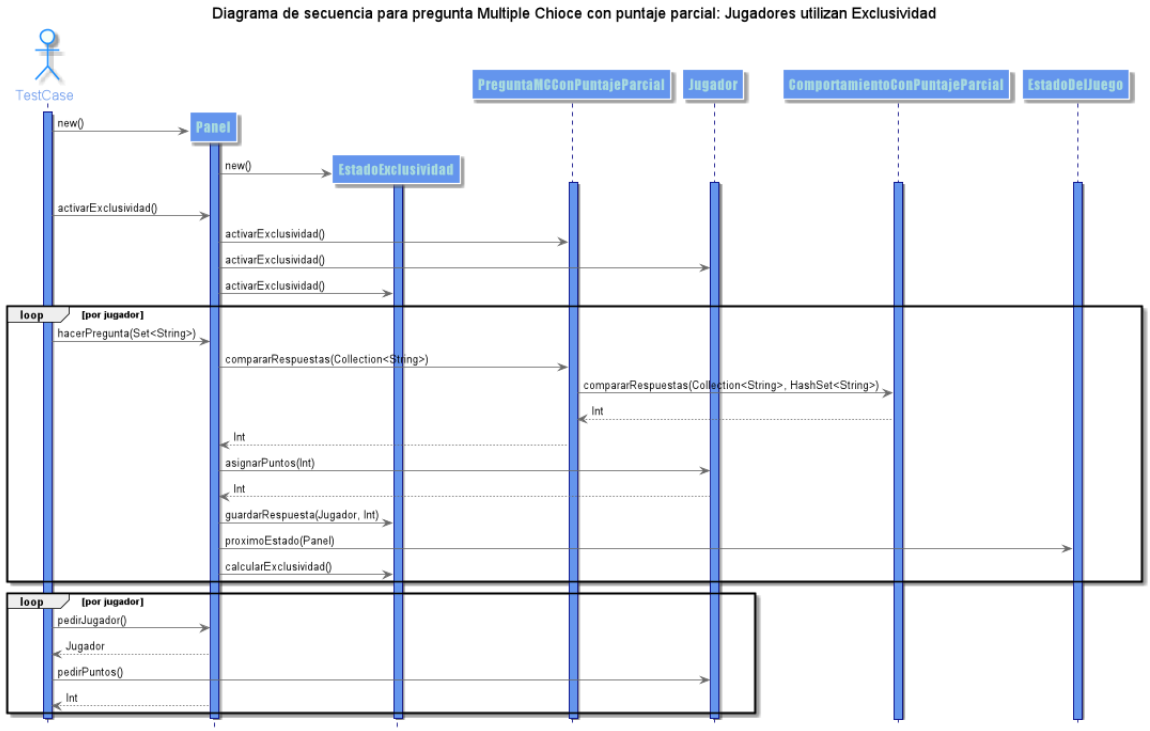

Figura 13: Diagrama de secuencia del flujo de una pregunta Multiple Choice con Exclusividad.

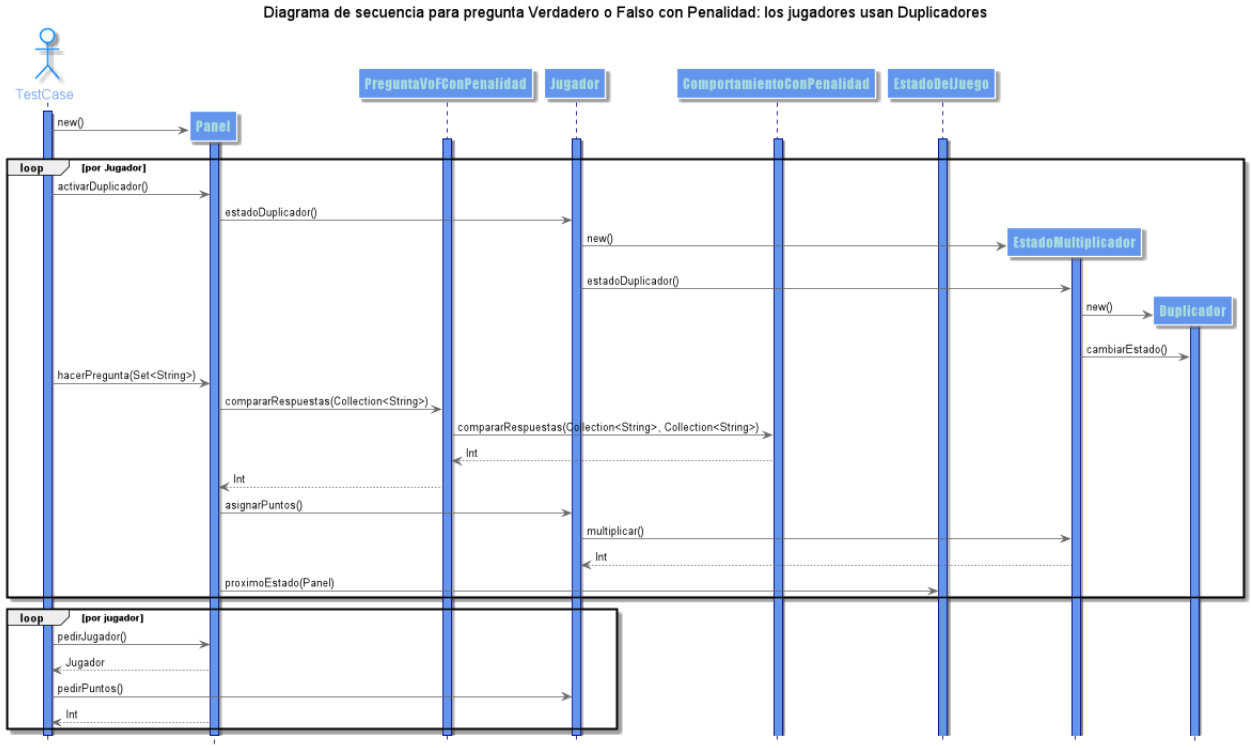

Figura 14: Diagrama de secuencia del flujo de una pregunta Verdadero o Falso con Penalidad con Duplicador.

5. Diagramas de paquetes

En la imagen de la figura 15 se muestra el diagrama de paquetes del modelo terminado. En el se puede ver como `Panel` es la entidad de mayor nivel que lleva el control del modelo, teniendo a los jugadores y a las preguntas y llevando el desarrollo del juego.

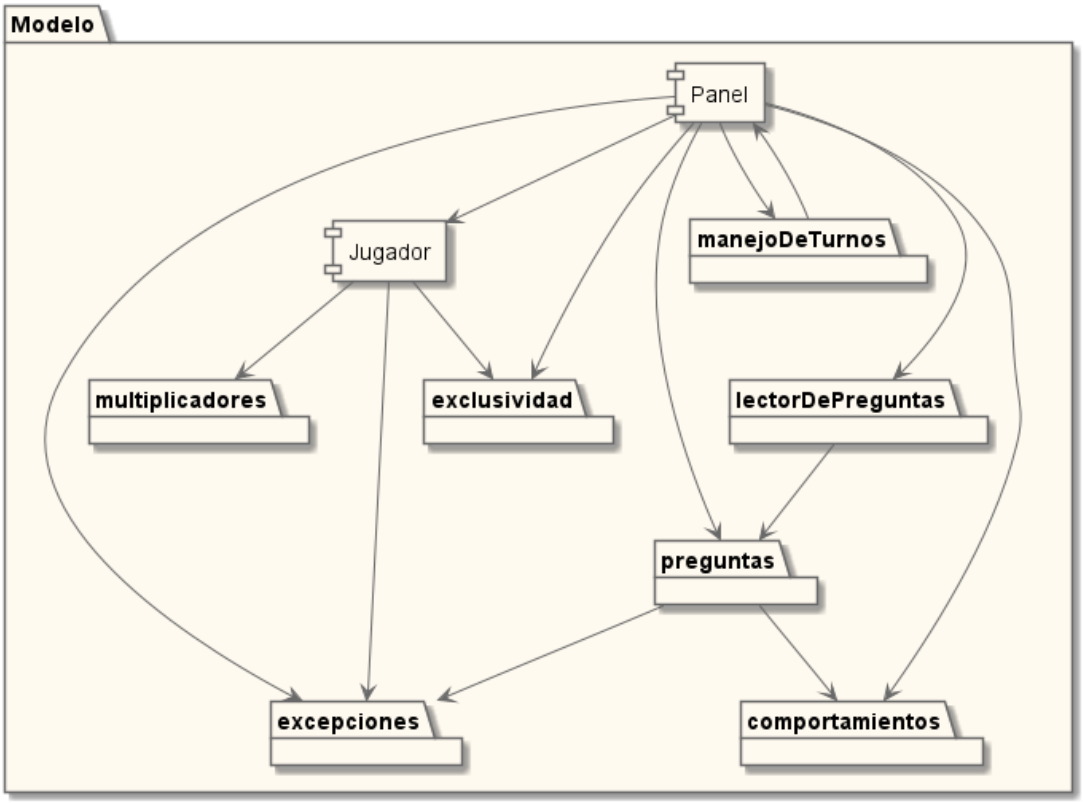

Figura 15: Diagrama de paquetes para el modelo del programa.

Por otra parte en el diagrama de la figura 16, se muestra el diagrama de paquetes de la interfaz gráfica. En el se puede ver como cada controlador le avisa al modelo para que realice acción sobre la vista. Además se muestra la dificultad que tuvimos para mantener este patrón ya que hay flechas del controlador hacia la vista que no son del todo correctas.

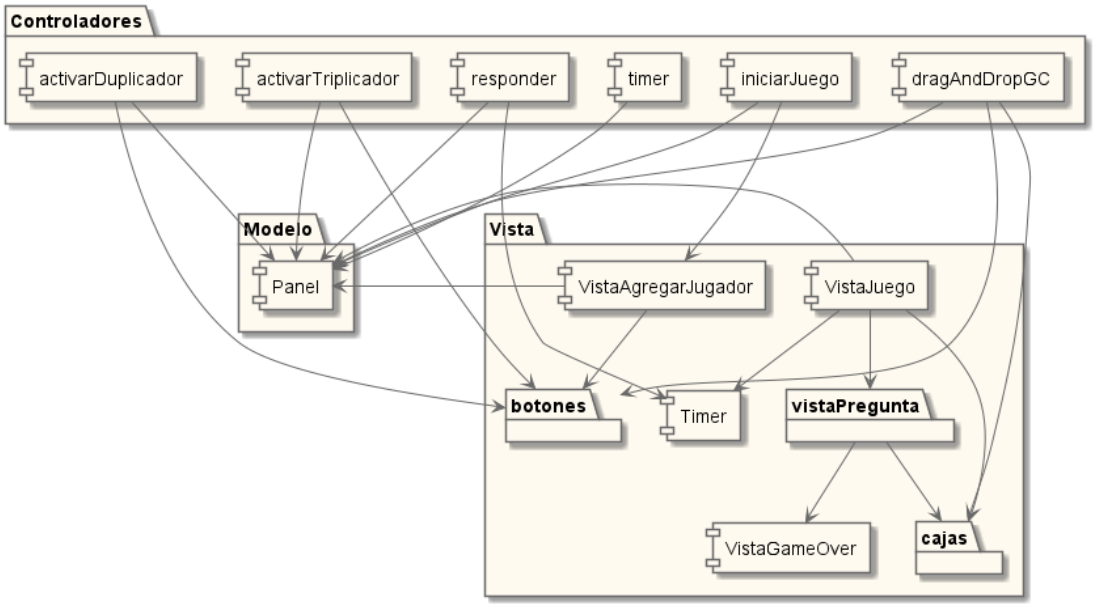

Figura 16: Diagrama de paquetes para la interfaz gráfica del programa.

6. Diagramas de estados

En la imagen de la figura 17 se muestra la máquina de estados que se implementó para el manejo de turnos y ronda. Para poder pasar de turno y de ronda se utilizó la clase `EstadoFlowDelJuego`. En esta clase se tiene como atributo una instancia de la clase `SigueinteRonda` o `SigueinteTurno`. La idea es que cada vez que el jugador 1 termine de contestar, desde panel se llama a el método `proximoEstado` del objeto " `estadoDelJuego`". Si el que respondió es el jugador uno, entonces `estadoDelJuego` es una instancia de `sigueinteTurno`, lo que permite cambiar de jugador y hace que `estadoDelJuego` apunte a una instancia de `sigueinteTurno`. Por su parte, cuando contesta el jugador dos, el método `proximoEstado`, cambia de ronda y hace que `estadoDelJuego` apunte a una isntancia de `Siguienteturno` volviendo a empezar el ciclo.

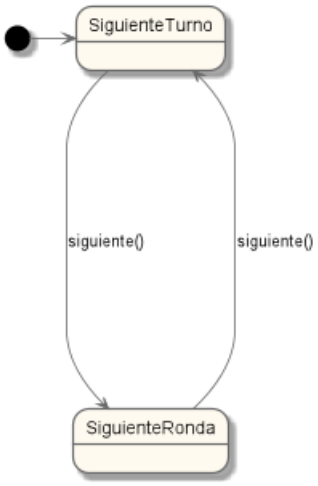

Figura 17: Diagrama de Estado para los turnos de los jugadores.

Para los multiplicadores, se implementó la máquina de estado que se muestra en la imagen de la figura 18. La idea principal es que el atributo `estadoMultiplciador` de cada jugador empiece con una instancia de `Uniplicador`. Luego, si el jugador decide utilizar el duplicador o triplicador, el `Panel` le avisa al jugador que cambie de estado, haciendo que su atributo sea una instancia de la clase `Duplicador` 0 `Triplciador` según corresponda. La responsabilidad que tendrán estas clases es la de poder cambiarse y multiplicar los puntos cuando el jugador se los asigna. Además cada tipo de multiplicador sabrá si el jugador le qiedan multplicaodres disponibles.

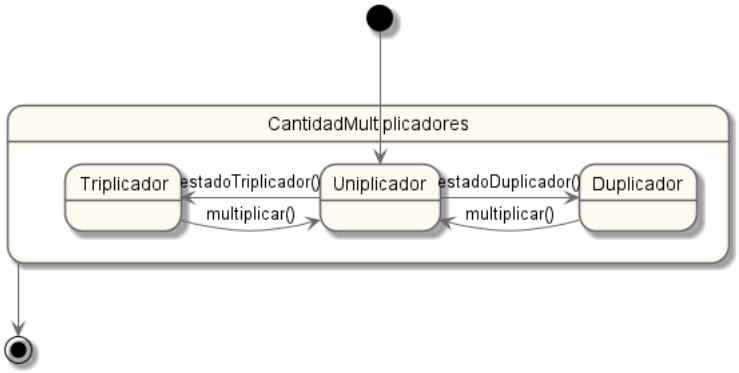

Figura 18: Diagrama de Estado para el uso de Multiplicadores.

La tercer máquina de estado que se implementó fue la de la exclusividad. Esta se puede ver en la imagen de la figura 19. La idea de esta maquina es poder manejar las exclusividades de cada jugador. En primer lugar se empieza con la exclusividad desactivada para pasar a una exclusividad simple si el jugador uno la activió y a una exclusividad doble si ambos la activaron. Esto se logra por medio del método `proximo()` que tiene la interfaz `Exclusividad`. Luego, al terminar la ronda de la pregunta, se llama al método `calcularExclusividad` que es responsabilidad de la clase

`EstadoExclusividad`, que dependiendo del tipo (desactivada, simple y doble) multiplica por dos o por 4 de acuerdo a lo lógica establecida y vuelve al estado a la exclusividad desactivada.

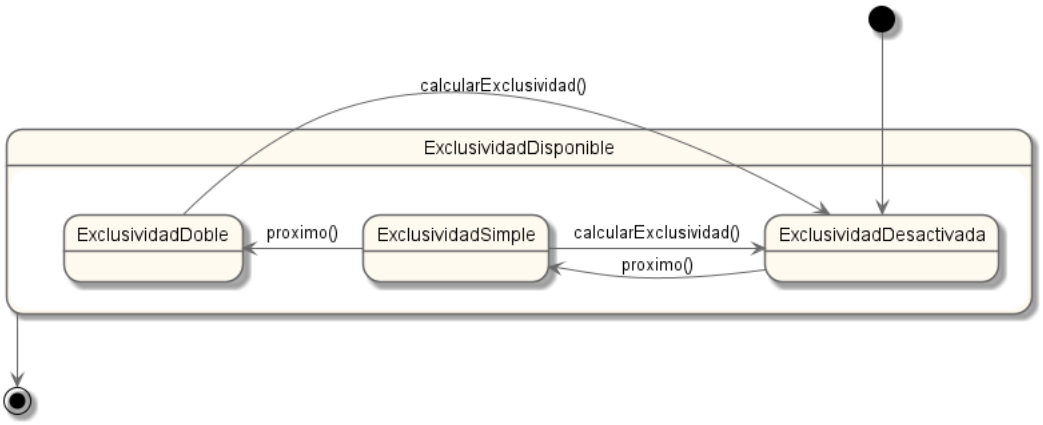

Figura 19: Diagrama de Estado para el uso de Exclusividad.

7. Detalles de implementación

A continuación se detalla un poco más exhaustivamente ciertos detalles que se usaron para el desarrollo del juego.

- Para la creación de las preguntas se utilizó el **patrón de diseño *Factory***. Se decidió implementarlo con el objetivo de liberarle a Panel la responsabilidad de la creación de las preguntas. De este modo, se encapsula la creación de las mismas en una clase externa. Existe una desventaja, que consiste en que la fábrica se implementa con un `switch case` que habría que modificar si se quisiera agregar otro tipo de pregunta en un futuro. Lo mencionado anteriormente rompe con el principio SOLID Open-Closed. Sin embargo, se concluyó que las ventajas que provee el patrón son mayores a las desventajas.

- Se implementaron tres tipos de máquinas de estados. Por un lado, se creó una para la implementación de los multiplicadores. Existe la interfaz *Multiplicador*; las clases que la implementan son **Duplicador**, **Triplicador** y **Uniplicador**. Esta última se utiliza en el caso en que no se haga uso de los multiplicadores. La clase **EstadoMultiplicador** se encarga de los cambios de estado. Por otro lado, existe otra máquina para las exclusividades. Las clases que implementan la interfaz *Exclusividad* son: **ExclusividadSimple**, **ExclusividadDoble** y **ExclusividadDescativada**. La clase **EstadoExclusividad** tiene el propósito del cambio de estados. La tercera máquina de estados es utilizada para el manejo del flujo de los turnos y las rondas del juego, en donde los turnos se definen por jugador y las rondas por pregunta. Para el cambio de estados se utiliza la clase **EstadoFlowDelJuego**. La interfaz

utilizada en este caso es *FlowDelJuego* y las clases que la implementan son **SiguienteTurno** y **SiguienteRonda**.

- Para la lectura de archivo `JSON` se utilizó la librería de JAVA de serialización/deserialización `GSON`. Por cada pregunta se utilizó el siguiente formato de json, donde se enceuintra el campo `tipoPregunta` que le idica a la fábrica de preguntas el tipo de pregunta a generar, `pregunta` que es la oración en sí que se mostrará en la pantalla, `opcionesPosibles` que son todas las opciones que se mostrarán en pantalla, `opcionesCorrectas` que le indicaran a cada pregunta las opciones correctas que utilizará en la comparación y `grupos` que son los grupos que se muestran en las preguntas *order choice*.

```
{
   "tipoPregunta": "preguntaVoFClasica",
   "pregunta": "(!true && false) || (true == false) && (!false || true)",
   "opcionesPosibles": [
     "Verdadero",
     "Falso"
   ],
   "opcionesCorrectas": [
     "V"
   ],
   "grupos": [
     {
       "grupoAComparar": "",
       "otroGrupo": ""
     }
   ]
 }
```

- Para la comparación del tipo de pregunta *order coice* se embebió dentro de las opciones correctas el nombre del grupo a la cual pertenecían, de esta forma la comparación se podía hacer como en las preguntas clásicas. La incorporación del grupo en las respuestas correctas es una responsabilidad que tiene el constructor del *order choice*.

- Al igual que en la creación de las preguntas, para las pantallas de la interfaz gráfica se utilizó el patrón de diseño ***Factory***. Los distintos tipos de pantalla se dividen en, preguntas clásicas,

con penalidad, *order choice* y *group choice*.En un primer momento no se tenía pensado crear las vistas de cada pantalla con este patrón, pero luego de un primer desarrollo de la interfaz, se decidió utilizar implementar vista que este en una capa superior a las de las preguntas y que se encargue de crearlas. Bajo este escenario, el patrón *factory* ayuda a generar un código más entendible. Así la `vista juego` implementa una fabrica de vistas que luego de cada ronda se actualiza creando la pantalla de la próxima pregunta.

- Para el manejo de la interacción entre el usuario y la interfaz se utilizó el patrón **modelo vista controlador (MVC)** y el patrón **Observer**. El juego comienza con la creación del `lector de preguntas` y el `Panel` (que en su constructor se leen y se crean las preguntas del juego). Luego se crea la " vista principal", denominada `vista juego`. Esta vista, que se encarga de manejar a las vistas de las preguntas, observa al `Panel`, por lo cual luego de cada ronda el `Panel` le avisa a la vista para que se actualice y crea la vista de la nueva pregunta. La primera acción de la `vista juego` es crear la pantalla donde se cargan a los jugadores. Una vez cargados y creados, la vista va creando las vistas de las distintas preguntas. En sí cada vista se compone de distintas " cajas ". Se puede pensar que cada caja es una sección de la pantalla. En la vista principal se crean las cajas de los jugadores y del tiempo. La caja de los jugadores observa a cada jugador, para así de esta forma poder ir mostrando el turno del jugador correspondiente, los puntos y las opciones de exclusividad y multiplicadores. Por último cada pregunta genera la caja pregunta donde se muestran la pregunta en sí y las opciones con el botón de responder. Lamentablemente, no pudimos utilizar el patrón correctamente en todo el desarrollo, ya que se complicó el manejo de las excepciones, por lo cual se puso un poco de lógica en los controladores, algo que sabemos no es del todo correcto.

- Para las vistas se tienen distintos tipos de botones y cada botón con su controlador, para saber que accion tomar y avisar al controlador. por ejemplo:

  - Los botones de los multiplicadores, llaman a panel al método `activarDuplicador()` o `activartriplicador()` que pone en funcionamiento a la máquina de estado de los multiplicadores.
  - Por otro lado el botón de exclusividad, llama al método `activarExclusividad()` que pone en marcha a la maquina de estado de exclusividad.
  - El botón de responder, activa el método `hacerPregunta()`, que una vez que se calculan los puntos pone en marcha a la máquina de estados del flow del juego para, cambiar de turno o de ronda.
  - Además se encuentra el control del *drag and drop* para poder ordenar y agrupar las respuestas del *order choice* y *group choice*. Se decidió utilizar esta opción para el *order*

choice ya que se muestra gráficamente como va quedando la secuencia elegida.

- Por último se tienen los botones para iniciar el juego y el de opción que hacen acción sobre el modelo para comenzar a jugar (y por ejemplo empezar a escuchar la música del juego) e ir colocando las opciones en el arreglo de la respuesta del jugador.

- Además de las especificaciones solicitadas, se agregaron 3 sonidos al juego. Una música que acompaña durante todo el transcurso de la partida y dos audios que indican si la respuesta fue correcta o incorrecta. De esta forma se cree que se mejora la jugabilidad ya que además de ver los puntos, el jugador tiene un indicio auditivo de si contestó correcta o incorrectamente.

8. Excepciones

1. ExcepcionPreguntaGCInvalida: se lanza cuando la cantidad de opciones recibidas en la pregunta de tipo Group Choice es menor a dos o es mayor a seis. También cuando no se reciben los grupos que contendrán las respectivas respuestas.

2. ExcepcionPreguntaMCInvalida: se lanza cuando la cantidad de opciones recibidas en la pregunta de tipo Multiple Choice es menor a dos o es mayor a cinco.

3. ExcepcionPreguntaOCInvalida: se lanza cuando la cantidad de opciones recibidas en la pregunta de tipo Ordered Choice es menor a dos o es mayor a cinco.

4. ExcepcionPreguntaVOFInvalida: se lanza cuando la cantidad de opciones correctas recibidas en la pregunta de tipo Verdadero o Falso es distinta de uno.

5. ExcepcionSoloPreguntaConPenalidadPuedeUsarMultiplicador: se lanza cuando se quiere usar un Multiplicador en preguntas de los tipos Clasicas o con Puntaje Parcial.

6. ExcepcionSoloPreguntasClasicasYPuntajeParcialPuedeUsarExclusividad: se lanza cuando se quiere usar un Multiplicador en preguntas del tipo con Penalidad.

7. ExcepcionTipoPreguntaInvalida: se lanza cuando se recibe un tipo de pregunta inexistente.

8. ExcepcionYaUsasteTuDuplicador: se lanza cuando el jugador quiere usar de nuevo el Multiplicador x2.

9. ExcepcionYaUsasteTuTriplicador: se lanza cuando el jugador quiere usar de nuevo el Multiplicador x3.

10. ExcepcionYaActivasteTuExclusividad: se lanza cuando el jugador quiere usar nuevamente su exclusividad en su turno.

11. ExcepcionYaUsasteLasExclusividades: se lanza cuando el jugador quiere usar su exclusividad cuando ya no tiene más.

12. ExcepcionYaNoHayPreguntasParaHacer: se lanza cuando ya no hay preguntas disponibles.

9. Conclusión

En primer lugar se puede afirmar que se alcanzó el objetivo planteado al principio del trabajo, logrando el desarrollo de un juego con diversas preguntas y distintos tipos de sumas de puntaje. Pero el camino que se transitó no fue directo. Viendo en retrospectiva, tuvimos que realizar diversos cambios (algunos más profundos que otros) durante el desarrollo. Esto se dio, en parte a las constantes iteraciones que se hacían sobre el código y a las reuniones semanales con el corrector.Por mencionar unos de los cambios más drásticos en un principio se decidió implementar una clase por por tipo de pregunta. Luego, se notó que era innecesario tener tantas clases y se decidió hacer clases de pregunta por tipo de comportamiento. Con este razonamiento, se vio que no se podía generalizar el comportamiento para el tipo de pregunta *Group Choice*. Como resultado, se decidió implementar una clase por tipo de pregunta y a la vez, definir una interfaz de comportamiento, en la cual, cada tipo de pregunta implementa un comportamiento.

También el trabajo se vio afectado por los tiempos. Más que nada en la fase de implementación de la interfaz gráfica. A pesar de tener una planificación estricta, debido a la falta de experiencia del grupo con las herramientas de javaFX, resultó dificil crear una interfaz "bonita", por lo cual se optó por una interfaz mas simple y minimalista, pero funcional . Como consecuencia, se priorizó el funcionamiento sobre lo estético.

A pesar de los contratiempos y problemas que presentaron, se los pudo ir solucionando y avanzando con el desarrollo, agregando algunos detalles extras para mejorar el funcionamiento y la experiencia del usuario. Todo esto se logro aplicando las metodologías y criterios vistos durante la cursada sobre el paradigma de objetos y las buenas prácticas de desarrollo de *software*.