FACULTAD DE INGENIERIA
Universidad de Buenos Aires

Trabajo Práctico 2 — Algohoot

[7507/9502] Algoritmos y Programación III
Curso 1
Primer cuatrimestre de 2024

Alumno	**Número de padrón**	**Email**
Aguilar Bugeau, Pedro José	100685	paguilar@fi.uba.ar
Mamani, Daniel	109932	dmamani@fi.uba.ar
Mortimer, Federica	108034	fmortimerfi.uba.ar
Pelliciari, Agustin	108172	apelliciari@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 una aplicación completa de un sistema al estilo de la pagina "Kahoot.[en] Java utilizando los conceptos del paradigma de la orientación a objetos vistos hasta ahora en el curso, incluyendo el modelo de clases, sonidos e interfaz gráfica. La aplicación deberá ser acompañada por pruebas unitarias e integrales y documentación de diseño.

2. Supuestos

- El usuario siempre ingresa un nombre válido, en caso de ingresar un nombre ya utilizado se lanzara una excepción.
- El orden de respuesta a las preguntas es el orden en el que se ingresaron a los jugadores en el inicio del juego.
- El puntaje de los jugadores puede ser negativo, iniciando siempre en 0. Esto se debe a que algunas preguntas pueden restar puntos en lugar de sumarlos. Sería injusto que un jugador no reciba penalización por respuestas incorrectas solo porque su puntaje inicial es 0.
- Se pueden utilizar varios poderes al mismo tiempo.

3. Modelo de dominio

Las clases que conforman principalmente a la aplicación son:

- **Game:** Clase principal del Algohoot, maneja tanto la lógica de juego (elección aleatoria de preguntas, actualización del tablero de puntos, validación de respuestas, etc.), como la creación de las pantallas de la interfaz gráfica y el registro de entrada de respuestas de cada jugador
- **Player:** Encapsula la identidad de cada participante en el juego, almacena atributos esenciales para el comienzo del juego, ya sea el nombre y, además, un 'score' que refleja su rendimiento en cada una de las respuestas. Por último tiene definido los poderes (multiplicator, nullifier y exclusivity) que el mismo jugador va a poder usar a lo largo del juego. Tiene métodos para tanto responder a la pregunta asignada, almacenando en una lista la/s respuesta/s contestadas por el jugador, como para activar el poder que se desee utilizar.
- **Question:** Representa una estructura abstracta para las preguntas en el juego, tiene como subclases que heredan de esta a: 'OrderedChoice' (Preguntas donde las opciones deben ser seleccionadas en un orden específico), 'MultipleChoice' (Preguntas de opción múltiple donde los jugadores eligen una o varias respuestas correctas), 'TrueOrFalse' (Preguntas de verdadero o falso, donde los jugadores seleccionan entre dos respuestas posibles), 'GroupChoice' (Preguntas donde se agrupan en dos grupos distintos elementos según ciertos criterios definidos en la consigna). 'Question' tiene como atributo un identificador único (ïd"), el contenido ('content'), la temática ('theme') y una lista de opciones ('choices') que esta compuesta de entre 2 a 5 opciones de tipo 'Choice' entre las cuales los jugadores pueden elegir la respuesta correcta. Finalmente cuenta con un modo ('mode') que define exactamente el tipo de pregunta. Esta puede ser 'Classic' (Modo común de juego, si se acierta se suma un punto y si no, no ocurre nada), 'Partial' (Por cada pregunta acertada se suma un punto, siempre y cuando no se haya elegido la correcta) y, por último 'Penalty' (Por cada opción correcta se suma un punto y por cada incorrecta se resta uno).

- **Choice:** Define una opción de una pregunta del juego. Cada opción tiene como atributo el contenido de esta, 'correction' que es de tipo 'Correction' (clase abstracta que tiene como subclases que heredan de ella a 'Correct' e 'Incorrect') que indica si la opción es correcta o incorrecta y cómo, dependiendo de esto, debe afectar el puntaje del jugador. Este objeto se asigna usando un patrón de diseño "Factory"(en la aplicación 'FactoryCorrection') explicado más detalladamente en el inciso **5.3 "Patrones de diseño".**

  Score: Modela el puntaje de un jugador en el juego. Utiliza el patrón de diseño 'State' para gestionar su estado (explicado más detalladamente su uso en el inciso **5.3 "Patrones de diseño")**. Tiene como atributos 'totalScore', entero que representa el puntaje acumulado del jugador, y 'state' que nos indica el estado actual del score que puede ser 'NormalState' o 'CanceledState'. De este patrón salen los métodos 'cancelScore' (Cambia el estado del puntaje a CanceledState, indicando que el puntaje ha sido cancelado) y 'restoreScore' (Restaura el estado del puntaje a 'NormalState', reactivando el puntaje después de haber sido cancelado). Además tenemos 'addScore()' y 'substractScore()' que dependiendo el método se suma o resta a 'totalScore' el numero pasado por parámetro.

- **Power:** Por último tenemos la clase abstracta 'Power', como mencionamos anteriormente, cada jugador tiene a su disposición una cierta cantidad de poderes que podrá utilizar para beneficio propio al comenzar el juego. Al igual que la clase 'Score', 'Power' utiliza el patrón de diseño 'State' para una mejor manipulación de la activación y uso de poderes. Cada poder cuenta con un estado ('PowerState') que puede ser tanto activo ('ActiveState') para indicar que el poder esta siendo utilizado, inactivo ('InactiveState') para mostrar que el poder no fue utilizado por el jugador aún y usado ('UsedState') que ocurre una vez que el poder termina de ser usado. Gracias a este patrón 'State' tenemos métodos como 'isActive()' o 'wasUsed()'.

Por último tenemos clases como 'JsonParser' o los packages 'handlers' y 'screens'. 'JsonParser' se encarga de de leer y convertir el contenido del archivo 'preguntas.json' a objetos de Java.

En cuanto a los paquetes 'handlers' y 'screens', estos encapsulan la lógica de la interfaz de usuario y la gestión de eventos de los botones. El paquete 'screens' contiene las clases que definen las diferentes pantallas de la aplicación, como la pantalla de ingreso de jugadores o la pantalla de juego.

El paquete 'handlers', por otro lado, se encarga de manejar las acciones del usuario, como clics de botones, entradas de texto y manejo de scrollbars. Estas clases de manejadores actúan como intermediarios entre la interfaz de usuario y el funcionamiento interno del código.

4. Diagramas de clase

Para explicar las relaciones entre todos los objetos que interactúan dentro de nuestra aplicación decidimos, dividir los diagramas de clase de la siguiente manera:

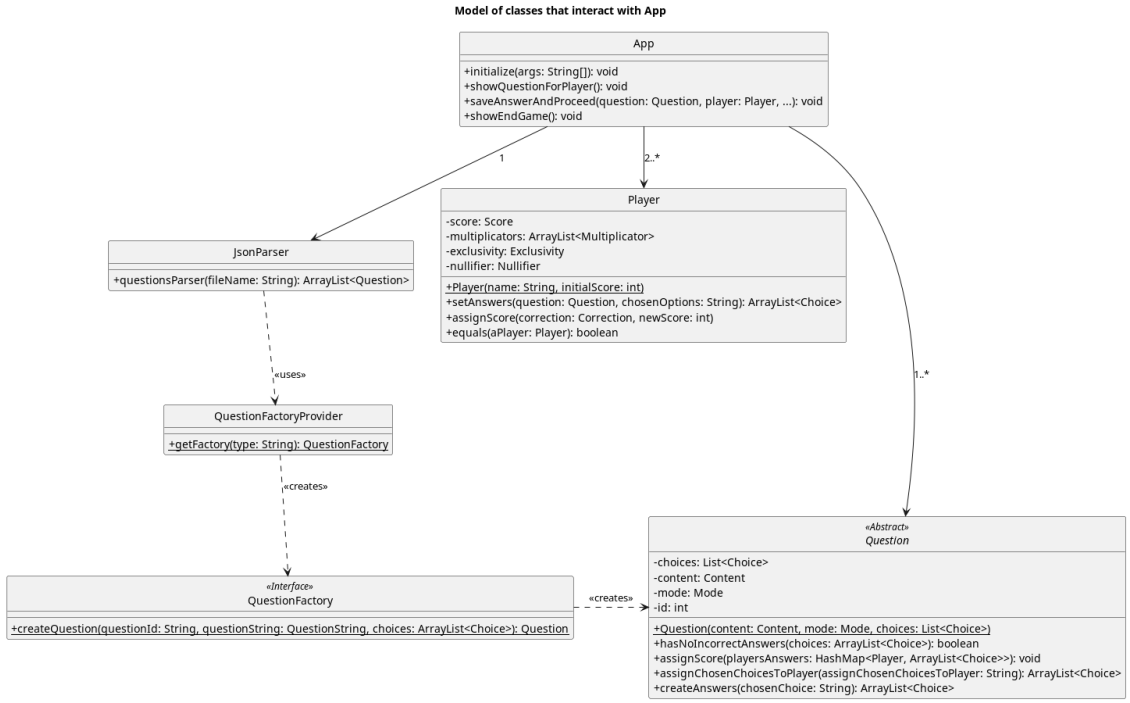

Figura 1: Diagrama de clase que modela a los objetos que interactúan con 'App'.

En este modelo participan los objetos en los cuales 'App' delega comportamiento: está 'Player', que 'App' instancia durante la partida, que estos son los encargados de responder las preguntas que se le van mostrando durante la partida. Luego participa 'JsonParser', que se encarga de la lectura y procesamiento del archivo de las preguntas, dentro de este objeto se utiliza a 'QuestionFactoryProvider', muy importante dentro de el sistema, la cual se encarga de instanciar al objeto que implementa a la interfaz 'QuestionFactory' determinado por los parámetros procesados del archivo. Este es el encargado de instanciar a la pregunta y su sistema lo modelaremos en el próximo diagrama. Por ultimo esta 'Question' el cual su modelo de clases lo dividimos en dos: el modelo de los objetos que interactúan en su instanciación y el modelo de los objetos que interactúan durante la partida con este objeto. El primer modelo es el siguiente:

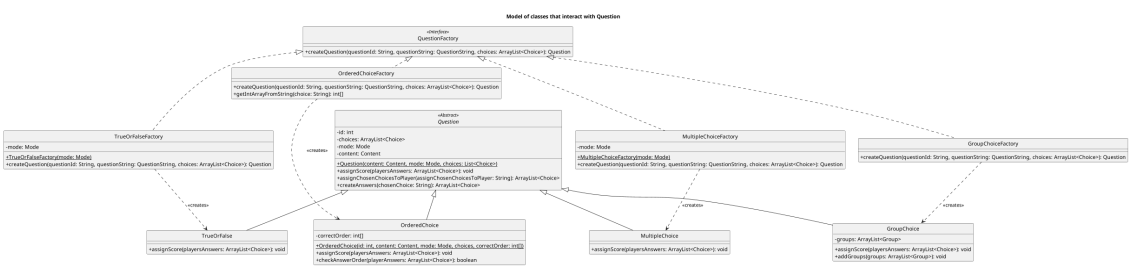

Figura 2: Diagrama de clases de los objetos que interactúan en la instanciación de 'Question'.

En el modelo anterior se puede observar a la clase abstracta 'Question' y sus clases herederas las cuales redefinen el mensaje 'assignScore' el cual se encarga de delegar la asignación de puntaje a un determinado jugador en los demás objetos que participan. También se puede observar a la interfaz 'QuestionFactory' la cual modelamos anteriormente, esta, dependiendo de cual sea la 'Factory' que la implemente, instanciara a la clase heredera de 'Question' correspondiente. Dentro de esta implementamos el patron de diseño 'Abstract Factory' ya que en este objeto se encapsula toda la lógica de la creación de preguntas. El segundo modelo representa a los objetos que interactúan con 'Question' durante la partida:

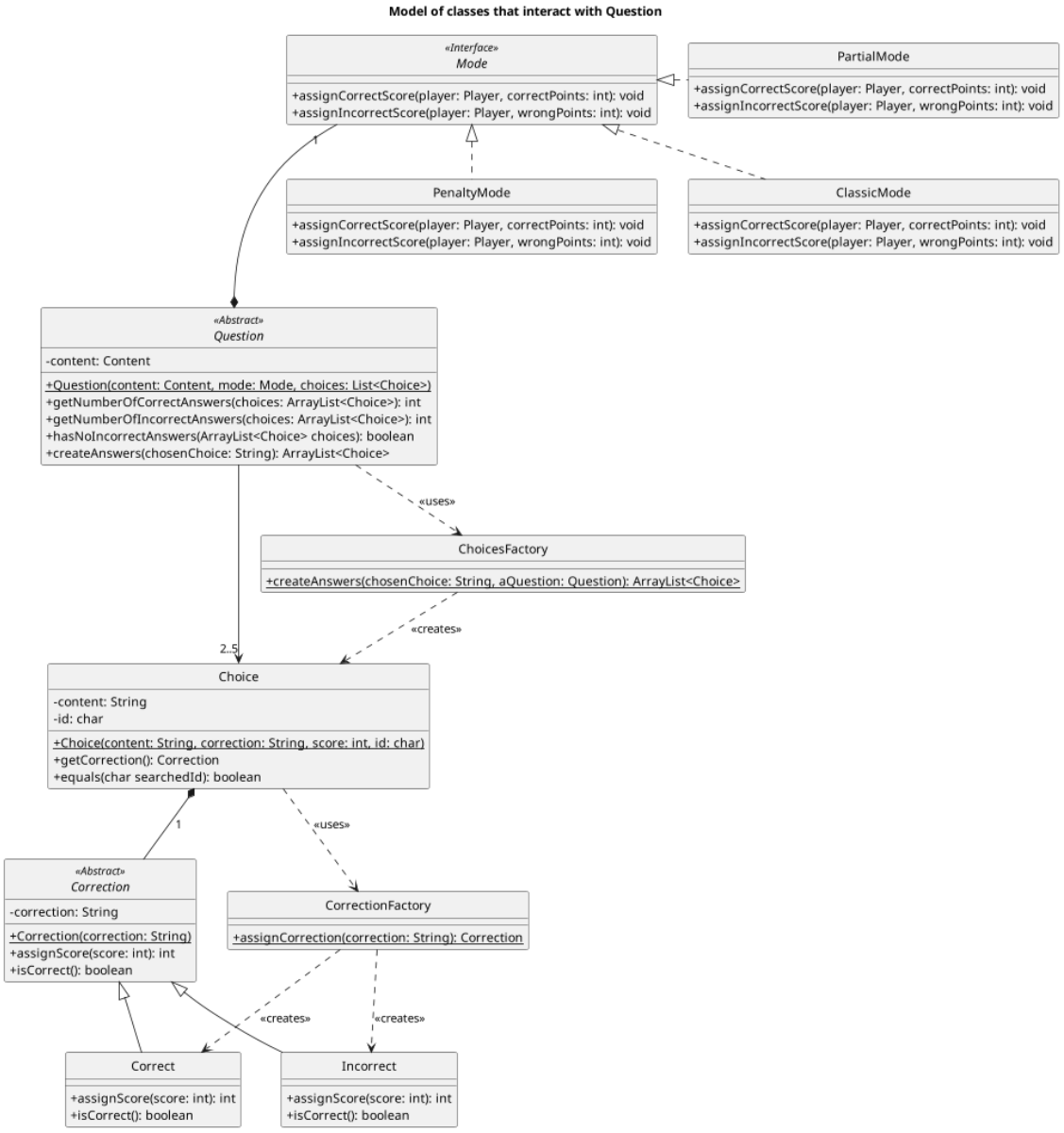

Figura 3: Diagrama de clases de los objetos que interactúan con 'Question'.

Una de las clases que se encarga de determinar de que manera se le asignara o restara el puntaje a 'Player' es la interfaz 'Mode', implementada por 'PartialMode', 'PenaltyMode' y 'ClassicMode' las cuales dependiendo de sus condiciones redefinen los mensajes de distinta manera. En

este modelo el objeto que se encarga de generar las respuestas de los jugadores es 'ChoicesFactory' que compara la cadena de texto ingresada por el usuario con las opciones que tiene almacenada 'Question' y dependiendo si su 'Correction' es 'Correct' o 'Incorrect' instanciara el 'Choice' correspondiente. Acá nuevamente aplicamos el patron de diseño 'Factory' ya que se encapsula en esta clase la creacion de elecciones correctas o incorrectas.

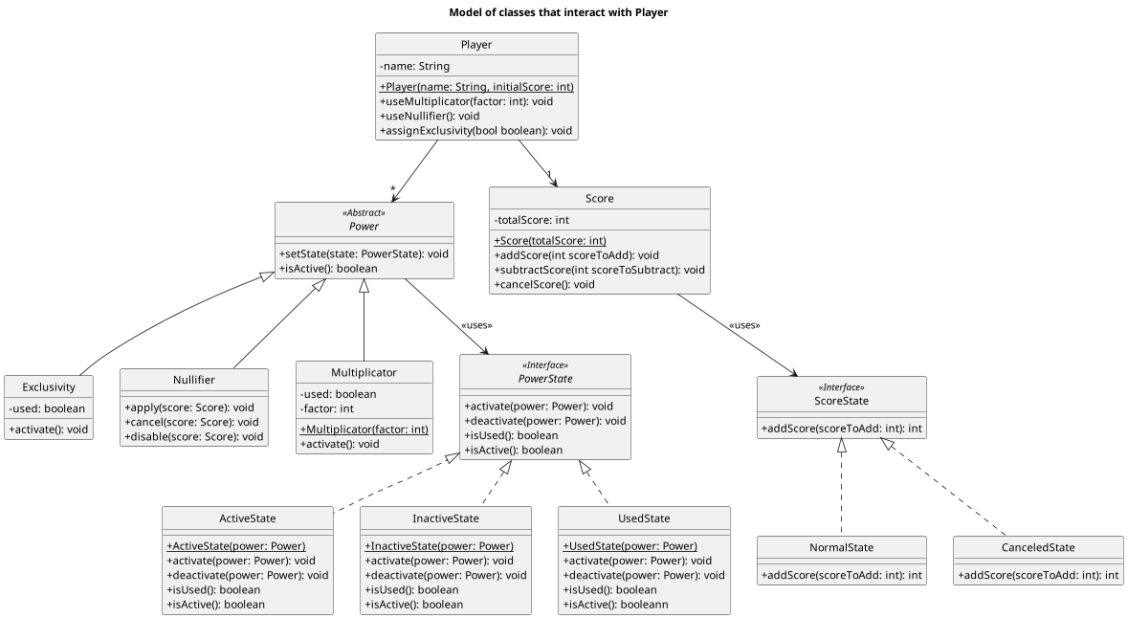

Figura 4: Diagrama de clases de los objetos que interactúan con 'Player'.

Este modelo se enfoca en el objeto 'Player' el cual se instancia con una preestablecida cantidad de diferentes subclases de 'Power', el cual puede utilizar durante determinados momentos de la partida para alterar el puntaje obtenido luego de responder una pregunta. En este caso implementamos el patron de diseño State donde 'Power' utiliza la interfaz 'PowerState' para indicar si cualquiera de los poderes se encuentra activo, inactivo o ya utilizado.

Por ultimo, 'Player' tiene un objeto 'Score' el cual se encarga de manejar el puntaje que el jugador a medida que va respondiendo las preguntas va aumentando o decrementando durante la partida. Dentro de este volvimos a implementar el patron de diseño 'State' ya que, dependiendo de si se le aplico o no el anulador de puntaje, 'Score' cambia su estado utilizando una interfaz llamada 'ScoreState' la cual se encarga de permitir la adición de puntaje ('NormalState') o no permitirla ('CanceledState').

5. Diagrama de Paquetes

Para mostrar la relación entre los distintos paquetes que se armaron a lo largo del desarrollo del proyecto se hizo un diagrama de paquetes que representa como se relacionan los paquetes principales usados en el modelo.

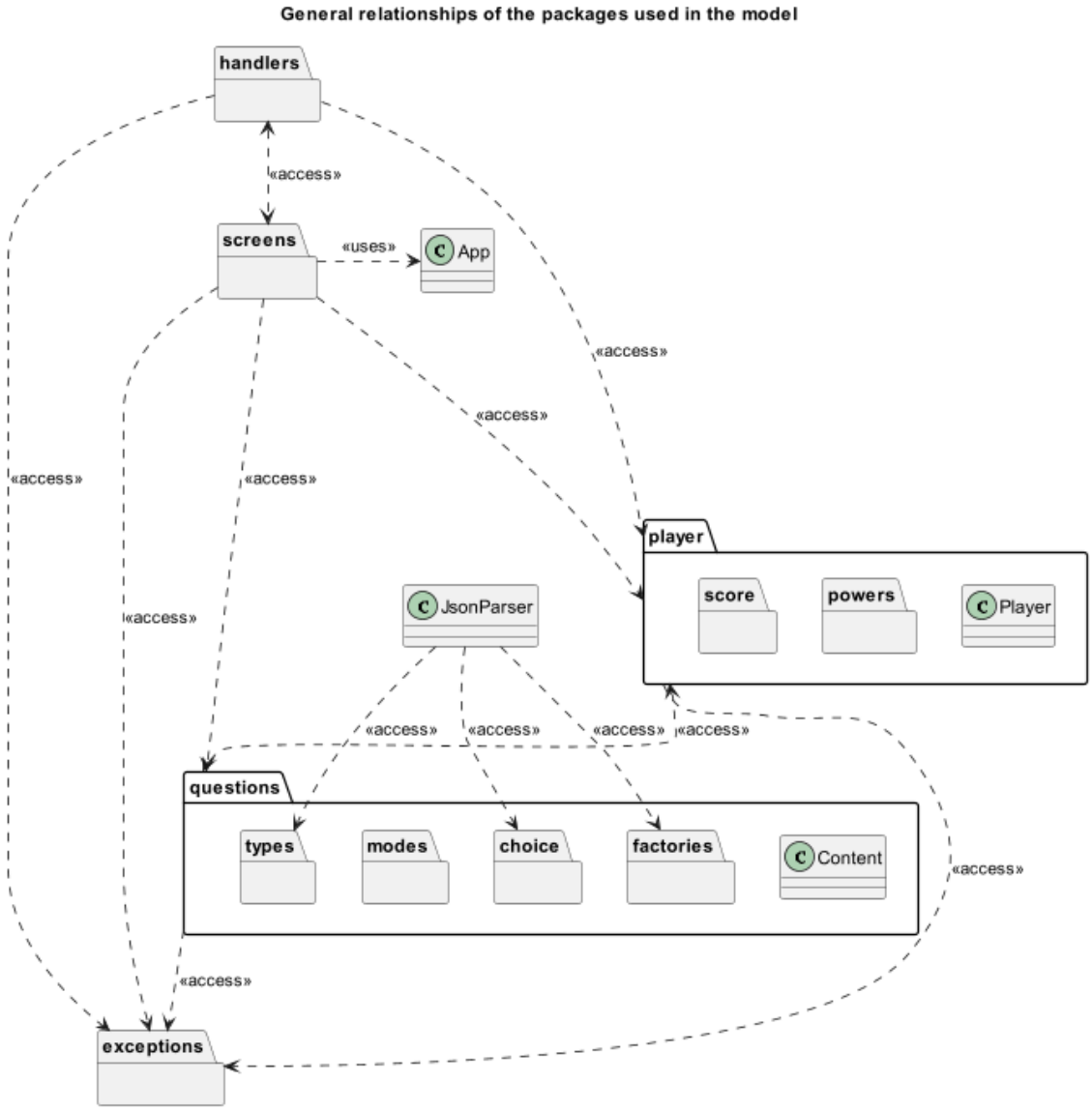

Figura 5: Diagrama de paquetes que muestra las relaciones de los paquetes principales del modelo.

6. Detalles de implementación

A continuación, detallaremos las estrategias empleadas para abordar y resolver los puntos más conflictivos del trabajo práctico.

6.1. Herencia vs Delegación

A estos dos pilares de la programación orientada a objetos los implementamos en distintas secciones de nuestro código. En el caso de Herencia, la implementamos cuando en las clases herederas encontramos varios métodos en común y era necesario que estas hereden el comportamiento de su clase "madre". Por ejemplo, para la familia de preguntas ('Question'), decidimos aplicar Herencia debido a nos dimos cuenta que podíamos generalizar el comportamiento de las diferentes preguntas, ahorrándonos duplicar el código. Otra razón por la que aplicamos este pilar es que estas subclases cumplían la relación 'es una' con respecto a la clase madre.

En cambio a delegación lo aplicamos en caso de que una clase deba derivar funcionalidad en otras clases. Uno de los ejemplos mas importantes de delegación en nuestra aplicación es el de la clase juego ('Game'), esta clase delega funcionalidad en los distintos objetos que componen el sistema. Desde elementos de la interfaz gráfica ('Panel', 'PlayersInputScreen', 'ConfirmButton', etc) hasta los elementos que forman parte de la partida ('Player', 'Question', etc), cada uno de estos elementos recibe comportamiento a partir de el objeto juego, que es el encargado de delegar responsabilidades a medida que se ejecuta la aplicación.

6.2. Principios de diseño

Uno de los principios de diseño que aplicamos en el trabajo practico fue el de Inversión de Dependencia, este establece que se debe depender de las abstracciones por sobre las implementaciones. En un principio habíamos pensado la lógica de las preguntas en el que cada par (tipo, pregunta) era un objeto, por ejemplo, verdadero o falso (pregunta) con penalidad (tipo) era un objeto 'TrueOrFalseWithPenalty' y así para cada par distinto. El problema que tuvimos fue en el caso de querer refactorizar o agregarle funcionalidad a las preguntas con penalidad, se debía modificar cada uno de estos objetos '...WithPenalty' generando código repetido, por lo que decidimos implementar este principio. Este problema lo solucionamos agregando una interfaz modo ('Mode') la cual es implementada y redefinida por tres objetos: clásico ('ClassicMode'), con penalidad ('PenaltyMode') y parcial ('PartialMode'). En este caso simplificamos el código, integrándolo y definiéndolo en una interfaz, de la cual depende cada tipo de pregunta ('TrueOrFalse', 'MultipleChoice', ..., etc).

6.3. Patrones de diseño

En el caso de los patrones de diseño, los fuimos implementando a medida que fuimos desarrollando y refactorizando varias secciones de la aplicación. Uno de los que mas utilizamos fue el **Patrón Factory** y **Abstract Factory**, estos los implementamos para encapsular la lógica de creacion de objetos dentro de una clase. A partir de la utilización de estos patrones pudimos resolver el conflicto de la creacion y armado de las preguntas procesadas del archivo. Para esta resolución creamos una Factory interfaz ('QuestionFactory') de la cual, para cada tipo de pregunta ('TrueOrFalse', 'MultipleChoice', ..., etc), una Factory de ese tipo de pregunta implementaba su comportamiento ('TrueOrFalseFactory', 'MultipleChoiceFactory', ..., etc), cada una de estas redefiniendo el método de creación de preguntas. Para establecer el modo de la pregunta (clásico, parcial o con penalidad) definimos una clase "proveedora"('QuestionFactoryProvider') de cada tipo de fabrica y dependiendo de la cadena de texto procesada del archivo se le aplica el modo correspondiente con el que después se inicializa a la pregunta.

```
public static QuestionFactory getFactory(String type) {
```

```
    return switch (type) {
        case "Verdadero Falso Simple" -> new TrueOrFalseFactory(new ClassicMode());
        case "Verdadero Falso Penalidad" -> new TrueOrFalseFactory(new PenaltyMode());
        case "Multiple Choice Simple" -> new MultipleChoiceFactory(new ClassicMode());
        case "Multiple Choice Puntaje Parcial" -> new MultipleChoiceFactory(new PartialMode());
        case "Multiple Choice Penalidad" -> new MultipleChoiceFactory(new PenaltyMode());
        case "Ordered choice", "Ordered Choice" -> new OrderedChoiceFactory();
        case "Group Choice" -> new GroupChoiceFactory();
        default -> throw new IllegalStateException();
    };
}
```

El patrón Factory también lo utilizamos en la creación de las opciones de la pregunta a partir del archivo procesado y, también, a partir de las respuestas del jugador durante la partida ('ChoiceFactory'). Nuevamente encapsulamos la lógica de creación de objetos, en este caso de tipo 'Choice', y dependiendo de si este fuese correcto o incorrecto, se instanciara a la clase determinada ('Correct' o 'Incorrect').

Otro de los patrones de diseño que integramos en nuestra aplicación fue el **Patrón State**, este permite a un objeto cambiar su comportamiento cuando su estado interno es alterado. Este patrón lo utilizamos para resolver la problemática que presentaba el manejo del sistema de poderes ('Nullifier', 'Multiplicator' y 'Exclusivity') que posee el jugador durante la partida. A medida que se desarrolla la partida el jugador tiene la posibilidad de activar y utilizar una cantidad preestablecida de poderes, para implementar esto decidimos que cada poder tenga un estado, este puede ser activo ('ActiveState'), inactivo ('InactiveState') o usado ('UsedState'), todos estos implementadores de la interfaz 'PowerState'. Estos diferentes estados son los que determinan y modifican el comportamiento de los poderes del jugador en la partida, cada uno de estos cumpliendo una función determinada. Si el poder tiene un estado activo, se deberá ejecutar (en el caso que se pueda) el poder seleccionado modificando el comportamiento del puntaje del jugador, si el poder tiene un estado inactivo no hará modificaciones ni se aplicara y si el poder esta usado no se podrá volver a utilizar ese poder.

Además, este patrón lo volvimos a aplicar para manejar el estado del puntaje del jugador ('Score'). Este objeto puede tener un estado normal ('NormalState') o un estado cancelado ('CanceledState'), estos dos implementadores de la interfaz 'ScoreState'. Estos estados definen el comportamiento del puntaje del jugador, por ejemplo, en caso de que acierte la respuesta si el puntaje del jugador se encontrara en estado cancelado, este no sumara puntos, en cambio, si el puntaje del jugador tuviese un estado normal su comportamiento no cambiaría y si se sumarían los puntos.

7. Excepciones

InvalidAnswerFormatException Se levantara la excepción en caso de que el jugador ingrese una respuesta en el formato inadecuado. Por ejemplo, si en una pregunta de múltiple choice el jugador ingresa "1 2 3,2 2"se le mostrara un mensaje con el formato valido para la respuesta.

UsedPowerException Esta excepción se arrojara luego de que un jugador quiera usar un poder después de ya haberlo utilizado, es decir que si el poder tiene un estado usado ('UsedState'), se le mostrara un mensaje por pantalla al jugador indicándole que ya no puede usar ese poder.

8. Diagramas de secuencia

En esta sección vamos a modelar algunas secuencias interesantes que implementamos en nuestro código.

La primer secuencia que decidimos modelar es la inicialización de la aplicación y como se comporta esta con el usuario que la utiliza. Por lo que para este caso decidimos dividir la secuencia en dos diagramas, en esta primera secuencia, como mencionamos anteriormente, modelamos a los objetos que participan al momento del armado y ejecución de la interfaz gráfica de nuestra aplicación. Por lo que nos quedo el siguiente modelo:

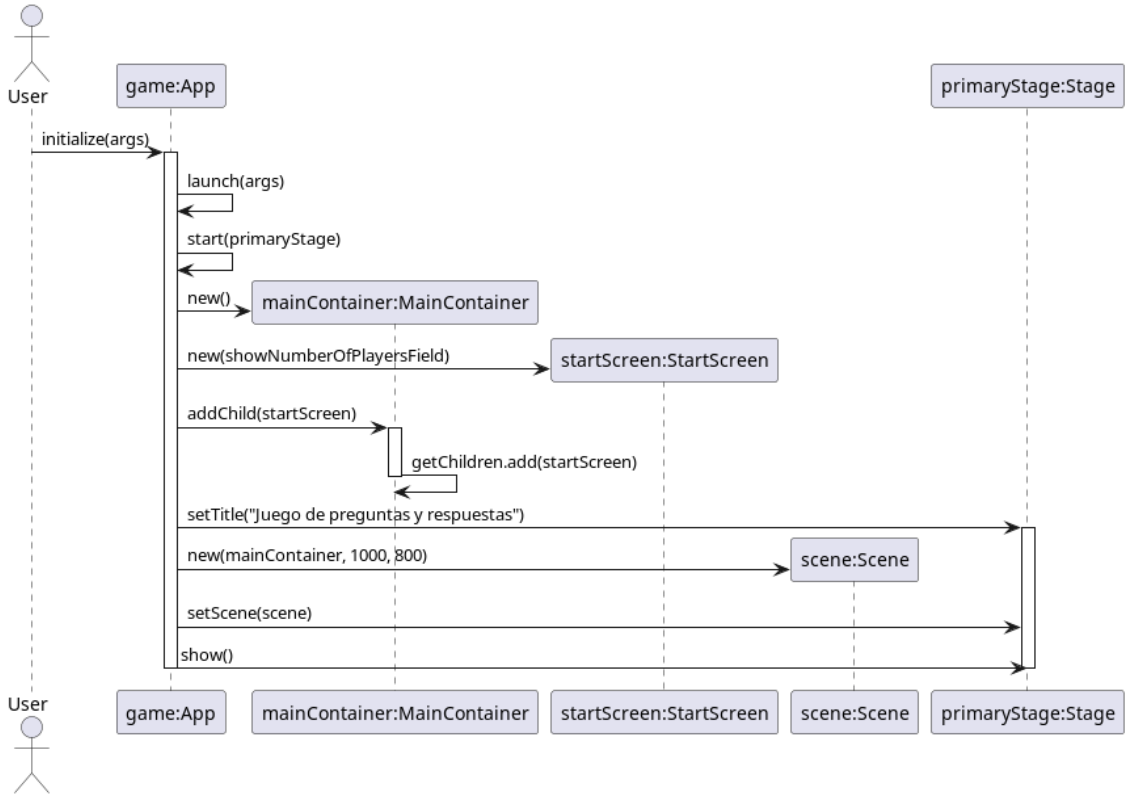

Figura 6: Diagrama de secuencia de la interacción de objetos cuando se inicializa la aplicación.

En este caso se puede observar la secuencia de mensajes que se desencadenan cuando el usuario ejecuta la aplicación. También se puede visualizar como se va seteando la escena de la etapa inicial de la aplicación y como se le van agregando distintos elementos a esta.

Luego en este segundo diagrama modelamos la continuidad de la secuencia anterior:

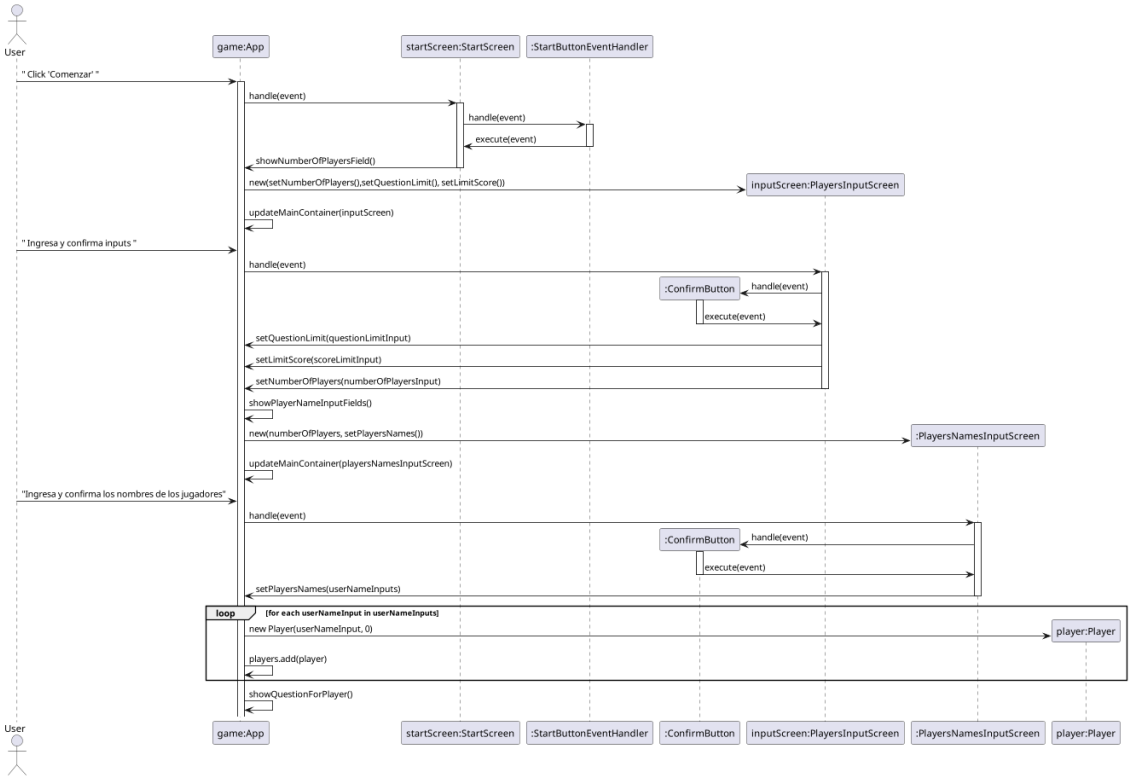

Figura 7: Diagrama de secuencia donde el usuario interactúa con la aplicación

En el diagrama anterior se puede apreciar como el usuario va interactuando con la aplicación y como esta delega en los objetos que componen su interfaz distintos comportamientos. La aplicación a medida que se desarrolla la partida va manejando los eventos que se crean a partir del usuario, en este caso los eventos son tres: en el primero el usuario clickea el boton para comenzar la partida, en el segundo el usuario ingresa la cantidad de jugadores y limites que tendrá la partida y en el tercero el usuario ingresa los nombres que tendrán los jugadores que participaran de la partida.

En la próxima secuencia modelaremos el caso de uso 5 y 6 el cual una pregunta 'TrueOrFalse' con el modo 'PenaltyMode' recibe una lista de respuestas y les asigna su respectivo puntaje. Lo decidimos dividir en 3 diagramas distintos: el primer diagrama contiene la instanciación de los objetos, el segundo modela la asignación de puntaje en el caso de que el jugador haya contestado correctamente y el tercero el caso en el que el jugador haya contestado incorrectamente.

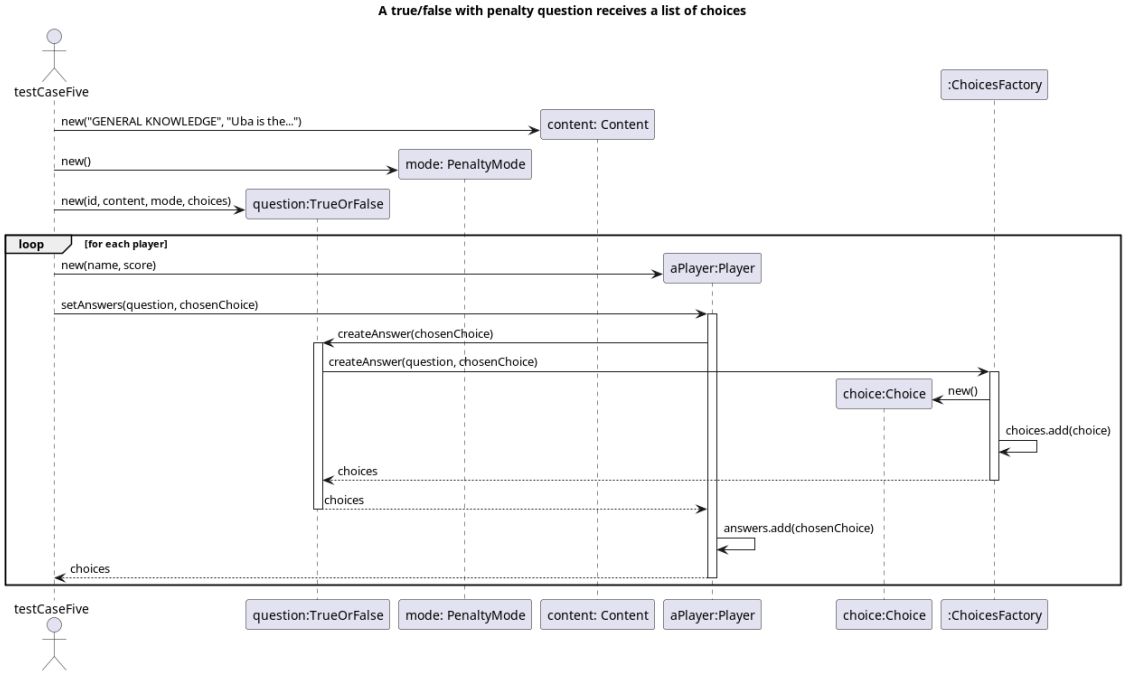

Figura 8: Diagrama de secuencia donde 'Question' recibe las opciones elegidas por 'Player'.

En esta primera parte se puede visualizar la instanciación de los objetos del sistema, con sus determinados parámetros. También se puede observar como 'Question' recibe la respuesta del jugador y delega el comportamiento de crear las elecciones del jugador a la clase 'ChoiceFactory' que devuelve una lista con objetos de tipo 'Choice'.

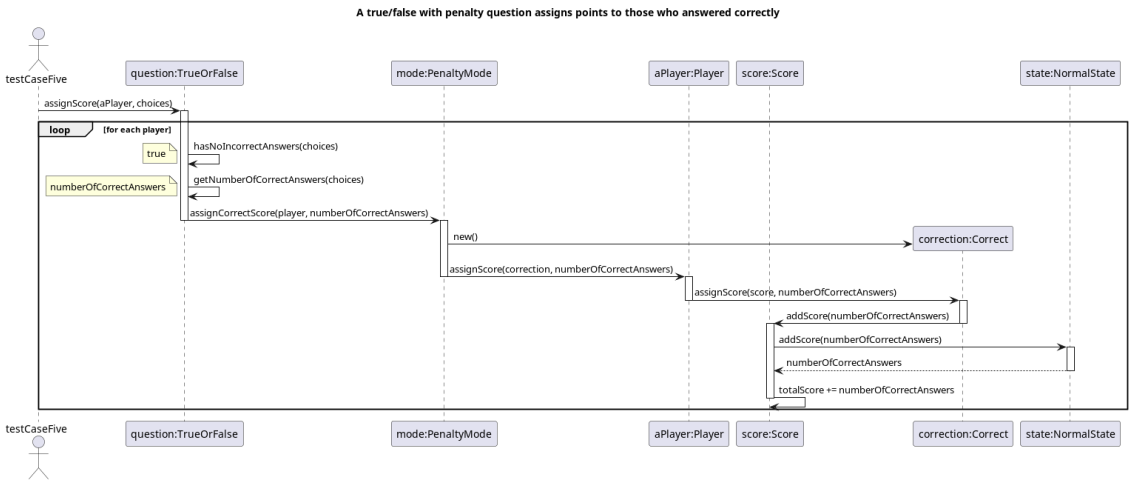

Figura 9: Diagrama de secuencia donde 'Question' asigna el puntaje a 'Player' que contesto correctamente.

Este segundo diagrama demuestra como funciona el sistema de asignación de puntaje en caso de que la o las respuestas del jugador sean correctas. Se puede observar como pregunta ('TrueOrFalse') delega el comportamiento de la asignación de puntaje en los diferentes objetos del sistema.

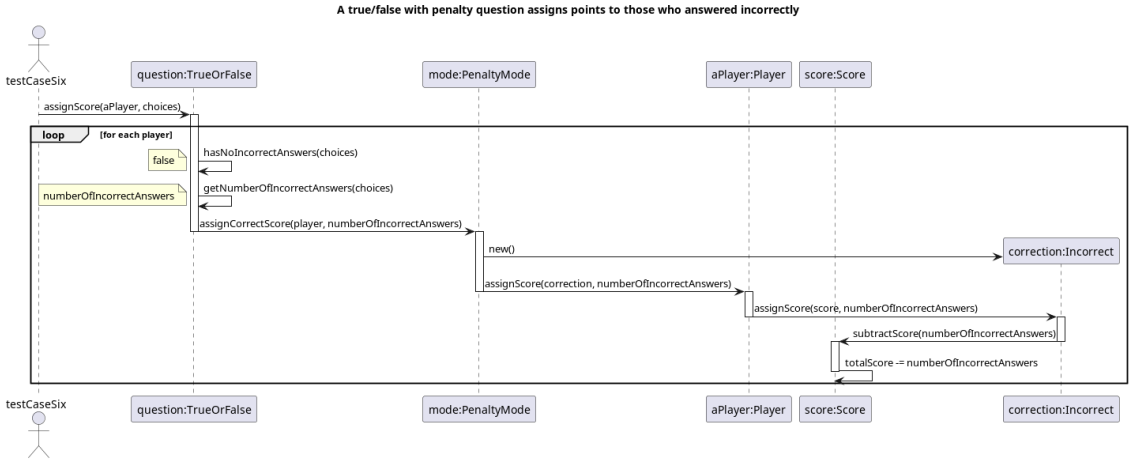

Figura 10: Diagrama de secuencia donde 'Question' asigna el puntaje a 'Player' que contesto incorrectamente.

Este ultimo diagrama demuestra como funciona el sistema de asignación de puntaje en caso de que la o las respuestas del jugador sean incorrectas. En este caso, la pregunta al tener el modo de penalidad ('PenaltyMode') restara la cantidad de respuestas incorrectas que haya tenido el jugador.

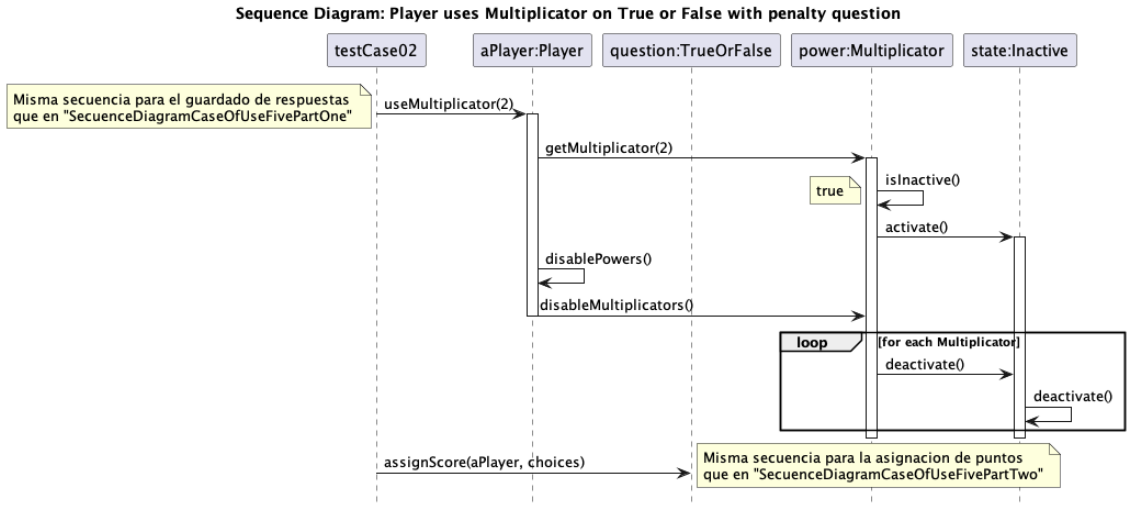

Figura 11: Diagrama de secuencia donde el usuario usa el poder de Multiplicador al responder una pregunta

En este diagrama se puede observar como un jugador decide activar el poder del multiplicador. Se ve detalladamente el uso del patrón de diseño "State"para poder desactivar y activar el poder cuando sea necesario