FACULTAD DE INGENIERIA
Universidad de Buenos Aires

Trabajo Práctico 2 - AlgoHoot

[7507/9502] Algoritmos y Programación III
Curso 2 (Suarez)
Primer cuatrimestre de 2024

Alumno	Numero de Padron	Email
Lucas Ezequiel Araujo	109867	learaujo@fi.uba.ar
Ignacio Latorre	101305	ilatorre@fi.uba.ar
Nicolas Cardone	111148	ncardone@fi.uba.ar
Salvador Perez	110198	sperezm@fi.uba.ar

Índice

1. Introducción **2**

2. Supuestos **2**

3. Modelo de dominio **2**

4. Diagramas de clase **3**

5. Diagramas de secuencia **6**

6. Diagrama de Paquetes **8**

7. Detalles de implementación **8**
7.1. Herencia vs Delegacion 8
7.2. Uso de Patron Decorator para Bonificadores 9
7.3. Uso del Patron de Diseño MVC (Modelo-Vista-Controlador) 9
7.4. Uso del Patron Observer) 9
7.5. Principios de diseño que se aplicaron 9
7.5.1. Tell don't Ask 9
7.5.2. Separacion de intereses 9
7.5.3. Principio de Responsabilidad Unica 9
7.5.4. Principio de Abierto-Cerrado 10
7.5.5. Principio de Sustitucion de Liskov 10
7.5.6. Principio de Segregacion de Interfaces 10
7.5.7. Principio de Inversion de la Dependencia 10

8. Excepciones **10**

1. Introducción

El presente informe reune la documentacion de la solucion del segundo trabajo práctico de la materia Algoritmos y Programación III que consiste en desarrollar una aplicación de un juego de preguntas y respuestas llamado Algohoot en Java utilizando los conceptos del paradigma de la orientación a objetos vistos en el curso y siguiendo los principios de diseño SOLID.

2. Supuestos

- El juego debe comenzar con 2 o mas jugadores
- Anulador vence siempre por sobre otros bonificadores

3. Modelo de dominio

El objeto principal del modelo es el Juego, que se encarga de administrar las preguntas y los jugadores. Este objeto centraliza la lógica del juego, coordinando las diferentes interacciones entre los componentes.

El objeto CreadorDePreguntas se encarga de leer el archivo y transformar los objetos JSON en objetos Pregunta.

Cada objeto Pregunta contiene el enunciado, un Tipo de Pregunta y una Penalidad. El TipoDePregunta es una interfaz que es implementada por las subclases VerdaderoFalso, Multiple Choice, Ordered Choice, Group Choice y se encarga de determinar si la Respuesta enviada por el usuario es correcta o no.

La Penalidad es la clase que se encarga de asignar los puntajes correspondientes a la respuesta. Esta puede ser SinPenalidad (Simple), Con Penalidad o Parcial. La Pregunta delega la correccion de la pregunta en el enunciado, y luego delega la asignacion de puntaje a la Penalidad.

La Respuesta del jugador contiene una referencia al jugador que responde y las opciones que elige. Luego de que TipoDePregunta determina si la Respuesta es correcta o no, se crea una clase RespuestaCorregida que recibe la penalidad y crea una RespuestaPuntuada que le asigna el puntaje al jugador

Las Opciones encapsulan los valores que corresponden a las respuestas.

Los Bonificadores, son objetos pueden ser activados por los jugadores durante las rondas. Estos bonificadores se integran con las preguntas al finalizar la ronda utilizando el patrón Decorator. Las clases MultiplicadorDecorador, ExclusividadDecorador y AnuladorDecorador contienen la logica para que estos funcionen, y se unen a la clase BonificadorConcreto.

4. Diagramas de clase

Los diagramas de clases:

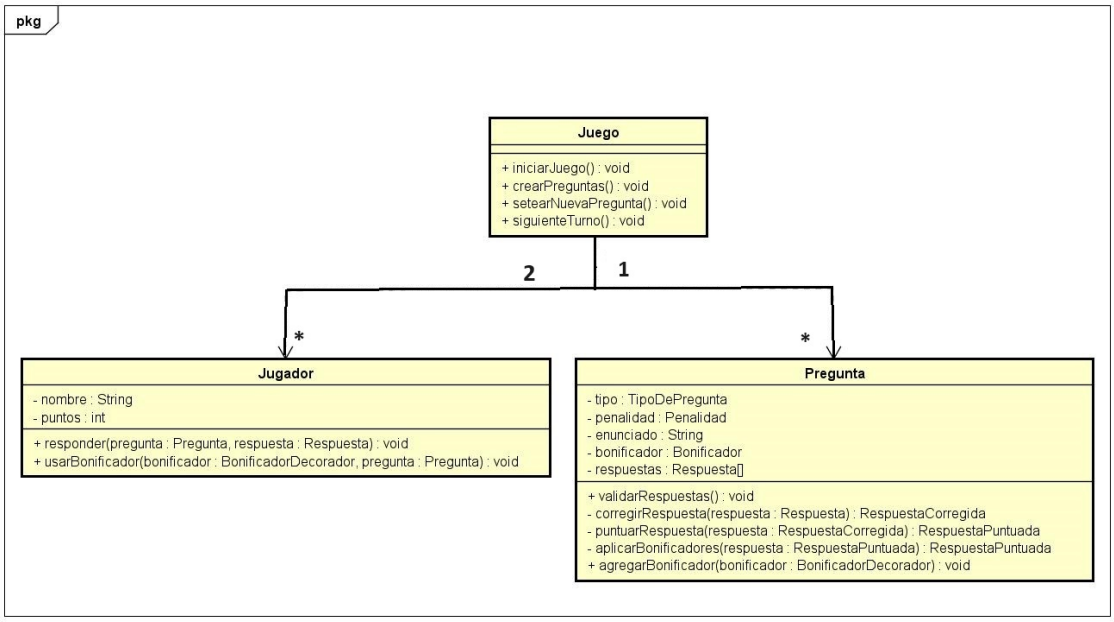

Figura 1: Diagrama de clases que representa la logica central del modelo

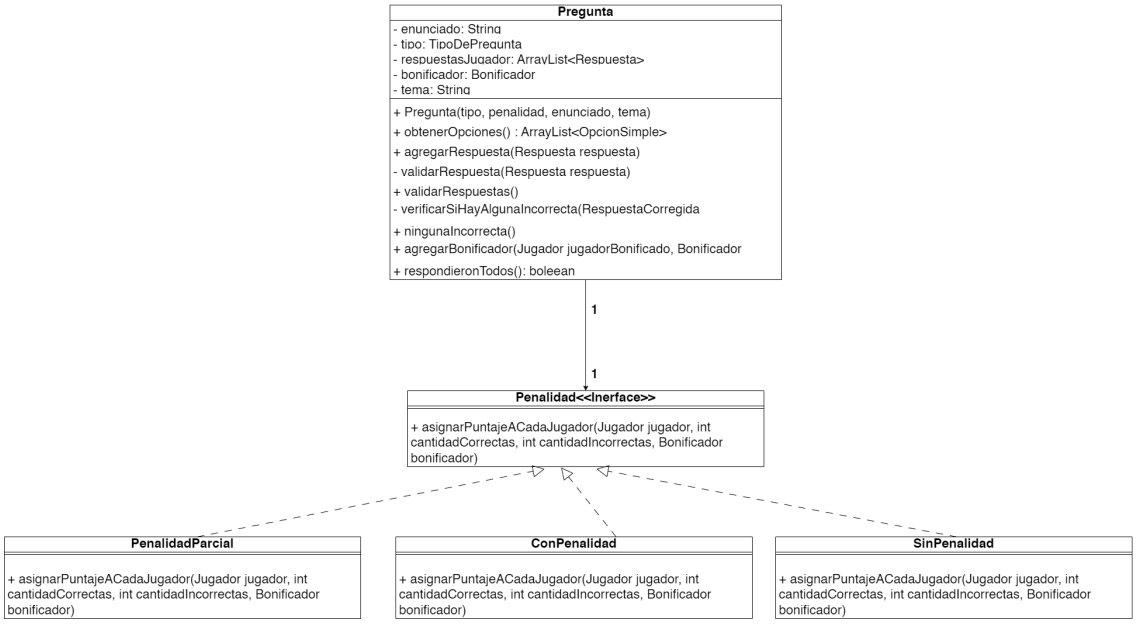

Figura 2: En este diagrama se muestra la asociacion de pregunta con penalidad

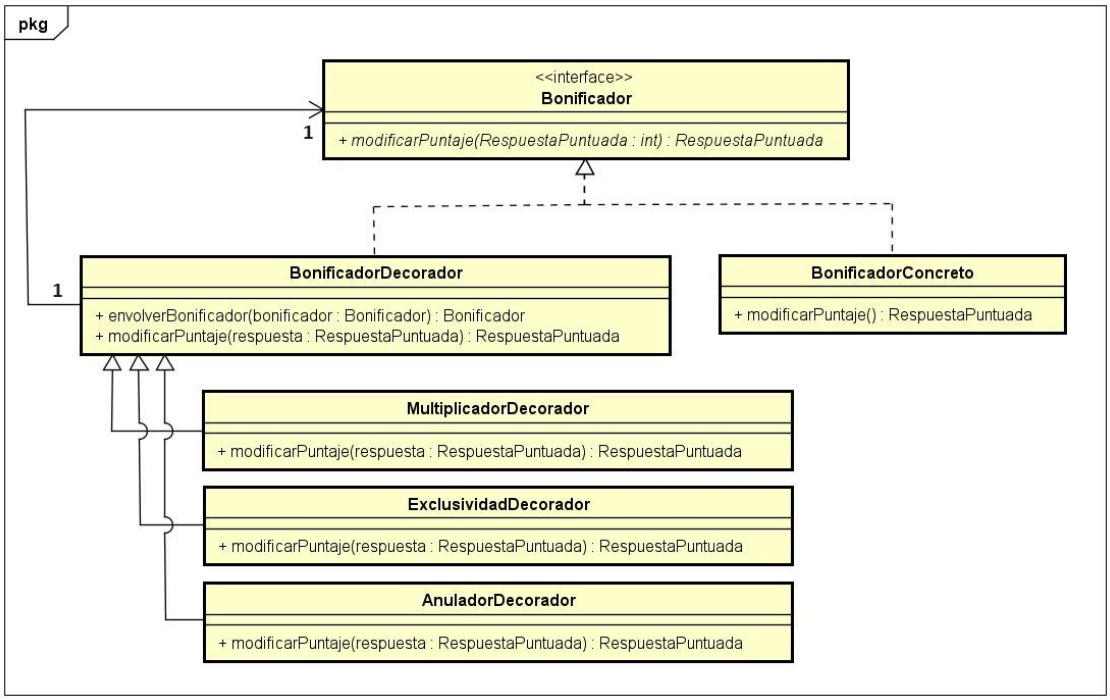

Figura 3: Los bonificadores tienen la estructura de un decorador. El bonificador Concreto implementa una interfaz modificarPuntaje. El DecoradorBase tiene una coleccion de Bonificadores que se iran guardando recursivamente en el. Los Decoradores Concretos Heredan los metodos de DecoradorBase para sobrecargarlos a segun cada decorador. El bonificadorDecorador tiene una relacion 1 a 1 con Bonificador porque el Jugador solamente puede usar un solo bonificador por cada turno

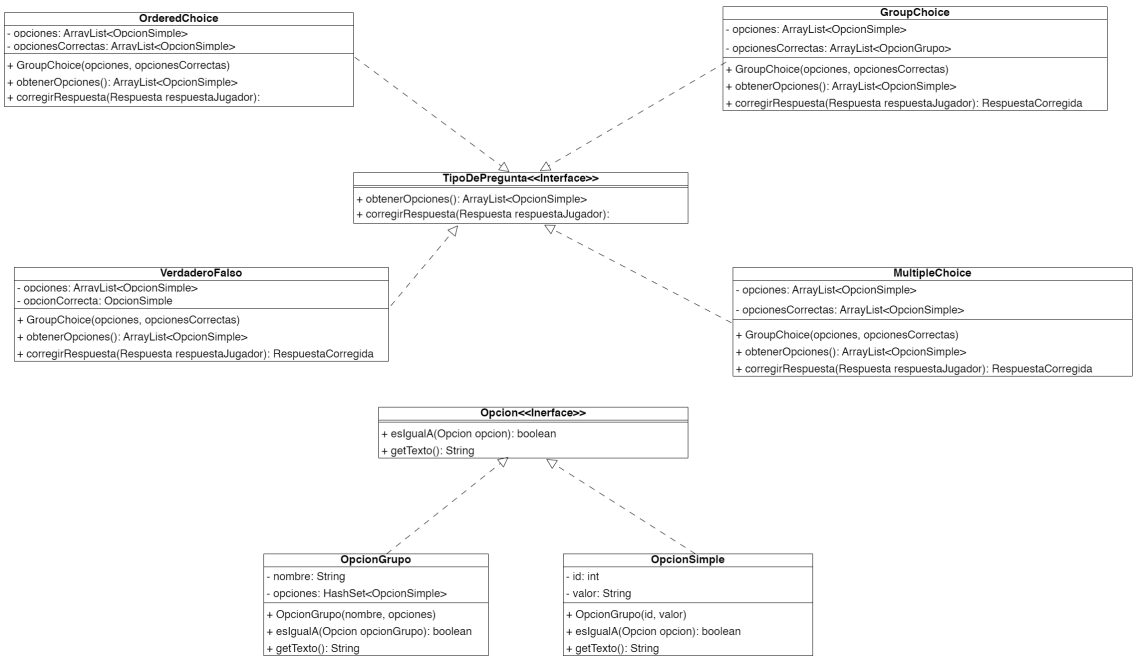

Figura 4: El diagrama muestra difrentes tipos de preguntas que utilizan una interfaz en comun para lograr el polimorfismo.
Mas abajo tambien podemos notar la interfaz Opcion que la usan varios de los objetos como Respuesta, Pregunta, Los tipos de preguntas. Todos aquellos objetos se comunican a traves de la interfaz de Opcion

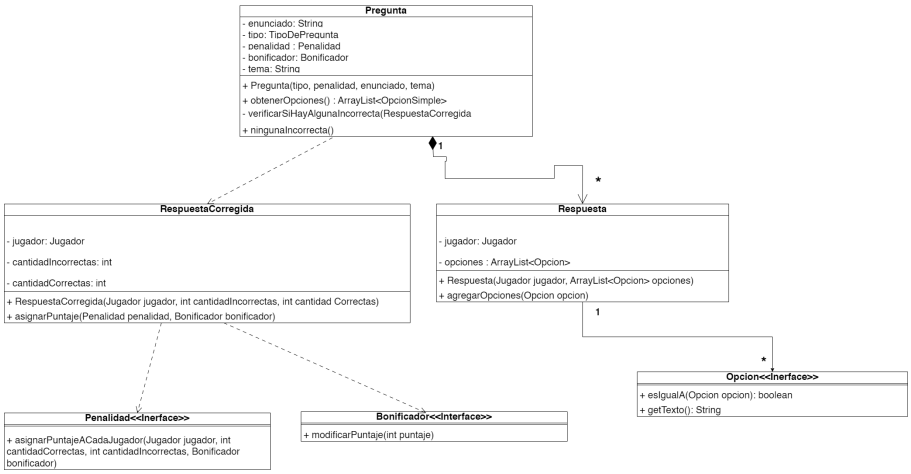

Figura 5: Diagrama de clases que muestran como Pregunta se relaciona con Respuesta y RespuestaCorregida. Pregunta tiene un composicion de Respuesta ya que si no existe esa Pregunta tampoco existira Respuesta
Como se ve Pregunta usa una RespuestaCorregida para delegarle la responsabilidad de asignar el puntaje a RespuestaCorregida que a traves de las interfaces Penalidad, Bonificador se conecta a dichos objetos delegandole la responsabilidad de asignar el puntaje

5. Diagramas de secuencia

En los diagramas de secuencia del proyecto, nos enfocamos en representar interacciones simples pero significativas entre los objetos clave del sistema.

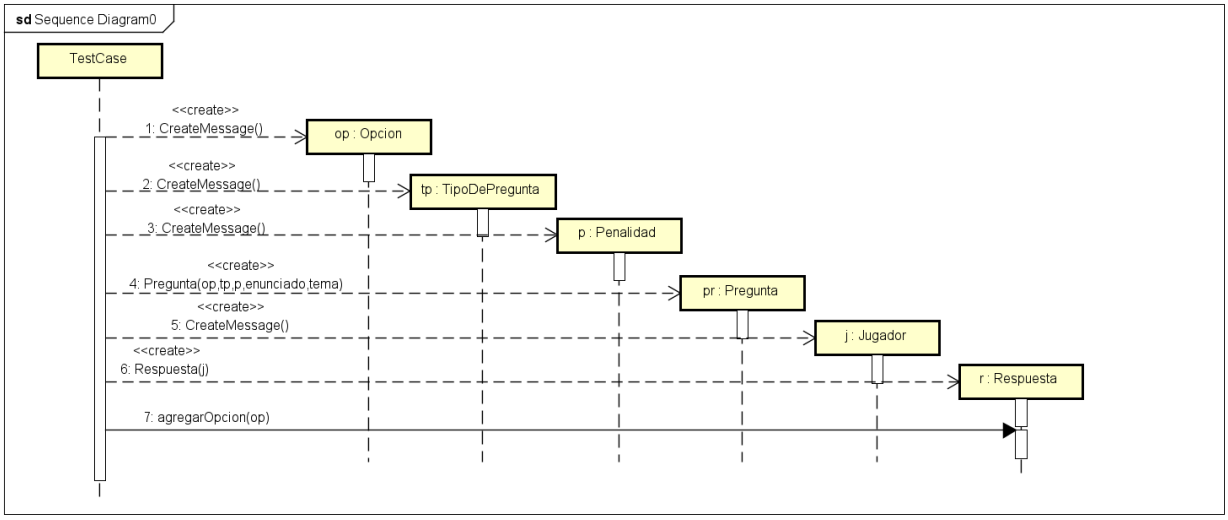

Figura 6: Diagrama de secuencia que nos muestra como se inicializan los objetos

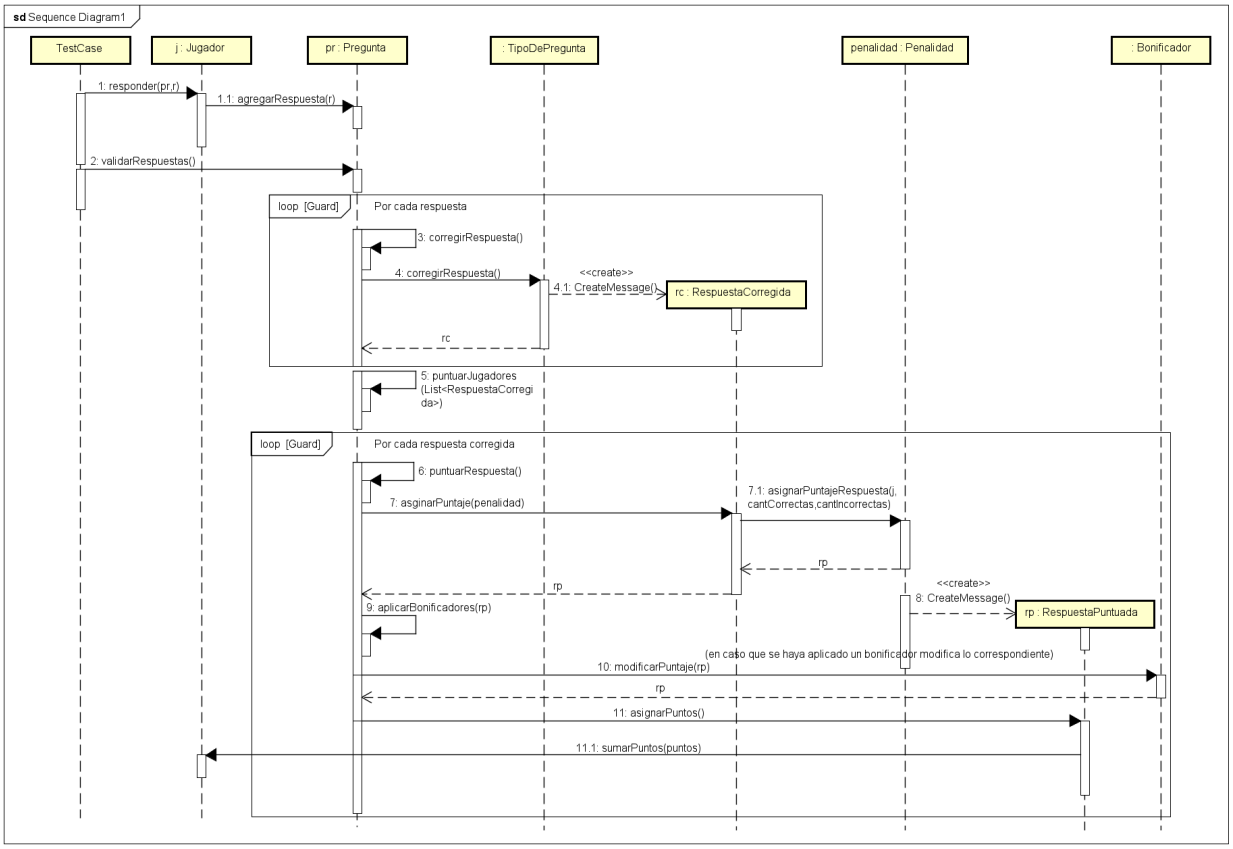

Figura 7: Continuacion del diagrama de secuencia anterior, mostrando como se suman los puntos a un jugador luego de responder la pregunta

6. Diagrama de Paquetes

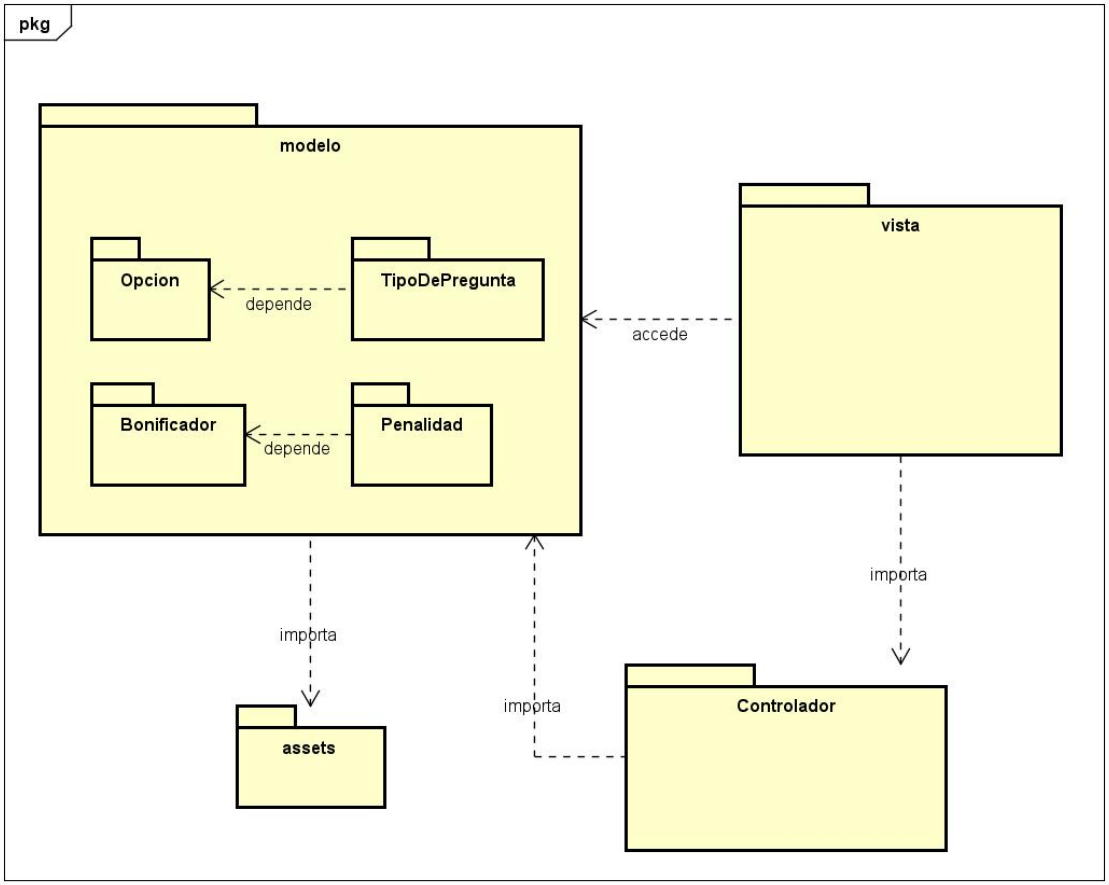

Figura 8: En este diagrama de paquetes se muestra la estructura del proyecto, el cual esta dividido en tres paquetes principales: modelo, vista y controlador. El objetivo de este diagrama es mostrar el bajo acoplamiento que se logra aplicando el patron MVC logrando que el modelo sea totalmente independiente de la vista que se quiera implementar para este

7. Detalles de implementación

7.1. Herencia vs Delegacion

Herencia: La herencia se utiliza para estructurar clases que comparten características y comportamientos comunes, lo que promueve la reutilización de código. En nuestro caso lo usamos para los decoradores que heredan el comportamiento de la clase Decoradora y la caracteristica en comun de wrapped que es donde se iban agregando los Bonificadores recursivamente.

Delegacion: La delegacion ofrece desacoplamiento al permitir que una clase transfiera responsabilidades a otra clase especializada, lo que mejora la cohesion y reduce las dependencias directas. La delegacion la usamos todo el tiempo por ejemplo la clase Pregunta delega en las clases que implementan TipoDePregunta la validacion de las respuestas y en las clases que implementan Penalidad, delega la determinacion del puntaje correspondiente a la respuesta del jugador que esta tambien delega responsabilidades a los bonificadores. Esto sirve para evitar la explosion de clases y para quitarle la responsabilidad excesiva a una clase cuando estamos delegamos estamos apo-

yandonos en la Abstraccion y Encapsulamiento que son dos de los pilares de POO y si delegamos correctamente cumplimos con el Principio de Responsabilidad Unica.

7.2. Uso de Patron Decorator para Bonificadores

Para gestionar los bonificadores, implementamos el patrón Decorator. Este patrón nos permitió añadir responsabilidades adicionales al objeto Pregunta de manera dinamica. Cuando un jugador utiliza un bonificador decorador, este se resta de su arreglo de bonificadores y envuelve el bonificador concreto de la pregunta. Al finalizar la ronda, el bonificador decorado modifica el puntaje de la Respuesta Puntuada llamando a los bonificadores internos. La Respuesta Puntuada es la encargada de asignar los puntajes al jugador.

7.3. Uso del Patron de Diseño MVC (Modelo-Vista-Controlador)

En nuestro juego de preguntas y respuestas, hemos implementado el patrón MVC para separar claramente la lógica de negocio de la interfaz de usuario. El Juego gestiona los turnos, preguntas y jugadores, manteniendo los datos y la logica de negocio. La Vista se encarga de presentar la interfaz de usuario, mostrando preguntas, puntajes y bonificadores. El Controlador maneja la interaccion del usuario, procesando las entradas y actualizando el modelo y la vista en consecuencia. Esta separación facilita el mantenimiento, la escalabilidad y la posibilidad de cambiar la interfaz de usuario sin afectar la logica del juego.

7.4. Uso del Patron Observer)

En nuestro caso utilizamos el patron observer para determinar cuando terminaba el juego. El Observador del juego (JuegoObserver) se crea al comienzo cuando se inicia la partida y se lo agrega como un Observador de Juego ya que JuegoObserver implementa la interfaz Observer y el Objeto Juego implementa una interfaz Observable.
JuegoObserver esta interesado en recibir una notificacion cuando termino el juego. El juego termina cuando un jugador tiene X puntos o cuando se acaban las rondas que se determinaron al inicio de la partida. Entonces cuando se le manda una notificacion a JuegoObserver este va a cambiar la escena que muestra quien fue el ganador.

7.5. Principios de diseño que se aplicaron

7.5.1. Tell don't Ask

En nuestro diseño, no hemos consultado el estado de otros objetos en cambio le deciamos al objeto que hacer delegandole toda la responsabilidad. Por ejemplo, cuando una Pregunta necesita calcular el puntaje de una respuesta, no pregunta directamente por el estado de la RespuestaCorregida para despues asignarle el puntaje al Jugador. En cambio, invoca a RespuestaCorregida para asignar un puntaje y le pasa por parametro la infromacion necesaria para lograr esta tarea. A su vez, RespuestaCorregida delega esta responsabilidad a Penalidad, quien finalmente con la informacion necesaria asigna el puntaje al Jugador.

7.5.2. Separacion de intereses

Aplicando el Modelo-Vista-Controlador (MVC) desacoplamos la logica del modelo de la vista, evitando dependencias entre si, disponibilizando solamente las partes del modelo necesarias para cualquier vista que se quiera desarrollar.

7.5.3. Principio de Responsabilidad Unica

Cada componente del juego (como Juego, Pregunta, Jugador, Bonificador, Penalidad, etc) tiene responsabilidades bien definidas y específicas. Por ejemplo, Juego gestiona el flujo general del juego,

Pregunta es encargado de administrar las preguntas y asignarle el puntaje correspondiente a cada jugador delegando responsabilidades en varios objetos de los que depende. Gracias a cumplir con el principio de Inversion de la Dependencia tambien nos ayuda indirectamente a cumplir la Responsabilidad Unica al desacoplar componentes y promover la modularidad.

7.5.4. Principio de Abierto-Cerrado

Hemos desarrollado las clases y jerarquias de forma tal que sea facil expandir la aplicacion, donde solo hay que agregar una clase que implemente un nuevo tipo de pregunta, penalidad o bonificacion para que esta funcione y no modificar otras.

7.5.5. Principio de Sustitucion de Liskov

Se respeta el Principio de Sustitucion ya que la clase Pregunta por ejemplo puede reemplazar un Tipo de Pregunta o Penalidad por otro objeto que implemente la misma interfaz y el programa va a funcionar correctamente.

7.5.6. Principio de Segregacion de Interfaces

Hemos seguido el principio de segregación de interfaces al diseñar varias interfaces más pequeñas y específicas. No tenemos clases implementando metodos que no les competen.

7.5.7. Principio de Inversion de la Dependencia

Este principio lo cumplimos en varias ocasiones al hacer que los objetos dependan interfaces, es decir modulos de alto nivel en vez de que depender de objetos Concretos. Por ejemplo, Pregunta depende de los siguientes modulos de alto nivel: Bonificador, Penalidad, TipoDePregunta.
Jugador depende de la interfaz BonificadorDecorador.
RespuestaCorregida depende de las interfaces Bonificador y Penalidad
La Respuesta depende de la interfaz Opcion

8. Excepciones

Decidimos no utilizar excepciones y limitar al usuario desde la interfaz para evitar que se llegue a un error en la aplicacion.