DP1 2020-2021

Documento de Diseño del Sistema

Proyecto parchís y oca

https://github.com/gii-is-DP1/dp1-2021-2022-g1-05.git

Miembros:

- Juan José Casamitjana
- Mario Espinosa Rodríguez
- Manuel Padilla Tabuenca
- Javier Terroba Orozco

Tutor: Irene Bedilia Estrada

GRUPO G1-L5

Versión 1

05/01/2022

Historial de versiones

Fecha	Versión	Descripción de los cambios	Sprint
07/12/21	V1	● Creación del documento ● Introducción, portada y formateo	3
05/01/22	V1	● Diagrama de capas ● Decisiones de diseño ● Patrones de diseño y arquitectónicos aplicados	3

Contenidos

Introducción

El proyecto consiste en la organización, documentación y desarrollo de un sistema que ofrezca la posibilidad de jugar a parchís y oca, el principal objetivo es conseguir una aplicación funcional, en la que jugadores puedan acceder y crear partidas, jugar con otras personas y elegir el juego con el que interactuar.

El número de personas mínimo es dos, ya que los dos juegos consisten en ganar contra otros jugadores, y el máximo 4, siendo en el parchís el número de colores disponibles, y en la oca, el número de fichas disponibles.

En la oca, el tiempo promedio de partida es de 30 minutos, cada jugador tira los dados en su turno, y el número conseguido es el número de casillas que debe avanzar, cada casilla tiene algunas bonificaciones o penalizaciones, como por ejemplo avanzar 3 casillas o perder 3 turnos.

Por otro lado, el parchís consiste de ‘’dos fases’’, primero, cada jugador comienza en su zona inicial, y hasta que no consigue un 5 al tirar los dados no comienza su partida. Una vez que lo consigue, el juego consiste en llegar a la casilla final de cada jugador, recorriendo todo el tablero y debiendo caer exactamente en esta. Sin embargo, los otros jugadores podrán bloquear el avance del jugador, o incluso eliminar sus fichas.

En los dos juegos la partida termina cuando el penúltimo jugador llega al final, decidiendo así el último lugar.

En cuanto a las funcionalidades principales que ofrece la aplicación, podemos destacar la posibilidad de creación y unión a partidas tanto de parchís como de oca mediante códigos de 6 dígitos o letras generados aleatoriamente. Además, un sistema de estadísticas y logros para que los jugadores puedan disfrutar de progresión. Por último, hemos implementado toda la gestión de usuarios, partidas, logros, etc… por parte de los administradores.

Diagrama(s) UML:

Diagrama de Dominio/Diseño [No realizado aún]

En esta sección debe proporcionar un diagrama UML de clases que describa el modelo de dominio, recuerda que debe estar basado en el diagrama conceptual del documento de análisis de requisitos del sistema pero que debe:

- *Especificar la direccionalidad de las relaciones (a no ser que sean bidireccionales)*
- *Especificar la cardinalidad de las relaciones*
- *Especificar el tipo de los atributos*
- *Especificar las restricciones simples aplicadas a cada atributo de cada clase de domino*
- *Incluir las clases específicas de la tecnología usada, como por ejemplo BaseEntity, NamedEntity, etc.*
- *Incluir los validadores específicos creados para las distintas clases de dominio (indicando en su caso una relación de uso con el estereotipo <<validates>>.*

Un ejemplo de diagrama para los ejercicios planteados en los boletines de laboratorio sería (hemos omitido las generalizaciones hacia BaseEntity para simplificar el diagrama):

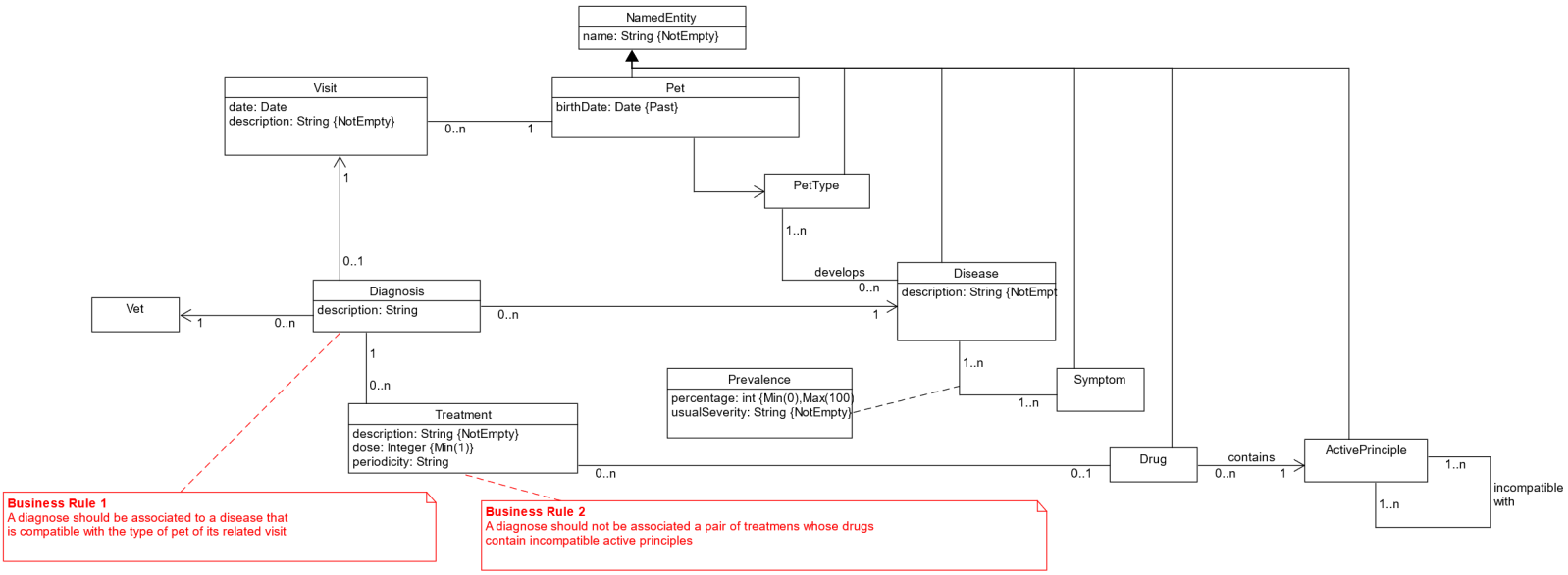
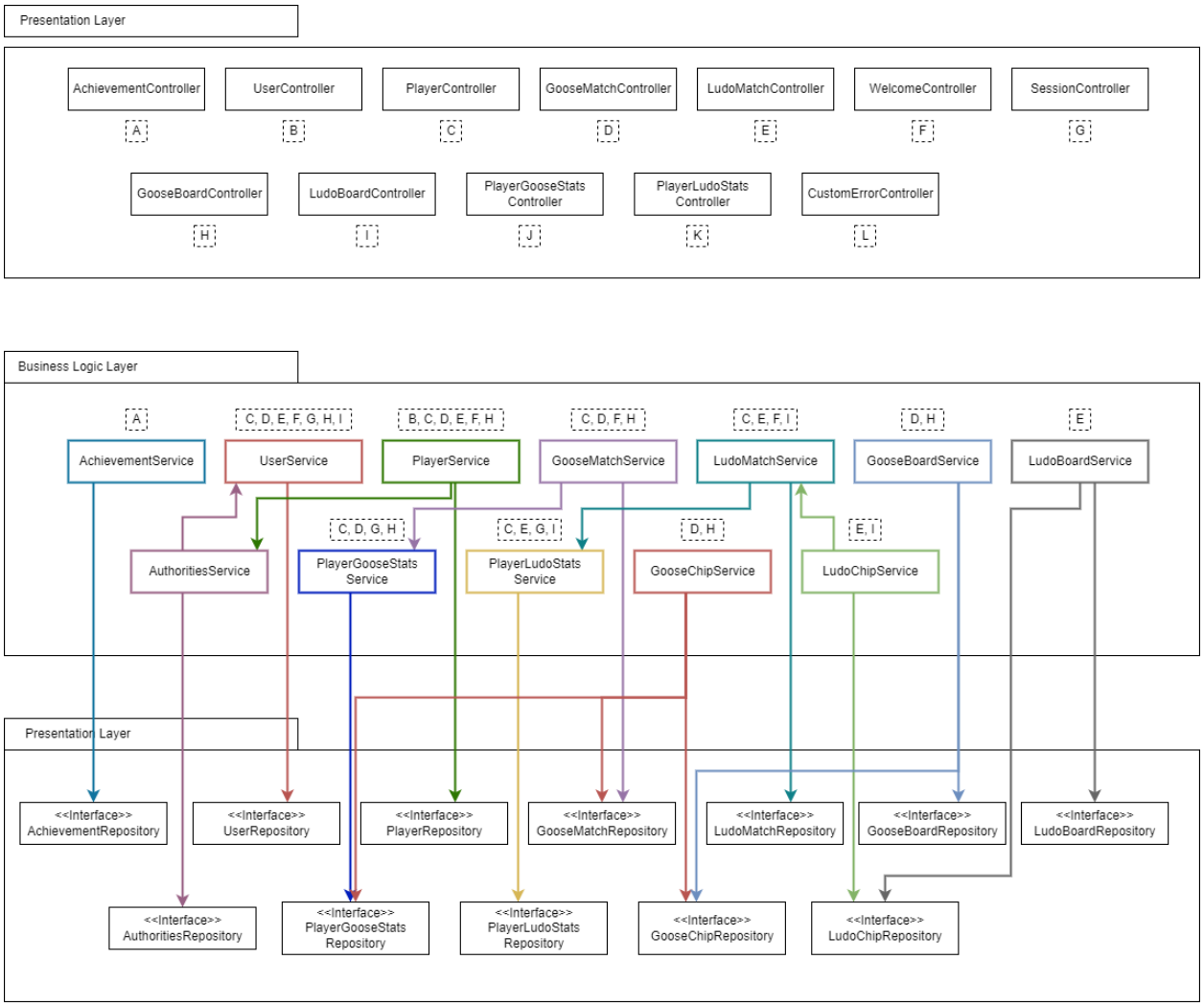


Diagrama de Capas (incluyendo Controladores, Servicios y Repositorios)



Cabe destacar que se ha optado por utilizar letras para mostrar las relaciones entre controladores y servicios, y colores para servicios a servicios y repositorios para manejar la complejidad del diagrama. Si se necesita mayor resolución, se adjunta en el repositorio de GitHub junto al resto de documentos.

Patrones de diseño y arquitectónicos aplicados

Patrón: Command

Tipo: de Diseño

Contexto de Aplicación

En el caso de la aplicación hacemos uso de este patrón de cara a la separación de las vistas y los controladores junto con los servicios. Las vistas proporcionan un conjunto de acciones a poderse realizar y los controladores y servicios se encargan de la lógica de negocio correspondiente a estas acciones.

Clases o paquetes creados

Principalmente todas las entidades que interactúan de forma directa con las vistas, es decir que tienen un controlador asociado:

- AchievementController
- GooseBoardController
- GooseMatchController
- LudoBoardController
- LudoMatchController
- PlayerController
- UserController
- SessionController
- WelcomeController

Ventajas alcanzadas al aplicar el patrón

Reduce el tamaño del problema separando las responsabilidades en distintas capas como lo son las vistas, los controladores y los servicios.

Patrón: State

Tipo: de Diseño

Contexto de Aplicación

Se usa en ciertas entidades para definir su comportamiento dependiendo del estado en el que se encuentren. En el caso de la aplicación lo hemos usado para las fichas del parchís ya que funcionan de forma distinta dependiendo de si se encuentran en la casa, en el circuito del tablero o en la recta final de su color correspondiente.

Clases o paquetes creados

- LudoChip: Clase que hace uso de los estados
- GameState: Enumerado que define los estados posibles que puede tener una determinada ficha de parchís.
- LudoChipService y LudoChipController: Clases que controlan la lógica de los cambios de estado

Ventajas alcanzadas al aplicar el patrón

Simplifica la lógica necesaria a la hora de desplazar las fichas e interactuar entre ellas reduciendo el número de comprobaciones que hay que hacer para decidir qué acciones puede realizar una determinada ficha o no.

Patrón: Template view

Tipo: de Diseño

Contexto de Aplicación

Se usa principalmente en el contexto de las vistas a la hora de generar listados de forma automática. Se ha usado principalmente en la representación de datos globales y personales de los jugadores en listas y tablas.

Clases o paquetes creados

- AchievementController
- GooseMatchController
- LudoMatchController
- PlayerController

Ventajas alcanzadas al aplicar el patrón

Agiliza el desarrollo de la aplicación reduciendo el tiempo invertido en diseñar nuevas vistas

Patrón: Transaction Script

Tipo: de Diseño

Contexto de Aplicación

Organiza la lógica de negocio de la aplicación en métodos sencillos que solo hacen una cosa determinada. Usado principalmente en los servicios.

Clases o paquetes creados

- AchievementService
- GooseBoardService
- LudoBoardService
- GooseMatchService
- LudoMatchService
- GooseChipService
- LudoChipService
- PlayerService
- PlayerGooseStatsService
- PlayerLudoStatsService
- AuthoritiesService
- UserService

Ventajas alcanzadas al aplicar el patrón

Reduce el tamaño de los problemas más grandes en un conjunto de pasos más sencillos mejorando así la mantenibilidad del código.

Patrón: Table Module

Tipo: de Diseño

Contexto de Aplicación

Organiza la lógica de la aplicación con una clase por tabla de una de una forma similar a como lo hacen los servicios. Cada entidad tiene su propio servicio que maneja la lógica de los datos entre los atributos y las tablas.

Clases o paquetes creados

- AchievementService
- GooseBoardService
- LudoBoardService
- GooseMatchService
- LudoMatchService
- GooseChipService
- LudoChipService
- PlayerService
- PlayerGooseStatsService
- PlayerLudoStatsService
- AuthoritiesService
- UserService

Ventajas alcanzadas al aplicar el patrón

Mejora la mantenibilidad del código y la manejabilidad de los posibles cambios que se puedan hacer sobre este.

Patrón: Domain Model

Tipo: de Diseño

Contexto de Aplicación

Se usa para diferenciar atributos más complejos y sus relaciones como por ejemplo los jugadores y sus logros un logro puede tener unas características al igual que su jugador y estos están claramente relacionados entre ellos.

Clases o paquetes creados

- Player
- PlayerGooseStats
- PlayerLudoStats
- User

- Authorities

Ventajas alcanzadas al aplicar el patrón
Distribuye la lógica de la aplicación en distintas clases reduciendo el tamaño del problema y su manejabilidad.

Patrón: Service layer

Tipo: Arquitectónico

Contexto de Aplicación
Se usa en los servicios y se encargan de la comunicación entre los controladores y los repositorios además de manejar la mayor parte de la lógica sobre los datos.

Clases o paquetes creados
- AchievementService
- GooseBoardService
- LudoBoardService
- GooseMatchService
- LudoMatchService
- GooseChipService
- LudoChipService
- PlayerService
- PlayerGooseStatsService
- PlayerLudoStatsService
- AuthoritiesService
- UserService

Ventajas alcanzadas al aplicar el patrón
Encapsula claramente las funcionalidades más importantes dentro de estas clases y mejora la mantenibilidad de la aplicación.

Patrón: Active Record

Tipo: Arquitectónico

Contexto de Aplicación
Se ha aplicado en diferentes contextos, el primero siendo el seguimiento de los movimientos de las fichas, que quedan almacenados en la base de datos. El segundo viene dado por la auditación de algunas tablas y entidades.

Clases o paquetes creados
Tablas auditadas:

- Achievements
- GooseMatch
- LudoMatch

- Player
- PlayerGooseStats
- PlayerLudoStats
- Authorities
- User

Ventajas alcanzadas al aplicar el patrón
Nos permite tener un mejor seguimiento de los cambios realizados en las partidas y en las tablas.

Patrón: Meta-data Mapper
Tipo: de Diseño
Contexto de Aplicación
Nos ayuda a establecer un contexto entre las relaciones de los atributos y entidades a la base de datos y es usado generalmente para indicar cómo se relacionan los atributos de ciertas entidades con otras clases facilitando así el manejo de la base de datos.

Clases o paquetes creados
- Achievements
- GooseMatch
- LudoMatch
- GooseBoard
- LudoBoard
- Player
- PlayerGooseStats
- PlayerLudoStats
- User
- Authorities
- GooseChip
- LudoChip

Ventajas alcanzadas al aplicar el patrón
Una correcta aplicación puede mejorar el rendimiento de la aplicación y la disposición de los datos.

Patrón: Identify field
Tipo: Arquitectónico | de Diseño
Contexto de Aplicación
Nos permite identificar una fila de una tabla de forma unívoca, en el contexto de nuestra aplicación se ve claramente con el atributo username de los jugadores que usamos para diferenciar a unos de otros

Clases o paquetes creados

Ventajas alcanzadas al aplicar el patrón

Reduce la cantidad de memoria necesaria a la hora de hacer cambios sobre filas específicas en las tablas y facilita la identificación de individuos o datos concretos.

Patrón: Layer supertype

Tipo: de Diseño

Contexto de Aplicación

En este caso hacemos uso de este patrón de diseño en clases que heredan ciertas características de otras clases, como es el caso de BaseEntity que lega alguna de sus características a clases como Achievements.

Clases o paquetes creados

- BaseEntity
    - Achievement
    - Authorities
    - PlayerLudoStats
    - PlayerGooseStats
    - LudoMatch
    - GooseMatch
    - LudoBoard
    - LudoChip
    - GooseBoard
    - GooseChip
    - Person
        - Player

Ventajas alcanzadas al aplicar el patrón

Simplifica el desarrollo de la aplicación reduciendo significativamente el código necesario para generar y almacenar algunas de las características de las entidades.

Decisiones de diseño

Decisión 1: Clave primaria de usuarios

Descripción del problema:

La estructura proporcionada para el manejo de cuentas en el proyecto plantilla declara el nombre de usuario como clave primaria de la entidad User, esto nos acabó generando problemas relacionados con la edición de los datos de un jugador, ya que editar el nombre de usuario supone editar dicha clave primaria, cosa que nos generaba un segundo usuario dejando el anterior “colgando”, sin jugador asociado.

Alternativas de solución evaluadas:

Alternativa 1.a: Cambiar la estructura de la entidad User, dejando el nombre de usuario como atributo corriente.

Ventajas:

- Lo más intuitivo y correcto.

Inconvenientes:

- A las alturas del desarrollo a la que nos planteamos esta alternativa, resultaba ya un cambio que afectaría a un gran número de clases relacionadas con esta entidad, retrasando el progreso del proyecto significativamente en el caso de que optáramos por esta opción.
- Entre esas clases afectadas se encontraban algunas clases de la infraestructura de seguridad que en principio no deberíamos cambiar.

Alternativa 1.b: Borrar el User “fantasma” que queda sin Player asociado en caso de que se cree uno nuevo al modificar el nombre de usuario.

Ventajas:

- Se mantiene la funcionalidad de edición completa.
- Se evita tener que cambiar la infraestructura ya creada de la aplicación.

Inconvenientes:

- Nos resulta una solución arcaica y casi temporal.

Alternativa 1.c: Restringir la edición del nombre de usuario.

Ventajas:

- La más sencilla y directa.
- Es una restricción relativamente común entre las páginas web con sistema de cuentas.

Inconvenientes:

- La funcionalidad de edición deja de ser “completa”.
- Puede resultar un problema si los usuarios no son avisados de esto y luego deciden cambiar su nombre de usuario.

Justificación de la solución adoptada

Finalmente optamos por la opción 1.c., añadiendo la regla de negocio correspondiente al documento e implementándola en el proyecto. Además, decidimos añadir un aviso a la vista de registro de nuevo usuario alertando de esta funcionalidad.

Decisión 2: Diferenciación entre lobby y partida

Descripción del problema:

En uno de los momentos previos al comienzo del desarrollo del proyecto, se nos planteó la duda de cómo diferenciar una partida en proceso de creación de una que ya se ha creado (sea esta de parchís o de oca).

Alternativas de solución evaluadas:

Alternativa 2.a: Considerar que una partida es un lobby cuando la entidad partida a la que está asociada no tiene fecha de inicio. Dicha fecha se crea una vez a partida da comienzo. En adición, para identificar un lobby de cara a los jugadores, decidimos crear el atributo matchCode, un código generado aleatoriamente que cualquiera que esté en el lobby puede ver, y cualquier otro jugador de la aplicación puede introducir para unirse a este (efectos prácticos un segundo Id, solo que de longitud uniforme, en lugar de un número que incrementa automáticamente).

Ventajas:

- Fácil implementación, lo único que diferencia una partida de un lobby es la ausencia o no de fecha de inicio. Esto puede extenderse para diferenciar también una partida en curso de una terminada, según la ausencia o no de fecha de fin.
- La longitud uniforme del matchCode contribuye a un mejor aspecto de la interfaz de la aplicación.

Inconvenientes:

- Al ser matchCode generado aleatoriamente, siempre existe la ínfima posibilidad (1,45e-9 %) de que se genere un código que ya existe y esté asociado a otra partida, provocando un error 500 de cara al jugador al violar el servidor la unicidad de la clave alternativa.

Justificación de la solución adoptada

Dado que la primera opción considerada nos resultó la más eficiente de primeras, no consideramos otras, más allá del matiz de utilizar directamente el id de la partida para el lobby en lugar del matchCode.

Respecto a matchCode, a pesar de que posee el inconveniente comentado anteriormente, puede ser potencialmente evitado aumentando la complejidad del código aleatorio, la cual hemos situado en una cadena de longitud 6, conteniendo mayúsculas, minúsculas y dígitos del 0 al 9. En el hipotético caso de que se generase un código ya existente, tras ser notificado del error el jugador en cuestión tendría simplemente que intentar generar una partida de nuevo, por lo que tampoco resulta un problema mayor.

Decisión 3: Entidades para cada casilla

Descripción del problema:

Inicialmente, cuando hicimos la primera versión del modelo de datos, señalamos la necesidad de guardar las casillas asociadas a cada tablero como entidades, aunque una vez comenzó la implementación nos tuvimos que replantear esta funcionalidad.

Alternativas de solución evaluadas:

Alternativa 3.a: Mantener la estructura inicial del modelo de datos.

Ventajas:

- Es lo que teníamos en mente en un principio.

Inconvenientes:

- Resulta redundante para cada entidad partida que se crea, junto con una entidad tablero, crear más de cincuenta entidades casilla con valores que, especialmente en el caso de la oca, son fijos y no varían en absoluto.

Alternativa 3.b: Desechar la idea de las casillas como entidad y manejar el tema de las casillas especiales de forma interna (en los servicios de las fichas y los tableros, principalmente).

Ventajas:

- De cara a la eficiencia, se generan muchas menos entidades por partida.

Inconvenientes:

- Sería necesario rediseñar el modelo de datos teniendo esto en cuenta.
- Al gestionar todo esto como lógica de negocio, los servicios en los que se manejan dichas casillas especiales inevitablemente pasan a tener métodos más largos.

Justificación de la solución adoptada

Tras comentarlo entre nosotros, acabamos optando por la opción 3.b., ya que nos resultó más sencillo de entender, a pesar de que de esta manera los servicios de las fichas y los tableros han acabado ligeramente más grandes.

Decisión 4: Flags en las estadísticas

Descripción del problema:

Este problema surge al plantear cómo representar ciertas situaciones como el hecho de que ya haya ganado la partida un jugador, un jugador tenga el turno o haya tirado dobles.

Alternativas de solución evaluadas:

Alternativa 4.a: Representarlos mediante flags, de forma que si el jugador ha ganado, ha tirado dobles o tiene turno, la flag correspondiente a esa situación valdrá 1 para representarla. Cabe destacar la excepción de la flag hasTurn que puede tomar además los valores -1 y -2 para representar que ha perdido ese número de turnos (al caer en posada o cárcel) y poder gestionar esas situaciones.

Ventajas:

- Facilita el manejo de ciertas situaciones de la aplicación y de los turnos en general.

Inconvenientes:

- La existencia de flags en la clase en la que se almacenan las estadísticas puede resultar poco intuitivo de cara a la implementación.

Justificación de la solución adoptada
Finalmente implementamos la alternativa 4.a, ya que nos permite manejar de una forma relativamente sencilla todos los eventos relacionados con turnos, tiradas dobles y la propia victoria en la partida, además de identificarlos claramente cuando ocurren.

Decisión 5: Unirse a partida estando ya en una

Descripción del problema:
Mientras diseñábamos el sistema de creación y unión a partidas nos dimos cuenta de que un jugador lógicamente no puede estar en más de una partida al mismo tiempo, por lo que necesitábamos implementar algún método para informar de esto al usuario y tratar con este error.

Alternativas de solución evaluadas:
Alternativa 5.a: Se impide al usuario la creación de nueva partida de cara tanto a la vista como a la lógica de negocio (se le deshabilita temporalmente el acceso a la creación de partida) y el controlador además impide esta situación. En caso de que intente unirse a otra no se le permite y se le muestra un mensaje informando de la situación y que además contiene el código de la partida para que pueda ingresar a la misma.

Ventajas:

- Relativamente sencilla implementación.
- Muestra la información de forma clara al usuario

Inconvenientes:

- No es la solución más directa posible en cuanto a la redirección.

Alternativa 5.b: Se impide al usuario la creación de nueva partida de cara tanto a la vista como a la lógica de negocio (se le deshabilita temporalmente el acceso a la creación de partida) y el controlador además impide esta situación. En caso de que intente unirse a otra partida se le redirecciona automáticamente a la partida que tuviese en curso.

Ventajas:

- Más directa en cuanto al funcionamiento

Inconvenientes:

- Implementación ligeramente más compleja.

Justificación de la solución adoptada
Finalmente nos decantamos por la alternativa 5.a ya que consideramos que era más cómoda de implementar e intuitiva de usar a la hora de probar la aplicación y el correcto funcionamiento de esta parte.

Decisión 6: Reglas de la oca

Descripción del problema:

A la hora de buscar las reglas de la oca, nos topamos con un problema, más personal que técnico, que decidimos abordar inmediatamente: no existe un estándar definido para estas reglas, y cada uno de nosotros había jugado una versión ligeramente distinta anteriormente, que a su vez difieren de las reglas que se pueden encontrar en wikipedia o en páginas especializadas en juegos de mesa.

Alternativas de solución evaluadas:

Alternativa 6.a: Implementar al pie de la letra las reglas encontradas en internet.

Ventajas:

- Lo más correcto, técnicamente hablando

Inconvenientes:

- Existen ciertas casillas especiales cuyo efecto afecta negativamente a la experiencia de juego, como el pozo, el cual te quita el derecho a jugar una vez caes, hasta que alguien más caiga, momento en el cual esa persona pasa a ser la afectada por la casilla.

Alternativa 6.b: Llegar a un consenso entre las reglas encontradas y las que cada uno de nosotros conoce.

Ventajas:

- Podemos aprovechar para cambiar ligeramente las reglas para hacerlas más justas.

Inconvenientes:

- Se podría discutir que esta decisión influye negativamente a la identidad del juego de la oca como tal si se realizan cambios demasiado drásticos.

Justificación de la solución adoptada

Tras comentarlo entre nosotros, acabamos optando por la opción 1.b., realizando cambios lo más sutiles posibles a la jugabilidad de la oca para que se ajustase más a la experiencia que nosotros recordamos, y adicionalmente eliminando la casilla del pozo definitivamente.

Se pueden contrastar las reglas que hemos creado, las cuales constan en el documento de requisitos y análisis del sistema, con las que proporciona la siguiente página:
https://www.juegodelaoca.com/Reglamento/reglamento.htm

Decisión 7: Dobles y casillas especiales en oca

Descripción del problema:

Este problema surgió mientras pensábamos en el funcionamiento de los dados dobles. Era necesario decidir cómo se gestionarían a partir de entonces las tiradas dobles y cómo interactuarían con las casillas especiales.

Alternativas de solución evaluadas:

Alternativa 7.a: Gestionar los dobles después de las casillas especiales. Después de la casilla especial se le daría el turno extra al jugador.

Ventajas:

- Técnicamente hablando, lo más correcto.

Inconvenientes:

- Dificultad añadida a la gestión de turnos.

Alternativa 7.b: Ignorar las tiradas dobles en caso de caída en casillas especiales, ya que todas las especiales, además de algún otro efecto, o te hacen perder el turno o directamente te dan una tirada extra, por lo que no es necesario considerar también las tiradas dobles.

Ventajas:.

- Implementación más sencilla.

Inconvenientes:

- Pérdida de la doble tirada porque la casilla especial tiene prioridad.

Justificación de la solución adoptada

Finalmente implementamos la alternativa 7.b, ya que consideramos que esa forma de gestionar las tiradas dobles se ajusta más a nuestra visión del juego de la oca. De cara al jugador, el hecho de perder la tirada extra en caso de caída en casilla especial realmente no supone un cambio significativo en la jugabilidad debido a lo ya comentado.

Decisión 8: Guardado de estadísticas de las casillas especiales de la oca y gestión de los dados y turnos

Descripción del problema:

Este problema surgió al plantearnos qué estadísticas de la oca íbamos a guardar y cómo guardarlas de forma interna en la aplicación, además del problema de la gestión de ciertos eventos como el cambio de turno o las tiradas dobles, que en ciertos casos podían entrar en conflicto con las casillas especiales.

Alternativas de solución evaluadas:

Alternativa 8.a: Guardar en un solo atributo la suma de todas las casillas especiales en las que caiga el jugador en esa partida y pasar turno al siguiente después de gestionar el movimiento. El turno se representaría mediante una flag en 0 o 1. La tirada doble se realizaría después de la gestión del movimiento antes del paso de turno.

Ventajas:

- Simple de implementar
- No es necesario almacenar las estadísticas de cada una por separado.

Inconvenientes:

- Se producían interacciones extrañas entre ciertas casillas especiales y los turnos y tiradas dobles que producían un funcionamiento erróneo de la aplicación.

Alternativa 8.b: Guardar en muchos atributos las estadísticas relacionadas a las distintas casillas especiales de la partida y usar las mismas flags para tiradas dobles y turnos que en el caso anterior, pero gestionadas de forma distinta en el servicio. De esta nueva forma primero se tendría en cuenta las casillas especiales, si cae en una de estas se desactiva la posibilidad de tirada doble sin importar cuál sea la casilla especial en la que caiga. En esa casilla se actualizan las estadísticas y además se realizan las acciones necesarias para implementar el funcionamiento de esa casilla. De esta forma se evitan las extrañas interacciones previamente mencionadas. También se evitan errores con los turnos.

Ventajas:

- Guardado de una mayor cantidad de estadísticas de interés.
- Permite obtener en caso de así quererlo el número total de casillas especiales en las que cae.
- Soluciona todos los problemas relacionados a turnos y dados.

Inconvenientes:

- Hay que guardar un mayor número estadísticas.

Justificación de la solución adoptada

Finalmente implementamos la alternativa 8.b ya que nos permitía guardar un mayor número de estadísticas, evitar problemas relacionados con las tiradas dobles y los turnos y guardar de forma sencilla la información de los turnos.

Decisión 9: Movimiento de fichas de oca

Descripción del problema:

Al diseñar el juego de la oca, nos dimos cuenta que en lo referente al movimiento el usuario no tiene que realizar ninguna acción de forma directa más allá de la propia tirada de dados, por lo que en este caso es posible un sistema que gestione el movimiento de forma automática.

Alternativas de solución evaluadas:

Alternativa 9.a: Diseño de un sistema de movimiento automático de las fichas en función de la tirada realizada y que además gestione las casillas especiales automáticamente, debido a que no es necesaria la selección de ficha a mover ni similares por parte del jugador.

Ventajas:

- Evita pasos innecesarios al jugador
- Agiliza la partida

Inconvenientes:

- Puede desorientar levemente al jugador al realizar automáticamente el movimiento y la gestión de las casillas especiales.

Justificación de la solución adoptada

Optamos por implementar la solución 9.a ya que es lo más intuitivo y evita pasos innecesarios al usuario que no aportan ningún tipo de valor al gameplay. Además para amortiguar un poco el principal inconveniente decidimos añadir unos mensajes que dieran un cierto feedback al usuario acerca de lo que estaba ocurriendo en la partida. Finalmente incorporamos un tablero en el que se ven las fichas para facilitar aún más la visualización de la partida.

Decisión 10: Movimiento de fichas de parchís

Descripción del problema:

Al contrario que en la oca, en el parchís cada jugador tiene 4 fichas que puede mover, por lo que es necesario dar la posibilidad de elegir qué ficha mover cada vez que se tiran los dados, e incluso elegir cuál de los dos dados se quiere sumar antes (suponiendo el caso de que se decide sumar los dos a una misma ficha, podría resultar más beneficioso para el jugador sumar el primero antes, o viceversa).

Alternativas de solución evaluadas:

Alternativa 10.a: Formulario de asignación de resultados de los dados, con tres elecciones radiales (a qué ficha sumar primer dado, a cuál sumar el segundo, y cuál sumar antes).

Ventajas:

- Relativamente intuitivo de cara al usuario

Inconvenientes:

- Necesidad de poder diferenciar visualmente cada ficha de cada jugador, para poder seleccionarla en el formulario sin riesgo a equivocarse.
- Solución poco eficiente de cara a la jugabilidad.
- Supone una interfaz de usuario menos limpia.

Alternativa 10.b: Crear enlaces para elegir el orden de la suma de dados, y en estos mostrar el tablero como en el controlador default de la partida, pero envolviendo las fichas con enlaces que llamen al controlador que gestiona el movimiento.

Ventajas:

- La opción más intuitiva, y la interfaz más limpia.
- Se evita tener que diferenciar visualmente las fichas, más allá de representarlas en el tablero.

Inconvenientes:

- Debemos gestionar casos en los que se acceda a dichos controladores manualmente aún en momentos en los que un jugador no debería poder avanzar sus fichas.
- La elección en dos pasos supone la creación de dos controladores.

Justificación de la solución adoptada

A pesar de aparentemente resultar ligeramente más complicada de implementar, finalmente optamos por la opción 10.b., dado que es una solución muy limpia que beneficia tanto a la experiencia de juego como al aspecto de la interfaz del tablero.

Decisión 11: Representación de la posición de las fichas en el parchís

Descripción del problema:

Al comenzar a plantear el movimiento de las fichas nos dimos cuenta de que teníamos un problema con las casillas tanto de las casas de los jugadores como con las del final de la partida. Esto se debe a que estas casillas no estaban numeradas y además en las de casa había cuatro fichas a la vez en la misma casilla. Además estas casillas no numeradas tienen un funcionamiento completamente distinto al del resto.

Alternativas de solución evaluadas:

Alternativa 11.a: Tratarlas como casillas normales con una numeración interna que no se representase en el propio tablero.

Ventajas:

- Una forma sencilla de tratar con las posiciones, siendo estas números sin más.

Inconvenientes:

- Hay que transformar luego las posiciones de la ficha en las posiciones adecuadas para el tablero.
- Se hace tedioso el tener que tener en cuenta más casillas de las realmente existentes e identificar cuál es cuál en el caso de las que no están numeradas.

Alternativa 11.b: Crear un atributo de la ficha que represente el estado de la partida en que se encuentra, siendo estos estados EarlyGame, MidGame y EndGame. EarlyGame representaría que se encuentra en casa mientras que MidGame representaría que se encuentra en el tablero “normal” (casillas numeradas) y EndGame serían las casillas finales de cada color.

Ventajas:

- Permite visualizar de forma más rápida e intuitiva el estado actual de la ficha.
- Es más sencillo comprobar si hay bloqueos ya que aunque estén en la misma posición solo hay que comprobar también el estado de la partida en que se encuentra.
- Fácil representación en el caso de las casillas no numeradas.

Inconvenientes:

- Además de con la posición hay que trabajar con el estado de cada ficha.

Justificación de la solución adoptada

Consideramos que la alternativa 11.b es mejor ya que representa mejor el estado global de la ficha y facilita realizar ciertas comprobaciones que ya tenemos planteadas para el desarrollo del propio juego.