DP1 2020-2021

Documento de Diseño del Sistema

Proyecto Idus Martii

https://github.com/gii-is-DP1/dp1-2021-2022-G6-4

Miembros:

- Carlos Garrido Rodríguez
- Juan Carlos Gómez Ramírez
- Angel Lorenzo Casas
- Mario Pérez Coronel
- Álvaro Úbeda Ruiz

Tutor: Carlos Guillermo Müller Cejas

GRUPO G6-04

Versión 1

21/01/2022

Historial de versiones

Fecha	Versión	Descripción de los cambios	Sprint
13/12/2020	V1	● Creación del documento	2
26/12/2021	V2.01	● Apartado de introducción	3
5/12/2021	v2.02	● Apartado de patrones de diseño ● Apartado de decisiones de diseño	3
06/01/2022	V2.03	● Añadido diagramas de dominio del módulo Juego, Friends y Estadísticas ● Añadido Diagrama de capas del modulo Estadisticas	3
19/01/2022	v2.04	● Añadido los patrones de diseño ● Explicado a más detalle las decisiones de diseño	4

Contents

Introducción

El proyecto consiste en la implementación de una aplicación web para facilitar a personas de todo el mundo a jugar al juego de mesa titulado como “idus martii”. El objetivo principal de esta implementación es poder conseguir que cualquier persona con un dispositivo electrónico, el cual tenga instalado en él un navegador web, y, además, tenga una conexión a internet, pueda jugar a este juego sin tener que comprar el juego en su versión física y sin depender de ninguna otra persona ni de donde se encuentre geográficamente.

Idus martii es un juego de mesa de rol oculto en el que pueden participar entre 5 y 8 jugadores. En este juego, los jugadores formarán parte del Senado durante una conspiración secreta que intenta acabar con la vida del César. Cada jugador deberá escoger un bando, según su objetivo secreto, e intentar descubrir quién piensa igual que él para poder forzar la balanza hacia un lado u otro. Sin embargo, los jugadores deberán de tener cuidado ya que a algunos senadores sólo les importa su bolsillo y se venderán al mejor postor. La duración aproximada de la partida es de aproximadamente vente o veinticinco minutos y no tiene ningún tipo de límite de tiempo.

Antes de empezar la partida, hay que hacer la siguiente preparación:

1. En primer lugar, hay que separar tantas parejas de cartas de facción “leal” y “traidor” como jugadores haya menos uno. Es decir, si hay cinco jugadores en la partida, habrá que seleccionar ocho cartas de facción, de las cuales cuatro son de la facción “leal” y otras cuatro de la facción “traidor”. Posteriormente, hay que añadir dos cartas de facción “mercader” a la baraja seleccionada anterior.
2. En segundo lugar, se barajan todas las cartas seleccionadas anteriormente y se reparten dos cartas a cada jugador boca abajo. Cada jugador podrá ver sus dos cartas, pero en ningún momento podrá mostrárselas a los otros jugadores.
3. En tercer lugar, se separan las seis cartas de voto y las cuatro cartas de rol y se dejan en el centro de la mesa boca arriba.
4. En cuarto lugar, se coloca la carta de “Svffragivm” en el centro de la mesa, boca arriba, y se colocan los marcadores de facción al lado. Esta carta actuará de contador y nos servirá, por tanto, para contar los votos de cada facción durante el desarrollo de la partida.
5. En quinto y último lugar, se asignarán los roles a los jugadores. El Cónsul será el jugador que haya tocado el cuchillo más recientemente, el jugador a su izquierda obtendrá el rol de Pretor y los dos siguientes cogerán los roles de Edil.

La partida se desarrolla en dos rondas, cada ronda tiene tantos turnos como jugadores haya en mesa. La primera ronda se compone de varias fases:

- Votación: donde cada Edil coge una carta de voto verde y una roja. Posteriormente, seleccionan su voto secretamente y colocan dicha carta frente a sí, boca bajo, sin que ninguno de los otros jugadores la vea.

- Veto: En esta fase el Pretor escoge una de dichas cartas de voto y la mira. Si lo desea puede forzar a dicho Edil a intercambiar dicha carta de voto por la carta que le queda en la mano, es decir, lo obligaría a cambiar su voto.
- Conteo: el Cónsul recoge los votos finales, los mezcla sin que nadie los vea y los muestra al resto de jugadores. El Cónsul contabiliza los votos en la carta de "Svffragivm". Por cada voto el marcador de dicha facción avanza un espacio. Dependiendo del número de jugadores en la partida, hay un límite de votos marcados en la carta "Svffragvm", en el caso de que se superase, se procedería al final de la partida.
- Fin del turno: el Cónsul descarta, boca abajo, una de sus cartas de facción a su elección. De esta forma escogen su bando, lo que será crucial para el final de la partida (El jugador inicial que tenga el rol de Cónsul ignora este paso en su turno). Posteriormente, cada jugador con carta de rol entrega dicha carta al jugador a su izquierda (el Pretor pasará a ser Cónsul, el primer Edil pasará a ser Pretor, el segundo Edil seguirá siendo Edil, etc) y empieza un nuevo turno.

En la primera ronda habrá tantos turnos como jugadores haya en la mesa. Una vez el jugador inicial obtenga de nuevo el rol de Cónsul, comenzará la segunda ronda. En esta segunda ronda, se siguen los mismos pasos de cada fase de la ronda uno, con los siguientes añadidos:

- Votación: el Cónsul asigna el resto de las cartas de rol, cada una a un jugador diferente, sabiendo que los jugadores no pueden repetir rol dos turnos seguidos (a excepción de partidas a 5 jugadores donde se podrá repetir un Edil). Adicionalmente, cada Edil coge una carta de voto amarilla, una roja y una verde para poder votar con cualquiera de ellas.
- Veto: si el Pretor escoge una carta amarilla debe mostrarla al resto de jugadores, y el Edil correspondiente la sustituye boca abajo por uno de sus otros dos votos.
- Conteo: el voto amarillo es un voto nulo, y no hace avanzar ningún marcador de facción.
- Fin del turno: el Cónsul descarta, boca abajo, una de sus cartas de facción a su elección (Sólo el jugador inicial realizará este paso, ya que el resto de los jugadores tienen ya una sola carta de facción). Sólo se entrega la carta de rol de Cónsul al jugador de la izquierda, el resto de las cartas se asignan al inicio del paso de votación del próximo turno.

En esta segunda ronda, habrá tantos turnos como jugadores haya en la mesa. Una vez finalizada, es decir, cuando el jugador inicial reciba por tercera vez la carta de Cónsul, se produce el final de la partida.

La partida termina inmediatamente cuando se cumple una de las siguientes condiciones:

- Conspiración fallida: un marcador de facción sobrepasa el límite de jugadores, tal y como está indicado en la carta de "Svffragivm".
- IDUS DE MARZO: el jugador inicial recibe la carta de rol por tercera vez.

En cualquier caso, cada jugador revela su carta de facción y se determina la facción ganadora:

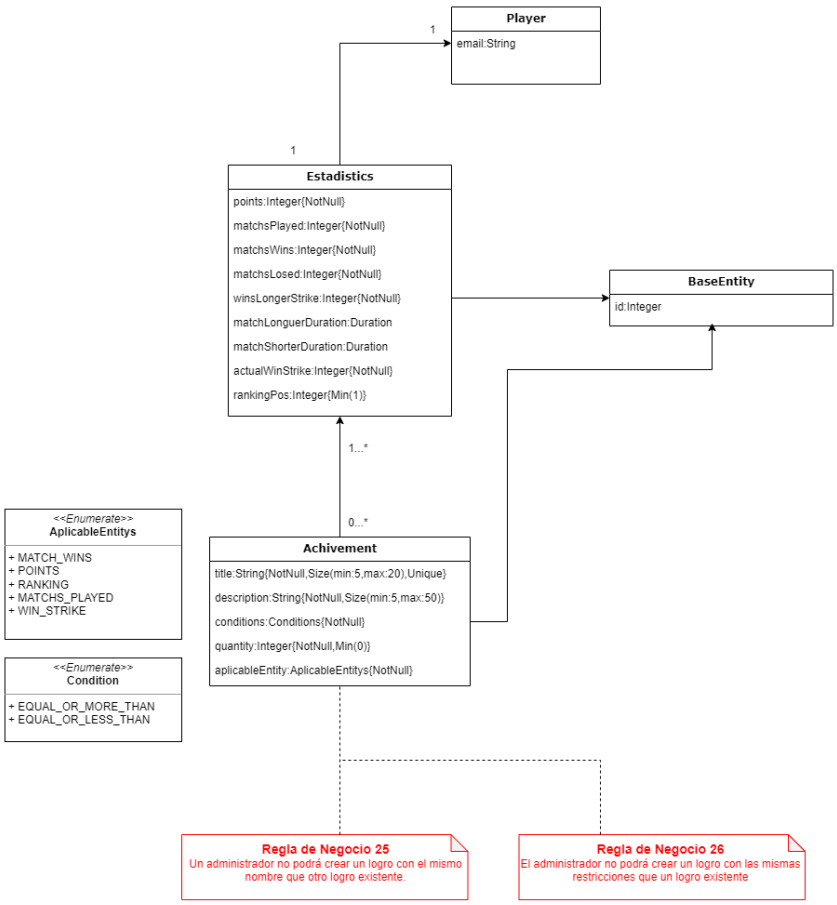
- En caso de conspiración fallida, la facción cuyo marcador sobrepase el límite de jugadores es descubierta por la facción rival y sus integrantes son asesinados. En consecuencia, gana la facción rival. En el caso de que no hubiera ningún jugador de la facción rival, gana la facción “Mercader”.
- En caso de IDUS DE MARZO, la facción que tenga dos o más votos que la facción rival gana la partida ya que consigue frustrar los planes de sus oponentes. En caso contrario, si hay igualdad de votos, diferencia de un voto, o no hubiera ningún jugador de la facción rival, gana la facción “Mercader” ya que se han conseguido mantener el status quo… hasta los próximos Idus de Marzo.

A continuación podemos ver algunas de las cartas del juego:

Diagrama(s) UML:

Diagrama de Dominio/Diseño

- Diagrama de Dominio del modulo de estadisticas:



- Diagrama de Dominio para el módulo de Juego y Friend:

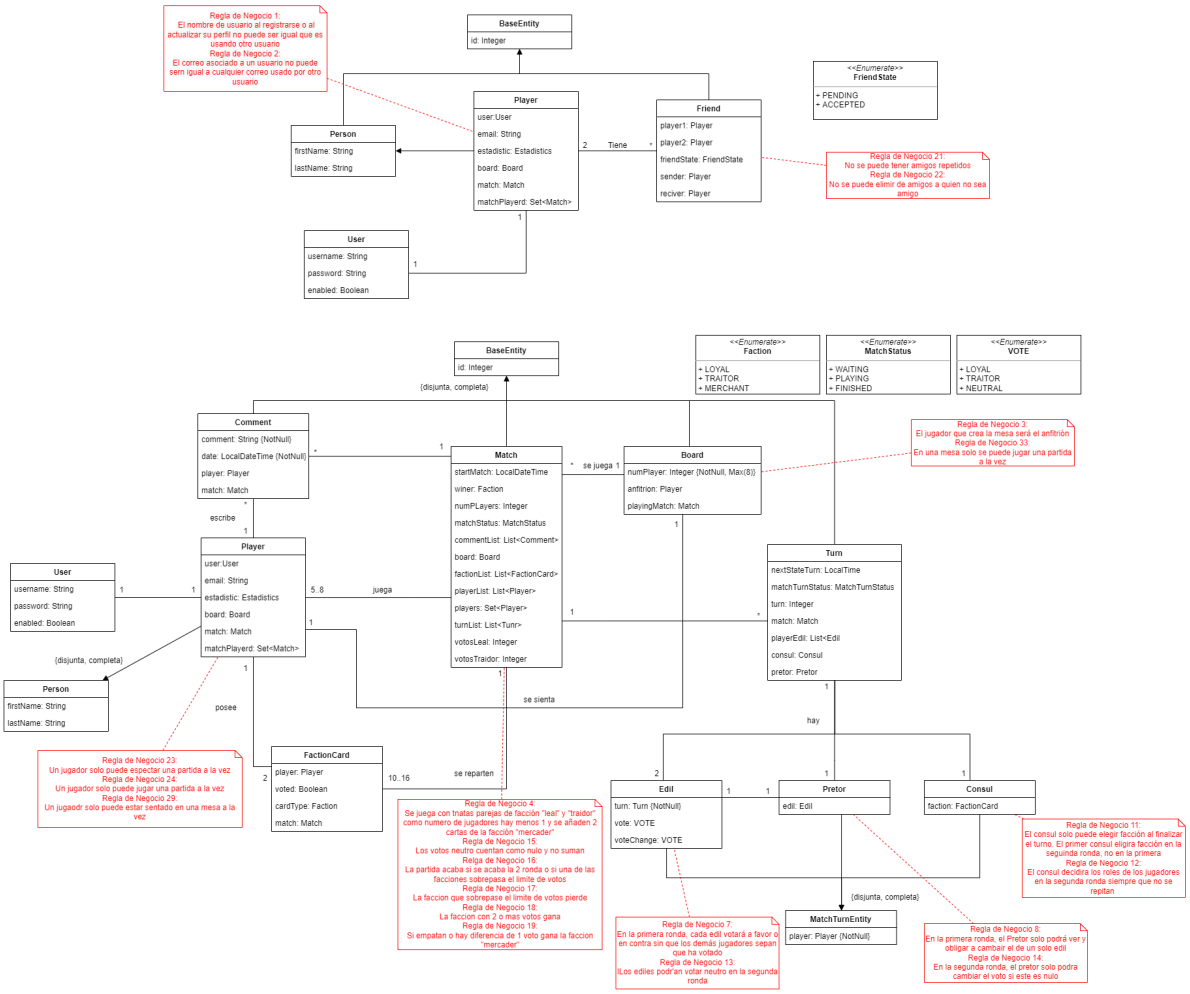
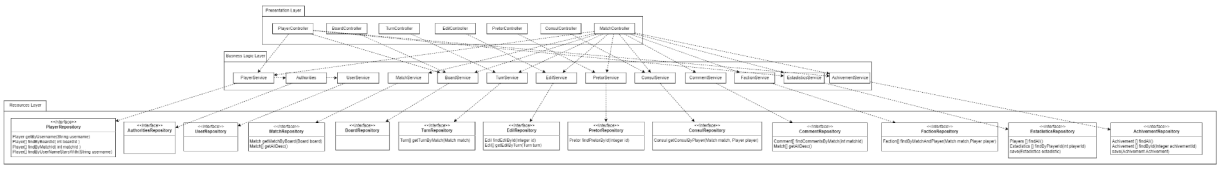
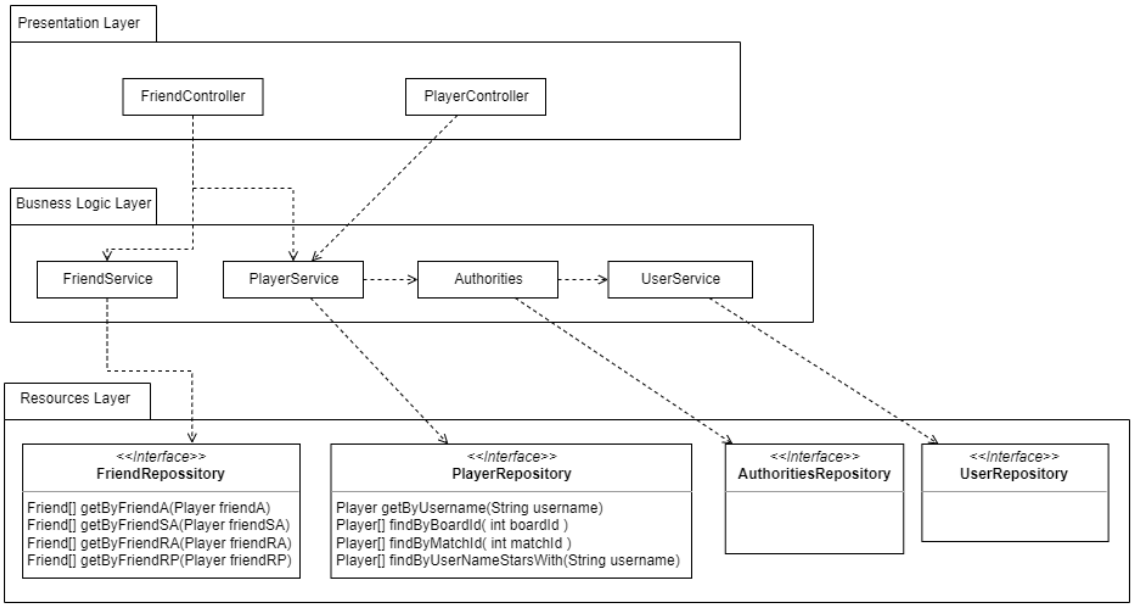


Diagrama de Capas (incluyendo Controladores, Servicios y Repositorios)

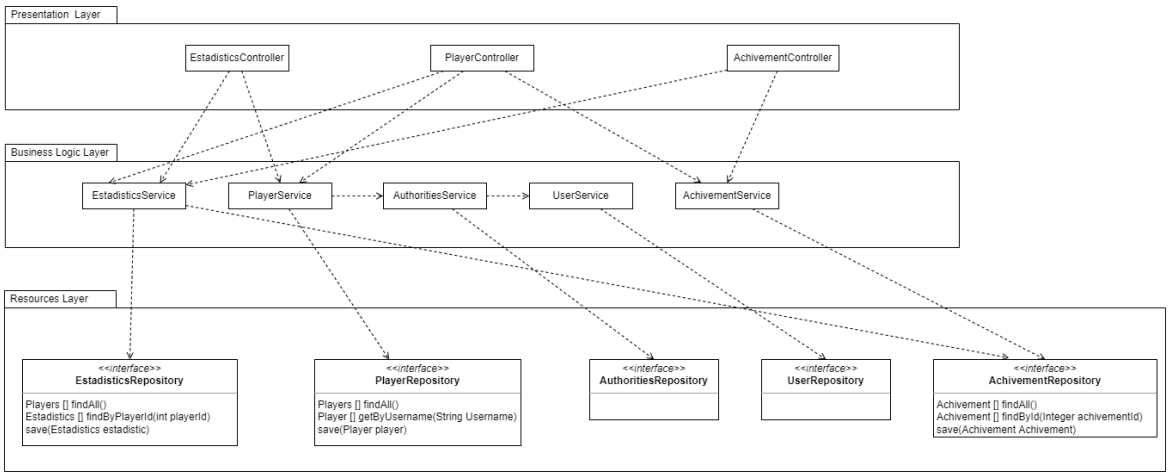
- Diagrama de Capas para el modulo de Juego:



- Diagrama de Capas para el modulo de Friends:



- Diagrama de Capas para el modulo de Estadisticas:



Patrones de diseño y arquitectónicos aplicados

En esta sección de especificar el conjunto de patrones de diseño y arquitectónicos aplicados durante el proyecto. Para especificar la aplicación de cada patrón puede usar la siguiente plantilla:

Patrón: FrontController

Tipo: Diseño

Contexto de Aplicación

Para el desarrollo del proyecto usaremos el patrón de diseño Front Controller, el cual nos permite crear funciones llamadas “contoller” la cual se encarga de manejar todas las peticiones de una vista y las gestiona al manejador apropiado.

Clases o paquetes creados

Se ha creado una clase controlador por entidad (exceptuando un par de entidades que no tienen su propia clase controller porque no hacía falta) las cuales contienen todos sus controladores.

Ventajas alcanzadas al aplicar el patrón

Se encarga de manejar todas las peticiones de la página y las gestiona al manejador apropiado. Nos ayudan a centralizar el control, además también nos ofrece otras funcionalidades útiles, como pueden ser la validación o la transformación. Más concretamente:

- Transforman las entidades al modelMap para poder procesadas en las vistas.
- Validar los valores de entrada que nos proveen las vistas.
- Transformar las entradas de las vistas en entidades si hiciera falta.
- Llamar a los servicios pero concretar alguna acción.
- Devolver la llamada a la vista que debe ser cargada

Patrón: Domain Model

Tipo: Diseño

Contexto de Aplicación

Para el desarrollo de nuestro proyecto, hemos usado el patrón diseño Domain Model que nos ofrece Spring, para poder representar los datos en las diferentes vistas. Para ello se ha hecho uso de ello en todos los controladores creados.

Clases o paquetes creados

Se ha usado en todos los controladores creados para el desarrollo del proyecto.

Ventajas alcanzadas al aplicar el patrón

Este patrón de diseño nos permite implementar complejos lógicos de negocios y poder representar nuestros datos a las vistas muy fácilmente.

Patrón: Service layer

Tipo: Diseño

Contexto de Aplicación

Para el desarrollo de nuestro proyecto, también hemos usado el patrón de diseño Service Layer, para así poder interactuar desde el controlador hasta el repositorio más fácilmente. La capa de servicio define el esqueleto de la aplicación estableciendo un conjunto de operaciones y coordinando la respuesta de la aplicación en cada operación. La capa de presentación interactúa con el dominio a través de la capa de servicio.

Clases o paquetes creados

Se ha creado una clase de servicio por cada entidad desarrollada para el proyecto, donde se implementan los diferentes métodos necesarios.

Ventajas alcanzadas al aplicar el patrón

- Nos define el esqueleto de la aplicación.
- Establece un conjunto de operaciones y coordina la respuesta de la aplicación en cada operación.

Patrón: Repositorio

Tipo: Diseño

Contexto de Aplicación

Son clases que encapsulan la lógica requerida para acceder a los datos. Su objetivo es dar la impresión de que estás accediendo a una colección de objetos de memoria en vez de la base de datos. Crearemos un repositorio por entidad.

Clases o paquetes creados

Se ha creado una clase de repositorio por cada entidad desarrollada para el proyecto, donde se implementan los diferentes métodos necesarios.

Ventajas alcanzadas al aplicar el patrón

- Nos permite obtener los datos de la base de datos fácilemente.
- Podemos hacer consultas de todo tipo en la clase.

- No facilita la tarea de tener que acceder a la base de datos y obtener los diferentes datos que necesitemos.

Patrón: Estrategia

Tipo: Diseño

Contexto de Aplicación

El patrón de estrategia es un patrón de diseño de software de comportamiento que permite seleccionar un algoritmo en tiempo de ejecución. En lugar de un solo algoritmo directamente, el código recibe en tiempo de ejecución instrucciones sobre cuál en una familia de algoritmos a utilizar.

Clases o paquetes creados

Es un patrón que viene ya incluido en spring, por lo que afecta al proyecto en general.

Ventajas alcanzadas al aplicar el patrón

- Previene declaraciones condicionales, simplificando el código fuente.
- Promueve un bajo acoplamiento y cohesión.
- Promueve la extensibilidad de los algoritmos admitidos.
-

Patrón: Cadena de responsabilidad

Tipo: Diseño

Contexto de Aplicación

Este patrón el cual nos obliga a usar Spring, crea una cadena de objetos para examinar solicitudes. Cada objeto a su vez examina una solicitud y la maneja o la pasa al siguiente objeto de la cadena.

Clases o paquetes creados

Es un patrón que viene incluido en spring, por lo que afecta al proyecto en general.

Ventajas alcanzadas al aplicar el patrón

- Promueve el bajo acoplamiento.
- Simplifica su objeto porque no tiene que conocer la estructura de la cadena y mantener referencias directas a sus miembros.
- Le permite agregar o quitar responsabilidades de forma dinámica al cambiar de miembros u orden de la cadena.

Decisiones de diseño

Decisión 1: relación entre entidad mesa y entidad partida

Descripción del problema:

La primera decisión que hemos tenido que tomar es en diseñar una relación entre las entidades board-match manyToMany o oneToMany, ya que en el desarrollo del proyecto comenzamos con una relación OneToMany pero nos encontramos con un problema en el cual, para luego mostrar los jugadores de una partida finalizada no podíamos, ya que la id de la partida que se está jugando se guarda en la entidad player, pero en el momento de terminar la partida se elimina por lo que no podiamos consultarla.

Alternativas de solución evaluadas:

Alternativa 1.a: modificar todo para dejar todas las relaciones entre board-player manyToMany

Ventajas:

- Es lo más cómodo para todo el desarrollo posterior, ya que tener una relación manyToMany entre la entidad board y la entidad player nos facilita la implementación de otras historias de usuario.

Inconvenientes:

- Es la opción que más cambios conlleva ya que habría que modificar todo el código ya desarrollados de ambas entidades.

Alternativa 1.b: dejar todo como está (relación oneToMany), y crear una propiedad en player que sea una lista con todas las partidas que ha jugado, la cual sea una relación ManyToMany.

Ventajas:

- No conlleva ningún cambio de lo ya desarrollado, ya que solo habría que añadir una propiedad más en player, que sería una lista con todas las partidas jugadas..
- Más simple, ya que es el que menos cambios conlleva.

Inconvenientes:

- Añade más complejidad para el posterior desarrollo, ya que al contrario que en la alternativa 1.a para implementar otras historias de usuario, de esta manera tenemos más dificultades para obtener el historial de partidas jugadas de un jugador determinado.

Justificación de la solución adoptada

Finalmente se decidió por desarrollar la alternativa b, ya que era la más cómoda de implementar y posteriormente no había mucha cantidad de código a desarrollar. Además, analizando las historias de usuario que quedaban por desarrollar, nos dimos cuenta que tampoco implicaba mucha carga de trabajo adicional.

Decisión 2: Código html de desarrollo de la partida

Descripción del problema:

Como grupo queríamos hacer que el desarrollo de la partida se ejecutará en un mismo jsp, para así tenerlo todo en el mismo y no tener un jsp para cada sección de la partida. Debido a que no usamos javascript, esto complicaba la tarea y teníamos dos opciones.

Alternativas de solución evaluadas:

Alternativa 2.a: desarrollar todo el código de la partida en un jsp.

Ventajas:

- Simple, ya que solo tenemos que escribir el código en un jsp ordenadamente, y tendríamos toda la ejecución de una partida en un mismo jsp.

Inconvenientes:

- Añadía mucha complejidad, ya que para desarrollar la partida al completo hay muchas sentencias if y for anidadas. Además ni siquiera sabíamos si se podría realizar de esta manera.

Alternativa 2.b: desarrollar la columna principal del código en el jsp, y posteriormente desarrollar tags con las diferentes funcionalidades del juego, para así tener el código modulado e interactivo según las decisiones que tome cada usuario jugando la partida..

Ventajas:

- Tener el código modulado y más ordenado.
- Tener el código separado por funcionalidades en los diferentes tags.
- Menor número de vistas, ya que solo tendríamos que crear una.

Inconvenientes:

- Más esfuerzo al tener que modular todas las funcionalidades.
- Más esfuerzo ya que supone una investigación profunda para el buen uso de tags.

Alternativa 2.c: crear un jsp diferente por cada funcionalidad diferente que tiene la partida.

Ventajas:

- Tener el código modulado en diferentes jsp.

Inconvenientes:

- Mucha cantidad de archivos jsp.
- Más esfuerzo al tener que desarrollar tantos archivos jsp.

Justificación de la solución adoptada

Finalmente, como grupo nos decantamos por la opción b, ya que teníamos la ventaja de tener el código mejor organizado, a pesar de que nos conlleva más tiempo de desarrollo. Además, creemos que sería la opción más realista.

Decisión 3: Entidades de roles

Descripción del problema:

Como grupo nos encontramos ante una decisión importante la cual se debatía en cómo implementar las entidades de los diferentes roles de la partida (cónsul, edil, pretor).

Alternativas de solución evaluadas:

Alternativa 3.a: poner cada rol como un atributo en la entidad player.

Ventajas:

- Opción más simple de desarrollar ya que solo habría que añadir 3 atributos de tipo bool a la clase.

Inconvenientes:

- Posteriormente, en el desarrollo de la partida nos podría dar problemas para el desarrollo de otras historias de usuario ya que cada rol tiene diferentes funcionalidades.

Alternativa 3.b: desarrollar las diferentes entidades con sus respectivas propiedades y relaciones.

Ventajas:

- Nos facilita el desarrollo de posteriores historias de usuario ya que como cada rol tiene sus acciones y propiedades se podrían especificar en sus respectivas clases..

Inconvenientes:

- Más complejo de desarrollar en un principio, ya que habría que crear 3 clases diferentes, con sus respectivos controladores, servicios y repososítorios. Además, de crear sus diferentes relaciones.

Justificación de la solución adoptada

Finalmente, como grupo nos decantamos por la opción b,ya que nos aseguraba el no tener problemas en un futuro con las diferentes historias de usuario. Ya que la primera opción teníamos dudas si se podía implementar de esta manera ya que cada rol tiene diferentes funcionalidades.

Decisión 4: Problema inicio de partida

Descripción del problema:

Como grupo nos encontramos ante un problema en el cual no sabíamos cómo poder iniciar una partida.

Alternativas de solución evaluadas:

Alternativa 4.a: que todos los jugadores de la mesa tuvieran un botón de confirmado, y cuando todos confirmen se inicie.

Ventajas:

- Opción más lógica a desarrollar ya que en el caso de que haya un jugador ausente no se empezará la partida.

Inconvenientes:

- Puede ser una opción muy lenta, ya que se tiene que esperar a que todo el mundo confirme para poder iniciar una partida.
- Más complejo de desarrollar, ya que habría que crear x botones, y controlar cuantos han confirmado en cada momento.

Alternativa 4.b: crear un anfitrión que sea la primera persona que se siente en una mesa (si este se sale de la mesa, pasaría a otro, etc.) y que él tuviera la decisión de cuándo iniciar la partida.

Ventajas:

- Opción fácil de desarrollar, ya que solo habría que crear una propiedad en la entidad board que sea la id del jugador anfitrión.
- No depende de todos los jugadores, para una menor espera.

Inconvenientes:

- Si hay algún jugador ausente, la partida empezaría con el jugador ausente.

Justificación de la solución adoptada

Nos decidimos por la opción b, ya que era la más sencilla y la que creemos que mejor puede funcionar para evitar tiempos de esperas largos, etc.

Decisión 5: Problemas para elegir anfitrión

Descripción del problema:

Como grupo nos encontramos ante un problema de cómo evitar jugadores ausentes durante el desarrollo de la partida.

Alternativas de solución evaluadas:

Alternativa 5.a: que al cabo de x tiempo, el sistema automáticamente tome una decisión por el jugador.

Ventajas:

- Opción más lógica y confortable para los jugadores de la partida, ya que no notarán que hay un jugador ausente y por tanto podrían continuar la partida.

Inconvenientes:

- Opción más tediosa de desarrollar, ya que habría que realizar un contador, y que cuando finalice se tome una decisión por él, en el caso de que no haya realizado ninguna acción.

Alternativa 5.b: eliminar al jugador ausente después de x tiempo.

Ventajas:

- Opción útil en el caso de que haya jugadores ausentes, ya que los echaría de la partida.

Inconvenientes:

- Opción menos rentable ya que puede ser que se acabe la partida por tener menos jugadores de los permitidos, perjudicando así a los demás jugadores.
- Opción compleja ya que habría que añadir un contador de segundos para detectar cuando un jugador está ausente.

Alternativa 5.c: eliminar a todos los jugadores de la partida cuando haya un jugador ausente.

Ventajas:

- Opción fácil de desarrollar, ya que a partir de x segundos de un jugador sin responder se acabaría la partida.

Inconvenientes:

- Opción menos rentable ya que puede ser que se acabe la partida por tener menos jugadores de los permitidos, perjudicando así a los demás jugadores.

Justificación de la solución adoptada

Finalmente elegimos la opción a ya que es la más lógica y la que no perjudica a los demás jugadores. Además, también creemos que es la más fácil de desarrollar.

Decisión 6: Transcurso de las rondas

Descripción del problema:

Como grupo nos encontramos ante el problema de cómo se iba a ejecutar el juego mientras los jugadores hacían X acción.

Alternativas de solución evaluadas:

Alternativa 6.a: que al cabo de x tiempo la ronda pasaría a al siguiente estado automáticamente

Ventajas:

- La implementación es muy sencilla

Inconvenientes:

- Existe la posibilidad de que hayan null por jugador ausente

Alternativa 6.b: que cuando el jugadores termine su acción, pase al siguiente estado.

Ventajas:

- La implementación es muy sencilla

Inconvenientes:

- La partida es muy poco dinámica si hay que esperar un tiempo infinito

Justificación de la solución adoptada

Nos hemos decantado por la opción a porque era una forma sencilla de implementar en un principio, aun así después hemos tenido diferentes problemas con los null y hemos automatizado algunas elecciones si no se aplican por el jugador.

Decisión 7: Cálculo de estadísticas Individuales

Descripción del problema:

Como grupo nos encontramos en la tesitura de de en qué momento deberían calcularse las estadísticas de cada jugador individualmente.

Alternativas de solución evaluadas:

Alternativa 7.a: calcular las estadísticas cada vez que el propio jugador, administrador u otros jugadores accedan a sus estadísticas

Ventajas:

- Las estadísticas del jugador únicamente se actualizarán cuando alguno de estos 3 actores entrara en las estadísticas del jugador y solo en caso de que jugara más partidas.
- El único cálculo que debería hacer el sistema en caso de no necesitar actualizar las estadísticas sería su posición en el ranking.

Inconvenientes:

- Es una opción que puede hacer que el tiempo de carga de la vista sea mayor de lo esperado.
- En caso de no haber entrado en esa vista durante mucho tiempo, y haber jugado bastantes partidas, podría colapsar el sistema de consultas, ya que dependería de las partidas jugadas y más cálculos externos.

Alternativa 7.b: calcular las estadísticas cada cierto tiempo, por ejemplo al final de cada día

Ventajas:

- Las estadísticas del jugador únicamente se actualizarán en una determinada hora, donde habría menos peticiones al sistema y haciendo que hubiera menos posibilidades de colapsar.
- Solo habría que calcular los datos con las partidas de ese mismo día, por lo tanto se reducen el número de consultas a partidas.

Inconvenientes:

- Los jugadores no verían sus Estadísticas en tiempo real si no del día anterior.
- Al hacerse todas las Estadísticas al mismo tiempo, hay posibilidad que, aunque se hagan menos consultas a partidas, se hacen más por parte de todos los jugadores. Pudiendo colapsar el sistema.

Alternativa 7.c: calcular las estadísticas después de cada partida

Ventajas:

- Las estadísticas de los jugadores que juegan una partida se calculan al terminar esa partida, minimizando el número de peticiones.
- Solo involucra una partida y el número de jugadores que hayan jugado la partida.

Inconvenientes:

- Puede ralentizar un poco el cambio de vista de final de partida.
- Puede que haya problemas con el ranking si dos partidas terminan a la vez.

Justificación de la solución adoptada

Utilizamos finalmente la opción 7.c, al tener menos posibilidades esta opción de colapsar el sistema en producción. El resto de opciones a mayor número de jugadores y partidas totales seguramente el programa acabe completamente colapsado o como mínimo ralentizado durante un tiempo.

Decisión 8: Baraja

Descripción del problema:

Como grupo nos encontramos ante el problema de cómo implementar la baraja dentro de la partida.

Alternativa 8.a: Utilizar tres entidades diferentes según que tipo de carta sea

Ventajas:

- Ninguna

Inconvenientes:

- Difícil de implementar una baraja con diferentes entidades

Alternativa 8.b: Utilizar una entidad carta que tenga una propiedad con su tipo

Ventajas:

- Fácil de implementar una baraja

Inconvenientes:

- Añade una complejidad extra

- Añade una relación

Justificación de la solución adoptada

Hemos utilizado la opción b, ya que aunque en un principio era más tediosa, como grupos pensamos que iba a ser mejor que la baraja simplemente sea una lista de cartas que siempre se pueda consultar de una manera sencilla. También pensamos que si esta baraja se guardaba teníamos la posibilidad de saber que has elegido respecto de lo que tenias e íbamos a tener más datos para el modulo de estadística.

Decisión 9: Jugadores aleatorios

Descripción del problema:

Como grupo nos encontramos ante el problema de que la lista de jugadores sea diferente en cada partida.

Alternativa 9.c: utilizar randomizador con una seed que sería la id de la partida

Ventajas:

- Fácil de implementar
- Siempre sabrás el orden de la partida si tienes la id de la partida

Inconvenientes:

- No hay ninguno

Justificación de la solución adoptada

Utilizamos esta opción ya que nos confirmaba de que aunque hagas un petición en cualquier momento de los player siempre sabrás el orden y también nos aporta esa randomización que necesitábamos.

Justificación de la solución adoptada

Utilizamos finalmente la opción 7.c, al tener menos posibilidades esta opción de colapsar el sistema en producción. El resto de opciones a mayor número de jugadores y partidas totales seguramente el programa acabe completamente colapsado o como mínimo ralentizado durante un tiempo.

Decisión 10: Cálculo de estadísticas Individuales

Descripción del problema:

Como grupo nos encontramos en la tesitura de de en que momento deberían calcularse las estadísticas de cada jugador individualmente

Alternativas de solución evaluadas:

Alternativa 7.a: calcular las estadísticas cada vez que el propio jugador, administrador u otros jugadores accedan a sus estadísticas

Ventajas:

- Las estadísticas del jugador únicamente se actualizarán cuando alguno de estos 3 actores entrara en las estadísticas del jugador y solo en caso de que jugara más partidas.
- El único cálculo que debería hacer el sistema en caso de no necesitar actualizar las estadísticas sería su posición en el ranking.

Inconvenientes:

- Es una opción que puede hacer que el tiempo de carga de la vista sea mayor de lo esperado.
- En caso de no haber entrado en esa vista durante mucho tiempo, y haber jugado bastantes partidas, podría colapsar el sistema de consultas, ya que dependería de las partidas jugadas y más cálculos externos.

Alternativa 7.b: calcular las estadísticas cada cierto tiempo, por ejemplo cada día

Ventajas:

- Las estadísticas del jugador únicamente se actualizarán en una determinada hora, donde habría menos peticiones al sistema y haciendo que hubiera menos posibilidades de colapsar.
- Solo habría que calcular los datos con las partidas de ese mismo día, por lo tanto se reducen el número de consultas a partidas.

Inconvenientes:

- Los jugadores no verían sus Estadísticas en tiempo real si no del día anterior.
- Al hacerse todas las Estadísticas al mismo tiempo, hay posibilidad que, aunque se hagan menos consultas a partidas, se hacen más por parte de todos los jugadores. Pudiendo colapsar el sistema.

Alternativa 7.c: calcular las estadísticas después de cada partida

Ventajas:

- Las estadísticas de los jugadores que juegan una partida se calculan al terminar esa partida, minimizando el número de peticiones.
- Solo involucra una partida y el número de jugadores que hayan jugado la partida.

Inconvenientes:

- Puede ralentizar un poco el cambio de vista de final de partida.
- Puede que haya problemas con el ranking si dos partidas terminan a la vez.

Justificación de la solución adoptada

Utilizamos finalmente la opción 7.c, al tener menos posibilidades esta opción de colapsar el sistema en producción. El resto de opciones a mayor número de jugadores y partidas totales seguramente el programa acabe completamente colapsado o como mínimo ralentizado durante un tiempo.

Decisión 11: Amigos e invitaciones de amistad

Descripción del problema:

Como grupo nos encontramos que tenemos que establecer relaciones de amistad entre jugadores e invitaciones de amistad.

Alternativas de solución evaluadas:

Alternativa 11.a: Tener dos entidades separadas de amigos una de Friend y otra de FriendInvitation

Ventajas:

- Es fácil saber qué jugador es amigo de quien al hacer consultas con repository
- Es fácil saber qué invitaciones de amigo hay y consultarlas con repository

Inconvenientes:

- Tendría que implementar dos Entidades al cada una con su Controller, Service y Repository
- Los datos de amigos están separados y para que funcione la vista se deberían usar siempre las dos entidades y sus controllers y service

Alternativa 11.b: Tener una Entidad Friend con un atributo FriendState de tipo Enum que puede tomar dos valores Accepted y Pending

Ventajas:

- Solo habría que implementar una Entidad, Service, Controller y Repository
- Los datos que son importantes para el módulo de amistad(social) van a estar concentrados en una misma entidad

Inconvenientes:

- Las consultas a la BD usando JPQL son más complejas ya que tienen que tener en cuenta el Enum de tipo FriendState

Justificación de la solución adoptada

Utilizamos finalmente la opción 11.b, al tener que implementar menos métodos y tenerlos más concentrados mejora el tiempo de respuesta y la mantenibilidad con la única desventaja de que el Repository tiene que implementar Querys más complejas. La otra opción hace más complicado el mantenimiento y el entendimiento del código.

Decisión 12: Amigos Disponibles para Invitar

Descripción del problema:

Como grupo nos encontramos que tenemos que encontrar una forma de indicar que un jugador esta disponible para jugar. e invitar

Alternativas de solución evaluadas:

Alternativa 12.a: Implementar un Session Registry para saber los jugadores que están logeados en la aplicación

Ventajas:

- Permite saber cuando un jugador no está conectado permitiendo ahorrar tiempo a la hora de mandar invitaciones si el jugador no está en la web
- Permite saber cuando un jugador esta conectado por tanto facilita el lanzar invitaciones

Inconvenientes:

- Ralentiza levemente la página si hay muchos jugadores conectados
- Es bastante complejo de implementar y requiere realizar cambios en el login y el logout que de otra forma se utilizarían los por defectos de Spring

Alternativa 12.b: Añadir un atributo llamado Playing de tipo Boolean en la Entidad Player y cambiarlo en el BoardController

Ventajas:

- Solo habría que cambiar un Boolean en el controller de Board cuando se entra o sale de mesa
- Es rápido de implementar y no ralentiza la aplicación casi nada
- Permite saber en todo momento qué jugadores están ya jugando y por tanto cuales están disponibles

Inconvenientes:

- No se puede saber exactamente qué jugadores están conectados y cuáles no sólo quienes no están jugando y cuales estan jugando

Justificación de la solución adoptada

Utilizamos finalmente la opción 12.b, al tener que realizar cambios más pequeños en el código y funcionamiento de la aplicación y ralentizarla menos a pesar de permitir saber qué jugadores están o no disponibles con la única desventaja de que no sabemos qué jugadores están logeados y conectados a la pagina.

Decisión 13: Voto en partida

Descripción del problema:

Como grupo nos encontramos que tenemos el problema de cómo implementar el voto dentro de la partida o como reflejar quién puede votar o no.

Alternativas de solución evaluadas:

Alternativa 13.a: propiedad voto en player

Ventajas:

- Lo más rápido de implementar

Inconvenientes:

- Una vez terminada la partida no se sabría quien a votado

Alternativa 13.b: Entidad voto

Ventajas:

- Puedes saber qué ha votado quien en todo momento.

Inconvenientes:

- Más difícil de implementar

Alternativa 13.c: Entidad edil con un voto

Ventajas:

- Simplificación de entidad voto a un enum

Inconvenientes:

- La implementación sigue siendo compleja

Justificación de la solución adoptada

Pensamos en utilizar la opción a en un principio, pero como ya habíamos implementado los roles de la partida como una entidad simplemente añadimos la propiedad voto en la entidad edil.

Decisión 14:Problema con la relacion player y match

Descripción del problema:

En un principio teníamos hecho un ManyToOne entre player y match ya que estábamos implementado el match sin pensar que al no implementar un ManyToMany no iban a quedar registrados los match ya jugados.

Alternativas de solución evaluadas:

Alternativa 14.a: cambiar la relación ManyToOne a ManyToMany

Ventajas:

- Sería la relación ideal en esta situación

Inconvenientes:

- Difícil de implementar en la situación de desarrollo del match.

Alternativa 14.b: añadir otra relación ManyToMany para los match ya jugados

Ventajas:

- Lo más rápido de implementar

Inconvenientes:

- Una relación que no debería existir

Justificación de la solución adoptada

En el momento de esta decisión teníamos el match ya casi terminado y el cambio de esta relación afecta a todo lo implementado. Decidimos rápidamente que lo mejor sería añadir una nueva relación así no cambiar lo que ya funciona bien y adaptar el módulo de estadística a la nueva relación.

Decisión 15: Problema la partida

Descripción del problema:

Teníamos problemas al implementar las funcionalidades de la partida y las relaciones entre las entidades del juego.

Alternativas de soluciones:

Alternativa 15.a: Crear una partida temporal que se elimine una vez que acabe.

Ventajas

- Más fácil de implementar al no tener que guardar nada en base de datos y ser todo temporal.

Desventajas

- No se podría realizar búsquedas para las estadísticas globales al ser temporal y no guardarse en base de datos.

Alternativa 15.b: . Crear una partida (*Match*) que contenga todos los atributos necesarios y los guarde en la base de datos. Tendría una lista de Turnos (Turn) que contenga las acciones de cada player y que el jugador tuviera algunas propiedades de la partida.

Ventajas

- Se puede utilizar la información de turnos anteriores

Desventajas

- Las acciones se hacen como player y no como rol, es necesario tener en player propiedades que digan que rol es.

Alternativa 15.c: . Crear una partida (*Match*) que contenga todos los atributos necesarios y los guarde en la base de datos. Tendría una lista de Turnos (Turn) que contenga una Lista de ediles (Edil) un consul (Consul) y un pretor (Pretor) y estos tendrían un propiedad que sería la acción que el jugador hacer como rol.

Ventajas

- Se puede utilizar la información de turnos anteriores para las funcionalidades de la partida.
- Es un código más limpio al tener las diferentes propiedades de los roles en diferentes entidades y no todas en la entidad player.

Desventajas

- Se añaden demasiadas entidades que no son necesarias.

Justificación de la solución adoptada

Optamos por la opción c ya que decidimos no tener propiedades temporales dentro del player. Pensamos que repartirlo entre entidades iba a dejar el codigo mas limpio y ibamos a tener un mayor control del match