DP1 2020-2021

Documento de Diseño del Sistema

No times for heroes

https://github.com/gii-is-DP1/dp1-2021-2022-g7-02.git

Miembros:

- Pablo Espada Hoyo
- Javier Martínez Jaén
- Ramón Rodríguez Berejano
- Mario Rodríguez García
- Miguel Ángel Romalde Dorado
- Pedro Luis Soto Santos

Tutor: Belidia Estrada

GRUPO G3 L7-02

Versión 1

10/12/2021

Historial de versiones

Estos son ejemplo del contenido que debería tener el historial de cambios del documento a entregar a lo largo de los sprints del proyecto

Fecha	Versión	Descripción de los cambios	Sprint
10/12/2021	V1	● Creación del documento	2
13/12/2021	V2	● Añadido diagrama de dominio/diseño ● Explicación de la aplicación del patrón caché	3
29/08/2022	V3	Corrección de errores	-

++

Contents

Introducción

El proyecto es la implementación del juego No Time For Heroes. Es un juego de rol en el que no hay tablero, siendo basado en héroes que luchan contra una horda de monstruos liderada por un boss final, tratando de vencer a la horda, y posteriormente siendo el héroe con más gloria.

Es un juego fácil y rápido en el cual pasar un buen rato con amigos de forma on-line.

En una partida puede haber desde 2 hasta 4 jugadores, desarrollándose de la siguiente manera:
- Para empezar, cada jugador elige un héroe, no pudiendo haber dos del mismo tipo.
- Después se eligen dos cartas de la mano para ver quien es el jugador que empieza, siendo este el que tenga más ataque en las 2 cartas elegidas; estas dos cartas se descartan y se robarán 2 nuevas. Si hay empate, se decidirá por la edad de los jugadores, comenzando el mayor.
- Preparamos en la mesa 5 cartas de mercado del mazo destinado a estas, 3 cartas de enemigos del mazo de horda y una carta de escenario. La última carta de este mazo deberá ser el boss final.

Empieza el juego.

Los turnos se dividen en 3 fases:

- 1º Fase: el héroe utiliza sus cartas de habilidades para matar a los enemigos de la horda, cada vez que utiliza una carta de habilidad, esta va al mazo de descarte. Sus cartas irán infligiendo daño a los enemigos hasta matarlos,siempre y cuando este daño supere a sus puntos de fortaleza. Cada enemigo derrotado otorga unos puntos de gloria al jugador, y puede otorgar además puntos de gloria y/o monedas extras, que no se conocerán hasta derrotar al enemigo.
- 2º Fase: al terminar el turno, cada enemigo inflige al héroe un daño equivalente a la suma de las vidas que queden en las cartas de horda en el campo, (si haces daño a un enemigo, pero no lo matas, te hace el daño de su vida menos la vida que le has quitado).

El daño que recibe el héroe se traduce en el número de cartas del mazo de habilidades que tienes que enviar al mazo de descarte. Si no tienes suficientes cartas en el mazo, se voltea el mazo de descarte, se baraja y se restablece tu mazo de habilidades, pero el héroe pierde 1 punto de vida; si se queda sin vidas el héroe correspondiente morirá y no participará más en el juego.

- 3º Fase: es el turno de comprar en el mercado, utilizando las monedas que has recibido al matar a la horda, o de habilidades de ciertas cartas. Puedes comprar cartas de mercado, que suelen ser tanto ayudas como mejoras o armas para hacer más daño. Cada carta de mercado cuenta como una carta de habilidad más en el mazo.

Una vez terminadas las 3 fases, llega el turno al siguiente héroe. Se repondrán las cartas de mercado, hasta llegar a 5, y se añadirá un número de enemigos dependiendo de cuantos hayan sido derrotados. Si todos los enemigos de la mesa son derrotados en este turno, la carta de escenario cambiará, y se sacarán tres cartas nuevas del mazo de la horda. Si al finalizar aún quedan 1 o 2 enemigos con vida, se colocará una nueva carta de la horda.

- Las partidas suelen durar en torno a 20-30 minutos, y terminan cuando se mata al boss final de la horda.

Para elegir al vencedor de entre los héroes, se hará un recuento del número de fichas de gloria que tiene cada uno (estas fichas se consiguen con el botín de matar a las cartas de horda, o se te otorgan con las diferentes habilidades del mazo o del propio héroe) y el que tenga más será el vencedor.

Diagrama(s) UML:

Diagrama de Dominio/Diseño

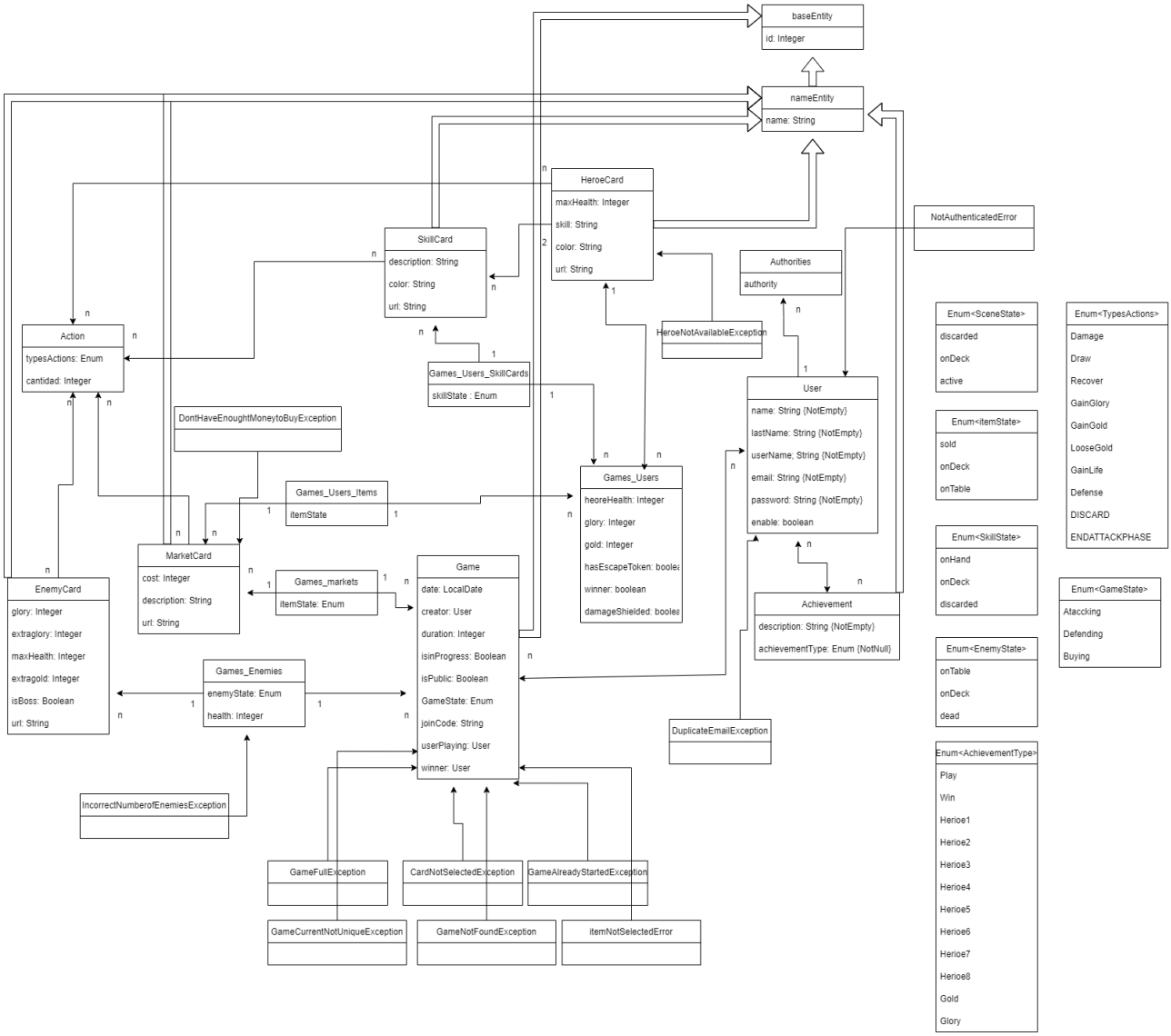
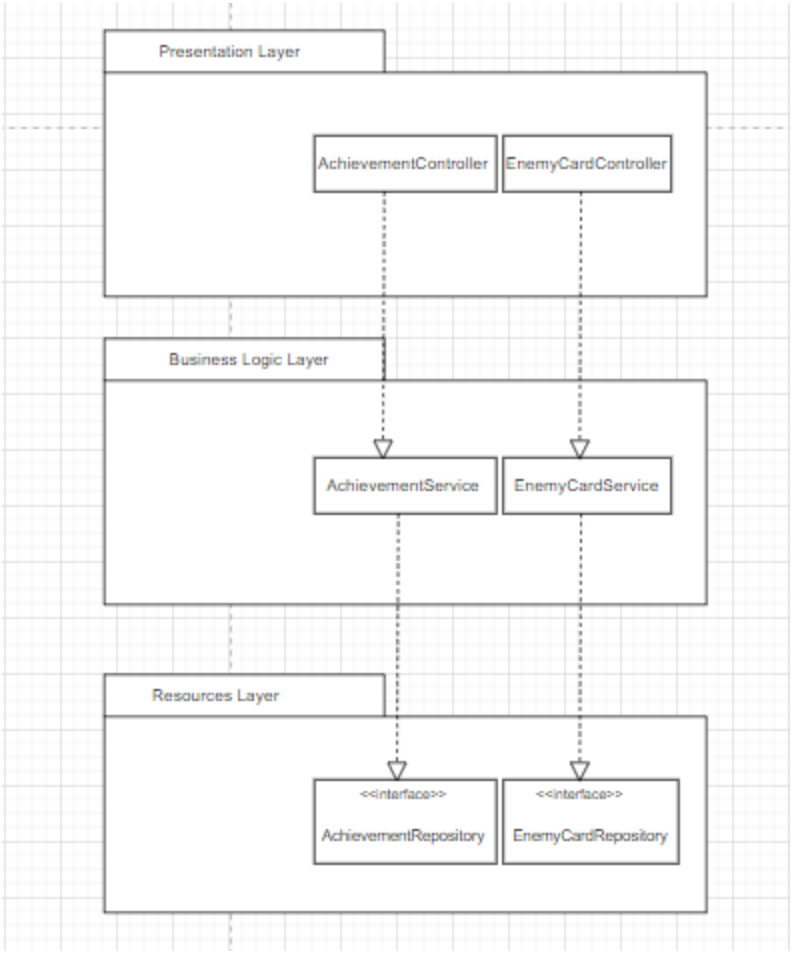
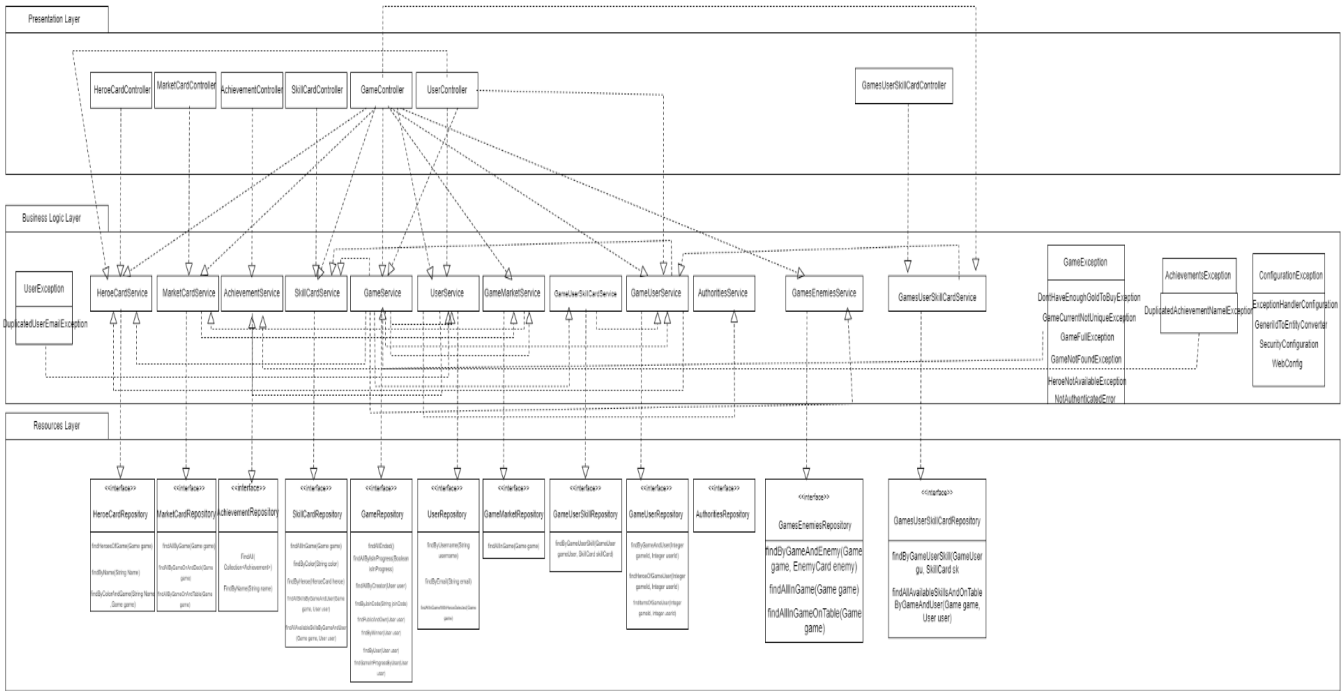


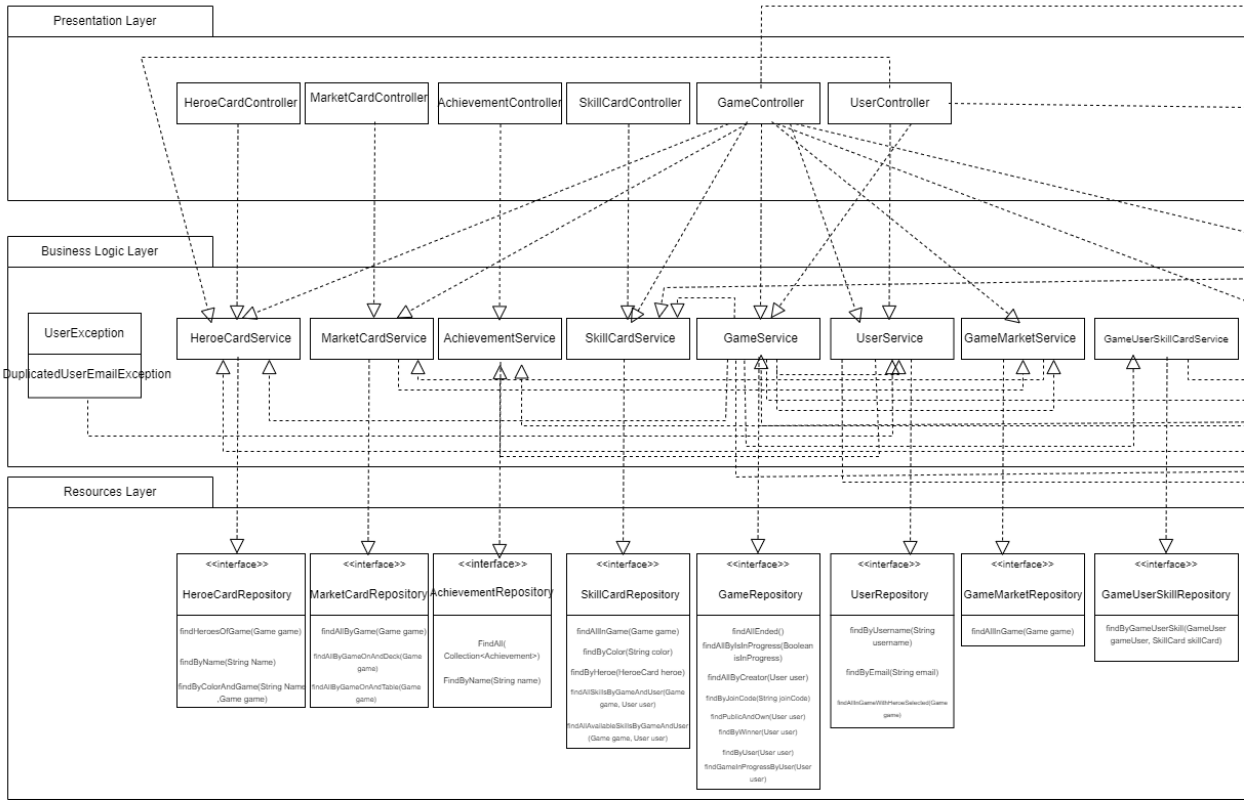
Diagrama de Capas (incluyendo Controladores, Servicios y Repositorios)

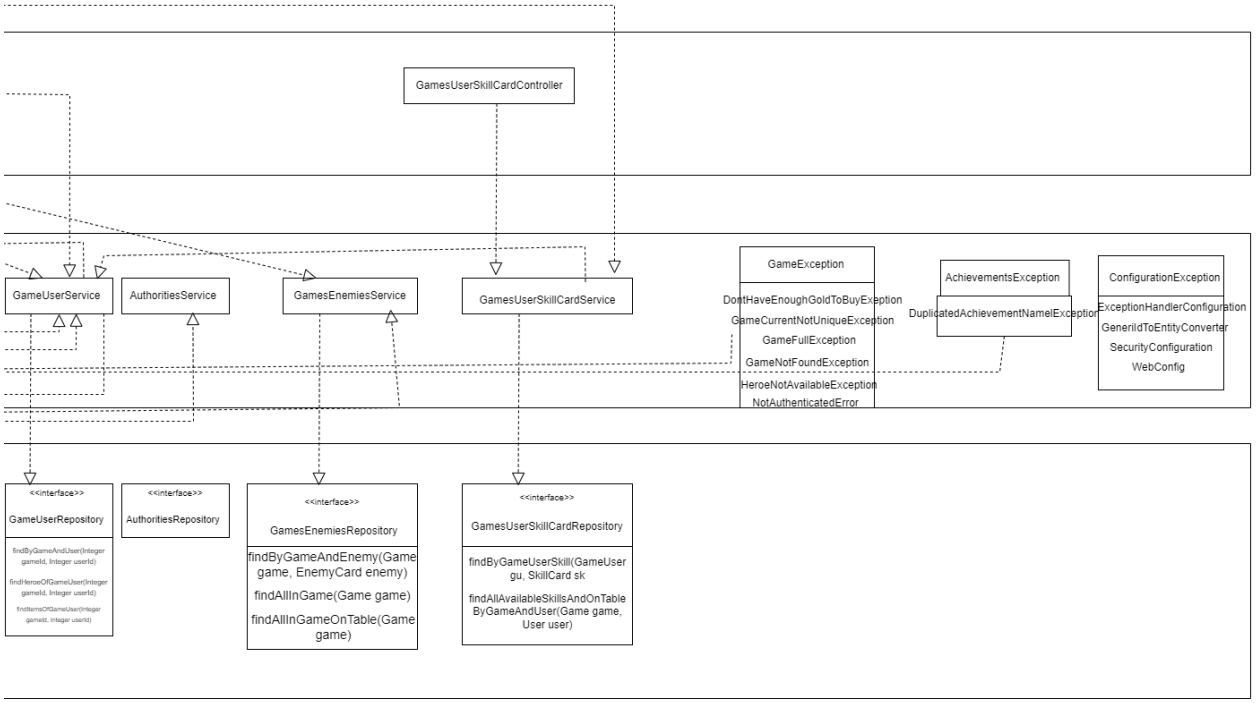
Diagrama de capas No times for Heroes





Anterior imagen dividida para poder verla mejor:


Presentation Layer
HeroeCardController
MarketCardController
AchievementController
SkillCardController
GameController
UserController
Business Logic Layer
UserException
DuplicatedUserEmailException
HeroeCardService
MarketCardService
AchievementService
SkillCardService
GameService
UserService
GameMarketService
GameUserSkillCardService
Resources Layer
<<interface>>
HeroeCardRepository
findHeroesOfGame(Game game)
findByName(String Name)
findByColorAndGame(String Name ,Game game)
<<interface>>
MarketCardRepository
findAllByGame(Game game)
findAllByGameOnAndDeck(Game game)
findAllByGameOnAndTable(Game game)
<<interface>>
AchievementRepository
FindAll(Collection<Achievement>)
FindByName(String name)
<<interface>>
SkillCardRepository
findAllInGame(Game game)
findByColor(String color)
findByHeroe(HeroeCard heroe)
findAllSkillsByGameAndUser(Game game, User user)
findAllAvailableSkillsByGameAndUser (Game game, User user)
<<interface>>
GameRepository
findAllEnded()
findAllBylsInProgress(Boolean isInProgress)
findAllByCreator(User user)
findByJoinCode(String joinCode)
findPublicAndOwn(User user)
findByWinner(User user)
findByUser(User user)
findGameInProgressByUser(User user)
<<interface>>
UserRepository
findByUsername(String username)
findByEmail(String email)
findAllInGameWithHeroeSelected(Game game)
<<interface>>
GameMarketRepository
findAllInGame(Game game)
<<interface>>
GameUserSkillRepository
findByGameUserSkill(GameUser gameUser, SkillCard skillCard)


GamesUserSkillCardController
GameUserService
AuthoritiesService
GamesEnemiesService
GamesUserSkillCardService
GameException
DontHaveEnoughGoldToBuyException
GameCurrentNotUniqueException
GameFullException
GameNotFoundException
HeroeNotAvailableException
NotAuthenticatedError
AchievementsException
DuplicatedAchievementNameException
ConfigurationException
ExceptionHandlerConfiguration
GenerilDToEntityConverter
SecurityConfiguration
WebConfig
<<interface>>
GameUserRepository
findByGameAndUser(Integer gameId, Integer userId)
findHeroeOfGameUser(Integer gameId, Integer userId)
findItemsOfGameUser(Integer gameId, Integer userId)
<<interface>>
AuthoritiesRepository
<<interface>>
GamesEnemiesRepository
findByGameAndEnemy(Game game, EnemyCard enemy)
findAllInGame(Game game)
findAllInGameOnTable(Game game)
<<interface>>
GamesUserSkillCardRepository
findByGameUserSkill(GameUser gu, SkillCard sk
findAllAvailableSkillsAndOnTable ByGameAndUser(Game game, User user)

Patrones de diseño y arquitectónicos aplicados

Patrón: MVC

Tipo: Diseño

Contexto de Aplicación

La aplicación se basa en un patrón de modelo, vista y controlador.

El modelo contiene los datos y la lógica de negocios, en nuestro caso contiene la lógica del juego de mesa con todas sus reglas y formas de jugar. En spring el modelo está formado por todos las entidades con sus respectivos servicios, toda la parte de base de datos y el repositorio el cual se encarga de acceder a la base de datos y obtener solo los datos necesarios.

La vista se encarga del diseño y la presentación de la aplicación para el usuario, es la parte con la que el usuario interactuara y se comunicará a través de estas interacciones con el modelo. La vista en spring es todo lo relacionado con el JSP el cual es la parte visual con la que el usuario interactúa.

El controlador es el encargado de manipular la información ya sea actualizando o añadiendo datos, el usuario hace uso de este a través de la vista la cual se comunica con el controlador para actualizar, añadir o eliminar datos del modelo. En este caso en spring los controladores son las clases controllers las cuales por lo general existen un controller por entidad.

Clases o paquetes creados

Indicar las clases o paquetes creados como resultado de la aplicación del patrón.

Patrón: Front Controller

Tipo: Diseño

Contexto de Aplicación

Front Controller se usa para proporcionar un mecanismo de manejo de solicitudes centralizado, de modo que todas las solicitudes sean manejadas por un único controlador. Este controlador puede realizar la autenticación/autorización/registro etc y luego pasar las solicitudes a los controladores correspondientes.

Patrón: Dependency Injection

Tipo: Diseño

Contexto de Aplicación

Dependency Injection es un patrón de diseño que se utiliza para implementar loC. Permite la creación de objetos dependientes fuera de una clase y proporciona esos objetos a una clase de diferentes maneras. Usando DI, movemos la creación y vinculación de los objetos dependientes fuera de la clase que depende de ellos.

Patrón: Proxy

Tipo: Diseño

Contexto de Aplicación

Un proxy, en su forma más general, es una clase que funciona como una interfaz para otra cosa. El proxy podría interactuar con cualquier cosa: una conexión de red, un objeto grande en la memoria, un archivo o algún otro recurso que sea costoso o imposible de duplicar. En resumen, un proxy es un contenedor o un objeto de agente que el cliente está llamando para acceder al objeto de servicio real detrás de escena.

Patrón: Domain Model

Tipo: Diseño

Contexto de Aplicación

El modelo de dominio se crea con el fin de representar el vocabulario y los conceptos clave del dominio del problema. El modelo de dominio también identifica las relaciones entre todas las entidades comprendidas en el ámbito del dominio del problema, y comúnmente identifica sus atributos.

Patrón: Service Layer

Tipo: Arquitectónico

Contexto de Aplicación

La capa de servicio es un patrón arquitectónico, aplicado dentro del paradigma de diseño de orientación de servicio, que tiene como objetivo organizar los servicios, dentro de un inventario de servicios, en un conjunto de capas lógicas. Los servicios que se clasifican en una capa particular comparten funcionalidad.

Patrón: Paginación

Tipo: Diseño

Contexto de Aplicación

La paginación en sitios web es una forma de estructurar el contenido, agrupando por una cantidad fija de espacio o cantidad de elementos. La paginación es simplemente el número de páginas que se muestran en la parte inferior de una página web que sirve para separar el contenido y facilitar la navegación.

Patrón: MetaDataMapper

Tipo: Diseño

Contexto de Aplicación

Gran parte del código que se ocupa del mapeo relacional de objetos describe cómo los campos de la base de datos se corresponden con los campos de los objetos de memoria. El código resultante tiende a ser repetitivo de escribir. Una asignación de metadatos permite a los desarrolladores definir las

asignaciones en una forma simple, que luego puede procesarse mediante un código genérico para llevar a cabo los detalles de lectura, inserción y actualización de los datos.

Patrón: Layer SuperType

Tipo: Diseño

Contexto de Aplicación

Layer Supertype es una idea simple que conduce a un patrón muy corto. Todo lo que necesita es una superclase para todos los objetos en una capa. Por ejemplo, todos los objetos de dominio en un Modelo de dominio pueden tener una superclase llamada Objeto de dominio. Las funciones comunes, como el almacenamiento y la gestión de los campos de identidad, pueden ir allí.

Patrón: Repository

Tipo: Diseño

Contexto de Aplicación

El patrón de repository ofrece una buena forma de integrar las necesidades de persistencia de datos. Se encarga de gestionar todas las operaciones de persistencia contra una tabla de la base de datos.

En el caso de nuestra aplicación, hemos implementado el patrón repository para poder consultar datos de partidas y jugadores para poder trabajar con estos datos o consultar información necesaria.

Patrón: Chain of Responsibility

Tipo: Diseño

Contexto de Aplicación

El patrón Chain of Responsibility es un patrón de diseño que permite pasar solicitudes a lo largo de una cadena de manejadores. Cuando un manejador recibe una solicitud, debe decidir si la procesa o la pasa al siguiente manejador

Estilo: Capas

Tipo: Arquitectónico

Contexto de Aplicación

Las capas del diseño pasan a formar el modelo de capas; capa de presentación(Vista y controlador), capa de negocio(Modelo) y capa de datos(Modelo). En este caso el modelo contiene parte de capa de negocio y capa de datos, la parte de entidades y servicios forma la capa de negocio y la parte del repositorio y la base de datos pasa a ser la capa de datos.

Ventajas alcanzadas al aplicar el patrón

El uso de este modelo nos ha facilitado el poder cambiar o añadir continuamente nuevas funcionalidades o métodos sin que afectara a toda la aplicación gracias a la forma de comunicación entre capas y su bajo acoplamiento y su alta cohesión.

Decisiones de diseño

Decisión 1: Separación de los tipos de cartas

Descripción del problema:

Como grupo hemos tenido problemas a la hora de implementar el tipo de carta ya que al principio lo realizamos mediante una clase carta siendo esta padre y teniendo 3 atributos comunes para todas las cartas y los demás tipos de cartas heredan de esta. Pero esto suponía un problema con el tema de relaciones a entidades ya que una clase padre da incompatibilidad con algunos tipos de relaciones.

Alternativas de solución evaluadas:

Alternativa 1.a: Hemos realizado cada tipo de carta por separado, ya que algunas cartas también tienen una serie de acciones que son diferentes entre los tipos de cartas que hay.

Ventajas:

- Ahora sí podemos asignar las habilidades del héroe al jugador.

Inconvenientes:

- Hemos tenido que repetir algunos atributos que están en algunos tipos de cartas comunes.
- Nos facilita todo el tema de relaciones entre tipos de cartas y otras entidades ya que cada tipo de carta tiene unas relaciones.

Alternativa 1.b: Añadir todas las cartas como un mismo tipo y añadir un atributo con un enumerate que catalogue al tipo de carta y las diferencia de otra.

Ventajas:

- Crearemos una sola entidad carta.

Inconvenientes:

- Cada tipo de carta tiene relaciones con diferentes tipos de entidades por lo que queda descartado solo por esta razón, las entidades no podrían completarse o se completan demasiadas relaciones para una sola entidad añadiendo muchas que son innecesarias.

Justificación de la solución adoptada

Hemos elegido la alternativa a ya que era la única que nos permitía hacerlo lo más rápido posible para poder seguir avanzando con el proyecto, además que simplifica el trabajo de tablas intermedias entre relaciones manyToMany.

Decisión 2: Uso de cartas mediante acciones

Descripción del problema:

Como grupo hemos tenido problemas a la hora de implementar el uso de las cartas, debido a que no todas las cartas hacen lo mismo si no que hacen una o varias determinadas acciones hemos decido crear acciones implementadas individualmente y relacionarlas con las cartas que hagan uso de esta.

Alternativas de solución evaluadas:

Alternativa 1.a: Hemos realizado cada tipo de acción por separado, para posteriormente asociar cada carta a la o las acciones que realizan al ser usadas.

Ventajas:

- Podemos asignar acciones a cartas indiferentemente de la acción básica que contiene cada carta.
- Ahorro de código repetido.

Inconvenientes:

- Es más complejo asignar las acciones a las cartas.

Alternativa 1.b: Podríamos añadir a la entidad skill un atributo que defina la acción y tras su uso que la diferencia y ejecute.

Ventajas:

- Sería más sencillo ya que nos ahorramos la entidad Action y con ello todas las relaciones y la tabla intermedia.

Inconvenientes:

- Realmente no podemos hacer esto ya que hay cartas con varias acciones y muchas de ellas con lógica única, lo cual no nos permitiría hacer un método general para el uso de cartas con lógica más diferencial.

Justificación de la solución adoptada

Hemos elegido la alternativa A, ya que era la más eficaz a la hora de implementar el uso de acciones, nos ahorramos implementar cada carta por separado y así no repetir código ya que muchas de estas comparten alguna acción, aunque se complique más el tema relacional.

Decisión 3: Dividido el desarrollo de la partida en casos

Descripción del problema:

Como grupo hemos tenido problemas con el uso de las url para cada fase de la partida, al tener una url para cada fase podían saltarse fases al acceder a la url directamente.

Alternativas de solución evaluadas:

Alternativa 1.a: Hemos juntado las fases en la misma url y hemos cambiado las fases con el uso de cases.

Ventajas:

- Ahorramos métodos y movimientos entre url innecesarios.

Inconvenientes:

- El método en el controller al usar una url en vez de una por fase se queda un poco más largo.

Alternativa 1.b: Como ya hemos comentado la otra alternativa era la del uso de una url por cada fase

Ventajas:

- Los métodos quedarían más ordenados y nos ahorramos el uso de casos el cual puede a llegar a ser un poco complicado.

Inconvenientes:

- Era más complicado el no permitir al jugador saltarse fases obligatorias.

Justificación de la solución adoptada

Hemos elegido la alternativa A, ya que era la más intuitiva para controlar el flujo del jugador en el juego y asegurar que este juegue todas las fases obligatorias.

Decisión 4: Conversión de cartas de mercado a skill tras la compra

Descripción del problema:

Las cartas de mercado no son más que cartas de skill, solo que el jugador debe comprarlas antes de usarlas, pero nosotros las tenemos como diferentes entidades ya que las de mercado contienen atributos que las de skill no tienen.

Alternativas de solución evaluadas:

Alternativa 1.a: Añadir las cartas de mercado como cartas de skill desde la selección del héroe al principio de la partida con las cartas de su mazo, estas cartas de mercado quedan con un estado de la carta que las oculta por lo que no pueden ser usadas ni robadas hasta que el usuario compre la carta y esta se actualiza.

Ventajas:

- Es la única manera de actualizar el mazo con las cartas de mercado tras la compra de esta sin que tengamos que crear otro mazo desde 0 en mitad de la partida, así no se pierde la información del mazo y sus cartas de estado.

- Seguimos manteniendo las dos entidades MarketCard y SkillCard solo que añadimos un findByName() para que al comprar la MarketCard encuentre su correspondiente SkillCard y actualice su estado.

Inconvenientes:

- Hay que añadir muchas entidades en la base de datos a mano.

Alternativa 1.b: Crear un método que a través de los atributos de market cree una nueva carta skill y la añada al mazo del jugador tras comprarlo.

Ventajas:

- Es el método más intuitivo.

Inconvenientes:

- Es un poco complicado añadir una carta de skill al mazo con la partida empezada, ya que estas tienen relaciones con otras tablas y se perderían datos al actualizar el mazo.

Justificación de la solución adoptada

Hemos elegido la alternativa a ya que era la única que no estropea las relaciones de username y cartas ya que añade las cartas al principio con un estado oculto que no afecta al mazo del jugador, solo tras la compra del objeto se actualiza solamente este y ya puede ser usada por el usuario.

Decisión 5: Añadidos metodos de acciones personalizadas

Descripción del problema:

Las cartas tienen acciones muy variadas, algunas son más generales como las típicas de hacer daño o recuperar vidas, pero otras necesitan de lógica adicional para una única carta.

Alternativas de solución evaluadas:

Alternativa 1.a: Añadir los métodos generales de las acciones y crear luego los métodos específicos para cada carta.

Ventajas:

- Es la manera más visual y organizada de crear los métodos de acción de una carta, ya que cada carta con lógica adicional tendrá su propio método y se podrá encontrar fácilmente al estar nombrado por su nombre.

Inconvenientes:

- Muchos métodos para tan solo unas acciones que aparecen una vez.
- Tener que comprobar continuamente si la carta usada usa lógica adicional y llamar al método de esta al usarla.

Alternativa 1.b: Dividir las acciones más específicas en un conjunto de generales.

Ventajas:

- Ahorramos métodos para una sola carta que usa una acción en específico.

Inconvenientes:

- No podemos dividir todas las acciones específicas en varias generales ya que algunas son demasiado complejas o no depende de acciones básicas del juego.

Justificación de la solución adoptada

Hemos elegido la alternativa A, ya que era la única que realmente nos dejaba crear todas las acciones de las cartas más complejas aunque se haya tenido que crear muchos métodos para tan solo unas pocas cartas.

Decisión 6: El creador puede eliminar a otros jugadores si estos dejan de jugar

Descripción del problema:

Otro problema que se planteó fue el hecho de actuar con la caída o con la inactividad de algún jugador, ya que si un jugador no quería seguir jugando y no se salía de la partida cuando llegue su turno la partida no avanzara más.

Alternativas de solución evaluadas:

Alternativa 1.a: Dar el poder al creador de poder expulsar de la partida a los demás jugadores si ve que ha dejado de jugar.

Ventajas:

- Es una manera fácil de resolver el problema.

Inconvenientes:

- Puede que el creador no sea paciente y expulse antes de tiempo.

Alternativa 1.b: Crear un timer que empezase una cuenta atrás desde el inicio de un turno y si pasaban x segundos eliminase al jugador.

Ventajas:

- Se quita de responsabilidad al creador.

Inconvenientes:

- Incompatibilidades y lag durante la partida.

Justificación de la solución adoptada

Hemos elegido la alternativa a, primero probamos con la alternativa b pero la estabilidad de la partida se veía afectada, iba muy lagueada y a veces daba incompatibilidad con la BD al estar continuamente actualizando.