DP1 2020-2021

Documento de Diseño del Sistema

Proyecto Dobble

https://github.com/gii-is-DP1/dp1-2021-2022-ling-5

Miembros:

- Blanco Mora, David
- Cabra Morón, Manuel
- Claros Barrero,Fernando
- Domínguez Garoz, Irene
- Ortega Soldado, Gregorio
- Sampedro Bernal, Marta

Tutor: José María García Rodríguez

GRUPO LIng-5

Versión 1

6/01/2022

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**
12/11/2021	V1	● Creación del documento	2
03/01/2022	V1	● Adecuación del documento a nuestro proyecto	3
05/01/2022	V1	● Diagrama UML	3
06/01/2022	V1	● Decisiones de diseño ● Patrones de diseño	3
06/01/2022	V1	● Diagrama de capas	3

Contents

Introducción

Nuestro proyecto será una versión web del juego Dobble. El objetivo del juego siempre será el mismo, buscar el símbolo idéntico entre dos cartas. Durante el juego habrá más de 50 símbolos, 55 cartas y 8 símbolos por cartas. Implementaremos varios modos de juego los cuales llamaremos minijuegos. Se necesitan al menos 2 jugadores hasta un máximo de 8 para comenzar una partida.

Incluiremos tres minijuegos. El primero de ellos será la torre infernal: cada jugador debe descubrir si su carta tiene un dibujo en común con la del centro, si esto es así, entonces el jugador se lleva la carta del centro, una vez ya no queden más cartas en el centro, ganará el que más cartas tenga. El segundo, el foso: el primer jugador que detecte que comparte un símbolo con la carta del centro, se descarta de su carta; ganará el primer jugador que se quede sin ninguna. Y el último, el regalo envenenado: cada jugador tiene que encontrar el símbolo idéntico entre la carta de cualquiera de sus compañeros y la primera del mazo; el primero que encuentre el símbolo roba la carta central y se la coloca al jugador en cuestión.

La obtención de puntos y reglas serán distintas en cada minijuego y se especificarán al comienzo de la partida. El ganador final será aquel que haya conseguido la mayor puntuación.

El juego incluirá entre sus opciones la identificación de los usuarios para jugar al Dobble, crear partidas, unirse a ellas, enviar solicitudes de amistad, conseguir logros y recompensas al alcanzar ciertos objetivos, comentar las partidas en un foro o incluso ver las estadísticas por jugador. También habrá un administrador que será el encargado de controlar y supervisar el juego y auditar los datos de todos los usuarios de la aplicación. Por último, habrá un modo espectador para que los distintos usuarios puedan acceder a las partidas en curso.

Diagrama(s) UML:

Diagrama de Dominio/Diseño

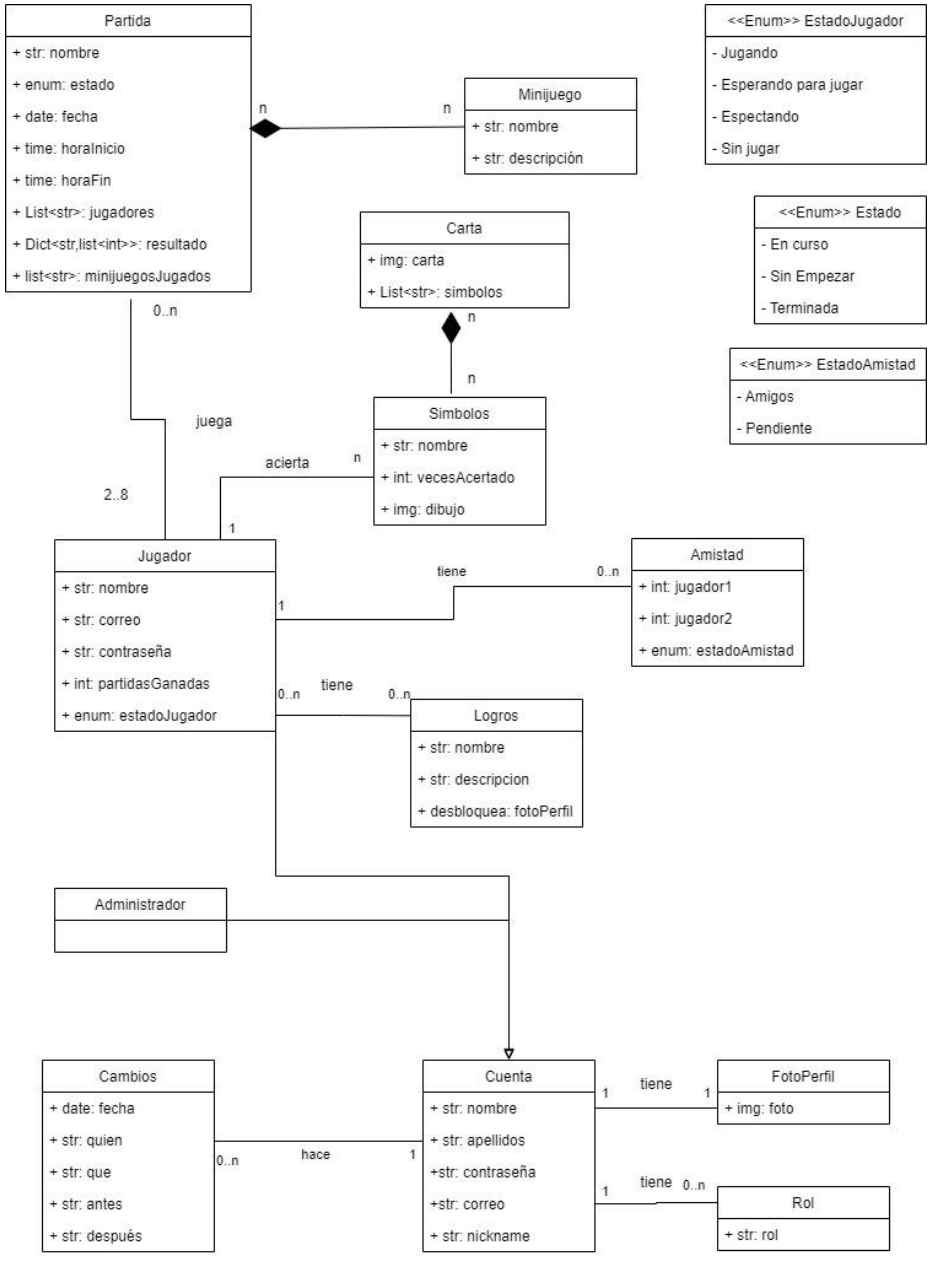
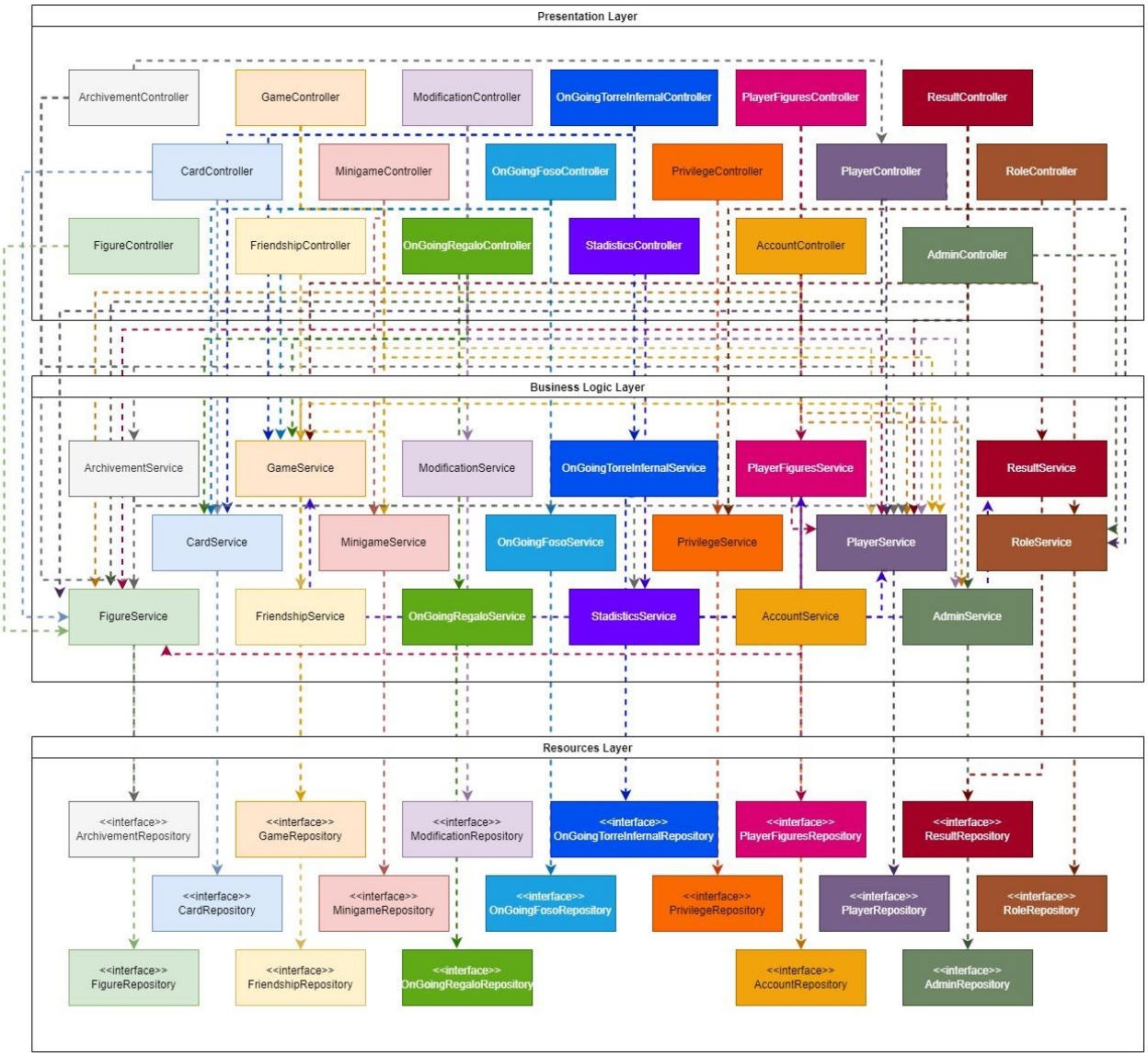


Diagrama de Capas (incluyendo Controladores, Servicios y Repositorios)



Patrones de diseño y arquitectónicos aplicados

Patrón: Monolítico

Tipo: Arquitectónico

Contexto de Aplicación

El proyecto se ha desarrollado acorde con este patrón, tenemos 3 capas: la de presentación, la lógica y la de recursos. La capa de presentación y lógica se comunican mediante una API, y la lógica con la de recursos mediante Spring y Hibernate.

Ventajas alcanzadas al aplicar el patrón

Sencillez estructural y capas bien definidas entre sí, las capas son independientes entre sí.

Patrón: Singleton

Tipo: de Diseño

Contexto de Aplicación

Se ha aplicado a las partes de la aplicación encargadas de manejar los datos de un minijuego en curso mientras se está jugando.

Clases o paquetes creados

Los paquetes: “com.example.accessingdatamysql.ongoingminigame”, “com.example.accessingdatamysql.ongoingRegaloEnvenenado” y “com.example.accessingdatamysql.ongoingfoso”

Ventajas alcanzadas al aplicar el patrón

Mayor velocidad para procesar las peticiones a la API puesto que no necesitan consultar la base de datos.

Decisiones de diseño

Decisión 1: Uso del Modelo de Dominio

Descripción del problema:

Teníamos que decidir cómo organizar la capa lógica de negocio.

Alternativas de solución evaluadas:

Alternativa 1.a: Transacción de datos

Ventajas:

- Útil para lógica simple.
- Fácil de implementar.

Inconvenientes:

- Tiende a duplicar código.
- Difícil de mantener si el dominio lógico es complejo.

Alternativa 1.b: Módulo de tablas

Ventajas:

- Fácil de mapear en la estructura de la base de datos.

Inconvenientes:

- No te da todo el poder de los objetos (no hay relaciones ni herencias).

Alternativa 1.c: Modelo de dominio

Ventajas:

- Permite implementar lógica de negocio compleja.

- Soportado por la mayoría de frameworks.

Inconvenientes:

- El mapeo a la base de datos es más complicado.

Justificación de la solución adoptada

Finalmente nos decidimos por la última alternativa ya que es la más usada y la más completa.

Decisión 2: Uso de la Capa de Servicios

Descripción del problema:

Se nos planteaban casos de usos que involucraban diferentes entidades y había que decidir si usar o no una capa de servicios.

Alternativas de solución evaluadas:

Alternativa 2.a: Diseño impulsado por dominio

Ventajas:

- Agilidad como principio.
- Puramente flexible.
- Mejor código.
- Mantenibilidad.

Inconvenientes:

- Costo relativamente alto.
- Curva de aprendizaje alta
- Sugerido para aplicaciones donde el dominio sea complejo, no para simples CRUDs.

Alternativa 2.b: Capa de servicios

Ventajas:

- Reduce el nivel de acoplamiento.
- Fácil testeo.
- Mantenibilidad, reutilizabilidad, escalabilidad, interopabilidad.
- Mapeo directo entre procesos y sistemas.
- Favorece el desarrollo en paralelo.

Inconvenientes:

- Depende de la implementación de estándares.
- No es para aplicaciones con alto nivel de transferencia de datos.
- Incrementalmente se hace difícil y costoso el ser capaz de cumplir con los protocolos y hablar con un servicio.
- Implica conocer los procesos del negocio, clasificarlos, extraer las funciones, estandarizarlas y formar con ellas las capas de servicios.

- En la medida en que un servicio de negocio, vaya siendo incorporado en la definición de los procesos de negocio, dicho servicio aumentará su nivel de criticidad.

Justificación de la solución adoptada

Debido a todas las ventajas planteadas y también al conocer más ese tipo de organización, finalmente decidimos optar por usar la capa de servicios.

Decisión 3: Uso de Data Mapper para Modelo de Dominio

Descripción del problema:

Se nos plantea cómo podemos mover los datos de los objetos Java en memoria a la base de datos y viceversa

Alternativas de solución evaluadas:

Alternativa 3.a: Un objeto que envuelve una fila en una tabla o vista de la base de datos, encapsula el acceso a la base de datos y añade lógica de dominio a esos datos.

Ventajas:

- Útil para modelos de dominio sencillos

Inconvenientes:

- Casi imposible de mantener en aplicaciones de nivel empresarial donde varias personas trabajan en el mismo proyecto pues una sola clase contiene lógica de interacción de bajo nivel con la base de datos, lógica de negocio y lógica de validación.

Alternativa 3.b: Un objeto que mueve los datos entre objetos y una base de datos manteniéndolos independientes entre sí y del propio mapeador

Ventajas:

- Usado con el patrón de mapeo de metadatos, este mapeo se realiza automáticamente.
- Cuando se trabaja en el modelo de dominio se puede ignorar la base de datos, tanto en el diseño como en el proceso de construcción y prueba.
- Puedes entender y trabajar con los objetos del dominio sin tener que entender cómo están almacenados en la base de datos

Inconvenientes:

- Capa extra
- Complejidad de la lógica de negocio

Justificación de la solución adoptada

Debido al uso de un Modelo de Dominio que haremos y la complejidad de nuestro proyecto consideramos el uso de Data Mapper para refactorizar el comportamiento de la base de datos en una capa separada.

Decisión 4: Uso de la estrategia MappedSuperclass en la herencia

Descripción del problema:

En nuestro proyecto, tenemos dos tipos de usuarios (administrador y jugador), ambos tienen atributos comunes por lo que lo más conveniente era usar la herencia.

Alternativas de solución evaluadas:

Alternativa 4.a: MappedSuperclass

Ventajas:

- Simple, la clase hija extiende de la clase padre y su tabla tendrá las columnas declaradas en la propia entidad más las heredadas.

Inconvenientes:

- Las clases padre no pueden ser entidades, por lo que no persistirán en la base de datos por ellas mismas.
- Las clases padre no pueden contener asociaciones con otras entidades.

Alternativa 4.b: Estrategia Single Table

Ventajas:

- Usada por defecto.

Inconvenientes:

- Todas las subclases se almacenan en la misma tabla especificando el valor discriminador que nos dirá de qué tipo es para diferenciarlas.

Alternativa 4.c: Estrategia Joined Table

Ventajas:

- Cada clase en la jerarquía se mapea en su propia tabla.

Inconvenientes:

- Para encontrar entidades se necesita hacer “join” entre tablas, lo que puede dar lugar a un menor rendimiento para cantidades grandes.
- El número de “joins” es mayor consultando a la clase padre porque se unirá con cada hijo.

Alternativa 4.d: Estrategia Table per Class

Ventajas:

- Mapea cada entidad en su propia tabla la cual contiene todos los atributos de la entidad, incluyendo los heredados.
- Similar a *MappedSuperclass* solo que la clase padre será también una entidad, lo que permitirá asociaciones y consultas polimórficas.

Inconvenientes:

- El uso de “UNION” puede conllevar un rendimiento inferior.
- No se puede usar la generación de claves de identidad.

Justificación de la solución adoptada

Al principio la solución elegida fue la alternativa 4.d debido a que parecía más eficiente tener una tabla por entidad. Además tener la clase padre también como entidad, permitía realizar asociaciones que eran comunes en las subclases (evitaba duplicación de operaciones). Finalmente se optó por la alternativa *MappedSuperclass* debido al inconveniente de no poder usar la generación Identity en el id. Esto conllevó muchos cambios posteriores pero mereció la pena por las ventajas del tipo de generación del id ya que es mucho más eficiente, el id se incrementa automáticamente y está altamente optimizado.

Decisión 5: Uso de Layer Supertype

Descripción del problema:

Nos planteamos, al tener muchas clases con el mismo campo de identidad, si usar una clase abstracta común para todas las entidades que contenga el campo de identidad

Alternativas de solución evaluadas:

Alternativa 5.a: Uso de Layer Supertype para los campos *NamedEntity* y *BasedEntity*

Ventajas:

- Evitar duplicación de código

Inconvenientes:

- Dificultad de organizar la interacción entre las clases
- Las consultas SQL se vuelven más complejas

Alternativa 5.b: Uso de Bloated Domain Model

Ventajas:

- Fácil para un Modelo de Dominio sencillo

Inconvenientes:

- Duplicación de código
- Cada clase modela diferentes tipos de objetos que implementan interfaces dispares, lo que significa que es algo difícil abstraer la lógica sin terminar con una jerarquía torpe

Justificación de la solución adoptada

Dada la cantidad de clases que tenemos en una misma capa consideramos que el uso de Supertype nos permite eliminar grandes porciones de implementaciones duplicadas sin mucha preocupación durante todo el proceso de refactorización.

Decisión 6: Uso de singleton

Descripción del problema:

A la hora de jugar, datos tales como las cartas que se van usando o los puntos, deben guardarse de alguna forma.

Alternativas de solución evaluadas:

Alternativa 6.a: Guardar los datos en la BD de la misma forma que se guardan otros tipos de datos como los jugadores, partidas, amigos, logros, etc.

Ventajas:

- Sencillo ya que se trataría de hacer lo mismo que ya se hizo previamente con otras entidades.

Inconvenientes:

- Los datos se guardarían de forma permanente y una vez terminada la partida no servirían de mucho además de consumir mucho espacio.

Alternativa 6.b: Usar modelo singleton

Ventajas:

- Se pueden guardar los datos de forma temporal y se destruirían al finalizar la partida.
- Ayuda a mejorar el rendimiento del sistema.

Inconvenientes:

- La responsabilidad de la clase singleton es demasiado pesada, lo que viola el “principio de responsabilidad única”

Justificación de la solución adoptada

Finalmente se optó por el modelo singleton ya que aportaba lo que se necesitaba.