DP1 2021-2022

Documento de Diseño del Sistema

Proyecto **Parchis&Oca**

https://github.com/dantolvil/dp1-2021-2022-g1-septiembre

Miembros:

- Toledo Villalba, Daniel

Tutor:

- Parejo, José Antonio

GRUPO G1-septiembre 2022

Versión 3.2

01/09/2022

Historial de versiones

Fecha	Versión	Descripción de los cambios	Sprint
01/05/2022	V1.1	● Creación del documento. ● Añadidos primeros puntos y configuración del documento.	Entregable Septiembre
24/05/2022	V1.2	● Añadidos diagrama UML de dominio/diseño.	Entregable Septiembre
08/06/2022	V2.1	● Añadidas las decisiones 1 y 2. ● Añadido diagrama de capas. ● Actualizado diagrama UML.	Entregable Septiembre
22/06/2022	V2.2	● Añadido diagrama de capas. ● Actualizadas las decisiones	Entregable Septiembre
04/07/2022	V2.3	● Actualizados los diagramas de capas y UML.	Entregable Septiembre
22/07/2022	V2.4	● Añadidos los patrones arquitectónicos FrontController, Domain Model, Repository, Service Layer, Layer Supertype.	Entregable Septiembre
12/08/2022	V3	● Actualizado diagrama de capas. ● Actualizado diagrama UML de dominio.	Entregable Septiembre
25/08/2022	V3.1	● Actualizadas las decisiones de diseño. ● Añadidas últimas modificaciones pre entrega para cerrar el documento.	Entregable Septiembre
01/09/2022	V3.2	● Revisión del documento y actualización de los puntos faltantes. ● Finalización del documento	Entregable Septiembre

Índice

Introducción

La idea principal de la plataforma Parchis&Oca es la de crear un sistema informático en el cual los usuarios puedan registrarse en el sistema como jugadores para jugar a dos juegos clásicos: Parchís y Oca.

Las funciones principales son las de implementar un sistema para jugar a estos dos juegos de modo que los usuarios que se registren en la plataforma puedan iniciar un juego y seleccionar el juego que deseen jugar en cada momento. Además, los administradores serán capaces de gestionar la aplicación con los diferentes juegos y jugadores. Todos los datos de los jugadores, tablero de juego, juegos y administradores serán almacenados en la base de datos.

Diagrama(s) UML:

Diagrama de Dominio/Diseño

Todos los modelos incluidos en el proyecto heredan de la entidad BaseEntity, se ha incluido esta estructura de entidades/relaciones hacia BaseEntity:

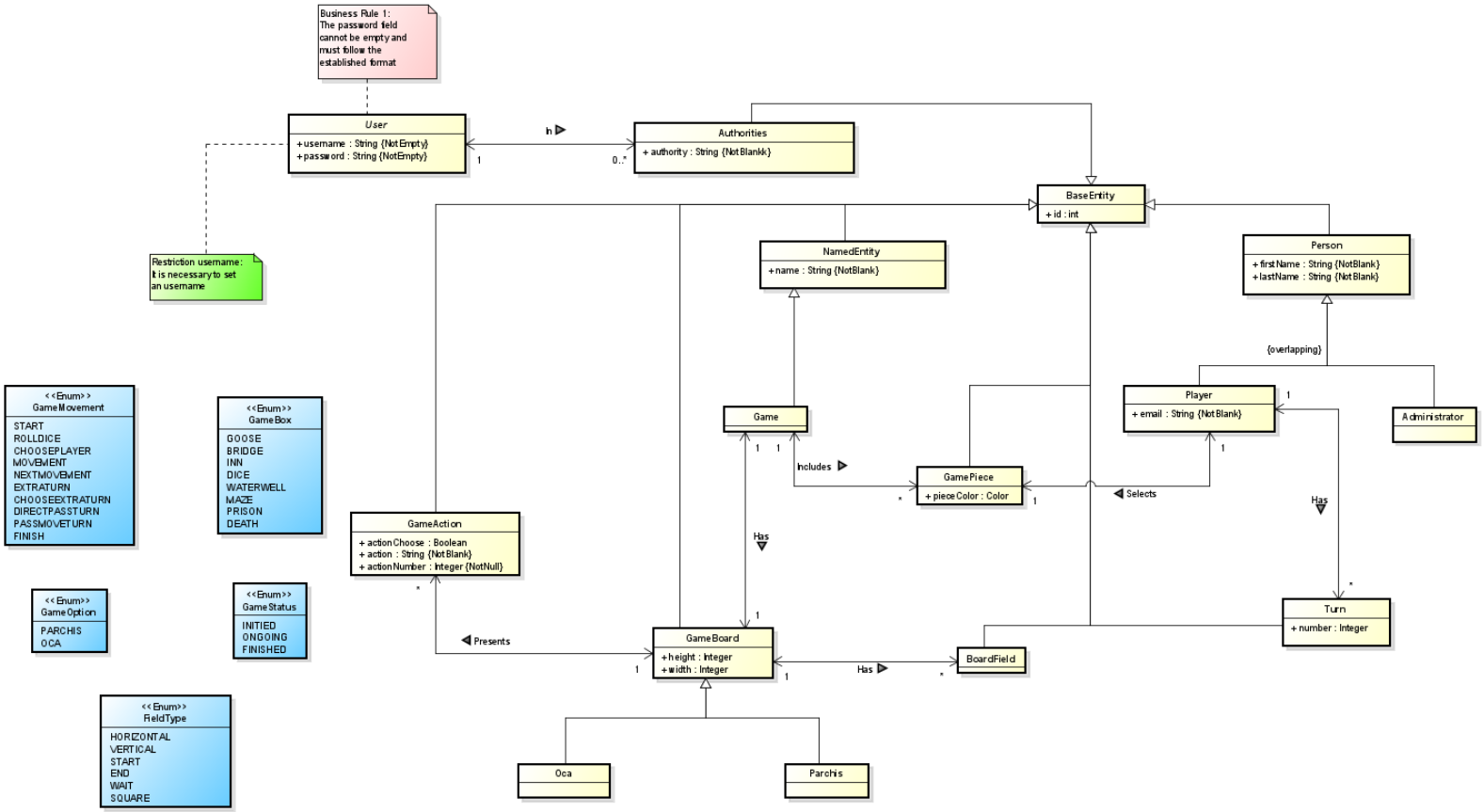
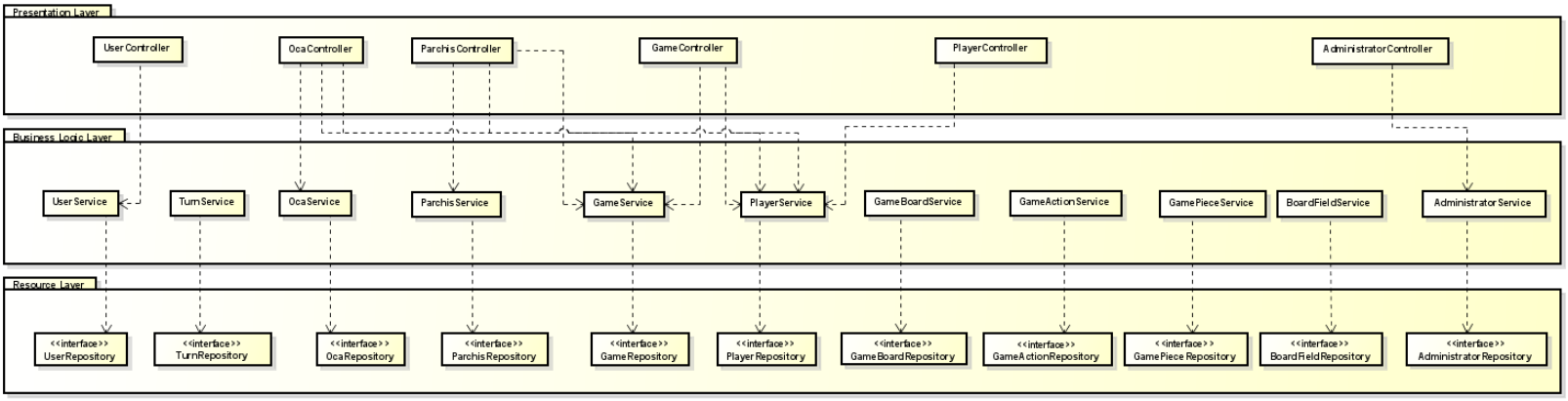


Diagrama de Capas (incluyendo Controladores, Servicios y Repositorios)

En este apartado se muestran las relaciones entre los controladores, servicios y repositorios mediante las capas que se han implementado y forman la aplicación.



Patrones de diseño y arquitectónicos aplicados

Patrón: Modelo Vista Controlador (MVC)

Tipo: de Diseño

Contexto de Aplicación

El proyecto se ha estructurado según el patrón MVC (Modelo-Vista-Controlador).

Las clases de modelo representan el modelo del proyecto de cada una de las entidades, los archivos.jsp representan las vistas y las clases Controller para los controladores.

Clases o paquetes creados

En este punto se indican las clases o paquetes creados como resultado de la aplicación del patrón.

- model(Modelos): Administrator, Authorities, BaseEntity, BoardField, Game, GameAction, GameBoard, GamePiece, NamedEntity, Player, Oca, Parchis, Person, Turn, User.

- web(controladores): AdministratorController, DicesOnSessionController, CrashController, GameController, OcaController, ParchisController, PlayerController, UserController, WelcomeController.

- jsp(vistas): Se han dividido en tres carpetas: administrators, game y players.

    1. En la carpeta administrators:
       Se han creado los archivos adminCreatePlayer.jsp, adminEditProfile.jsp y adminPlayerDetails.jsp.

    2. En la carpeta game:
       Se han creado los archivos createGameForm.jsp, ocaGame.jsp y parchisGame.jsp.

3. En la carpeta players:
   Se han creado los archivos createPlayerForm.jsp y editPlayerProfileForm.jsp.

Ventajas alcanzadas al aplicar el patrón

- Identificación de forma clara del tipo de lógica aplicado en cada parte, facilidad de mantenimiento y escalabilidad de la aplicación.
- Facilidad para implementar distintas representaciones de los mismos datos.
- Facilidad para la realización de pruebas unitarias de cada uno de los componentes y para aplicar un desarrollo guiado por pruebas.
- Reutilización de los componentes desarrollados en el modelo.
- Facilidad para el diseño de la aplicación web.
- Gran control sobre el comportamiento de la aplicación web.

Patrón: Front Controller.

Tipo: de Diseño

Contexto de Aplicación

Se ha usado para mostrar los datos en la interfaz de usuario, para la creación de las vistas necesarias para el proyecto y para establecer las URLs del mismo.

Se encuentra en el paquete /springframework/samples/parchis_oca/web

Clases o paquetes creados

AdministratorController, DicesOnSessionController, CrashController, GameController, OcaController, ParchisController, PlayerController, UserController, WelcomeController.

Ventajas alcanzadas al aplicar el patrón

- Flexibilidad a la hora de establecer un controlador apropiado para las solicitudes a través de las URLs.
- Procesar los datos de los formularios y poder validarlos y transformarlos en el propio controlador.

Patrón: Domain Model.

Tipo: de Diseño

Contexto de Aplicación

Creamos objetos en el modelo a partir del dominio, donde guardaremos todos los datos que necesitemos relacionados con el dominio, como son las clases y los atributos.

Se encuentran en el paquete /springframework/samples/parchis_oca/model

Clases o paquetes creados

Administrator, Authorities, BaseEntity, BoardField, Game, GameAction, GameBoard, GamePiece, NamedEntity, Player, Oca, Parchis, Person, Turn, User.

Ventajas alcanzadas al aplicar el patrón

- Permite implementar una lógica de negocio más compleja.

Patrón: Repository

Tipo: de Diseño

Contexto de Aplicación

Lo usamos para encapsular la lógica requerida para acceder a los datos.

Clases o paquetes creados

- repository(Repositorios): AdministratorRepository, AuthoritiesRepository, BoardFieldRepository, GameActionRepository, GameBoardRepository, GamePieceRepository, GameRepository, OcaRepository, ParchisRepository, PlayerRepository, TurnRepository UserRepository.

Ventajas alcanzadas al aplicar el patrón

- Centraliza la lógica de datos
- Proporciona una arquitectura flexible
- Si se quiere modificar el acceso a los datos, no es necesario cambiar la lógica del repositorio.

Patrón: Service Layer

Tipo: de Diseño

Contexto de Aplicación

Lo utilizamos para establecer un conjunto de operaciones disponibles.

Clases o paquetes creados

- service(servicios): AdministratorService, AuthoritiesService, BoardFieldService, GameActionService, GameBoardService, GamePieceService, GameService, OcaService, ParchisService, PlayerService, TurnService, UserService.

Ventajas alcanzadas al aplicar el patrón

- Ayuda a reducir la sobrecarga conceptual relacionada con la gestión de servicios.

Patrón: Layer Supertype.

Tipo: de Diseño

Contexto de Aplicación

Se crea una clase abstracta común que contiene el campo identidad para los modelos que implementamos en la aplicación.

Se encuentran en el paquete /springframework/samples/parchis_oca/model/baseEntity.java

Clases o paquetes creados
BaseEntity

Ventajas alcanzadas al aplicar el patrón
- Al crear una clase común para todas las entidades no se sobrecargan con más información de la necesaria.

Patrón: Capas.
Tipo: Arquitectónico

Contexto de Aplicación
Lo hemos usado para dividir las responsabilidades del proyecto en 3 capas. La capa de recursos formada por el repositorio; la capa de lógica de negocio formada por los servicios y las clases que definen las entidades; y la capa de presentación formada por las vistas y los controladores.

Clases o paquetes creados
Capa de recursos:

- Repository(Repositorios): AdministratorRepository, AuthoritiesRepository, BoardFieldRepository, GameActionRepository, GameBoardRepository, GamePieceRepository, GameRepository, OcaRepository, ParchisRepository, PlayerRepository, TurnRepository UserRepository.

Capa de lógica de negocio:

- model(Modelos): Administrator, Authorities, BaseEntity, BoardField, Game, GameAction, GameBoard, GamePiece, NamedEntity, Player, Oca, Parchis, Person, Turn, User.
- service(servicios): AdministratorService, AuthoritiesService, BoardFieldService, GameActionService, GameBoardService, GamePieceService, GameService, OcaService, ParchisService, PlayerService, TurnService, UserService.

Capa de presentación:

- web(controladores): AdministratorController, DicesOnSessionController,CrashController, GameController, OcaController, ParchisController, PlayerController, UserController, WelcomeController.

- jsp(vistas): Están divididos en tres carpetas con el nombre admins, game y players, hacen referencia a las entidades de admin, game y player. La carpeta admins incluye tres vistas .jsp, la carpeta game también incluye tres vistas .jsp y la carpeta players, dos vistas .jsp.

Ventajas alcanzadas al aplicar el patrón
- Alta cohesión (cada capa se centra en una tarea determinada) y necesitan pocas dependencias.
- Facilidad de separación de responsabilidades.
- Las capas son independientes unas de otras
- Facilidad para reemplazar las capas.

Decisiones de diseño

En esta sección se describen las decisiones de diseño aplicadas a lo largo del desarrollo de la aplicación "Parchis&Oca" que no están incluidas en la aplicación de patrones de diseño o arquitectónicos.

Decisión 1: Importación de datos al sistema

Descripción del problema:

Para el correcto funcionamiento y prueba de la aplicación es fundamental tener una base de datos lo más completa posible que cubra todas las casuísticas contempladas.

Para ello se necesitan datos con los que se puedan ejecutar todas las pruebas y comprobar el funcionamiento de todos los métodos implementados en la aplicación.

En esta tarea se han encontrado problemas al incluir los datos como parte del script de inicialización de la base de datos. El arranque del sistema se ralentiza y las pruebas resultan más tediosas.

A continuación, se detallan una serie de alternativas exponiendo las ventajas y desventajas de cada una de ellas para elegir la mejor.

Alternativas de solución evaluadas:

Alternativa 1.a: ***Incluir los datos en el script de Inicialización de la BD (data.sql).***

Ventajas:

- Simple, no requiere nada más que escribir el SQL que genera los datos.

Inconvenientes:

- Se ralentiza el sistema a la hora de trabajar en la implementación.
- Es necesario poblar el script de la Base de datos con datos suficientes para cubrir todas las casuísticas.

Alternativa 1.b: ***Crear otro script con datos adicionales (data2.sql). Además, sería necesario implementar un nuevo controlador encargado de leer este script y ejecutar las consultas para cargar estos nuevos datos.***

Ventajas:

- Reutilización de los datos contemplados en el script original (data.sql).
- Se mejora el funcionamiento del sistema a la hora de desarrollar y realizar pruebas.

Inconvenientes:

- Impacto en modelo de división en capas si no se usa un servicio para cargar los datos que se relacione con el controlador asociado a la carga de datos desde el script.
- Necesidad de carga de más datos para rellenar el script (data2.sql).

Justificación de la solución adoptada

Puesto que la división en capas del sistema es fundamental y al revisar las diferentes alternativas, tras valorar las ventajas e inconvenientes, se ha elegido usar la alternativa 1.a.

Con esta alternativa se reduce notablemente la cantidad de datos que es necesario crear al sólo hacer uso de un script (data.sql). Con ello se consigue disponer de los diferentes datos necesarios para las pruebas de una manera sencilla y rápida.

Decisión 2: Layer Supertype

Descripción del problema:

Para evitar duplicidad de código en la implementación de las entidades del proyecto se ha planteado el uso de Layer Supertype para la creación del campo Identidad.

Alternativas de solución evaluadas:

Alternativa 2.a: Uso de Layer Supertype para la entidad *BaseEntity*

Ventajas:

- Evitar duplicación de código.

Inconvenientes:

- Dificultad en la organización para la interacción entre las diferentes clases del proyecto.
- Complejidad en la implementación de las consultas SQL.

Alternativa 2.b: Uso de Bloated Domain Model

.

Ventajas:

- Facilidad de uso en un modelo UML sencillo.

Inconvenientes:

- Duplicidad de código.
- Dificultad para abstraer la lógica de negocio sin uso de una jerarquía compleja.

Justificación de la solución adoptada

Al hacer uso de una gran cantidad de clases para cada entidad en una misma capa (capa de lógica de negocio) se ha optado por el uso de Supertype para evitar duplicidad de código en la implementación de cada entidad.

Decisión 3: Uso de la Capa de Servicios

Descripción del problema:

Al usar casos de uso entre diferentes entidades se hace necesario el uso de la capa de servicios

Alternativas de solución evaluadas:

Alternativa 3.a: Diseño impulsado por dominio

Ventajas:

- Agilidad y flexibilidad.
- Mantenibilidad del código.

Inconvenientes:

- Coste de implementación alto.
- Curva de aprendizaje alta.
- Uso para aplicaciones con el modelo de dominio más complejo.

Alternativa 3.b: Uso de la capa de servicios.

Ventajas:

- Fácil testeo.
- Mantenibilidad y escalabilidad.
- Mapeo directo entre procesos y sistemas.
- Mejora el desarrollo en paralelo.

Inconvenientes:

- No es recomendable su uso en aplicaciones con un alto nivel de transferencia de datos.
- Implica conocer los procesos del negocio, clasificarlos, extraer las funciones, estandarizarlas y desarrollar la capa de servicios.

Justificación de la solución adoptada

Se ha optado por usar la opción 3.b puesto que es la opción desarrollada durante la asignatura y debido a las ventajas que plantea su uso.

Decisión 4: MappedSuperclass

Descripción del problema:

Se hace uso de dos tipos de usuarios registrados en la plataforma (administrador y jugador), en estas dos entidades se usan varios atributos comunes por lo que el uso de herencia es una buena opción.

Alternativas de solución evaluadas:

Alternativa 4.a: MappedSuperclass

Ventajas:

- Su uso es muy simple, la clase hija extenderá de la clase padre y esa clase hija podrá hacer uso de los atributos de la clase padre y los nuevos.

Inconvenientes:

- Imposibilidad de que las clases padres sea entidades, no persisten en la base de datos.
- La clase padre no puede contener asociaciones con otras entidades.

Alternativa 4.b: Single Table.

Ventajas:

- Uso por defecto.

Inconvenientes:

- Implica que todas las subclases se almacenan en la misma tabla.

Alternativa 4.c: Joined Table.

Ventajas:

- Uso de una tabla por cada clase.

Inconvenientes:

- Necesidad de usar “JOIN” entre tablas, implica una bajada de rendimiento en la plataforma.

Alternativa 4.d: Table per Class.

Ventajas:

- Uso de una tabla por cada clase, contiene todos los atributos incluso los heredados.
- Similitud con MappedSuperclass pero a diferencia de esta, la clase padre también es considerada una entidad.

Inconvenientes:

- Necesidad de usar “UNION”, implica una bajada de rendimiento en la plataforma.
- Imposibilidad de uso de claves de identidad.

Justificación de la solución adoptada

Se ha optado a usar la opción 4.a debido al inconveniente de no poder usar la generación Identity en el id, el id se incrementa automáticamente y mejora la optimización.