DP1 2021-2022

Documento de Diseño del Sistema
Proyecto Idus Martii

https://github.com/gii-is-DP1/dp1-2021-2022-g5-04

Miembros (Grupo G5-04):

- Arias Expósito, José Ramón
- Carnero Vergel, Manuel
- Moreno Pérez, Juan Carlos
- Sabugueiro Troya, David
- Santos Pérez, Pablo
- Serrano Mena, Antonio Roberto

Miembros en Septiembre (Grupo SEPT 3):

- Moreno Pérez, Juan Carlos
- Santos Pérez, Pablo

Tutor: Manuel Resinas

GRUPO G5-04

GRUPO SEPT3

31/08/2022

Versión 5.0

Historial de versiones

Fecha	Versión	Descripción de los cambios	Sprint
26/12/2021	V1	● Creación del documento ● Introducción ● Diagrama de capas primera versión	3
04/01/2022	V2	● Añadido diagrama de dominio/diseño ● Explicación de la aplicación del patrón caché	3
06/01/2022	V3	● Puesta a punto del documento	3
21/01/2022	V4	● Arreglos de diagramas y finalización del documento	4
12/07/2022	V5	● Arreglos de diagramas	SEP
18/08/2022	V5	● Patrones	SEP
22/08/2022	V5	● Patrones	SEP
25/08/2022	V5	● Decisiones de diseño	SEP
29/08/2022	V5	● Corrección de diagramas, patrones y decisiones ● Puesta a punto del documento	SEP

Contenido

Introducción

Para la entrega del proyecto en septiembre, nuestro principal objetivo es el funcionamiento al 100% de manera correcta de todas las historias de usuario principalmente, ya que algunas cómo la auditoria de usuarios o eliminar un usuario de una partida, entre muchas otras, no funcionaban. A parte de eso, hemos tratado de mejorar, optimizar y limpiar el código y la estructura de este. Después de esta entrega, el juego será totalmente funcional.

Todo el código producido, junto con las pruebas unitarias de cada historia de usuario, conforman nuestro proyecto completo.

Al analizar de nuevo el código nos hemos dado cuenta de que, para la entrega final de la convocatoria de febrero, no teníamos un proyecto digno para aprobar. Este verano vamos a trabajar de manera correcta y con tiempo suficiente para aprobar esta parte tan importante de la asignatura.

Diagrama(s) UML:

Diagrama de Dominio/Diseño

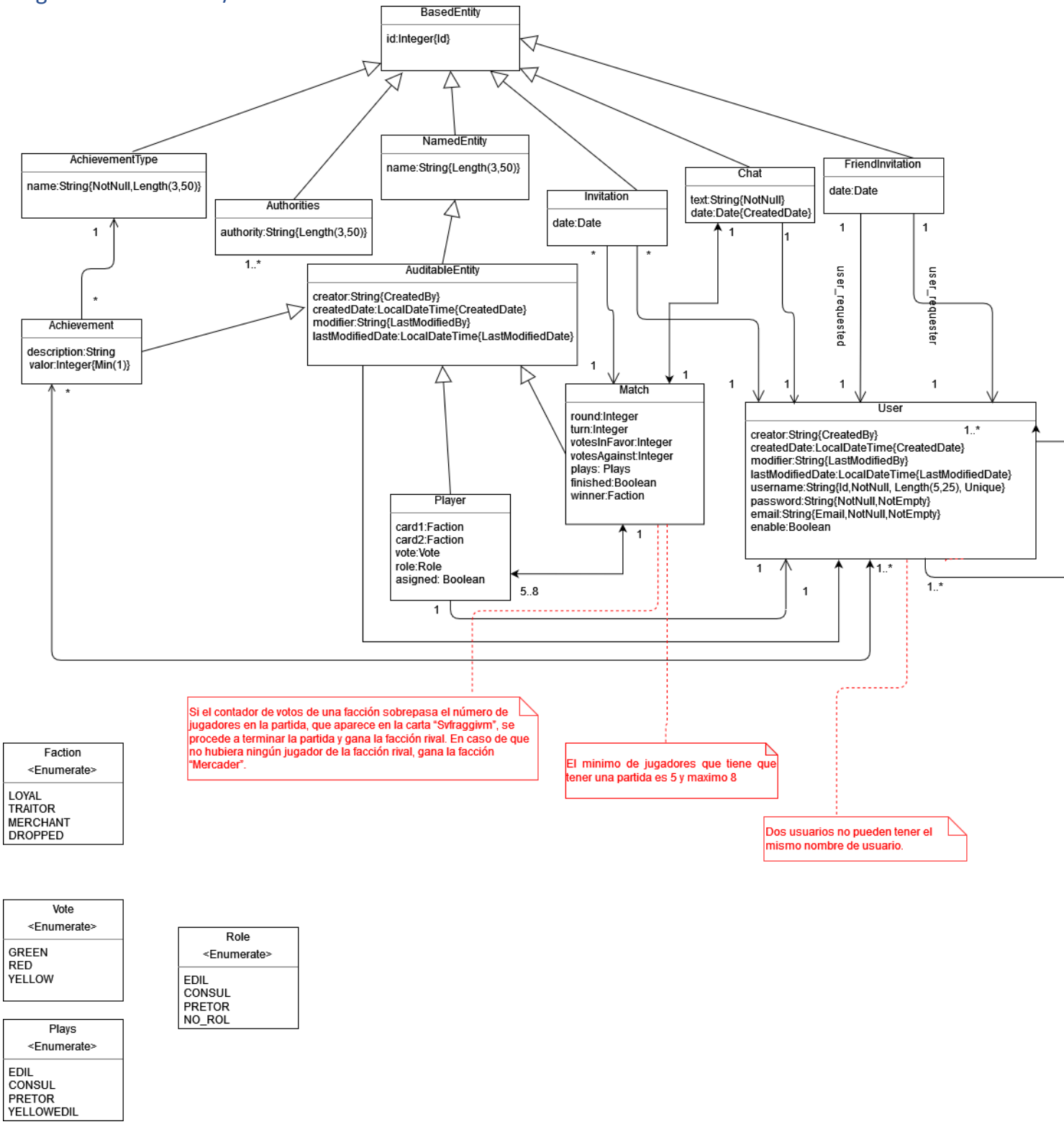
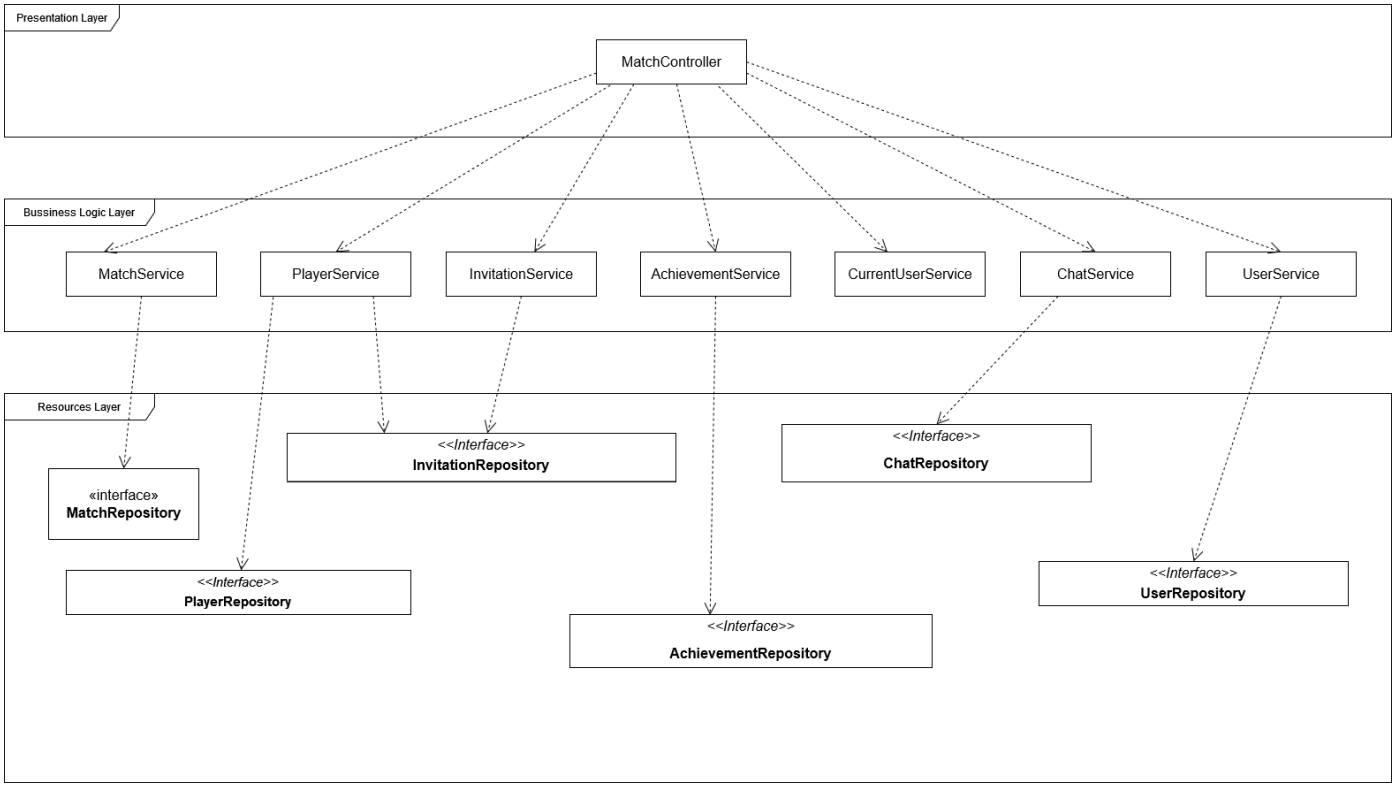
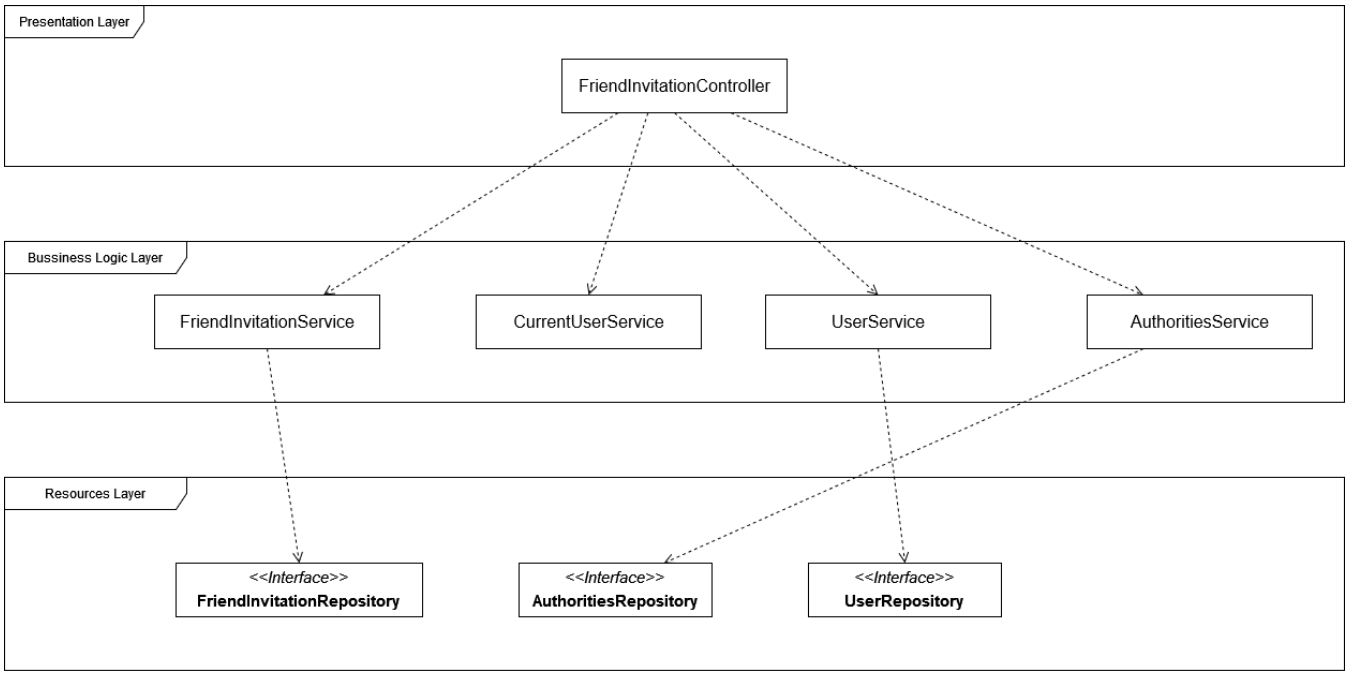
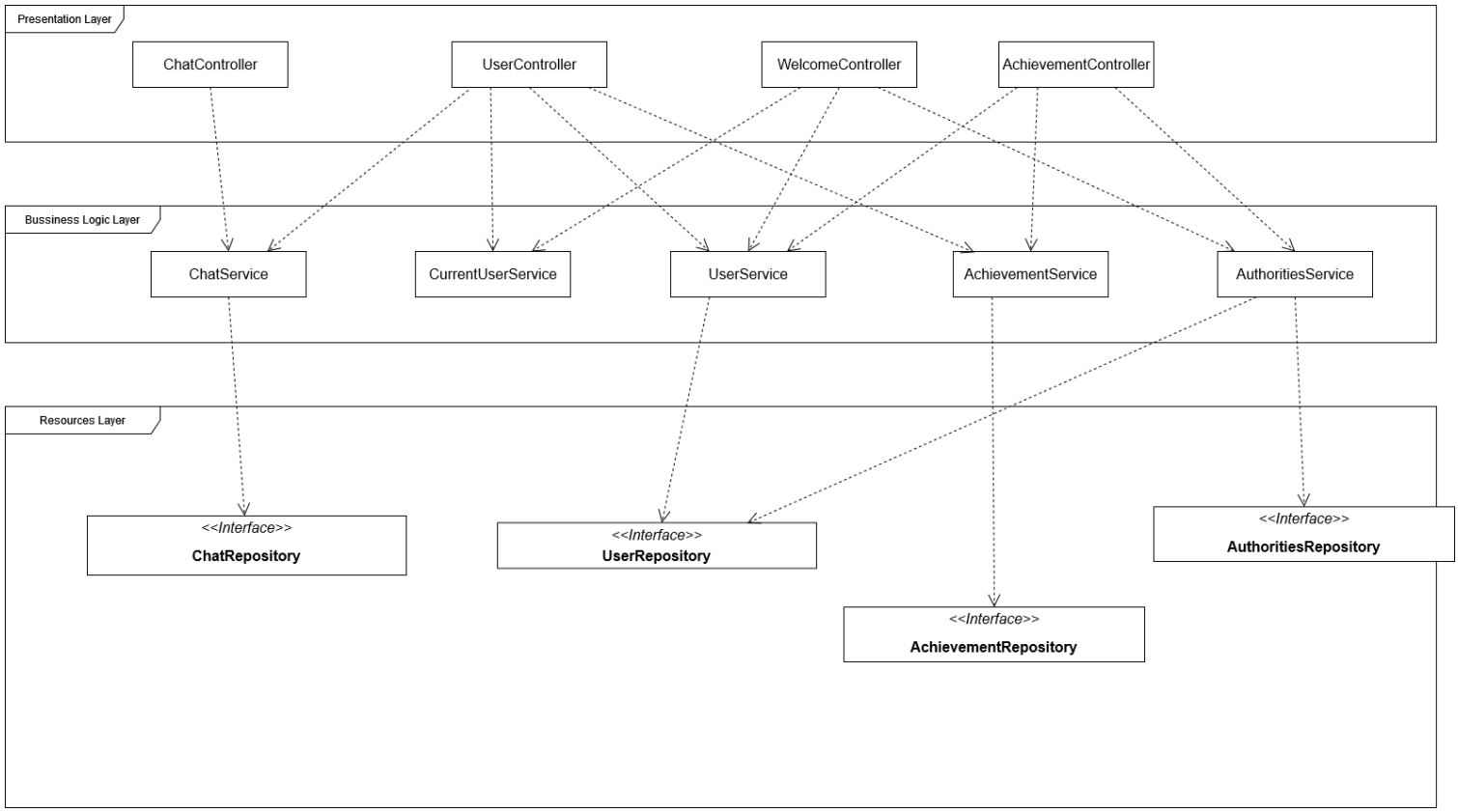
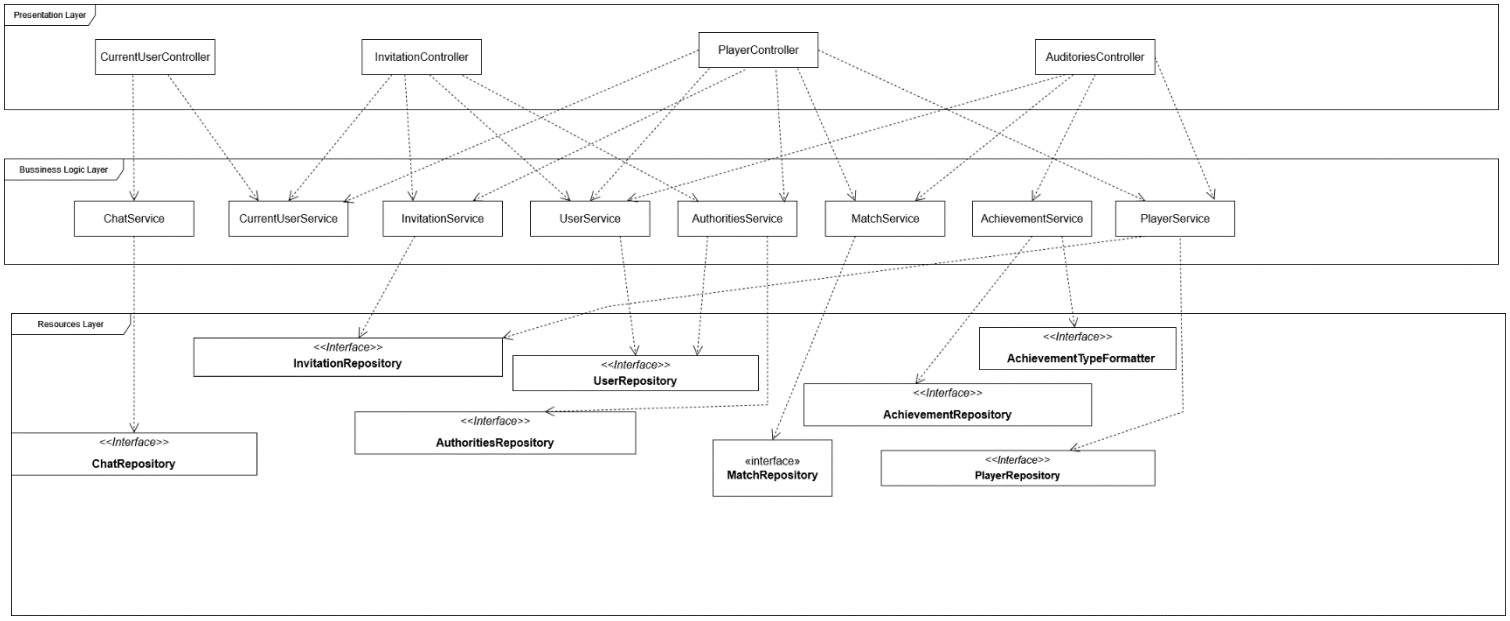


Diagrama de Capas (incluyendo Controladores, Servicios y Repositorios)






Presentation Layer
ChatController
UserController
WelcomeController
AchievementController
Bussiness Logic Layer
ChatService
CurrentUserService
UserService
AchievementService
AuthoritiesService
Resources Layer
<<Interface>>
ChatRepository
<<Interface>>
UserRepository
<<Interface>>
AuthoritiesRepository
<<Interface>>
AchievementRepository


Presentation Layer
CurrentUserController
InvitationController
PlayerController
AuditoriesController
Bussiness Logic Layer
ChatService
CurrentUserService
InvitationService
UserService
AuthoritiesService
MatchService
AchievementService
PlayerService
Resources Layer
<<Interface>>
InvitationRepository
<<Interface>>
UserRepository
<<Interface>>
AchievementTypeFormatter
<<Interface>>
AchievementRepository
<<Interface>>
AuthoritiesRepository
<<Interface>>
ChatRepository
«interface»
MatchRepository
<<Interface>>
PlayerRepository

Patrones de diseño y arquitectónicos aplicados

Patrón: Modelo-Vista-Controlador (MVC)

Tipo: Diseño

Contexto de Aplicación

El patrón se ha aplicado en prácticamente toda la aplicación:

- Modelo: carpetas “src/main/java” (excepto el paquete que termina con .web, el que termina en IdusMartii y el que termina en .exceptions) y “src/main/resources”
- Vista: carpeta “src/main/webapp”
- Controlador: carpeta “src/main/java” (únicamente el paquete que termina con .web)

Clases o paquetes creados

Los paquetes y sus clases creados con este patrón son:

- paquetes -> “org.springframework.samples.IdusMartii.configuration”, “org.springframework.samples.IdusMartii.enumerates”, “org.springframework.samples.IdusMartii.model”, “org.springframework.samples.IdusMartii.repository” , “org.springframework.samples.IdusMartii.service”, “org.springframework.samples.IdusMartii.util” y “org.springframework.samples.IdusMartii.web”
- clases -> paquete.configuration - {“ExceptionHandlerConfiguration.java”, “GenericIdToEntityConverter.java”, “SecurityConfiguration.java” y “WebConfig”} paquete.enumerates - {“Faction.java”, “Plays.java”, “Role.java” y “Vote.java”} paquete.model - {“Achievement.java”, “AuditableEntity.java”, “AuditorAwareImpl.java”,

“Authorities.java”, “BaseEntity.java”, “Invitation.java”, “Match.java”, “NamedEntity.java”; “Owner.java”, “Player.java”, “User.java” y “Chat.java”}
paquete.repository - {AchievementRepository.java”,
,“AuthoritiesRepository.java”, “FriendInvitationRepository.java“,
,InvitationRepository.java”, “MatchRepository.java”,
“PlayerRepository.java”, “UserRepository.java” y
“ChatRepository.java”}
paquete.service - {“AchievementService.java”, “AuthoritiesService.java”, “CurrentUserService.java”, “InvitationService.java”, “MatchService.java”, “PlayerService.java”, “UserService.java” ,“ChatService.java” y “FriendInvitationService.java}
paquete.util - {“CallMonitoringAspect.java” y “EntityUtils.java”}
paquete.web - {“AchievementController.java”, “AuditoriesController.java”,
“FriendInvitationController.java”, “CurrentUserController.java”, “InvitationController.java”, “MatchController.java”, “PlayerController.java”,
“UserController.java”,“WelcomeController.java” y “ChatController.java”}

Ventajas alcanzadas al aplicar el patrón

Con este patrón hemos podido separar los diferentes tipos de lógica, de forma que se hace más fácil el implementar el juego. Es sencillo a la hora de representar los datos y facilita la realización de pruebas unitarias. Muy útil también a la hora de reutilizar componentes. Crea independencia en el funcionamiento y a la hora del mantenimiento de los errores es eficaz.

Patrón: Front Controller

Tipo: Diseño

Contexto de Aplicación

Un controlador frontal es una clase/función que maneja todas las solicitudes de un sitio web y luego envía esas solicitudes al controlador apropiado. Esto lo hacemos a través de una implementación denominada DispatcherServlet.

Clases o paquetes creados

Todo lo que está en org.springframework.samples.IdusMartii.web

Ventajas alcanzadas

La aplicación de políticas en toda la aplicación, como el seguimiento de usuarios y la seguridad, entre otros.
La implementación de Front Controller, Dispatcher Servlet, es proporcionado por Spring, es decir, el propio framework lo proporciona. La decisión sobre quién es el manejador apropiado de una solicitud puede ser más flexible.
Proporciona funciones adicionales, como el procesamiento de solicitudes para la validación y transformación de parámetros.

Patrón: Domain Model

Tipo: Diseño

Contexto de Aplicación

Un modelo de dominio es un objeto cuyos datos se conservan en la base de datos y tiene su propia identidad. Tienen anotaciones que indican cómo mapear atributos a tablas en la base de datos. Son Entidades JPA. Este patrón nos ha ayudado a que el juego pase a ser sostenido en términos de objetos, estos objetos tienen comportamiento y su interacción depende de los mensajes que se comunican entre ellos, es el patrón orientado a objetos por excelencia y hace que los objetos modelados estén en concordancia con el juego.

Clases o paquetes creados

La mayoría de las entidades que están en org.springframework.samples.IdusMartii.model

Ventajas alcanzadas

Facilita guardar la información en la base de datos. Es el patrón base de todo el proyecto. Gracias a este patrón podemos guardar información relacionada con la jugabilidad del juego. Facilita la concordancia de los objetos modelados del negocio.

Patrón: Layer Supertype

Tipo: Diseño

Contexto de Aplicación

El patrón se ha aplicado con las clases BaseEntity, NamedEntity y AuditableEntity

Clases o paquetes creados

Los paquetes y sus clases creados con este patrón son:

- paquetes -> “org.springframework.samples.IdusMartii.model”
- clases -> paquete.model - {“BaseEntity.java” y “NamedEntity.java” y “AuditableEntity”}

Ventajas alcanzadas al aplicar el patrón

Con este patrón nos hemos ayudado con los id de nuestras clases, ya que con BaseEntity nos genera el id sin necesidad de definirlo en cualquier clase, ya que nuestras clases extienden a ellas. NamedEntity es capaz de general el nombre. También lo hemos usado para AuditableEntity que genera atributos auditables como la fecha de creación o de modificación.

Patrón: Repository

Tipo: Diseño

Contexto de Aplicación

El patrón se ha aplicado para encapsular la lógica de negocio necesaria para acceder a los datos.

Clases o paquetes creados

Los paquetes y sus clases creados con este patrón son:

- clases -> paquete.repository - {AchievementRepository.java”, AchievementUserRepository.java”,“AuthoritiesRepository.java”, “FriendInvitationRepository.java“,”FriendRepository.java”,InvitationRepository.java”, “MatchRepository.java”, “PlayerRepository.java”, “UserRepository.java” y “ChatRepository.java”}
- paquetes -> “org.springframework.samples.IdusMartii.repository.

Ventajas alcanzadas al aplicar el patrón

Con este patrón nos hemos ayudado de las Queries que nos deja usar este patrón, con ello podemos acceder a los datos de la base de datos de una manera más fácil, y con ello, implementar métodos del servicio, ya que el servicio llama al repositorio. El código de acceso a los datos puede ser reutilizado.
Es fácil de implementar la lógica del dominio. Nos ayuda a desacoplar la lógica de la aplicación. La lógica de negocio puede ser probada fácilmente sin acceso a los datos, por lo que con este patrón el código es más fácil de probar

Patrón: Arquitectura centrada en datos

Tipo: Diseño

Contexto de Aplicación

Se ha implementado base de datos donde están almacenados los datos del propio juego y donde se irán almacenando los cambios y nuevos datos.

Clases o paquetes creados

Se ha llevado a cabo la creación del archivo data.sql

Ventajas alcanzadas al aplicar el patrón

La principal ventaja es que nos ha permitido crear y almacenar datos y sobre todo poder acceder a ellos para la manipulación de estos. Mantiene a los datos en el centro mientras todo los demás se construye alrededor. Proporciona escalabilidad y reutilización. Reduce la sobrecarga de datos transitorios entre componentes de software.

Patrón: Service Layer

Tipo: Diseño

Contexto de Aplicación

Define la frontera de la aplicación con una capa de servicios que establece un conjunto de operaciones disponibles y coordina la respuesta de la aplicación en cada operación. La capa de presentación interactúa con la de dominio a través de la capa de servicio.

Clases o paquetes creados

El paquete creado para la adaptación de este patrón de diseño es el mismo que hemos mencionado antes cómo “service”.

Ventajas alcanzadas al aplicar el patrón

Nos permite tener muchos casos de uso que envuelven entidades de dominio e interactuar con servicios externos. Es un patrón que define una capa de servicios en el cual limita un conjunto de operaciones disponibles para englobar la lógica de negocio de la de aplicación.

Patrón: Pagination

Tipo: Diseño

Contexto de Aplicación

Trata de obtener un subconjunto de los datos de los resultados, separados por páginas, a fin de mejorar la experiencia de los usuarios. De esta manera se aumenta el rendimiento de la aplicación.

Clases o paquetes creados

Tanto en UserRepository.java y ChatRepository.java como en UserService.java y ChatService.java usamos paginación.

Ventajas alcanzadas al aplicar el patrón

Mejora la experiencia del usuario además del rendimiento de la aplicación. Permite que los resultados se puedan visualizar en un tamaño adecuado y no todos los resultados, ya que ocuparían mucho tamaño

Patrón: Capas

Tipo: Arquitectura

Contexto de Aplicación

Este patrón nos ha servido para organizar el sistema en 3 capas diferentes, en donde se agrupan los componentes con funcionalidades similares, dependiendo de la funcionalidad de la capa inferior. Estas 3 capas son: la capa de presentación, de lógica de negocio y la de recursos. En el contexto de nuestra aplicación en la capa de presentación hemos agrupado todos los controladores del sistema, en la capa de lógica de negocio todos los servicios, encargados de hacer cálculos y operaciones como autentificación, y por último en la capa de recursos agrupamos todos los repositorios del sistema los cuales van a gestionar nuestra base de datos.

Clases o paquetes creados

Se han creado varios diagramas para que se viera todo más limpio con las 3 capas mencionadas anteriormente

Ventajas alcanzadas al aplicar el patrón

Promueve el bajo acoplamiento y cohesión, la separación de responsabilidades y la independencia de las distintas capas. Es un patrón fácil de desarrollar, de mantener y de probar.

Decisiones de diseño

En esta sección describiremos las decisiones de diseño que se han tomado a lo largo del desarrollo de la aplicación que vayan más allá de la mera aplicación de patrones de diseño o arquitectónicos.

Decisión 1: Realización de pruebas de la partida

Descripción del problema:

Como grupo nos gustaría poder probar algunos estados de la partida, significando dichos estados, un conjunto de condiciones que se pueden dar en una partida en un determinado momento, porque nos resultaba necesario para poder asegurar la jugabilidad del juego. El problema es que hay ciertas pruebas que son muy complejas para realizarlas como tests.

Alternativas de solución evaluadas:

Alternativa 1.a: Incluir las condiciones en el propio script de inicialización de la BD (data.sql).

Ventajas:

- Simple, no requiere nada más que modificar el SQL que genera la partida.
- No afecta la elaboración de pruebas ni el trabajo diario.

Inconvenientes:

- Se necesita un tiempo extra para poder probar todos los posibles estados.

Alternativa 1.b: Crear un controlador y un servicio que permita incluir las condiciones sin tener que cambiar el script de inicialización de la BD (data.sql).

Ventajas:

- Permite reutilizar los datos ya existentes en (data.sql).

Inconvenientes:

- Afecta la elaboración de pruebas, ya que habría que realizar más pruebas para el controlador y el servicio.
- Supone añadir más complejidad al código.

Justificación de la solución adoptada

Como no queremos renunciar a trabajar de forma ágil, escogimos la alternativa 1.a, pues modificar la base de datos según nos convenga era más rápido que crear un controlador y un servicio y las pruebas asociadas a ellos.

Decisión 2: Refresco puntual de la partida

Descripción del problema:

Como grupo y posibles jugadores, nos gustaría que cuando el estado de una partida en curso cambie sea mostrado dicho cambio en las ventanas de todos los jugadores presentes para que por ejemplo cuando los ediles terminen de votar, se le muestren automáticamente al pretor los botones de revisar los votos y hacer de la experiencia de juego algo más amena.

Alternativas de solución evaluadas:

Alternativa 2.a: Añadir un autorrefresco a la página de duración moderada, por ejemplo 5-10 segundos para que los jugadores no tengan que estar refrescando la página para saber si es su turno o no.

Ventajas:

- Dificultad baja, es solo añadir poca cantidad de código en el controlador de la partida en curso
- No afecta la elaboración de pruebas ni el trabajo diario.

Inconvenientes:

- Si el tiempo de autorrefresco es de 10 segundos y el estado de la partida cambia muy poco después de haberse refrescado, el jugador debe esperar 10 segundos para que se le actualicen los cambios o refrescar manualmente.

Alternativa 2.b: Implementar un RestController que se encargue de actualizar la página cuando suceda un cambio.

Ventajas:

- La experiencia de usuario mejora considerablemente al no tener que esperar al autorrefresco.
- Optimización a la hora de probar funcionalidades desde el punto de vista del jugador.

Inconvenientes:

- Dificultad alta, al momento de conocer la existencia de esta alternativa, desconocemos la manera de implementarla.
- El desconocimiento presente implica que si queremos decantarnos por esta solución debemos destinar parte de nuestro tiempo a investigar su funcionamiento e implementación.

Justificación de la solución adoptada

Para no perder demasiado tiempo en una solución cuyo impacto tampoco es tan grande si el tiempo de autorrefresco es reducido, escogimos la alternativa 2.a para poder centralizar nuestro tiempo a labores y tareas de mayor relevancia en el proyecto.

Decisión 3: Eliminación de usuarios en el lobby de la partida

Descripción del problema:

Queremos decidir una manera eficaz de eliminar a los jugadores de la partida como host sin tener oportunidad de expulsarse a sí mismo.

Alternativas de solución evaluadas:

Alternativa 3.a: Implementar una excepción cuya función consiste en devolver un error en el caso de intentar eliminarse a uno mismo

Ventajas:

- Evita poder realizar esta acción a través de la URL del navegador.

Inconvenientes:

- Añade complejidad al código
- Peor rendimiento

Alternativa 3.b: Eliminar de la vista el botón de eliminar que aparecería en el host

Ventajas:

- Simplicidad en el código
- Mejor rendimiento

Inconvenientes:

- Posibilidad de acceder al método mediante peticiones HTTP

Justificación de la solución adoptada

Aunque realizar el borrado mediante una petición HTTP, vemos reducida la posibilidad de que los jugadores conozcan la petición correspondiente a ese método porque en ningún momento es visible para el usuario, por lo que la alternativa 3.b nos pareció una solución cómoda y eficaz para que el juego siga su curso sin errores graves.

Decisión 4: Chat de la partida

Descripción del problema:

Como jugador de la partida, me gustaría que el chat se pudiera ver bien, sin que nada lo obstaculice, para que se puedan ver bien los mensajes de los demás jugadores.

Alternativas de solución evaluadas:

Alternativa 4.a: Crear el chat por separado a la partida, con lo cual, se mostrará un enlace del chat en la propia partida, para tener las 2 cosas separadas y que no se pisen. Además de crear un chat pageable.

Ventajas

- Baja dificultad, ya que su implementación es algo muy básico y que se ha realizado antes.
- No afecta al transcurso de la partida

Inconvenientes:

- Los jugadores tienen que estar cambiando de ventana para cuando quieran chatear o jugar.

Alternativa 4.b: Implementar el chat dentro de la partida.

Ventajas:

- La experiencia de usuario mejora, en el sentido de no tener que cambiar de ventana para chatear y jugar

Inconvenientes:

- Dificultad alta, ya que tendríamos que implementar todo lo que conlleva la partida con su lógica junto con el chat.
- Demasiado cargada la vista de la partida, ya que nosotros utilizamos cartas, la vista de la partida se

vería demasiado llena de elementos

Justificación de la solución adoptada

Para ir a lo seguro, hemos implementado el chat en una ventana aparte a la de la partida, así nos aseguramos de que ambas partes funcionen correctamente.

Decisión 5: Revisión de voto nulo

Descripción del problema:

Como equipo, queremos encontrar una manera adecuada de hacerle saber a los jugadores que el Pretor ha descubierto un voto nulo a la hora de revisarlo.

Alternativas de solución evaluadas:

Alternativa 5.a: Redirigir a todos los jugadores a una nueva vista especializada para este anuncio

Ventajas:

- Solución bastante llamativa que da mucho juego por las posibilidades de aviso disponibles

Inconvenientes:

- Complejidad de código aumentada
- Creación de una nueva vista

Alternativa 5.b: Añadir un mensaje en la vista de la partida en curso

Ventajas:

- Extremadamente simple
- Claro y conciso
- Simplicidad en el código

Inconvenientes:

- Pérdida de oportunidad de hacer la experiencia más divertida

Justificación de la solución adoptada

Debido a decisiones de equipo, aunque nuestra alternativa favorita era dedicarle una vista individual a este aviso para dar más emoción a la partida, nos decantamos por la alternativa 5.b para mejorar nuestra organización y poder dedicar más tiempo a otros aspectos del proyecto.

Decisión 6: Creación y edición de logros

Descripción del problema:

A la hora de crear y guardar un logro en la base de datos,

Alternativas de solución evaluadas:

Alternativa 6.a: Se ha creado una entidad que es AchievementType que está asociada al Achievement. Esta entidad puede ser, por el momento "jugadas" o "ganadas", refiriéndose a partidas ganadas y jugadas y asociadas con el atributo "valor" de Achievement. Por ejemplo, si el valor es 100 y el name del AchievementType es "jugadas", el logro se obtendrá cuando el jugador juegue 100 partidas.

Ventajas:

- Solución bastante general del problema
- Sencilla

Inconvenientes:

- Por el momento, solo se pueden añadir AchievementType a través del data.sql.

Alternativa 6.b: Dejar que se pueden crea y editar un AchievementType, y así crear el logro.

Ventajas:

- Solución específica del problema

Inconvenientes:

- Sería una solución demasiado ambigua ya que podría haber muchísimos AchievementType.
- Compleja

Justificación de la solución adoptada

Para ir a lo seguro, hemos implementado la primera solución ya que la hemos visto sencilla y fácil para cualquiera que pueda editar y crear siendo administrador y así evitar ambigüedades.

Decisión 7: Invitación de usuarios a una partida

Descripción del problema:

Un usuario desea invitar a otro a una partida.

Alternativas de solución evaluadas:

Alternativa 7.a: Se ha creado una entidad Invitation donde almacenamos la fecha de la invitación, la partida y el usuario que está relacionado con la entidad User. Con esta entidad podemos mandar todas las solicitudes de invitación para unirse a la partida deseada.

Ventajas:

- Sencilla

Inconvenientes:

- Puede chocar con las invitaciones de amistad a la hora de crear el servicio,controlador y repositorio, por lo que habría un duplicado de cosas que están iguales en 2 archivos.

Alternativa 7.b: Se ha creado una entidad Invitation donde almacenamos la fecha de la invitación, la partida, el usuario que manda la invitación y quien la recibe. Con esta entidad podemos mandar las invitaciones tanto de amistad como de partida.

Ventajas:

- Solución bastante general del problema

Inconvenientes:

- Bastante compleja

Justificación de la solución adoptada

Debido a decisiones de equipo, lo más sencillo para todos ha sido hacer la alternativa 7.a y crear la invitación de partida separada a la de amistad.

Decisión 8: Invitación de amistad

Descripción del problema:

Un usuario desea mandar una invitación de amistad a otro usuario.

Alternativas de solución evaluadas:

Alternativa 8.a: Se ha creado una entidad FriendInvitation donde almacenamos el usuario que manda la invitación, el que la recibe y la fecha de dicha invitación. Con esta entidad podemos mandar todas las solicitudes de amistad.

Ventajas:

- Solución específica del problema
- Sencilla

Inconvenientes:

- Para que dos usuarios sea amigos, a nivel de código tendrían que pasar por varios métodos e servicios, controladores y repositorios.

Alternativa 8.b: Se ha creado una entidad Invitation donde almacenamos la fecha de la invitación, la partida, el usuario que manda la invitación y quien la recibe. Con esta entidad podemos mandar las invitaciones tanto de amistad como de partida.

Ventajas:

- Solución bastante general del problema

Inconvenientes:

- Bastante acoplamiento de datos.

Justificación de la solución adoptada

Debido a decisiones de equipo, lo más sencillo para todos ha sido hacer la alternativa 8.a y crear la invitación de amistad separada a la de partida.

Decisión 9: Uso de las cartas del juego

Descripción del problema:

Al principio no sabíamos cómo implementar las cartas para poder jugar, ya que son fundamentales para el desarrollo de la partida.

Alternativas de solución evaluadas:

Alternativa 9.a: Se crean 3 enumerados correspondientes a las diferentes clases de cartas: Cartas de facción, de voto, de rol y un enumerado que indica que rol juega en ese turno.

Ventajas:

- Solución bastante general del problema
- Sencilla

Inconvenientes:

- Al desarrollar los métodos del controlador, hay que estar constantemente indicando que valor del enumerado debe tener cada atributo de Match y Player para que pueda funcionar correctamente.

Alternativa 9.b: Se crea una entidad MatchCard en la que se añade todo a un mismo tipo y un atributo al que se le pasa un enumerate con la facción, el rol, etc...

Ventajas:

- Creamos solo una entidad para la carta.

Inconvenientes:

- Compleja de implementar y añadiría otra entidad a nuestros diagramas y varias relaciones con otras entidades, lo que hace sobrecargar los datos y generar más acoplamiento.

Justificación de la solución adoptada

Debido a decisiones de equipo, lo más sencillo para todos ha sido hacer la alternativa 9.a y crear los enumerados indicados, sí que va a ser más tedioso a la hora de implementar los métodos del controlador, pero así evitamos tener otra entidad más y evitamos el acoplamiento.

Decisión 10: Estructura de los archivos de código

Descripción del problema:

Cómo estructurar los diferentes archivos que conforman nuestro código.

Alternativas de solución evaluadas:

Alternativa 10.a: Se crea una carpeta para cada entidad con su servicio, repositorio y controlador, así para cada entidad.

Ventajas:

- Solución bastante general del problema
- Sencilla

Inconvenientes:

- Habría muchas carpetas y poco orden.

Alternativa 10.b: Se crea una carpeta para las entidades(“model”), otra para los servicios(“service”), otra para los controladores(“controller”), otra para los validadores(“validator”), etc.

Ventajas:

- Creamos las carpetas justas para cada tipo de archivo de código.
- Sencilla

Inconvenientes:

- Tendríamos que buscar dentro de cada carpeta el archivo determinado ya que en una carpeta se encontrarían diversos archivos del mismo tipo.

Justificación de la solución adoptada

Debido a decisiones de equipo, hemos decidido escoger la alternativa 10.b, ya que es la que más sentido tendría para tenerlo todo ordenado. Ya que vamos a trabajar una arquitectura en capas, hemos creído conveniente separar las capas desde un principio.

Decisión 11: Uso de listas en pruebas

Descripción del problema:

Como testear algunos de los métodos de nuestros servicios.

Alternativas de solución evaluadas:

Alternativa 11.a: Se crean carpetas vacías y se llenan con datos por nosotros, para poder comparar con métodos como buscar una lista de usuarios, jugadores, etc. En general, se usan para luego comprobar el tamaño de esas listas y así probar operaciones de creación y eliminación de datos.

Ventajas:

- Solución bastante general del problema
- Sencilla

Inconvenientes:

- Se crearán muchas listas vacías de un solo uso.

Alternativa 11.b: Se crea una función que devuelva la lista que queremos crear vacía y llenarla o una lista que sirva para todos los métodos de la prueba de ese servicio.

Ventajas:

- Solo se crearía una lista de un solo uso.
- Sencilla.

Inconvenientes:

- Habría que crear más de una lista y cambiaría dependiendo del tipo.
- Demasiado general para las pruebas.

Justificación de la solución adoptada

Debido a decisiones de equipo, hemos decidido escoger la alternativa 10.a, ya que fue la primera que se nos planteo, y debido a que funcionarán igual de bien las 2 opciones, optamos por mantener el uso de listas vacías en los test que lo requieran.

Decisión 12: Mensaje de error

Descripción del problema:

Como trabajar los mensajes de error

Alternativas de solución evaluadas:

Alternativa 11.a: Se redirigen los mensajes de error a una vista llamada “excepcion” donde nos saldrá el mensaje de error correspondiente.

Ventajas:

- Solución bastante general del problema
- Sencilla

Inconvenientes:

- Te redirige a otra vista, no sale en la misma vista donde se produce el error.

Alternativa 11.b: Se gestiona el error con rejectValue en el campo que nos interesa.

Ventajas:

- Solución especifica en el campo específico.

Inconvenientes:

- En el ámbito de nuestra aplicación, algunas veces funciona y otras no.

Justificación de la solución adoptada

Debido a decisiones de equipo, hemos decidido escoger ambas, dependiendo del error si es grave o no usamos el rejectValue o la vista de exception, también a tener en cuenta algunos errores que no hemos podido solucionar con rejectValue ni con otro tipo de código, hemos usado la vista exception.