DP1 2021-2022

Documento de Diseño del Sistema

Proyecto Vae Victis

https://github.com/gii-is-DP1/dp1-2021-2022-g4-5

Miembros:

- Daniel Díaz Nogales
- Gonzalo Martínez Martínez
- Álvaro Miranda Pozo
- Pedro Parrilla Bascón
- Luis Rodríguez Vidosa
- Álvaro Vázquez Ortiz

Tutor: Manuel Resinas Arias

GRUPO G4-05

Versión 3

04/01/2022

Historial de versiones

Fecha	Versión	Descripción de los cambios	Sprint
05/01/2022	V1	● Creación del documento	3
05/01/2022	V2	● Patrón de diseño “Estados” ● Decisión de diseño “WebSocket”	3
20/02/2022	V3	● Revisión y corrección final del documento	4

Contents

Introducción

El proyecto consiste en la implementación del juego de mesa ***"Vae Victis"***. Con ello pretendemos lograr un despliegue del juego en línea con múltiples jugadores en una misma partida a tiempo real, de forma que no será necesario disponer del juego en formato físico ni la presencialidad de los jugadores.

Nuestro proyecto intentará hacer del juego de mesa un entorno social a través de internet, en el que se nos permita jugar con gente de todo el mundo, e incluso comunicarnos entre usuarios. El juego de mesa "Vae Victis" aporta muchos valores a los jugadores, el principal es el sentimiento de comunidad aunque también destacamos valores como la competitividad y factores como la cantidad de riesgo que estás dispuesto a sufrir. Nuestro proyecto también haría que el juego de mesa fuese más accesible y que se cumpliesen las normas del juego sin que los jugadores tuviesen que hacer un estudio exhaustivo de estas.

Al ser un juego cooperativo, la cantidad mínima de jugadores es de 3 y máxima de 6, siendo recomendable que estos tengan una edad mayor a 10.

Cada partida comienza, con el reparto de dos cartas de objetivo (una es victoria militar en 2 guerras y otra una configuración del senado) a los jugadores, cartas que representan la victoria en el juego. El jugador que durante su turno, cumpla el objetivo de ambas cartas, ganará la partida.

El turno de cada jugador se resuelve en 3 fases de juego:
1ª Fase. – Lanzamiento de dados.
2ª Fase. – Acciones de la parte alta, donde el jugador podrá hacer una o dos acciones. Estas acciones permiten:

- Avanzar en los marcadores de las diferentes guerras.
- Recoger monedas del tesoro.
- Acciones del Domus, que nos permiten recuperar en los tracks de consecuencias.

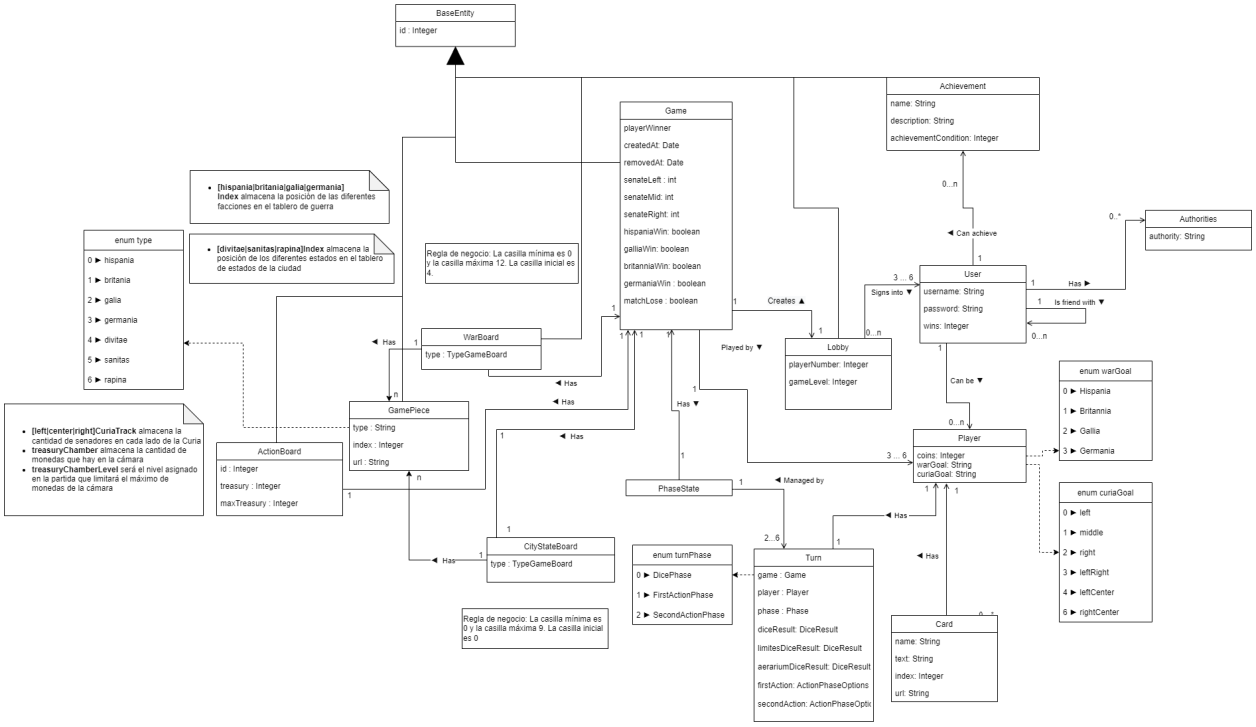
Cada una de estas acciones se puede hacer dos veces, pero si la repetimos una segunda vez en el mismo turno, tendremos un coste mayor y en algunos casos unos perjuicios.
3ª Fase. – Acciones de la parte baja del tablero, donde el jugador podrá hacer una o dos acciones. Estas acciones permiten:

- Las Termas, donde podremos conseguir cartas sobre el lanzamiento de dados.
- El Forum, donde podremos conseguir cartas con diversos efectos, que nos darán beneficios o perjuicios a los otros jugadores.
- La Curia, que nos permite variar las posiciones del senado, una de las condiciones de victoria.

Existen varios factores que hacen que se termine la partida. En primer lugar, el motivo más común que finaliza la partida es la derrota, por descender en cualquiera de los tracks (caminos de los tableros). Otro motivo es la victoria de alguno de los jugadores, esto ocurre con el cumplimiento de ambas cartas de objetivo por parte del mismo. Sin embargo, la partida puede terminar para un jugador si este llega a quedarse sin monedas. La duración de las partidas es muy relativa dependiendo de la cantidad de jugadores, aunque de manera general las partidas duran 1 hora.

Diagrama(s) UML:

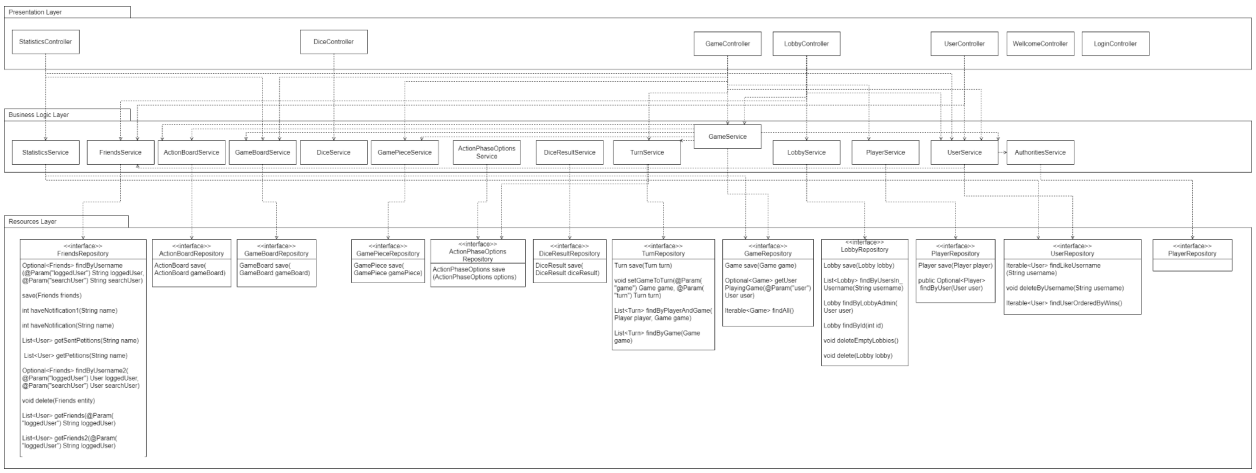
Diagrama de Dominio/Diseño



Para ver con más detalle, consultar el enlace:

[Diagrama de Diseño]

Diagrama de Capas (incluyendo Controladores, Servicios y Repositorios)



Para ver con más detalle, consultar el enlace:

[Diagrama de Capas]

Patrones de diseño y arquitectónicos aplicados

Patrón: Estado

Tipo: de Diseño

Contexto de Aplicación

Vae Victis divide su dinámica de turnos en 3 etapas o fases: fase de desgracia, fase de apoyo y fase de influencia. Cada una de ellas cuenta con una implementación distinta.

src\main\java\vaevictis\game\turn\Phases

Clases o paquetes creados

Paquetes:

- Action
- ActionOptions
- Doom

Clases:

- ActionFirstState.java
- ActionSecondState.java
- AerariumFirstState.java
- AerariumFirstState.java
- CuriaFirstState.java
- CuriaSecondState.java

- DomusPalatinaFirstState.java
- DomusPalatinaSecondState.java
- EndTurnState.java
- ForumFirstState.java
- ForumSecondState.java
- LimitesFirstState.java
- LimitesSecondState.java
- ThermaeFirstState.java
- ThermaeSecondState.java
- DoomDiceDeleteState.java
- DoomDiceThrowState.java
- DoomState.java
- Phase.java
- PhaseRepository.java
- PhaseService.java
- PhaseState.java
- PhaseStateRepository.java
- PhaseStateService.java

Ventajas alcanzadas al aplicar el patrón

- Aplicando el patrón eliminamos una estructura “Switch-Case” con más de 3 posibilidades, simplificando así la detección de fases.
- El código queda más separado, estructurado y legible.
- Cada fase tiene su clase asociada y su lógica queda implementada dentro de la misma de forma que cualquier cambio en la lógica de la fase es ajena a la de las otras fases evitando así errores.
- Toda la implementación de las fases queda localizada en un paquete.

Patrón: MVC

Tipo: Arquitectónico

Contexto de Aplicación

El patrón Modelo Vista Controlador (MVC) es el patrón arquitectónico que estructura nuestra aplicación completa.

Clases o paquetes creados

Todas las clases de la aplicación siguen la estructura del patrón MVC.

Ventajas alcanzadas al aplicar el patrón

- Soporte para utilizar numerosas vistas, dependiendo de la interfaz que se vaya a mostrar.

- Favorece la alta cohesión, el bajo acoplamiento y la separación de responsabilidades.

Patrón: Front Controller

Tipo: Arquitectónico

Contexto de Aplicación

El patrón Front Controller es implementado internamente en el proyecto por el propio Spring (su implementación se llama *DispatcherServlet*).

Clases o paquetes creados

Todas las clases con la notación @Controller.

Ventajas alcanzadas al aplicar el patrón

- El control centralizado permite hacer cumplir las políticas de toda la aplicación, como el seguimiento y la seguridad del usuario.
- La implementación del patrón normalmente ya la proporciona el propio framework de desarrollo.
- Proporciona funcionalidades adicionales, como el procesamiento de solicitudes para la validación y transformación de parámetros.

Patrón: Inyección de dependencias

Tipo: Arquitectónico

Contexto de Aplicación

El patrón de inyección de dependencias es implementado internamente en el proyecto por el propio Spring.

Clases o paquetes creados

Todas las clases del proyecto consideradas como *beans*.

Ventajas alcanzadas al aplicar el patrón

- Certeza de que solo se crea una instancia de cada clase, y de que las clases que las necesitan, pueden acceder a ellas.
- Se le permite al framework explotar la idea del patrón proxy y de la programación orientada a aspectos.

DP1 2021/22
Vae Victis Grupo: G4-05

Patrón: Proxy

Tipo: Arquitectónico

Contexto de Aplicación

El patrón proxy es implementado internamente en el proyecto por la propia inyección de dependencias de Spring.

Clases o paquetes creados

Todas las clases del proyecto consideradas como *beans*.

Ventajas alcanzadas al aplicar el patrón

- Certeza de que solo se crea una instancia de cada clase, y de que las clases que las necesitan, pueden acceder a ellas.

Patrón: Template View

Tipo: de Diseño

Contexto de Aplicación

El patrón Template View es implementado en el proyecto por el uso de JSP/JSTL.

\src\main\webapp\WEB-INF\jsp

\src\main\webapp\WEB-INF\tag

Clases o paquetes creados

Todos los archivos con extensión .jsp y .tag.

Ventajas alcanzadas al aplicar el patrón

- Permite escribir código Java en las páginas html.
- Permite reutilizar código para las vistas mediante el uso de tags.

Patrón: Domain Model

Tipo: de Diseño

Contexto de Aplicación

El patrón Domain Model define la forma de organizar la lógica de negocio del proyecto.

Clases o paquetes creados

Todos los archivos referentes a la lógica de negocio.

Ventajas alcanzadas al aplicar el patrón

- Permite implementar lógica de negocio compleja.
- La mayoría de los frameworks lo soportan hoy en día.

Patrón: Capa de Servicios

Tipo: de Diseño

Contexto de Aplicación

El patrón capa de servicios define la forma de organizar la lógica de negocio del proyecto.

Clases o paquetes creados

Todos los archivos referentes a la lógica de negocio.

Ventajas alcanzadas al aplicar el patrón

- Permite establecer límites dentro de la lógica de negocio para establecer separación de responsabilidades.

Patrón: Meta-data Mapper

Tipo: Arquitectónico

Contexto de Aplicación

El patrón Meta-data Mapper define la forma de mover datos de objetos Java en memoria a la base de datos y viceversa.

Clases o paquetes creados

Todos los archivos referentes a objetos Java.

Ventajas alcanzadas al aplicar el patrón

- Permite mantener independencia entre objetos Java y la base de datos.

Patrón: Campo de identidad

Tipo: de Diseño

Contexto de Aplicación

El patrón de campo de identidad se aplica a todas las clases Java consideradas entidades.

Clases o paquetes creados

Todas las clases Java consideradas entidades.

Ventajas alcanzadas al aplicar el patrón

- Permite añadir funcionalidades para la identificación de entidades.
- Se corresponde con la clave primaria en la base de datos.

Patrón: Capa supertipo

Tipo: de Diseño

Contexto de Aplicación

El patrón de capa supertipo se aplica a prácticamente todas las entidades del proyecto.

Clases o paquetes creados

Todas las entidades que heredan de las clases BaseEntity o NamedEntity.

Ventajas alcanzadas al aplicar el patrón

- Permite añadir funcionalidades comunes a varias entidades a la vez.
- Refactoriza y limpia el código.

Patrón: Repositorio

Tipo: de Diseño

Contexto de Aplicación

El patrón repositorio se aplica a las clases que encapsulan la lógica necesaria para acceder a la base de datos.

Clases o paquetes creados

Todas las clases cuyo nombre acaba en *'Repository'*, las cuales heredan de Repository o CrudRepository.

Ventajas alcanzadas al aplicar el patrón

- Permite utilizar funciones de acceso a la base de datos ya implementadas.

Patrón: Carga inmediata

Tipo: de Diseño

Contexto de Aplicación

El patrón carga inmediata se aplica a las entidades que contengan relaciones denotadas con *fetch = FetchType.EAGER*, o bien, que contengan relaciones *one to one* o *many to one* sin modificar su *fetch*.

Clases o paquetes creados

Todas las entidades que contengan relaciones denotadas con *fetch = FetchType.EAGER*, o bien, que contengan relaciones *one to one* o *many to one* sin modificar su *fetch*.

Ventajas alcanzadas al aplicar el patrón

- No hay impacto negativo en el rendimiento.

Patrón: Carga perezosa

Tipo: de Diseño

Contexto de Aplicación

El patrón carga perezosa se aplica a las entidades que contengan relaciones denotadas con *fetch = FetchType.LAZY*, o bien, que contengan relaciones *one to many* o *many to many* sin modificar su *fetch*.

Clases o paquetes creados

Todas las entidades que contengan relaciones denotadas con *fetch = FetchType.LAZY*, o bien, que contengan relaciones *one to one* o *many to one* sin modificar su *fetch*.

Ventajas alcanzadas al aplicar el patrón

- Cargas iniciales más rápidas.
- Menos consumo de memoria.

Patrón: Paginación

Tipo: de Diseño

Contexto de Aplicación

El patrón de paginación se ha usado en la interfaz de búsqueda de usuarios para mostrar los resultados de forma paginada.

Clases o paquetes creados

Se ha editado *UserController.java*, *UserService.java* y la vista *userLists.jsp*

Ventajas alcanzadas al aplicar el patrón

- Mejor visualización de listas demasiado largas.

Decisiones de diseño

Decisión 1: Implementación del juego mediante tecnología de WebSocket

Descripción del problema:

Debido a que Vae Victis necesita de una actualización constante de tableros de juego y es un juego dinámico, necesitábamos algún tipo de actualización automática de las vistas de forma que todos los jugadores puedan seguir la partida a tiempo real.

Alternativas de solución evaluadas:

Alternativa 1.a: Recarga de página al final de cada fase o turno.

Ventajas:

- Simple, no requiere nada más que una función al cambiar de fase o turno que haga que la vista se actualice con los datos de la partida ya modificados una vez pasado el turno.

Inconvenientes:

- No es una actualización dinámica o “pasiva”, es decir, requiere de un evento específico como el cambio de turno o fase para que todo jugador de la partida reciba la información actualizada.
- Los jugadores no dispondrán de la información necesaria para realizar acciones fuera de turno, como por ejemplo bloquear dados de la fase de desgracia.
- La constante recarga de la vista provoca parpadeos con la información siendo poco estético y molesto.

Alternativa 1.b: Recarga de página con temporizador.

Ventajas:

- Simple, no requiere nada más que una función con un temporizador que haga que la vista se actualice con los datos de la partida ya modificados una vez pasado el turno.

Inconvenientes:

- Los jugadores no dispondrán de la información necesaria para realizar acciones fuera de turno de forma inmediata, necesitarán esperar a la actualización para que se muestre
- La constante recarga de la vista provoca parpadeos con la información siendo poco estético y molesto.

Alternativa 1.c: Uso de WebSocket.

Ventajas:

- Actualización instantánea. Se dispone en todo momento de toda la información actualizada.
- La vista se actualiza dinámicamente, de forma que no se sufren los parpadeos como en las otras opciones.

Inconvenientes:

- Complejidad añadida a la hora de programar para el equipo en general. Es una tecnología que involucra javascript, en la cuál debemos manejar 1 canal general para la partida y todos los datos que le correspondan. Y otro canal privado para cada uno de los jugadores de la partida.
- Necesidad de manejar o tener en cuenta por un lado la interfaz y por otro el servidor.

Justificación de la solución adoptada

Optamos por la implementación con WebSocket ya que priorizamos el conocimiento en todo momento de la información de la partida y el dinamismo o sensación de continuidad durante la misma de forma que no hay parpadeos debido a la recarga.

El hecho de decantarnos por el uso de esta tecnología, se vio en parte influenciado por el posible aumento de la calificación en el proyecto.

Decisión 2: Auditoría mediante Hibernate Envers

Descripción del problema:

Debíamos implementar la auditoría de la entidad User.

Alternativas de solución evaluadas:

Alternativa 2.a: Auditoría mediante Spring Data JPA.

Ventajas:

- Ayuda sobre la implementación disponible en las diapositivas de teoría de la asignatura.

Inconvenientes:

- No nos convenía utilizarla porque para implementarlo de manera sencilla, tendríamos que hacer la entidad User herdara de AuditableEntity, la cual heredaba a su vez de BaseEntity, la cual tiene ya un @Id distinto al que tenía la entidad User.

Alternativa 2.b: Auditoría mediante Hibernate Envers.

Ventajas:

- Ayuda sobre la implementación disponible en las diapositivas de teoría de la asignatura.

Inconvenientes:

- La consulta a la base de datos para obtener la información de la auditoría no viene explicada en las diapositivas, por lo que hay que investigar por cuenta propia.

Justificación de la solución adoptada

Optamos por la implementación mediante Hibernate Envers, ya que por el inconveniente del campo @Id mencionado en la alternativa de Spring Data JPA, nos convenía más la opción de Hibernate Envers.

Debido a la falta de información en las diapositivas sobre cómo acceder a los datos de la auditoría en la base de datos, hubo que investigar la manera de hacerlo, ya que al no ser Entidades mapeadas por nosotros, no podíamos hacer un repositorio para acceder a las tablas. Finalmente lo conseguimos.

Decisión 3: Estado online de los jugadores

Descripción del problema:

Debíamos implementar que un usuario pudiera ver quiénes de sus amigos estaban online.

Alternativas de solución evaluadas:

Alternativa 3.a: Atributo boolean *online* en entidad User.

Ventajas:

- Simpleza de implementación, ya que cuando el jugador iniciara sesión, solo habría que asignarle el valor true al atributo, y false al cerrar sesión.

Inconvenientes:

- Encontrar forma de poder implementar lógica de negocio propia al hacer login y logout.

Alternativa 3.b: Obtener lista de los usuarios cuya sesión está iniciada actualmente.

Ventajas:

- Comprobación simple mediante método *contains* de la lista .

Inconvenientes:

- Implementación compleja para poder obtener la lista.

Justificación de la solución adoptada

Optamos por la opción de la lista ya que no fuimos capaces de editar el controlador interno de logout de Spring. Descubrimos la posibilidad de implementarlo mediante la lista, e investigamos bastante hasta que dimos con la forma de hacerlo, lo cual incluía editar la clase *SecurityConfiguration.java*. Concretamente, añadir código a su método *configure* y crear dentro un par de @Beans. Además, como el estado online de los amigos se podía ver desde todas las vistas que contuvieran el panel lateral social, en cada controlador que redirigiera a dichas vistas, había que incluir un @Autowired especial y por último, añadir al *model* la información necesaria.

Decisión 4: Panel desplegable social

Descripción del problema:

Debíamos implementar una forma de acceder a la lista de amigos y peticiones de amistad desde varias vistas que la necesitaran.

Alternativas de solución evaluadas:

Alternativa 4.a: Vista aparte y exclusiva para lista de amigos.

Ventajas:

- Simpleza de implementación.

Inconvenientes:

- Tener que salir de la vista actual para poder invitar amigos a una partida o aceptar peticiones.

Alternativa 4.b: Div en cada vista.

Ventajas:

- Simpleza de implementación.

Inconvenientes:

- Ocupa espacio de la vista en cuestión.

Alternativa 4.c: Panel desplegable.

Ventajas:

- Al ser desplegable, no ocupa espacio de la vista, solo se despliega al hacerle click.
- Refactoriza el código

Justificación de la solución adoptada

Optamos por implementar el panel social desplegable, ya que aunque tuvimos que aprender a diseñarlo, nos permitió una forma compacta de tener acceso a los amigos y a las peticiones desde cualquier vista

que necesitáramos, sin tener que copiar el mismo div en cada vista, repitiendo código y utilizando espacio en la propia vista.

Decisión 5: Separar los estados de la fase en clases distintas

Descripción del problema:

Debíamos implementar una forma de navegar entre fases eficiente y clara, de forma que la lógica de cada fase y el cálculo de las próximas fases no quedara muy compleja.

Alternativas de solución evaluadas:

Alternativa 5.a: Hacer un *switch* en cada fase que determine qué acción se ha tomado.

Ventajas:

- No es muy complejo, solo se necesitan 3 clases de Fases.

Inconvenientes:

- Hay un *switch* muy grande y tendríamos que duplicar una gran cantidad de código, ya que toda la lógica de todas las acciones deben estar implementadas en cada una de las fases.

Alternativa 5.b: Separar los estados de la fase en clases distintas.

Ventajas:

- Toda la lógica de fases queda separada.
- Se elimina el *switch-case* a la hora de determinar cómo se resuelve una fase.

Inconvenientes:

- Mayor complejidad.
- Fase dividida en más posibles etapas.

Justificación de la solución adoptada

Optamos por implementar los estados de la fase en clases distintas, ya que simplifica la navegabilidad entre fases, y aunque requiera mayor complejidad, la lógica de las fases queda más ordenada al dividirse.