DP1 2022-2023

Documento de Diseño del Sistema

Proyecto Buscaminas

https://github.com/gii-is-DP1/dp1-2022-2023-l5-5.git

Miembros:

- Ángela Bernal Martín
- Mercedes Iglesias Martín
- Paola Martín Sánchez
- Pablo Quindós de la Riva
- Santiago Zuleta de Reales Toro

Tutor: José Antonio Parejo Maestre

GRUPO L5-05

Versión V2

13/01/2023

Historial de versiones

Fecha	Versión	Descripción de los cambios	Sprint
20/11/2022	V1	● Añadida la introducción	3
29/11/2022	V1	● Añadidos patrones MVC y Front Controller	3
30/11/2022	V1	● Añadidos patrones Template View y Domain Model	3
05/12/2022	V1	● Añadidos patrones Service Layer, Data Mapper, Repository pattern	3
27/12/2022	V2	● Añadidas decisiones de diseño	4
08/01/2023	V2	● Añadidas decisiones de diseño	4
10/01/2023	V2	● Revisión y realización de documento	4
12/01/2023	V2	● Añadidos los diagramas actualizados	4

Contents

Introducción

El objetivo en este proyecto consiste en diseñar e implementar un sistema y sus pruebas asociadas al juego del Buscaminas, el cual es muy intuitivo y fácil de aprender a jugar.

El Buscaminas es un juego de un solo jugador, cuyo objetivo es despejar todas las casillas de un tablero que no oculten una mina, e ir marcando con banderas aquellas en las que se crea que puede contener una mina.

Algunas casillas tienen un número, este número indica las minas que hay en todas las casillas circundantes. Así, si una casilla tiene el número 3, significa que de las ocho casillas que hay alrededor (si no es en una esquina o borde) hay 3 con minas y 5 sin minas. Si se descubre una casilla sin número indica que ninguna de las casillas vecinas tiene mina y estas se descubren automáticamente.

Se puede poner una marca en las casillas (normalmente indicada con un icono de una bandera roja) donde el jugador piensa que hay minas para ayudar a descubrir las que están cerca.

La partida termina cuando se descubre una casilla con una mina que quiere decir que el jugador ha perdido la partida, o bien, cuando éste despeja todas las casillas que no tengan minas, ganando así la partida. Por tanto, la duración del juego depende de cuánto tiempo emplee el jugador en ganar o perder, es decir, no hay un tiempo límite.

Después de cada partida, todos los datos relevantes de ella como pueden ser la duración, si se ha ganado o perdido y datos específicos de la partida en concreto, se almacenan para hacer cálculos estadísticos y construir distintos rankings de jugadores.

La funcionalidad que vamos a implementar en el sistema estará dividida en tres módulos, los cuales son:

- En primer lugar, el **módulo de juego,** este módulo proporciona a los jugadores las funcionalidades de crear y jugar partidas, además de ver un listado de partidas que ha creado y jugado el propio jugador.
- En segundo lugar, el **módulo de gestión de usuarios**, este módulo proporciona a los jugadores registrarse, loguearse, cerrar sesión y editar su perfil personal, mientras que el administrador podrá ver un listado de los jugadores registrados en el juego, realizar las operaciones CRUD de jugadores, y ver la auditoría de los datos de los perfiles de los jugadores.
- En tercer lugar **el módulo de estadísticas**, este permitirá al jugador ver estadísticas totales y por jugador relacionadas con el juego.

Una vez que un jugador esté registrado y haya iniciado sesión en la aplicación, podrá crear una partida, pero antes podrá elegir el nivel de dificultad del juego. Habrá 3 opciones, nivel “EASY”, “MEDIUM” y “DIFFICULT”.
A continuación, el jugador jugará la partida con el nivel de dificultad seleccionado, y deberá seguir todas las normas del juego comentadas al principio de esta sección del documento.

Por último, como está indicado antes también se va a implementar el módulo de estadísticas. Dentro de este módulo si estás registrado como jugador podrás acceder a una página de estadísticas donde se mostrarán las estadísticas del propio jugador registrado (partidas jugadas, partidas ganadas, partidas perdidas, tiempo medio de partida, mayor tiempo de partida, menor tiempo de partida y menor número de movimientos en finalizar una partida. Todas estas estadísticas para cada nivel de dificultad), además de una serie de estadísticas generales. Si se está registrado como administrador puedes ver las estadísticas de cualquier jugador.

Dentro del módulo de estadísticas hay tanto un apartado de logros como uno de ranking de jugadores, que saldrán en orden de jugadores que más partidas hayan ganado en general y sale para todos los usuarios, indiferentemente de cómo estemos registrados. Para cada jugador habrá una vista que muestre sus logros obtenidos. Mientras si se está registrado como administrador podrá ver los logros de cualquier jugador y también como administrador se podrán crear nuevos logros y editar logros existentes.

Diagrama(s) UML:

Diagrama de Dominio/Diseño

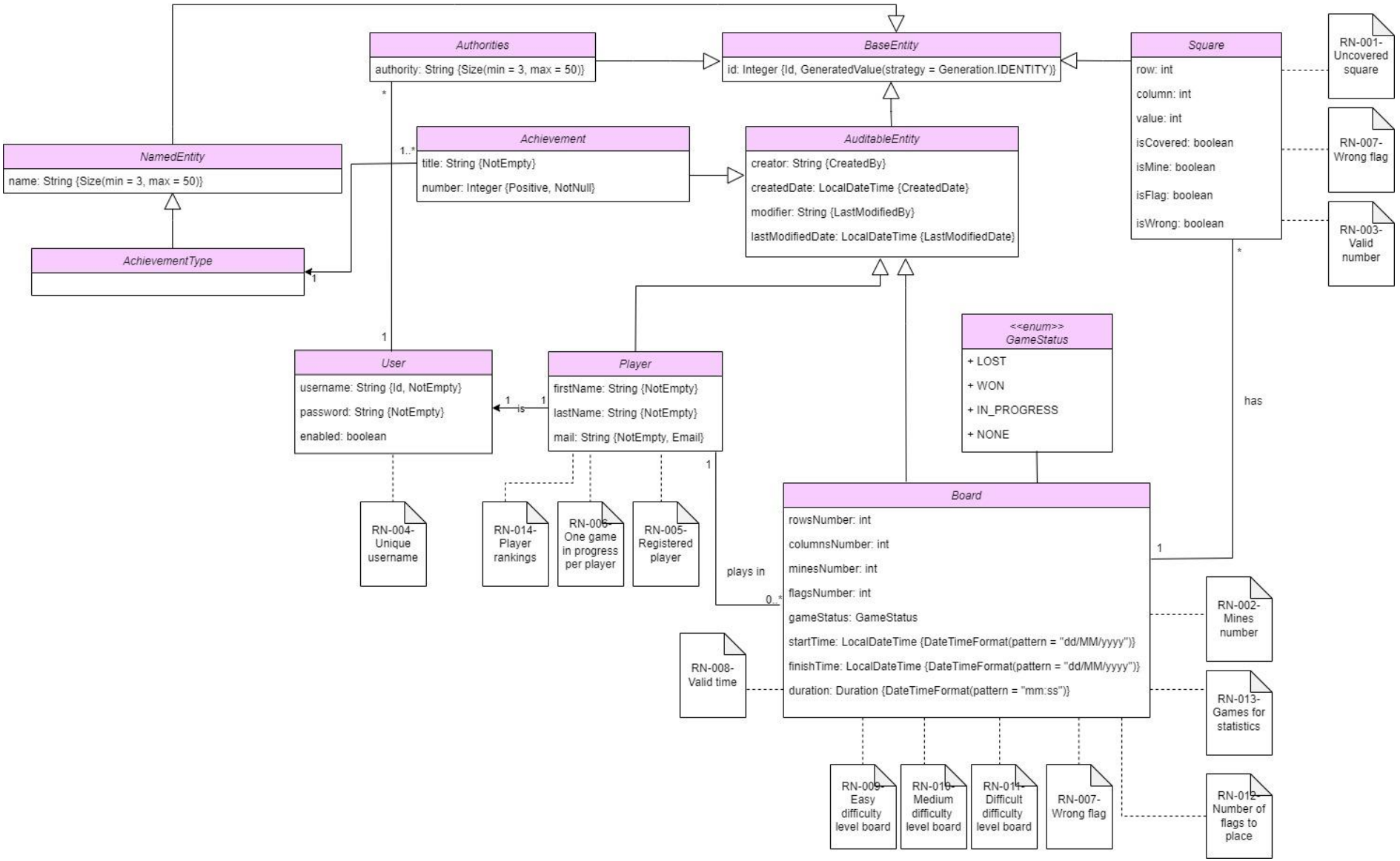
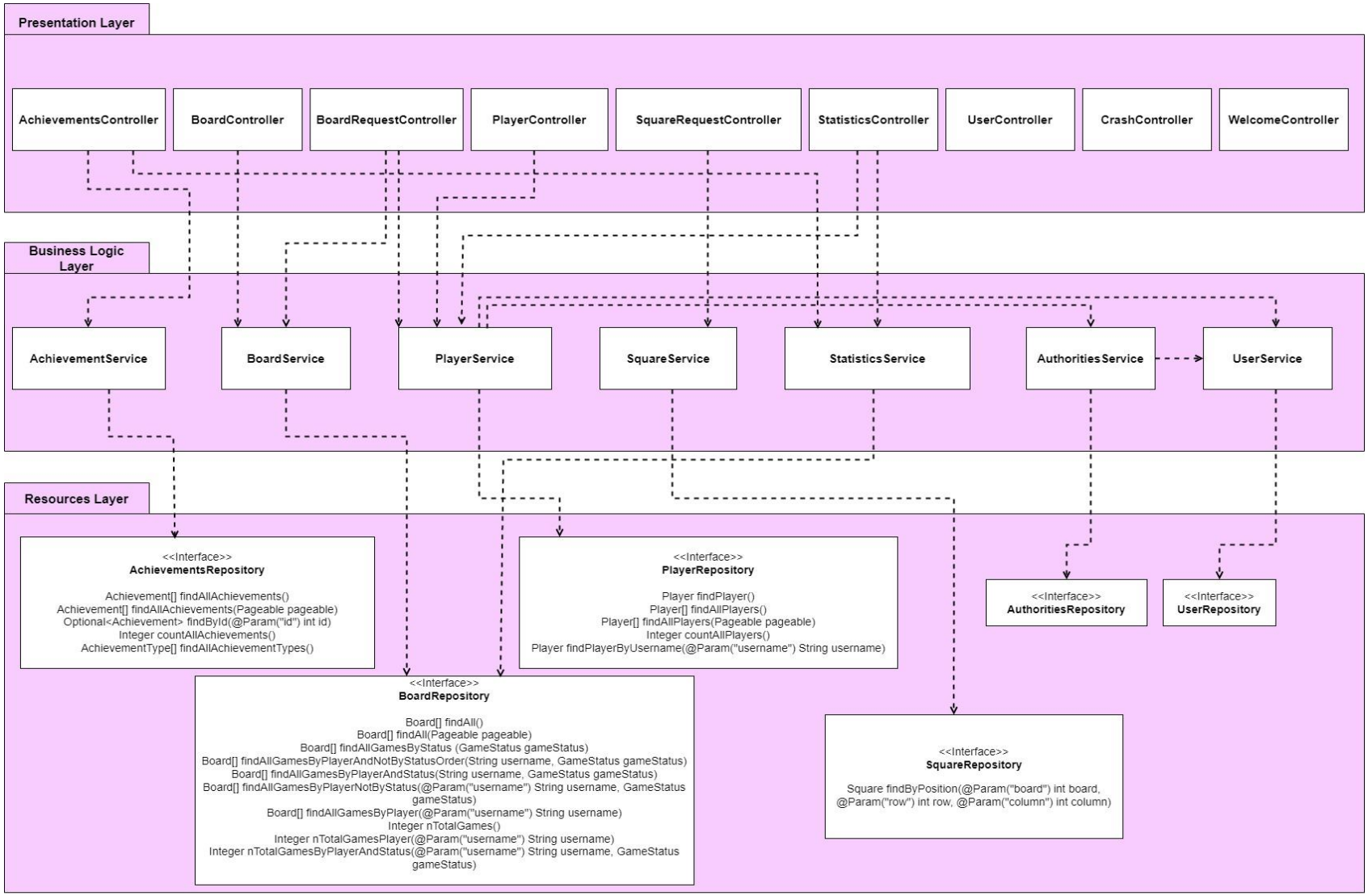

Authorities
authority: String {Size(min = 3, max = 50)}
BaseEntity
id: Integer {Id, GeneratedValue(strategy = Generation.IDENTITY)}
Square
row: int
column: int
value: int
isCovered: boolean
isMine: boolean
isFlag: boolean
isWrong: boolean
RN-001- Uncovered square
RN-007- Wrong flag
RN-003- Valid number
NamedEntity
name: String {Size(min = 3, max = 50)}
Achievement
title: String {NotEmpty}
number: Integer {Positive, NotNull}
AuditableEntity
creator: String {CreatedBy}
createdDate: LocalDateTime {CreatedDate}
modifier: String {LastModifiedBy}
lastModifiedDate: LocalDateTime {LastModifiedDate}
AchievementType
User
username: String {Id, NotEmpty}
password: String {NotEmpty}
enabled: boolean
Player
firstName: String {NotEmpty}
lastName: String {NotEmpty}
mail: String {NotEmpty, Email}
is
<<enum>> GameStatus
+ LOST
+ WON
+ IN_PROGRESS
+ NONE
has
RN-004- Unique username
RN-014- Player rankings
RN-006- One game in progress per player
RN-005- Registered player
plays in
Board
rowsNumber: int
columnsNumber: int
minesNumber: int
flagsNumber: int
gameStatus: GameStatus
startTime: LocalDateTime {DateTimeFormat(pattern = "dd/MM/yyyy")}
finishTime: LocalDateTime {DateTimeFormat(pattern = "dd/MM/yyyy")}
duration: Duration {DateTimeFormat(pattern = "mm:ss")}
RN-002- Mines number
RN-008- Valid time
RN-013- Games for statistics
RN-009- Easy difficulty level board
RN-010- Medium difficulty level board
RN-011- Difficult difficulty level board
RN-007- Wrong flag
RN-012- Number of flags to place

Diagrama de Capas (incluyendo Controladores, Servicios y Repositorios)



Patrones de diseño y arquitectónicos aplicados

En esta sección se especifica el conjunto de patrones de diseño y arquitectónicos aplicados durante el proyecto.

Patrón: Modelo-Vista-Controlador (MVC)

Tipo: Arquitectónico

Contexto de Aplicación

Este patrón se utiliza en toda la aplicación, separa los datos de la aplicación (capa de recursos), la interfaz de usuario (capa de presentación) y la lógica de negocio en 3 componentes distintos:

- **Modelo**: es la representación de la información. Incluye tanto los datos como la lógica empresarial necesaria para trabajar con ellos. Dentro del modelo están las entidades, los repositorios y los servicios.
- **Vista**: representa la información proporcionada por el responsable del tratamiento. Suele realizarse mediante una interfaz de usuario.
- **Controlador**: responde a eventos en la interfaz de usuario, invoca cambios en el modelo y, probablemente en la vista.

Clases o paquetes creados

Se han creado una serie de clases que hacen referencia al modelo de la aplicación. Como se ha indicado antes, el modelo incluye las entidades, repositorios y servicios con las que se han trabajado para el desarrollo de la aplicación.

- Entidades: Board.java, BaseEntity.java, NamedEntity.java, Person.java, Player.java, Square.java, Authorities.java y User.java, Achievement.java y AchievementType.java.
- Servicios: BoardService.java, PlayerService.java, SquareService.java, AuthoritiesService.java, UserService.java, StatisticsService.java y AchievementsService.java.
- Repository: BoardRepository.java, PlayerRepository.java, SquareRepository.java, AuthoritiesRepository.java, UserRepository.java y AchievementRepository.java.

En cuanto a la vista de la aplicación, podemos encontrar varias clases jsp relacionadas con las entidades del modelo, y que permiten mostrar la interfaz de usuario de la aplicación.

- boards: board.jsp, gamesList.jsp, gamesListInProgress.jsp, gamesListPlayer.jsp y setDifficulty.jsp
- players: createPlayerForm.jsp, playersdeleteAdmin.jsp, playersList.jsp, playersProfile.jsp y updatePlayerForm.jsp
- achievements: achievementsList.jsp, createAchievementsForm.jsp, updateAchievementForm.jsp y playerAchievements.jsp
- statistics: ranking.jsp y statis.jsp
- exception.jsp, error.jsp y welcome.jsp.

Por último, en cuanto a los controladores implementados son: BoardController.java, BoardRequestController.java, AchievementsController.java, StatisticsController.java, PlayerController.java, SquareController.java, UserController.java, CrashController.java, DicesOnSessionController.java y WelcomeController.java .

Ventajas alcanzadas al aplicar el patrón

- Tiene una fácil organización, puesto que solo cuenta con tres componentes.
- Es un patrón que se puede adaptar a diferentes frameworks.
- Se puede escalar fácilmente.
- Facilita el trabajo en equipo.
- Facilita el desarrollo y las pruebas de cada tipo de componente en paralelo.

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. La clase @controller tiene los métodos adecuados para el manejo de solicitudes.

Clases o paquetes creados

El paquete creado para la adaptación de este patrón de diseño es el org.springframework.samples.minesweeper.web

Ventajas alcanzadas al aplicar el patrón

En primer lugar este patrón conviene aplicarlo por 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 proporcionada 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: Template View

Tipo: Diseño

Contexto de Aplicación

SpringBoot utiliza JSP/JSTL para la visualización de las diferentes vistas de la aplicación.

Una página de servidor Java (JSP) nos permite escribir código java en las páginas html. Cada página JSP se traduce en un servlet la primera vez que se invoca y en adelante es el código del nuevo servlet el que se ejecuta.

Ciertas variables están disponibles en las páginas JSP sin necesidad de declararlas o configurarlas, entre otras:

- request. El objeto HttpServletRequest asociado a la petición.
- response. El objeto HTTPServletResponse asociado a la petición.
- out. Un objeto PrintWriter utilizado para enviar datos al cliente.
- session. El objeto HttpSession asociado a la petición.
- exception. Objeto de excepción con información sobre la excepción lanzada.

Clases o paquetes creados

Este patrón se ve reflejado en cualquier vista de la aplicación (mencionadas en el apartado del patrón Modelo-Vista-Controlador). Por ejemplo, las vistas para visualizar el listado de partidas, que tenemos varias ya que el listado de partidas en progreso y de todas las partidas jugadas solo pueden ser visualizadas por el administrador, gamesList.jsp, gamesListInProgress.jsp y gamesListPlayer.jsp.

Ventajas alcanzadas al aplicar el patrón

Nos permite utilizar una lógica de presentación separada (código html) del código Java (lógica de negocio). Proporciona etiquetas JSP incorporadas y también permite desarrollar etiquetas JSP personalizadas y utilizar etiquetas suministradas por terceros.

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 el paquete org.springframework.samples.minesweeper.model. También serían todas las clases .java (mencionadas en el apartado del patrón Modelo-Vista-Controlador) con el nombre de una entidad de nuestro diagrama de modelo de datos, como son Board.java, Player.java, Square.java, User.java…

Ventajas alcanzadas al aplicar el patrón

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

Patrón: Service Layer

Tipo: Arquitectónico

Contexto de Aplicación

Se usa en los servicios y se encargan de la comunicación entre los controladores y los repositorios además de manejar la mayor parte de la lógica sobre los datos.

Clases o paquetes creados

Se han utilizado las siguientes clases: BoardService.java, PlayerService.java, SquareService.java, AuthoritiesService.java, UserService.java, StatisticsService.java y AchievementsService.java.

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: Data Mapper

Tipo: Diseño

Contexto de Aplicación

Nos ayuda a establecer un contexto entre las relaciones de los atributos y entidades a la base de datos y, es usado generalmente para indicar cómo se relacionan los atributos de ciertas entidades con otras clases facilitando así el manejo de la base de datos.

Clases o paquetes creados

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

Ventajas alcanzadas al aplicar el patrón

Permite trasladar datos entre la base de datos y objetos manteniendo la independencia entre ambas. Además, proporciona escalabilidad y reutilización, y reduce la sobrecarga de datos transitorios entre componentes de software.

Patrón: Repository Pattern

Tipo: Diseño

Contexto de Aplicación

El patrón se ha aplicado para encapsular la lógica de negocio necesaria para acceder a los datos. En nuestro caso todos los repositorios extienden de CrudRepository.

Clases o paquetes creados

Se han utilizado las siguientes clases: BoardRepository.java, PlayerRepository.java, SquareRepository.java, AuthoritiesRepository.java, UserRepository.java y AchievementRepository.java.

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. 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: 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. En nuestro caso utilizamos este patrón para mostrar con paginación el listado de jugadores registrados en el juego.

Clases o paquetes creados

Se han utilizado las siguientes clases: PlayerController.java, PlayerRepository.java, PlayerService.java, AchievementController.java, AchievementRepository.java, AchievementService.java.

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: Identify Field

Tipo: Diseño

Contexto de Aplicación

Es esencial ya que permite identificar únicamente a las distintas instancias de las entidades. Corresponde a la clave primaria en la base de datos. En nuestro caso viene implícita en la entidad “BaseEntity.java” de la que heredan las entidades de nuestro proyecto.
Además se utiliza la clase NamedEntity.java para la implementación de la clase AchievementType.java, que solo tiene los atributos ‘id’ y ‘name’, ambos heredados de NamedEntity.java.

Clases o paquetes creados

La aplicación de este patrón se puede ver en cualquiera de las clases de las entidades, por ejemplo, “Square.java”, extendiendo de la clase “BaseEntity.java”. Aunque en el caso de la entidad “Player.java'', esta entidad extiende a “Person.java” que a su vez esta extiende a “BaseEntity.java”.

Ventajas alcanzadas al aplicar el patrón

Los diferentes objetos almacenados dispondrán de un identificador único que se autogenera lo cual facilita la lógica a la hora de acceder a unos datos específicos en memoria.

Patrón: Layer Supertype

Tipo: Diseño

Contexto de Aplicación

En este caso hacemos uso de este patrón de diseño en clases que heredan ciertas características de otras clases, como es el caso de “BaseEntity.java” que lega alguna de sus características a clases como “AuditableEntity.java” por ejemplo.

Clases o paquetes creados

Los paquetes y sus clases creados con este patrón son:

- paquetes: “org.springframework.samples.minesweeper.model”.
- clases: “BaseEntity.java”, “NamedEntity.java” y “AuditableEntity.java” en “org.springframework.samples.minesweeper.model”.

Ventajas alcanzadas al aplicar el patrón

Con este patrón nos hemos ayudado con los id de nuestras clases, ya que con “BaseEntity.java” nos genera el id sin necesidad de definirlo en cualquier clase, ya que nuestras clases se extienden a ellas y, “NamedEntity.java” es capaz de generar el nombre.

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: Quitar entidad Game

Descripción del problema:

Complejidad de la estructura del proyecto y dificultad con otras implementaciones debido a tener las clases Board y Game por separado.

Alternativas de solución evaluadas:

Alternativa 1.a:

Unir las clases Game y Board en una única clase Board.

Ventajas:

- Simplificación de la estructura del proyecto
- Mayor facilidad con otras implementaciones

Inconvenientes:

- Cambio de diagramas
- Cambio grande en la estructura general del proyecto

Alternativa 1.b:

Dejar las dos clases separadas y seguir con el proyecto.

Ventajas:

- No se introducían grandes cambios en la estructura del proyecto definida

Inconvenientes:

- Mucha complejidad
- Muchos datos que conlleva una gran carga en la base de datos

Justificación de la solución adoptada

Elegimos la alternativa de diseño 1.a, ya que eliminando la entidad Game, se reduce la complejidad y la carga de la aplicación significativamente.

Decisión 2: Realización de tablero cuadrado

Descripción del problema:

Como grupo tomamos la decisión de utilizar representaciones cuadradas del tablero en vez de formas rectangulares.
Decidir la forma geométrica de las representaciones del tablero.

Alternativas de solución evaluadas:

Alternativa 2.a:

Definir representaciones del tablero cuadradas.
Ventajas:

- Simplifica la generación del tablero
- El juego Buscaminas original es cuadrado

Inconvenientes:

- No había

Alternativa 2.b:

Se planteó utilizar representaciones con diferentes formas como rectangulares o incluso irregulares.

Ventajas:

- Sería más original

Inconvenientes:

- Más complejo

Justificación de la solución adoptada

Finalmente generamos tableros de forma cuadrada para todos los niveles de dificultad, por lo que decidimos seguir la alternativa de diseño 2.a, ya que consideramos que tenía más ventajas que la otra alternativa.

Decisión 3: Cambio del estado del juego por un enumerado

Descripción del problema:

La forma de almacenar el estado del juego.

Alternativas de solución evaluadas:

Alternativa 3.a:

Definir los distintos estados del juego en un enumerado, compuesto por Lost, Won, In_progress, None.

Ventajas:

- Simplificación del modelo de la aplicación
- Simplificación de la carga de datos
- Simplificación relaciones entre entidades

Inconvenientes:

- No encontramos

Alternativa 3.b:

Se planteó que el estado del juego fuera una clase, que extendiera a NamedEntity, de manera que sus atributos fueran el id y el nombre generados por la misma.

Ventajas:

- No encontramos

Inconvenientes:

- Aumento de la carga de datos al crear una nueva tabla y relación con otra tabla.

Justificación de la solución adoptada

Al utilizar un enumerado para el estado del juego obtenemos más ventajas, por lo que elegimos la alternativa 3.a finalmente.

Decisión 4: Uso de clases del proyecto petclinic ya implementado

Descripción del problema:

Hemos decidido reutilizar clases del proyecto de petclinic de spring.

Alternativas de solución evaluadas:

Alternativa 4.a:

Hacer las cosas desde cero, sin tener en cuenta ni reutilizar clases de petclinic.

Ventajas:

- Elección de atributos justos y necesarios para el sistema.

Inconvenientes:

- Mucho más trabajo para realizar sin una ventaja muy diferencial en el desarrollo del proyecto.

Alternativa 4.b:

Reutilizar clases del proyecto de petclinic.

Ventajas:

- Reutilización del código de las clases y las vistas ya implementado, facilitando el trabajo tanto de diseño como de implementación.

Inconvenientes:

- Tenemos que adecuar las próximas clases que hagamos a las ya existentes. También puede ocasionar que los desarrolladores no lleguen a un grado de compresión del código del proyecto tan alto como si se hubiese hecho desde cero.

La alternativa evaluada fue realizar nuestras propias clases de usuario y jugador

Justificación de la solución adoptada

Decidimos utilizar las clases ya implementadas para reducir la carga de trabajo y no invertir tiempo en una tarea que realmente estaba hecha, por lo que finalmente elegimos la alternativa 4.b.

Decisión 5: Utilizar canvas

Descripción del problema:

Decidimos utilizar la etiqueta canvas de HTML para la representación del tablero, lo cual aporta una interfaz mucho más amigable para el jugador.

Alternativas de solución evaluadas:

Alternativa 5.a:

Utilizar la etiqueta canvas de HTML para mostrar una imagen del tablero que permite interactuar con el mismo haciendo clicks sobre la imagen.

Ventajas:

- Interfaz del juego mucho más amigable y entendible

Inconvenientes:

- Aumenta la complejidad de la implementación en gran medida

Alternativa 5.b:

La alternativa fue implementar la visualización del tablero con una tabla y un formulario, esto simplifica mucho la implementación de la jugabilidad y de la representación del tablero, pero disminuye en gran medida la experiencia de usuario.

Ventajas:

- Simplifica en gran medida la implementación de la jugabilidad.

Inconvenientes:

- Interfaz de juego de baja calidad y poco amigable

Justificación de la solución adoptada

Al vernos con tiempo suficiente (en el momento de la toma de decisión) para implementar el juego con canvas, elegimos la alternativa 5.a para que el usuario tenga una interfaz adecuada y que se adapte a la interfaz original del buscaminas.

Decisión 6: Enumerado de contenido de casilla

Descripción del problema:

Hemos quitado el enumerado de contenido de la casilla por una propiedad boolean que sea isMine, porque si no tiene mina una casilla tienen número o nada, y nada es lo mismo que el número 0.

Alternativas de solución evaluadas:

Alternativa 6.a:

Crear un enumerado que indique cómo está la casilla (MINE,EMPTY,FLAG) .

Ventajas:

- Simplificación del modelo de la aplicación
- Simplificación de la carga de datos
- Simplificación relaciones entre entidades

Inconvenientes:

- La casuística de un enumerado que tome esos valores daba conflictos con el resto de implementación.

Alternativa 6.b:

Crear una propiedad en la entidad Square boolean que devuelva true si es una mina y false si está vacía, otra de si está cubierta o no y otra final de si es una bandera o no.

Ventajas:

- Sencillez en cuanto a entendimiento de funcionamiento y por tanto una implementación de los métodos del juego más intuitiva.

Inconvenientes:

- Más carga de datos
- Mayor número de propiedades dentro de la entidad Square

Justificación de la solución adoptada

La alternativa evaluada es la que se contempló en primer momento, es decir, representar el contenido de la celda con un enumerado que podía tomar valores vacío, bandera y mina, por lo que finalmente elegimos la alternativa 6.b.

Decisión 7: Eliminación de juego

Descripción del problema:

Hemos decidido que no se pueda eliminar un juego

Alternativas de solución evaluadas:

Alternativa 7.a: Eliminación de partidas por parte del administrador

El administrador tiene la opción de eliminar una partida en concreto desde el listado de partidas

Ventajas:

- El administrador tiene más control sobre el sistema

Inconvenientes:

- Invertir trabajo en implementación poco relevante

Alternativa 7.b: El administrador no puede borrar partidas

El administrador tiene la opción de eliminar una partida en concreto desde el listado de partidas.

Ventajas:

- Simplificación de la implementación del sistema

Inconvenientes:

- El administrador no tiene control absoluto sobre los registros del sistema

Justificación de la solución adoptada

Finalmente decidimos adoptar la alternativa 7.b, por simplificación de la implementación, de manera que el borrado de partidas se realiza con una eliminación en cascada al eliminar un jugador.

Decisión 8: Implementación de la jugabilidad con JavaScript

Descripción del problema:

Decidimos emplear JavaScript para implementar la jugabilidad

Alternativas de solución evaluadas:

Alternativa 8.a

Utilizar código Java con Spring y JSP. De esta forma se crea una instancia de la clase tablero al empezar la partida y cada vez que se realiza un ‘clic’ en el tablero. Cada vez que esto ocurra el repositorio añadiría un tablero a la base de datos.

Ventajas:

- Usar la arquitectura recomendada para la aplicación.

Inconvenientes:

- Dificultades al actualizar el canvas con los datos proporcionados por el controlador
- Altos tiempos de carga al tener que recargar la vista por cada click en el tablero

Alternativa 8.b

Utilizar Javascript para obtener un tablero dinámico que se actualiza sin necesidad de recargar la vista en el navegador.

Ventajas:

- Evitar que la información tenga que viajar por todos los componentes del sistema, incluyendo consultas e inserciones en la base de datos por cada click en el tablero.

Inconvenientes:

- Incluir un lenguaje más en la implementación de la jugabilidad, cuando podría haber sido prescindible.
- Uso de una arquitectura de la aplicación no recomendada para implementar la jugabilidad.

Justificación de la solución adoptada

Utilizamos la alternativa 8.b porque nos resulta más fácil utilizarlo con canvas, decisión que tomamos anteriormente y nos condiciona para esta. Además, de esta manera toda la jugabilidad se realiza en el frontend y ahorramos que la información transite por tantos componentes de la aplicación, reduciendo tiempos de carga.

Decisión 9: Auditoría

Descripción del problema:

La realización de la auditoría es una de las principales decisiones que hemos tenido que tomar en el desarrollo del proyecto, ya que teníamos diferentes alternativas, todas muy factibles.

Alternativas de solución evaluadas:

Alternativa 9.a:

Auditoría con Spring Data JPA

Ventajas:

- Muy visual y transparente, haciendo que su implementación sea sencilla y sin mucho trabajo para los programadores.

Inconvenientes:

- Algo limitados los campos de la auditoría.

Alternativa 9.b:

Auditoría con Hibernate Envers

Ventajas:

- Sencillo de usar y de implementar, haciendo que podamos implementar un control de cambios de una manera muy limpia y poco intrusiva en nuestro código.

Inconvenientes:

- No audita al usuario por defecto.

Justificación de la solución adoptada

Para la creación de auditorías optamos por la alternativa 9.a ya que nos ha parecido la forma más sencilla e intuitiva de hacerlo. Los datos que proporciona esta auditoría reflejan bien cada modificación dada y nos es suficiente en nuestro proyecto.

Decisión 10: Cambio de AchievementType de enum a namedEntity

Descripción del problema:

La forma de almacenar los tipos de logros.

Alternativas de solución evaluadas:

Alternativa 10.a:

Definir los distintos estados del juego en un enumerado, compuesto por todos los tipos de logros.

Ventajas:

- Simplificación del modelo de la aplicación
- Simplificación de la carga de datos
- Simplificación relaciones entre entidades

Inconvenientes:

- Mayor dificultad para la realización de un formulario donde poder elegir el tipo de logro que queremos añadir o modificar.

Alternativa 10.b:

Se planteó que el tipo de logro fuera una clase, que extendiera a NamedEntity, de manera que sus atributos fueran el id y el nombre generados por la misma.

Ventajas:

- Mayor facilidad para implementar un select y c:forEach en el formulario donde elegir el tipo de logros que queremos añadir o modificar.
- Mayor facilidad para introducir nuevos tipos de logros (cuando estamos hablando de un número grande de ellos). Ya que al tener un id cada uno de ellos el conteo de estos es más sencillo.

Inconvenientes:

- Aumento de la carga de datos al crear una nueva tabla y relación con otra tabla.

Justificación de la solución adoptada

Al utilizar un enumerado para el estado del juego obtenemos más ventajas, pero no hacía muy complicado la implementación del formulario. Por lo que tuvimos que usar la alternativa 10.b.

Decisión 11: No relacionar la entidad Achievements con un User

Descripción del problema:

Relacionar o no la entidad Achievements con un User.

Alternativas de solución evaluadas:

Alternativa 11.a:

No poder la relación con User.

Ventajas:

- Eliminar la relación con la entidad 'User' que realmente no era necesaria, simplificando el diseño y modelo de la aplicación.
- Se simplifica la carga de registros en la base de datos, al tener que almacenar más filas.
- Menor complejidad a la hora de hacer que el administrador añada nuevos logros.

Inconvenientes:

- Mayor complejidad a la hora de implementar el listado de logros conseguidos por un jugador.

Alternativa 11.b:

Poner la relación con User.

Ventajas:

- Facilidad para dividir los listados de los usuarios que cumplan o no los diferentes logros.

Inconvenientes:

- Aumento de la complejidad de la carga de registros en la base de datos.
- Aumento de la complejidad del modelo de la aplicación.
- Aumento de la complejidad a la hora de hacer que el administrador añada nuevos logros.

Justificación de la solución adoptada

Tanto para quitar complejidad a la aplicación y sobre todo para que el administrador pudiese crear los logros sin tener que relacionar a un usuario con un logro en el formulario, elegimos la alternativa 11.a.