DP1 2021-2022

Documento de Diseño del Sistema

Proyecto Buscaminas

https://github.com/gii-is-DP1/dp1-2021-2022-g7-04

Miembros:

- CERRATO SÁNCHEZ, LUIS
- GUTIÉRREZ CONTRERAS, ERNESTO
- MARTÍNEZ SUÁREZ, DANIEL JESÚS
- STEFAN, BOGDAN MARIAN

Tutor: IRENE BEDILIA ESTRADA TORRES

GRUPO L7-04

Versión V6

30/09/2022

Historial de versiones

Fecha	Versión	Descripción de los cambios	Sprint
13/12/2020	V1	● Creación del documento	2
13/12/2020	V2	● Añadido diagrama de dominio/diseño ● Explicación de la aplicación del patrón caché	3
17/01/2022	V3	● Corregido diagrama de dominio/diseño ● Añadido diagrama de capas	4
17/08/2022	V4	● Añadido más decisiones de diseño	Septiembre
30/08/2022	V5	● Actualizar y finalizar el documento	Septiembre
30/09/2022	V6	● Corrección de errores y finalización del documento ● Corrección decisiones de diseño	Noviembre

Contenido

Introducción

Nuestro objetivo en este proyecto es diseñar e implementar un sistema y sus pruebas asociadas al conocido videojuego Buscaminas, el cual creemos que es una opción interesante a implementar al ser un juego simple y conocido por todo el mundo.

Las características del sistema que vamos a implementar se dividen principalmente en dos módulos:

- En primer lugar, el módulo de juego, que provee todas las funcionalidades para que los jugadores puedan crear y jugar partidas.
- En segundo lugar, el módulo de interfaz de usuarios y administración, mediante el cual los jugadores tendrán disponibles las funcionalidades de iniciar y cerrar sesión, registrarse y editar su perfil, mientras que los administradores podrán tener el control sobre estos perfiles.

El objetivo del juego es ir despejando las casillas que no contengan minas, e ir marcando las que sí contengan minas con banderas.

El transcurso de la partida es sencillo, se trata de un tablero con un número determinado de casillas, dependiendo de la dificultad escogida.
Una vez descubierta una casilla, si no contiene una mina, nos informa con un número del número de minas que tiene en sus casillas circundantes.
La partida termina cuando un jugador despeja una casilla que contenga una mina, perdiendo así la partida, o cuando, por el contrario, consigue despejar todas las casillas libres de minas, identificando así las casillas que contengan minas.
La partida dura lo que tarda el jugador en localizar todas las minas o hasta que despeje una casilla que contenga una mina.

Creemos que la funcionalidad más interesante que hemos desarrollado es la de poder jugar una partida, con la posibilidad de ganarla o perderla, ya que es la más visual y entretenida.
Por otro lado, las funcionalidades relacionadas con el módulo de interfaz de usuario, a pesar de ser más tediosa, es muy útil saber desarrollarlo, ya que está en casi todos los sistemas de información.

Diagrama(s) UML

Diagrama de Dominio/Diseño

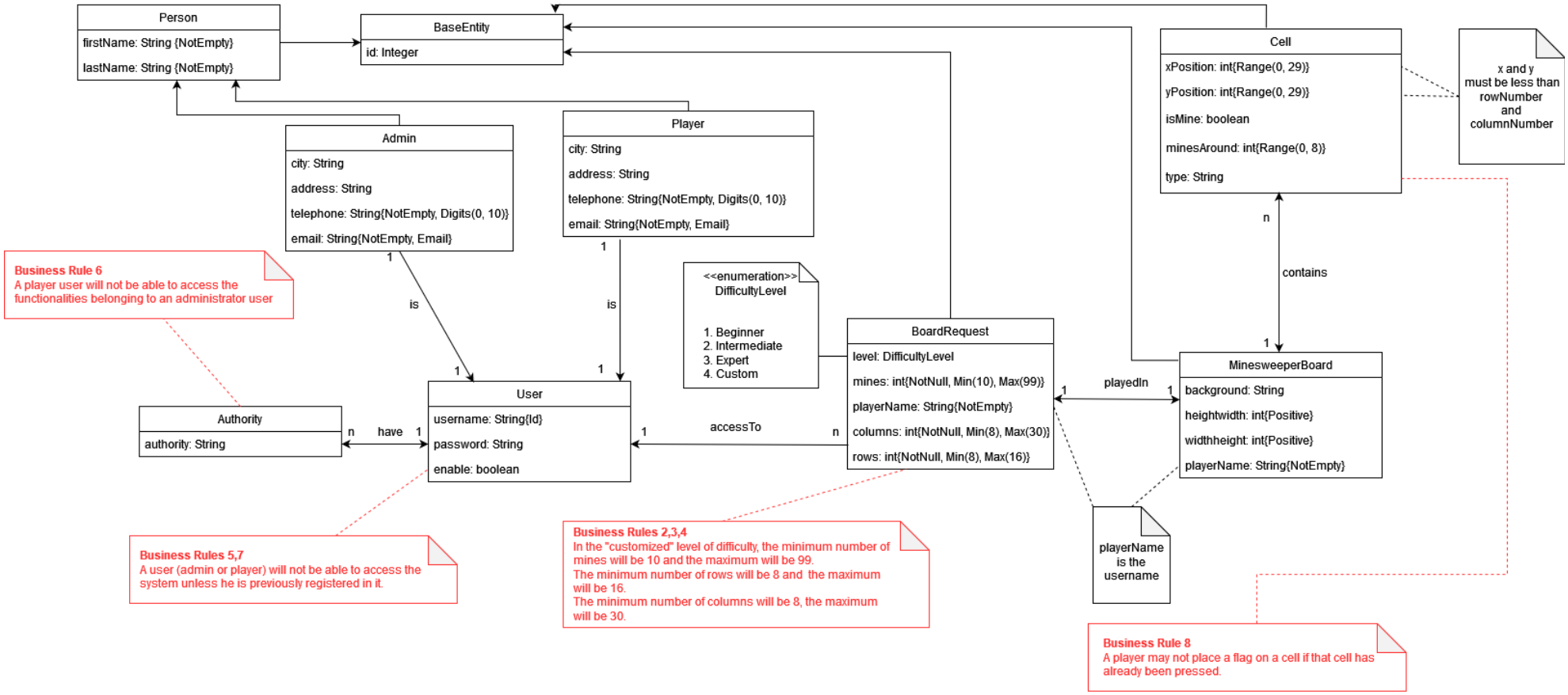
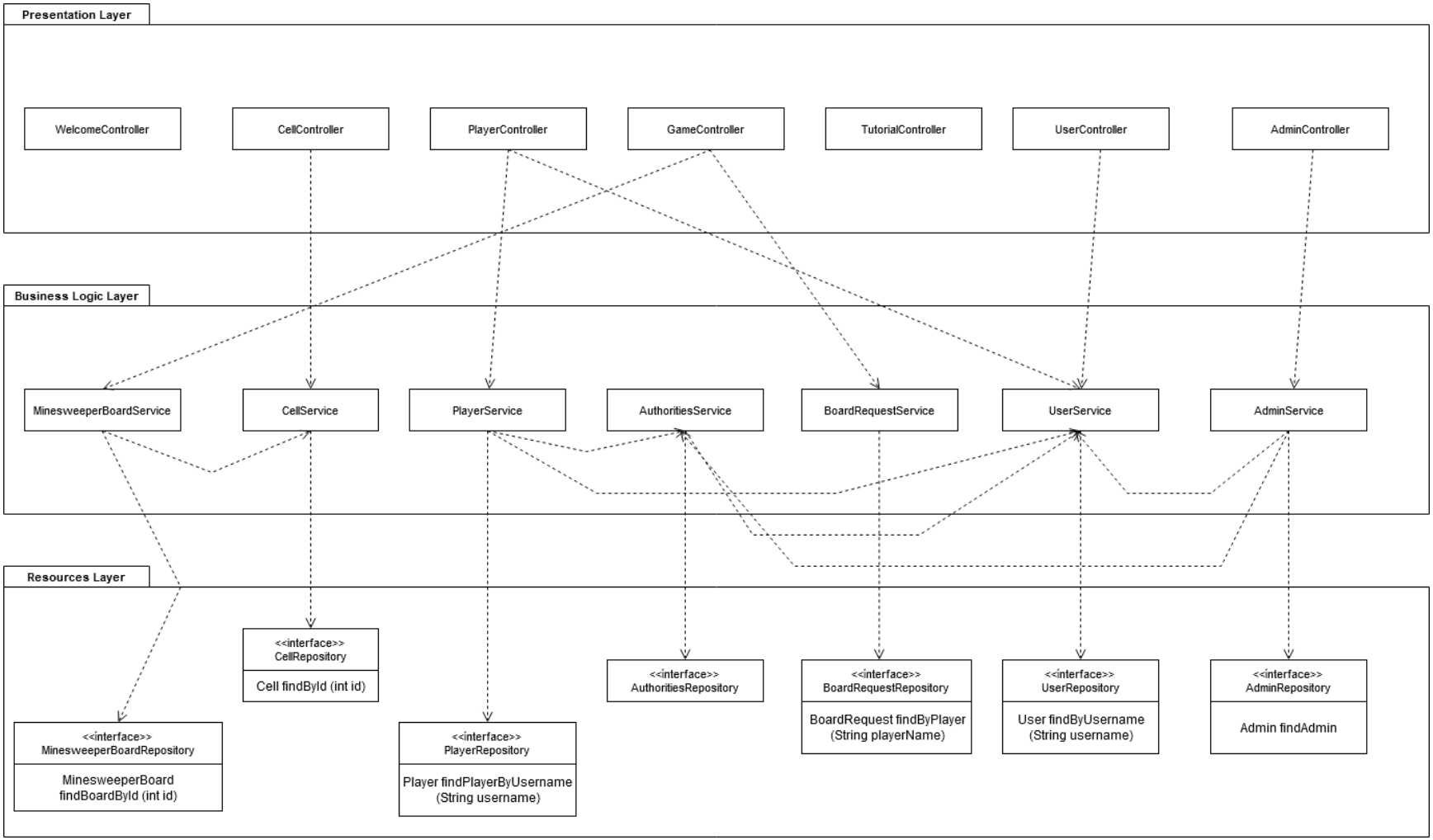


Diagrama de Capas (incluyendo Controladores, Servicios y Repositorios)



Patrones de diseño y arquitectónicos aplicados

Patrón: MVC

Tipo: Arquitectónico

Contexto de Aplicación

Se utiliza en el total de la aplicación, separa los datos de la aplicación, la interfaz de usuario y la lógica de negocio en 3 componentes distintos:

Modelo: Es la representación específica de la información con la que se opera. Incluye los datos y la lógica para operar con ellos. Dentro del modelo se incluyen tres elementos: entidades, repositorios y servicios.

Controlador: Responda a eventos de la interfaz de usuario e invoca cambios en el modelo y probablemente en la vista.

Vista: Es la representación del modelo de forma adecuada para interactuar con ella y representan la información proporcionada por el controlador.

Clases o paquetes creados

Se han creado clases, las cuales hacen referencia al modelo de la aplicación. En este se han implementado las distintas entidades, servicios y repositorios con las que se han trabajado para la construcción de la aplicación.

- **model:**
  Admin.java, Audit.java, BaseEntity.java, NamedEntity.java, Person.java, Player.java, Cell.java, MinesweeperBoard.java, Authorities.java, User.java, Revision.java, Cell.java, DificultyLevel.java.

- **service:**
  AdminService.java, AuditService.java, CellService.java, MinesweeperBoardService.java, BoardRequestService.java, PlayerService.java, AuthoritiesService.java, UserService.java.

- **repository:**
  CellRepository.java, MinesweeperBoardRepository.java, PlayerRepository.java, AuthoritiesRepository.java, UserRepository.java, AdminController.

Más adelante también creamos las clases correspondientes al controlador de la aplicación.

- **web:**
  AdminController.java, AuditController.java, CellController.java, GameController.java, PlayerController.java, TutorialController.java, UserController.java, CrashController.java, WelcomeController.java.

También podemos encontrar varias clases jsp relacionadas con las entidades del modelo.

- players: createOrUpdatePlayerForm.jsp, findPLayers.jsp, playerDelete.jsp, playerDetails.jsp, playersList.jsp.
- tutorial: tutorialDetails.jsp.

- users: createUserForm.jsp, updateUserForm.jsp, userDetails.jsp, usersList.jsp.
- admin: createOrUpdateAdminForm.jsp, adminDetails.jsp, viewAudits.jsp.

Ventajas alcanzadas al aplicar el patrón

Soporte para múltiples vistas, favorece la alta cohesión, el bajo acoplamiento y la separación de responsabilidades. Facilita el desarrollo y las pruebas de cada tipo de componente en paralelo.

Patrón: Front Controller

Tipo: Diseño

Contexto de Aplicación

Maneja todas las solicitudes de un sitio web y luego envía esas solicitudes al controlador apropiado. La clase @controller tiene los métodos adecuados para el manejo de solicitudes.

```java
@Controller
public class PlayerController {

    private static final String VIEWS_PLAYER_CREATE_OR_UPDATE_FORM = "players/createOrUpdatePlayerForm";

    @Autowired
    private PlayerService playerService;

    @GetMapping(value = "/players/find")
    public String initFindForm(Map<String, Object> model) {
        model.put("player", new Player());
        return "players/findPlayers";
    }


    @GetMapping(value = "/players/list")
    public String processFindForm(Player player, BindingResult result, Map<String, Object> model) {

        // allow parameterless GET request for /owners to return all records
        if (player.getFirstName() == null) {
            player.setFirstName(""); // empty string signifies broadest possible search
        }

        // find owners by username
        Collection<Player> results = this.playerService.findPlayers(player.getFirstName());
        if (results.isEmpty()) {
            // no owners found
            result.rejectValue("firstName", "notFound", "not found");
            return "players/findPlayers";
        }
        else {
            model.put("selections", results);
            return "players/playersList";
        }
    }
```

```java
@GetMapping(value = "/players/{playerId}/edit")
public String initUpdatePlayerForm(@PathVariable("playerId") int playerId, Model model) {
    Player player = this.playerService.findPlayerById(playerId);
    model.addAttribute(player);
    return VIEWS_PLAYER_CREATE_OR_UPDATE_FORM;
}

@PostMapping(value = "/players/{playerId}/edit")
public String processUpdatePlayerForm(@Valid Player player, BindingResult result,
        @PathVariable("playerId") int playerId) {
    if (result.hasErrors()) {
        return VIEWS_PLAYER_CREATE_OR_UPDATE_FORM;
    }
    else {
        player.setId(playerId);
        this.playerService.savePlayer(player);
        return "redirect:/players/{playerId}";
    }
}
```

Clases o paquetes creados
El paquete creado para la adaptación de este patrón de diseño es el mismo que hemos mencionado antes como “web”.

Ventajas alcanzadas al aplicar el patrón
El control centralizado permite la aplicación de políticas en toda la aplicación, como el seguimiento de usuarios y seguridad. La decisión sobre quién es el administrador apropiado de una solicitud puede ser más flexible y el FrontController puede proporcionar funciones adicionales, como el procesamiento de solicitudes para la validación y transformación de parámetros.

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 de una clase minesweeperData.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 posible manipulación de estos.

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.

Patrón: Domain Model

Tipo: Diseño

Contexto de Aplicación

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

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

Ventajas alcanzadas al aplicar el patrón

Facilita la concordancia de los objetos modelados del negocio.

Patrón: Repositories

Tipo: Diseño

Contexto de Aplicación

Para las funcionalidades necesarias acorde a nuestra aplicaciones hemos implementado repositorios de tipo CRUD y Repository por cada entidad para así poder interactuar con la base de datos y con los objetos Java del modelo de dominio.

Clases o paquetes creados

Todos los implementados dentro del paquete “repository” mencionado anteriormente.

Ventajas alcanzadas al aplicar el patrón

La ventaja al utilizar este patrón es que se ha permitido un acceso más funcional a la base de datos. Hemos podido definir nuestros propios métodos, y algunos de ellos mediante los queries.

Patrón: Layer super type

Tipo: Diseño

Contexto de Aplicación

Es esencial para la codificación de la aplicación ya que la mayoría de nuestros objetos se extienden de estas.

Clases o paquetes creados

El paquete creado para la adaptación de este patrón: BaseEntity.java, NamedEntity.java

Ventajas alcanzadas al aplicar el patrón

Hemos podido identificar entidades y se ha podido autoincrementar el nuevo id en la base de datos.

Patrón: Estrategia

Tipo: Diseño

Contexto de Aplicación

Nos ha resultado bastante útil, ya que al utilizar composición en vez de herencia y los comportamientos de este patrón están definidos como interfaces separadas y clases específicas que implementan esas interfaces. En el manejo de authorities y de seguridad ha sido cuando la hemos utilizado.

Clases o paquetes creados

Se han utilizado las siguientes clases: Authorities.java y SecurityConfiguration.java.

Ventajas alcanzadas al aplicar el patrón

Promueve extensibilidad en el manejo de algoritmos y el bajo acoplamiento y cohesión.

Decisiones de diseño

Decisión 1: Interacción con el tablero

Descripción del problema:

Para poder interaccionar con el tablero y jugar al buscaminas, necesitábamos una forma simple pero efectiva de descubrir celdas y poner y quitar banderas. Surgieron tres alternativas, una más simple pero igualmente funcional, la otra más estética pero más complicada de desarrollar y la última más fácil para el jugador.

Alternativas de solución evaluadas:

Alternativa 1.a: Incluir los datos mediante un formulario.

Ventajas:

- Simple, no requiere nada más que un formulario para los distintos datos que se introduzcan.

Inconvenientes:

- Estéticamente no destaca.
- Tardanza en la interacción.

Alternativa 1.b: Interacción con el tablero mediante botones.

Ventajas:

- Estéticamente mejor que un formulario.
- El tiempo de interacción para conseguir un resultado es mínimo.

Inconvenientes:

- Supone mucho más trabajo.
- Añade cierta complejidad al proyecto.
- Tiempos de carga elevados al tener que cargar tantos botones.

Alternativa 1.c: Incluir la casilla y acción desde un bloque de texto.

Ventajas:

- Más fácil que un formulario.
- Da menos lugar a un fallo para un jugador.

Inconvenientes:

- Supone mucho más trabajo.
- Añade cierta complejidad al proyecto.
- Supone más tiempo que las otras dos opciones.

Justificación de la solución adoptada

Como nuestro objetivo es desarrollar una aplicación en la que prime la parte funcional a la estética, nos decantamos por la alternativa de diseño 1.a, ya que nos permitía centrarnos y dedicar más tiempo a las funcionalidades de la aplicación.

Decisión 2: Tutorial del juego

Descripción del problema:

Para que cualquier jugador, tenga o no tenga experiencia, necesitábamos añadir una manera de que este aprendiese las normas y objetivos del juego. Surgieron 3 alternativas, añadir un vídeo explicando el juego, añadir un texto con el juego explicado y un nivel de dificultad “tutorial” para entender el funcionamiento del buscaminas.

Alternativas de solución evaluadas:

Alternativa 2.a: Vídeo explicativo del juego.

Ventajas:

- Es más visual.

Inconvenientes:

- Respecto a nuestro diseño de la página, no sería lo más estético.
- A la hora de consultar algo relacionado con el juego el usuario tendría que ver el vídeo completo hasta encontrar donde puede solucionar su consulta.

Alternativa 2.b: Texto explicativo del juego.

Ventajas:

- A la hora de consultar algo relacionado con el juego es mucho más fácil ya que solo tienes que mirar el punto relacionado con tu problema.
- Concuerda más con la estética de nuestra página.

Inconvenientes:

- Es menos visual.
- Es menos dinámico

Alternativa 2.c: Nivel “tutorial”. Al registrarse como nuevo jugador, para empezar una nueva partida, será obligatorio completar antes el nivel tutorial. Este nivel tutorial, sería idéntico en tamaño al de la

dificultad fácil, pero con la diferencia que las minas estarían visibles. Esto permitiría ver a los nuevos usuarios comprender con facilidad el juego.

Ventajas:

- Seguridad de que los nuevos usuarios conocen cómo jugar al juego.
- Un tutorial jugable es más llevadero que leer un texto o ver un vídeo.
- También sería más rápido que leer un texto o ver un vídeo.
- El juego sería más accesible al asegurarnos de que todos los jugadores conocen las normas.

Inconvenientes:

- Alta complejidad en el desarrollo en comparación a las otras opciones.
- La mayoría de los jugadores que juegan al buscaminas hoy en día conocen su funcionamiento, por lo que el tutorial sería una traba para empezar a jugar de forma normal.

Justificación de la solución adoptada

Nos gustaría haber desarrollado la alternativa 2.c, pero debido a su complejidad y teniendo en cuenta la carga de trabajo de los desarrolladores nos hubiera sido muy difícil haberla llevado a cabo.

Nos decantamos por la alternativa 2.b porque quedaba más estético en relación con el diseño de nuestra página y porque acelera mucho el consultar algo respecto al vídeo, ya que con el texto explicativo sabes dónde está tu consulta con un simple vistazo.

Decisión 3: Colocación de casillas en el tablero

Descripción del problema:

El buscaminas es un juego en el que vamos destapando casillas, las cuales pueden tener un número (el cuál indica cuantas minas hay alrededor) o una mina. Teniendo en cuenta esto se nos presentaba un problema, si colocar las minas y conforme a estas colocar el resto de casillas del tablero o al revés.

Alternativas de solución evaluadas:

Alternativa 3.a: Colocar primero los números.

Ventajas:

- Código más visual

Inconvenientes:

- Más cantidad de código
- Problemas con el nivel personalizado

Alternativa 3.b: Colocar primero las minas

Ventajas:

- Facilidades a la hora de implementar el nivel personalizado
- Menos cantidad de código

Inconvenientes:

- Código más complejo.

Justificación de la solución adoptada

Optamos por la opción 3.b para así evitar problemas y retrasos para el nivel personalizado. Esta opción además nos ha sido más fácil de implementar de lo que pensábamos en un principio. Así satisfacemos de mejor manera la regla de negocio 1 y 2.

Decisión 4: Seleccionar una partida

Descripción del problema:

Comenzar a jugar sin necesidad de registrarse o estar registrado.

Alternativas de solución evaluadas:

Alternativa 4.a: Sin necesidad de estar registrado.

Ventajas:

- Más dinámico.
- Puedes jugar sin perder tiempo en registrarte.
- Más privacidad al no aportar tus datos al juego.

Inconvenientes:

- Problemas con el módulo de estadísticas.

Alternativa 4.b: Necesidad de estar registrado.

Ventajas:

- Puedes ver tus estadísticas.
- A nivel de desarrollo, ayuda a implementar el módulo de estadística.

Inconvenientes:

- Es menos dinámico

Justificación de la solución adoptada

Nos decantamos por la alternativa 4.b primero porque hace que satisfaga la RDN-05 en la cual indicamos que un jugador no podrá acceder al sistema si no se encuentra previamente registrado en el mismo y, además, hace que nos resulte más dinámico y sencillo la implementación del módulo de estadística.

Decisión 5: Logs

Descripción del problema:

Logging, apartado obligatorio a realizar para poder avanzar con el proyecto. Para crear los logs nos encontrábamos con dos opciones, ambas vistas en clase.

Alternativas de solución evaluadas:

Alternativa 5.a: Vanilla logging.

Ventajas:

- Simple.
- Podemos especificar el nivel de registro que queremos mostrar.

Inconvenientes:

- Es estático.
- Todos los objetos de la clase usan la misma instancia.

Alternativa 5.b: Logging con Lombok.

Ventajas:

- Facilita la creación de logs.
- Simple

Inconvenientes:

- Añade cierta complejidad al proyecto.

Justificación de la solución adoptada

Para la creación de logs hemos decidido optar por la alternativa 5.b donde creamos los utilizando un @Slf4j y en cada clase definiéndolos con un Log.info(---). Hemos optado por esta opción por la cantidad de facilidades que nos daba para la creación de logs.

Decisión 6: Auditoría

Descripción del problema:

La realización de la auditoría es uno de los principales problemas que hemos encontrado en el desarrollo del proyecto, ya que teníamos diferentes alternativas, todas muy factibles.

Alternativas de solución evaluadas:

Alternativa 6.a: Auditoría con Spring Data JPA

Ventajas:

- Muy visual y transparente.

Inconvenientes:

- Muy limitado.
- Muchos inconvenientes.

Alternativa 6.b: Auditoría con Hibernate Envers.

Ventajas:

- Sencillo de usar.

Inconvenientes:

- No audita al usuario por defecto.

Justificación de la solución adoptada

Para la creación de auditorías optamos por la alternativa 6.b ya que como hemos visto era la más sencilla de usar y el inconveniente se podía arreglar muy fácilmente.