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

Tutor: IRENE BEDILIA ESTRADA TORRES

GRUPO DICIEMBRE 4

Versión V8

31/10/2022

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
20/10/2022	V7	● Actualización de la introducción ● Corrección diagrama de Dominio/Diseño ● Corrección decisiones de diseño	Noviembre
31/10/2022	V8	● Añadidos nuevos patrones de diseño: Estado, Paginación	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 tres módulos, los cuales explicaremos en profundidad más adelante:

- 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.
- En tercer lugar, el **módulo de estadísticas**, en el cual podremos ver estadísticas totales y por jugador relacionadas con el juego.

En la modalidad original de nuestro juego, el buscaminas, sólo puede jugar una persona simultáneamente. El objetivo del juego es ir despejando las casillas que no contengan minas, e ir marcando las que sí contengan minas con banderas.

Antes de empezar la partida habrá un selector de dificultad, el cual dará 4 opciones de dificultad al usuario. Las opciones son las siguientes:

- En el modo de dificultad "**Principiante**", el número de minas será de 10, mientras que el tablero tendrá 8 filas y 8 columnas.
- En el modo de dificultad "**Intermedio**", el número de minas será de 40, mientras que el tablero tendrá 16 filas y 16 columnas.
- En el modo de dificultad "**Avanzado**", el número de minas será de 90, mientras que el tablero tendrá 16 filas y 30 columnas.
- En el modo de dificultad "**Personalizado**", el número de minas será elegido por el propio jugador y estará entre los valores 10 y 99 (ambos incluidos), mientras para el tablero se podrán elegir valores de entre 8 y 16 filas, y de entre 10 y 99 columnas (ambos incluidos).

El transcurso de la partida es sencillo, se trata de un tablero con un número determinado de casillas, dependiendo de la dificultad escogida.
Existe la posibilidad de colocar una bandera en las casillas en las que creamos que se encuentra una mina, así como quitar la bandera en caso de que nos equivoquemos. Las casillas que estén marcadas con una bandera no será posible descubrirlas, lo cual será implementado con un aviso cuando se dé el caso.
Una vez descubierta una casilla, si no contiene una mina hay dos opciones, o nos informa del número de minas circundantes, o bien, si no tiene minas circundantes, se descubre esa casilla y varias de las cercanas a ella hasta que se descubran las que tengan minas 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.

La aplicación tendrá un apartado de auditoría, disponible únicamente para el administrador, en el que quedarán reflejadas todas las partidas jugadas por los jugadores con los siguientes datos: fecha de inicio y fin, nivel de dificultad, jugador que ha iniciado la partida, y estado de la partida, el cual puede ser empezada, cancelada, perdida o ganada.

Adicionalmente, habrá un apartado de estadísticas disponible para los jugadores, en el cual se podrá ver un top con los jugadores que más partidas han ganado, con el número de partidas ganadas por cada uno. A continuación aparecerá una tabla con estadísticas globales, que será la suma de las estadísticas de todos los jugadores. En ella podemos ver datos como el total de partidas ganadas en total y por jugador, la duración total de todas las partidas y la media, o la partida con más duración y con menos.
Más adelante, se podrá ver otra tabla con la misma información que la anterior, pero esta vez para el jugador que esté logueado. Adicionalmente, en esta tabla también figuran los datos totales de partidas ganadas, minas activadas, minas no activadas, banderas colocadas y celdas clicadas.
Por último, encontramos un apartado de logros con los siguientes logros: nivel bronce, has ganado 10 partidas, nivel plata, has ganado 25 partidas y nivel oro, has ganado 50 partidas.

Diagrama(s) UML

Diagrama de Dominio/Diseño

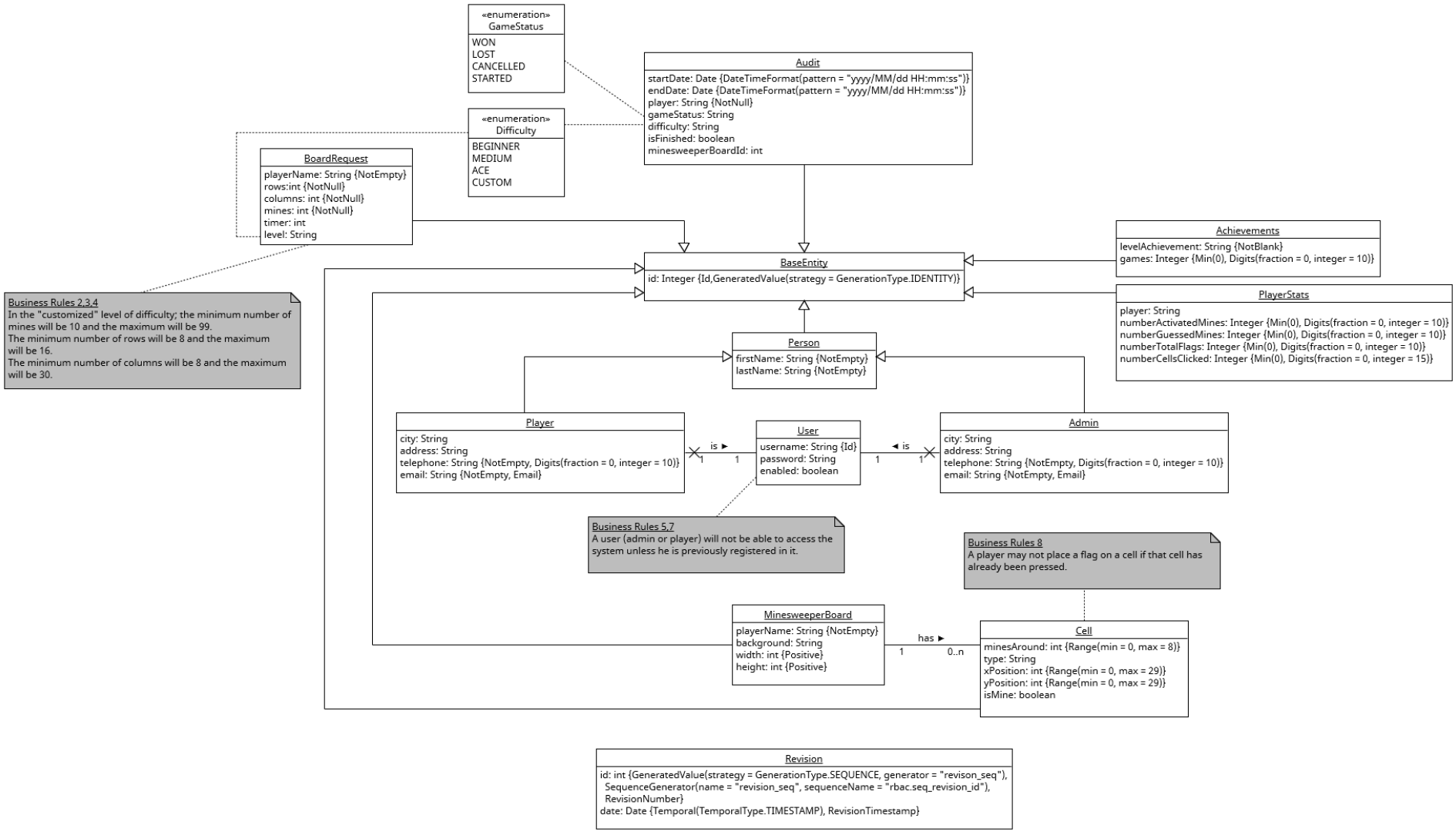
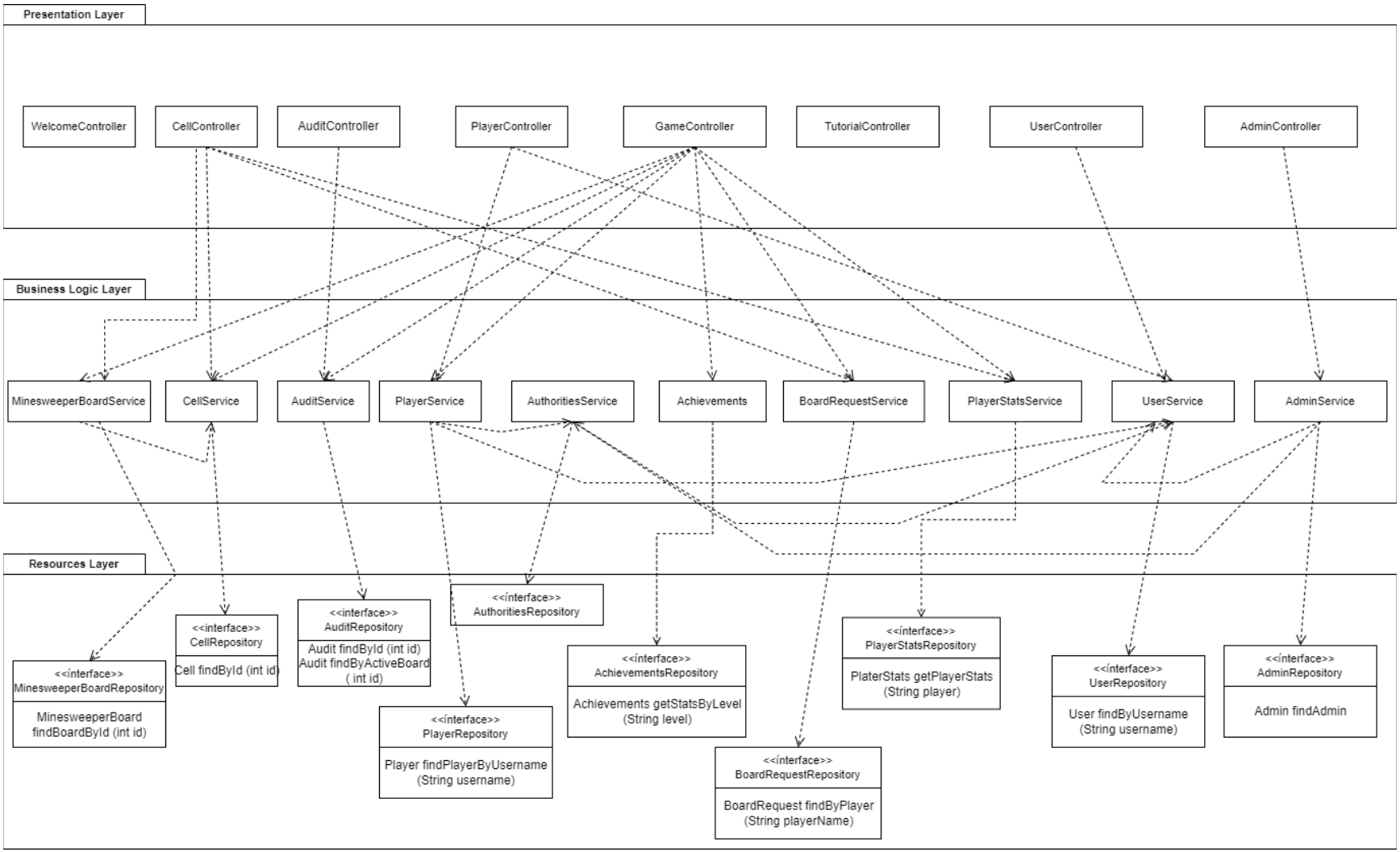


Diagrama de Capas (incluyendo Controladores, Servicios y Repositorios)



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";
        }
    }
```

Ejemplo de aplicación del Patrón Front Controller

```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}";
    }
}
```

Ejemplo de aplicación del Patrón Front Controller

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: Estado

Tipo: Diseño

Contexto de Aplicación

La principal aplicación del patrón de diseño Estado en nuestro proyecto ha sido aplicado a la hora de especificar el estado de las partidas jugadas por el jugador. Las partidas están clasificadas en uno de los siguientes estados una vez empezada: “WON”, “LOST”, “CANCELLED” o “STARTED”, dependiendo si ya ha sido ganada, perdida, si se ha cancelado pulsando el botón de “Finish Game” o simplemente si se ha empezado y no han ocurrido ninguna de las acciones anteriormente mencionadas.

Estos estados de la partida cambian completamente el comportamiento de la partida como objeto. Cuando una partida está en estado “WON”, “LOST” o “CANCELLED” esta queda reflejada tanto en el apartado “Audit” con sus datos correspondientes como en el apartado de “Game Stats”, además de no tener opción de continuar la partida.

Cuando una partida está en estado “STARTED”, no se tienen en cuenta los datos de esta ni para “Audits” ni para “Game Stats”, y se puede continuar jugando, a no ser que cambie de estado, ganando, perdiendo

o terminando la partida pulsando el botón de “Finish Game”. Otro de los cambios de comportamiento del objeto Game es que mientras siga en estado “STARTED” el cronómetro de la partida sigue corriendo.

Otro ejemplo de este patrón sería en el objeto “BoardRequest” el cuál tiene los diferentes niveles de dificultad especificados en “DifficultyLevel”, “BEGINNER”, “MEDIUM”, “ACE” O “CUSTOM”, los cuales cambian el comportamiento del tablero.

```
3  public enum GameStatus {
4      WON, LOST, CANCELLED, STARTED
5  }
```

Ejemplo de aplicación del Patrón Estado

```
        // Start audit game (STARTED GAME)
        Date newDate = this.minesweeperBoardService.getFormattedDate();
        Audit newGameAudit = new Audit();
        newGameAudit.setStartDate(newDate);
        newGameAudit.setPlayer(player.getName());
        newGameAudit.setGameStatus(GameStatus.STARTED.name());
        newGameAudit.setDifficulty(boardRequest.getLevel().name());
        newGameAudit.setFinished(false);
        newGameAudit.setMinesweeperBoardId(board.getId());
        this.auditService.saveAudit(newGameAudit);

        int flagsInMines = boardRequest.getMines();

        model.put("flagsInMines", flagsInMines);
        model.put("minesweeperBoard", board);
        model.put("boardRequest", boardRequest);
        model.put("timer", 0);
        return VIEWS_NEW_GAME;
    }
```

Ejemplo de aplicación del Patrón Estado

Clases o paquetes creados

Se han utilizado las siguientes clases: GameStatus.java, GameController.java, BoardRequest.java, “DifficultyLevel.java”.

Ventajas alcanzadas al aplicar el patrón

El código relacionado con determinados estados está organizado en clases separadas. (Alta cohesión y responsabilidad única).

Podemos introducir nuevos estados sin cambiar los estados existentes o el contexto (Abierto a la ampliación/cerrado a la modificación).

El código del contexto se simplifica al eliminar los condicionales repetitivos voluminosos (No se repite).

Patrón: Inyección de dependencias

Tipo: Diseño

Contexto de Aplicación

Es la base de todo lo relacionado con el control en la aplicación.

La inyección de dependencias es un patrón de diseño en el que un objeto o función recibe otros objetos o funciones de los que depende. La inyección de dependencias, que es una forma de inversión de control, pretende separar las preocupaciones de la construcción de objetos y su uso, lo que conduce a programas poco acoplados. El patrón garantiza que un objeto o función que quiera utilizar un servicio determinado no tenga que saber cómo construir esos servicios. En su lugar, el "cliente" receptor (objeto o función) recibe sus dependencias a través de un código externo (un "inyector"), del que no es consciente. La inyección de dependencias es útil porque hace explícitas las dependencias implícitas y ayuda a resolver los siguientes problemas.

```
@Autowired
private final PlayerService playerService;

@Autowired
private PlayerStatsService playerStatsService;

@Autowired
public PlayerController(PlayerService playerService) {
    this.playerService = playerService;
}
```

Ejemplo de aplicación del Patrón de Inyección de Dependencias

Clases o paquetes creados

El patrón MVC se basa en este principio, por lo que en todas las clases de modelo, vistas y controladores hacen uso de este patrón.

Ventajas alcanzadas al aplicar el patrón

Permite alcanzar el bajo acoplamiento y la modularización de la aplicación.

Patrón: Template View

Tipo: Diseño

Contexto de Aplicación

SpringBoot utiliza JSP/JSTL para la visualización de las diferentes vistas de la aplicación, tenemos una plantilla de la vista en HTML (o similar a HTML) con algunos elementos especiales que representan atributos del modelo.

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.

```
23    <script type="text/javascript">
24        document.addEventListener("DOMContentLoaded", function(event) {
25            var message = "${loserMessage}";
26            var message2 ="${winnerMessage}";
27            // When Lose game
28            if (message != "") {
29                var current_seconds = parseInt(document
30                        .getElementById("cont_timer").innerHTML);
31                var hour = Math.floor(current_seconds / 3600);
32                var minute = Math.floor((current_seconds - hour * 3600) / 60);
33                var seconds = current_seconds - (hour * 3600 + minute * 60);
34                if (hour < 10)
35                    hour = "0" + hour;
36                if (minute < 10)
37                    minute = "0" + minute;
38                if (seconds < 10)
39                    seconds = "0" + seconds;
40                document.getElementById("timer_finish").innerHTML = hour + ":"
41                        + minute + ":" + seconds;
42                document.getElementById("flagsRemaining").style.display = "none";
43                document.getElementById("nextMove").style.display = "none";
```

Ejemplo de aplicación del Patrón Template View

```
<c:if test="${loserMessage != null}">
    <div class="row text-center">
        <h1 style="color: red">
            <c:out value="${loserMessage}"></c:out>
        </h1>
        <span style="font-size: 18px"><b>Time: <span
                id="timer_finish"></span></b></span><br />
        <span style="font-size: 18px"><b>Flags
                remaining: ${flagsInMines}</b></span>
    </div>
</c:if>
```

Ejemplo de aplicación del Patrón Template View

Clases o paquetes creados

Este patrón puede verse en todas las vistas de la aplicación, algunas de las más complejas son, por ejemplo, newGame.jsp o selectGame.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: 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: Capa de Servicios

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: Modelo de Dominio

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: Registro Activo

Tipo: Diseño

Contexto de Aplicación

Este patrón consiste en un objeto que envuelve una fila en una tabla o vista de la base de datos, encapsula el acceso a la base de datos y añade lógica de dominio a esos datos.

- Relación 1:1 entre clase y tabla.
- Cada fila de la tabla es una instancia de este objeto.
- Cada registro activo es responsable de guardar y cargar en la base de datos y también de cualquier lógica de dominio que actúe sobre estos datos.
- Los métodos de acceso a los datos suelen ser estáticos.

Clases o paquetes creados
Este patrón puede verse aplicado en cada una de las distintas entidades creadas, como por ejemplo, “Player.java” o “Cell.java”.

Ventajas alcanzadas al aplicar el patrón
Facilita la implementación de las entidades y su relación con la base de datos.

Patrón: Repositorios

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: Campo de Identificación

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.

```
13 @Entity
14 @Getter
15 @Setter
16 @Table(name = "cells")
17 public class Cell extends BaseEntity {
18
19     @Range(min = 0, max = 8)
20     int minesAround;
21
22     String type;
23
24     @Range(min = 0, max = 29)
25     int xPosition;
26
27     @Range(min = 0, max = 29)
28     int yPosition;
29     boolean isMine;
30
```

Ejemplo de aplicación del patrón de Campo de Identificación

```java
@MappedSuperclass
public class BaseEntity {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    protected Integer id;

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public boolean isNew() {
        return this.id == null;
    }
}
```

Ejemplo de aplicación del patrón de Campo de Identificación

Clases o paquetes creados

La aplicación de este patrón se puede ver en cualquiera de las clases de las entidades, por ejemplo, “Cell.java”, heredando de la clase “BaseEntity.java”.

Ventajas alcanzadas al aplicar el patrón

Permite identificar únicamente una instancia de una entidad, permitiendo distinguirlas fácilmente.

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

Patrón: Paginación

Tipo: Diseño

Contexto de Aplicación

Demasiada cantidad de datos para ser mostrados en una misma página. Se implementa una interfaz que permite la navegación entre diferentes páginas en las que se muestra la información, además de las opciones de avanzar y retroceder de página

```
@GetMapping(value = "/players/list")
public String processFindForm(Player player, BindingResult result, Map<String, Object> model,
        @PageableDefault(page = 0, size = 5) @SortDefault.SortDefaults({
                @SortDefault(sort = "id", direction = Sort.Direction.ASC),
                @SortDefault(sort = "firstName", direction = Sort.Direction.DESC) }) Pageable pageable) {

    // allow parameterless GET request for /players to return all records
    player = this.playerService.checkPlayerSearched(player);

    Integer nresults = this.playerService.countFoundedPlayers(player.getFirstName());
    Integer page = 0;
    // find players by first name
    List<Player> results = this.playerService.findPlayers(player.getFirstName(), page, pageable);
    model.put("pageNumber", pageable.getPageNumber());
    model.put("hasPrevious", pageable.hasPrevious());
    model.put("firstName", player.getFirstName());
    Double totalPages = Math.ceil(nresults / (pageable.getPageSize()));
    model.put("totalPages", totalPages);
    if (results.isEmpty()) {
        // no players found
        result.rejectValue("firstName", "notFound", "not found");
        return VIEWS_FIND_PLAYER_FORM;
    } else {
        model.put("selections", results);

        return VIEWS_LIST_PLAYER;
    }
}
```

Ejemplo de aplicación del patrón de Paginación

Clases o paquetes creados

Se han utilizado las siguientes clases: PlayerController.java, PlayerRepository.java y PlayerService.java.

Ventajas alcanzadas al aplicar el patrón

El uso de la paginación puede ser conveniente por diversas razones. La más importante es que mejora la claridad y la usabilidad de una web al dividir un contenido muy extenso en varias páginas. Así pues, reducir el contenido que se muestra en una sola página también disminuye el tiempo de carga puesto que no hay que cargar todo el contenido del listado a la vez, mejorando así la usabilidad.

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. El mero hecho de interactuar con el tablero mediante un formulario, ya hace que el jugador llegue a tardar demasiado, pero si a eso le sumas el tener que fijarte y contar dónde quieres que se descubra la casilla que quieres o colocar una bandera, hace que la tardanza sea bastante considerable entre descubrir una casilla y otra.

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. Debido a que el tiempo que tardas en descubrir una casilla o colocar una bandera disminuye drásticamente en comparación con interactuar con un formulario, solo es clickar donde quieras.

Inconvenientes:

- Supone mucho más trabajo. Habría que dedicarle cierto tiempo a investigar cómo poder implementar la interacción mediante botones ya que ningún miembro del grupo tiene un conocimiento base para poder realizar dicha tarea.
- Añade cierta complejidad al proyecto. Ninguno de los integrantes del grupo tiene los conocimientos básicos para realizar esta tarea.
- Tiempos de carga elevados al tener que cargar tantos botones, estos tiempos pueden hacer que sea desesperante el jugar al buscaminas.

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

Ventajas:

- Da menos lugar a un fallo para un jugador.

Inconvenientes:

- En comparación con las otras alternativas, esta haría que le dedicaremos un tiempo extra ya que no es una forma que hayamos visto en teoría o en otras asignaturas.
- Añade cierta complejidad al proyecto. Ninguno de los compañeros de los compañeros se ve capacitado para realizar tanta cantidad de trabajo teniendo otras tareas que realizar.

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:

- Al tratarse de un video resulta más visual y llamativo para el jugador. También puede llegar a prestarle más atención que a las otras opciones.

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. Lo cual interferiría con la experiencia del jugador en un juego tan dinámico como es el buscaminas.
- A la hora de querer actualizar el tutorial, es más complicado, ya que habría que grabar de nuevo el vídeo o añadir un nuevo corte.

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.
- Al ser simplemente texto almacenado en la base de datos resulta más fácil de actualizar si se quiere añadir algún aspecto más del juego.

Inconvenientes:

- Es menos visual, haciendo que el jugador se aburra de leer tanto o deje de prestar atención en algún momento.
- En comparación con las otras 2 alternativas es mucho menos dinámico, lo que hace que la experiencia del jugador se vea afectada.

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, lo que supone un consumo mayor de recursos de la plantilla que haría dejar descuidadas otras partes más importantes del proyecto.
- 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. Haciendo que sea más fácil e intuitivo la búsqueda de cualquier error.

Inconvenientes:

- Más cantidad de código, lo cual implicaría un tiempo extra para modularizar ya que como bien nos explicaron en teoría, que se tenga un bloque con una gran cantidad de código no es una de las mejores formas de realizar un proyecto, por lo tanto hace que sea un inconveniente mayor.
- Problemas con el nivel personalizado, haciendo que tengamos que utilizar más tiempo para ver cómo podemos implementarlo.

Alternativa 3.b: Colocar primero las minas

Ventajas:

- Facilidades a la hora de implementar el nivel personalizado, ahorrándonos recursos y tiempo para realizar otras tareas.
- Menos cantidad, modularización más sencilla de código haciendo caso a las buenas maneras de realizar un proyecto tal y como se nos dijo en clase de teoría.

Inconvenientes:

- Código más complejo, el cual solo se pueden encargar un número limitado de miembros del grupo, ya que los demás no se ven capacitados para realizarlo debido a que no tienen los conocimientos para poder llevar a cabo esta tarea.

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. Lo que hace que el jugador disfrute más de la experiencia.
- Más privacidad al no aportar tus datos al juego.

Inconvenientes:

- Problemas con el módulo de estadísticas. Al no estar registrado, no debería contabilizarse los datos de las partidas que juegues, tanto a nivel global como personal. Esto hace que la carga de trabajo para realizar el módulo sea mucho mayor, incluso haciendo que tengamos que cambiar el proyecto base.

Alternativa 4.b: Necesidad de estar registrado.

Ventajas:

- Se guardan los datos asociados a cada partida con mayor facilidad e identificación, lo que hace que un jugador pueda ver sus estadísticas y estas se sumen a las estadísticas globales.
- A nivel de desarrollo, ayuda a implementar el módulo de estadística.

Inconvenientes:

- Es menos dinámico, haciendo que el jugador disfrute menos de la experiencia. Esto se debe a que a la hora de registrarse el jugador puede encontrarse con nombres de usuario ya cogidos por otros usuarios, contraseñas ya usadas, etc.

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. Como se indica en teoría puede ser la alternativa más fácil y que conlleve menos tiempo de implementar.
- 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. Lo cual nos beneficia en todos los aspectos, ya que dinamiza el trabajo y conlleva una menor carga de trabajo.
- Simple. Al igual que la otra alternativa, en teoría se nos indica y comprobamos que es una alternativa fácil de implementar y que conlleva poco tiempo.

Inconvenientes:

- Añade cierta complejidad al proyecto. Lo que hace que solo unos pocos miembros del equipo de trabajo puedan realizar dicha tarea.

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, haciendo que su implementación sea sencilla y sin mucho trabajo para los programadores.

Inconvenientes:

- Muy limitado. Concretamente los campos de auditoría son la fecha de creación, fecha de última modificación, el usuario creador y el último que modificó la entidad.
- Muchos inconvenientes para lo que buscamos en nuestro proyecto.

Alternativa 6.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 6.b ya que como hemos visto era la más sencilla de usar y el inconveniente se podía arreglar muy fácilmente.