DP1 2020-2021

Documento de Diseño del Sistema

Proyecto Buscaminas

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

Miembros:

- CERRATO SÁNCHEZ, LUIS
- ESCALERA MARTÍN, REGINA
- GUTIÉRREZ CONTRERAS, ERNESTO
- LOBATO TRONCOSO, JOSÉ MANUEL
- MARTÍNEZ SUÁREZ, DANIEL JESÚS
- STEFAN, BOGDAN MARIAN

Tutor: IRENE BEDILIA ESTRADA TORRES

GRUPO L7-04

Versión V1

07/01/2022

Historial de versiones

Estos son ejemplo del contenido que debería tener el historial de cambios del documento a entregar a lo largo de los sprints del proyecto

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

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

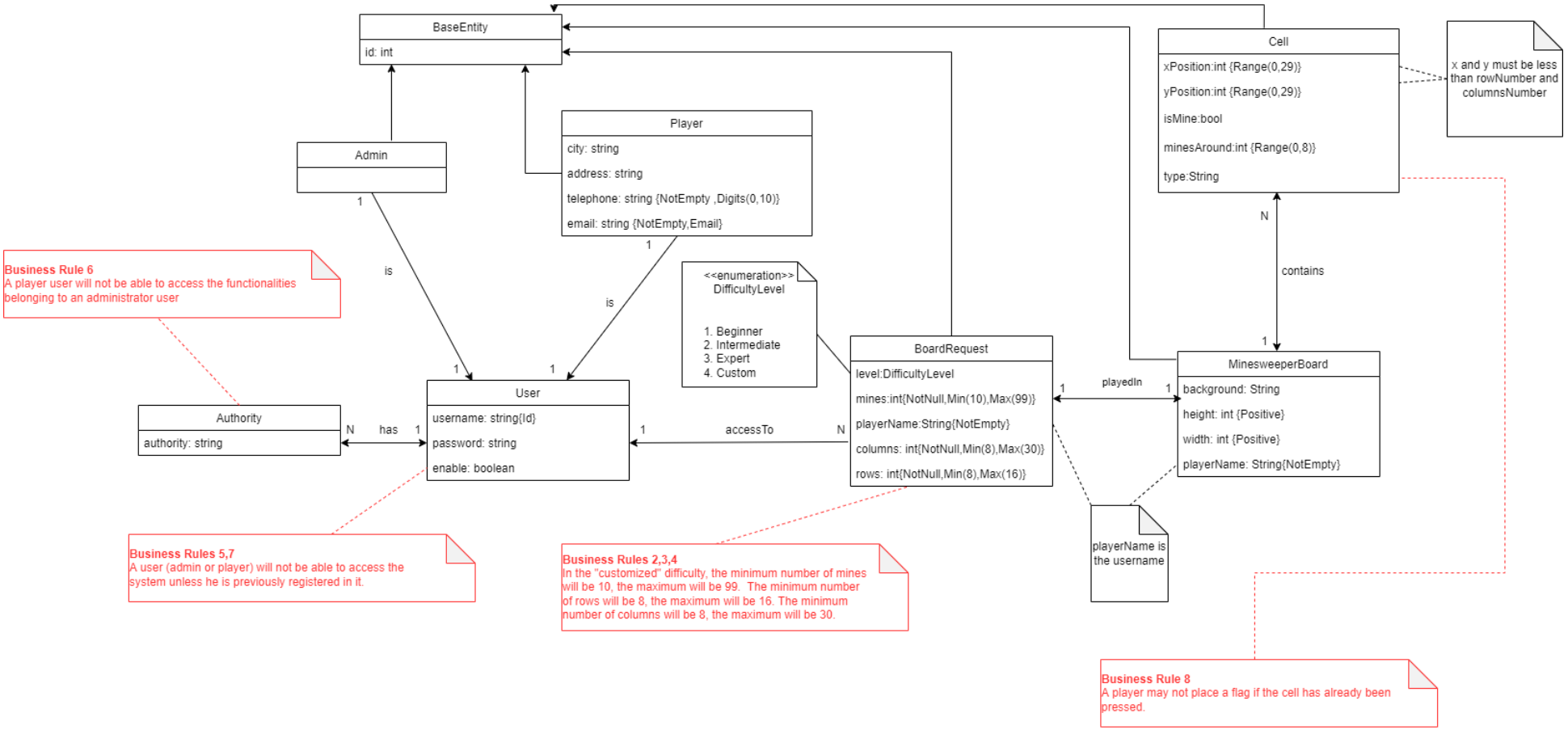
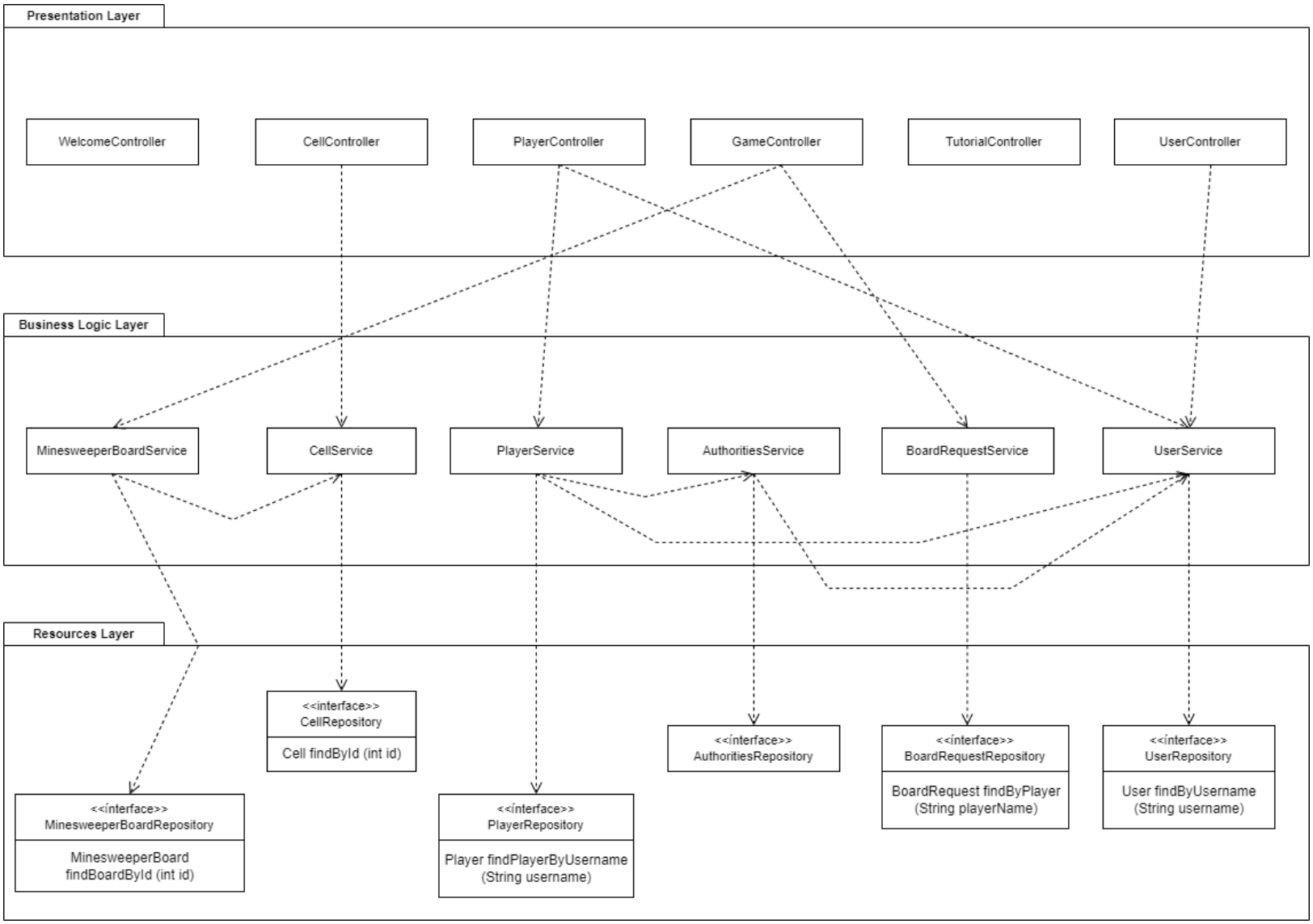


Diagrama de Capas (incluyendo Controladores, Servicios y Repositorios)



Patrones de diseño y arquitectónicos aplicados

Patrón: MVC

Tipo: Arquitectónico

Contexto de Aplicación

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

Modelo: Es la representación especifica 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:**
  BaseEntity.java, NamedEntity.java, Person.java, Player.java, Cell.java, MinesweeperBoard.java, Authorities.java, User.java
- **service:**
  CellService.java, MinesweeperBoardService.java, PlayerService.java, AuthoritiesService.java, UserService.java
- **repository:**
  CellRepository.java, MinesweeperBoardRepository.java, PlayerRepository.java, AuthoritiesRepository.java, UserRepository.java

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

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

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

- players: createOrUpdatePlayerForm.jsp, findPLayers.jsp, playerDetails.jsp, players.jsp
- tutorial: tutorialDetails.jsp
- users: createUserForm.jsp, updateUserForm.jsp, userDetails.jsp, userLogicDelete.jsp, usersList.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 quien 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 como “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 o ha permitido un acceso más funcional a la base de datos. Hemos podido definir nuestros propios métodos, y que algunos de ellos con querys.

Patrón: Layer super type

Tipo: Diseño

Contexto de Aplicación

Esencial para la codificación de la aplicación ya que la mayoría de nuestros objetos 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

Identifica entidades y autoincrementa 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 dos alternativas, una más simple pero igualmente funcional, la otra más estética pero más complicada de desarrollar.

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.

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: Acceso al perfil de un usuario

Descripción del problema:

Para poder acceder al perfil de un usuario, necesitábamos una forma discreta y que concuerde con la estética que estábamos llevando en la aplicación. Surgieron dos alternativas, añadir el botón de acceso en la misma barra de navegación, la cual no era estéticamente lo que buscábamos, o añadir este botón en un desplegable en el lugar donde aparece el perfil del usuario logueado.

Alternativas de solución evaluadas:

Alternativa 2.a: Incluir el acceso en la barra de navegación.

Ventajas:

- Se puede encontrar más a la vista.

Inconvenientes:

- No tenía nada que ver con lo que podemos encontrar en la barra de navegación.
- La barra de navegación está sobrecargada de elementos.

Alternativa 2.b: Incluir el acceso en el desplegable del usuario logueado.

Ventajas:

- Agrupar todas las operaciones referentes al usuario en un desplegable.
- La barra de navegación queda más limpia y clara.

Inconvenientes:

- Acceso más difícil que si estuviera en la barra de navegación.

Justificación de la solución adoptada

Como nuestra barra de navegación tenía accesos relacionados con el juego en sí, nos decantamos por la alternativa 2.b, ya que nos permitía tener una barra de navegación limpia y sin accesos que no tengan relación con jugar al juego.