# 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
- Andrés Francisco García Rivero
- 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 V1

18/11/2022

## 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** |
|---|---|---|---|
| 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 |

# 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.

Por último, indicar que el tiempo (expresado en segundos) es un parámetro muy importante, ya que si un jugador ha ganado dos partidas con el mismo nivel de dificultad, será mejor resultado aquella partida ganada en menos tiempo.

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.

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.

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). 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. 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. El ranking de jugadores se podrá filtrar por partidas ganadas o por menor tiempo de partida jugada, ambas para cada dificultad, y mostrará el top 3 de jugadores para la categoría dada. El ranking es indiferente de cómo estemos registrados.

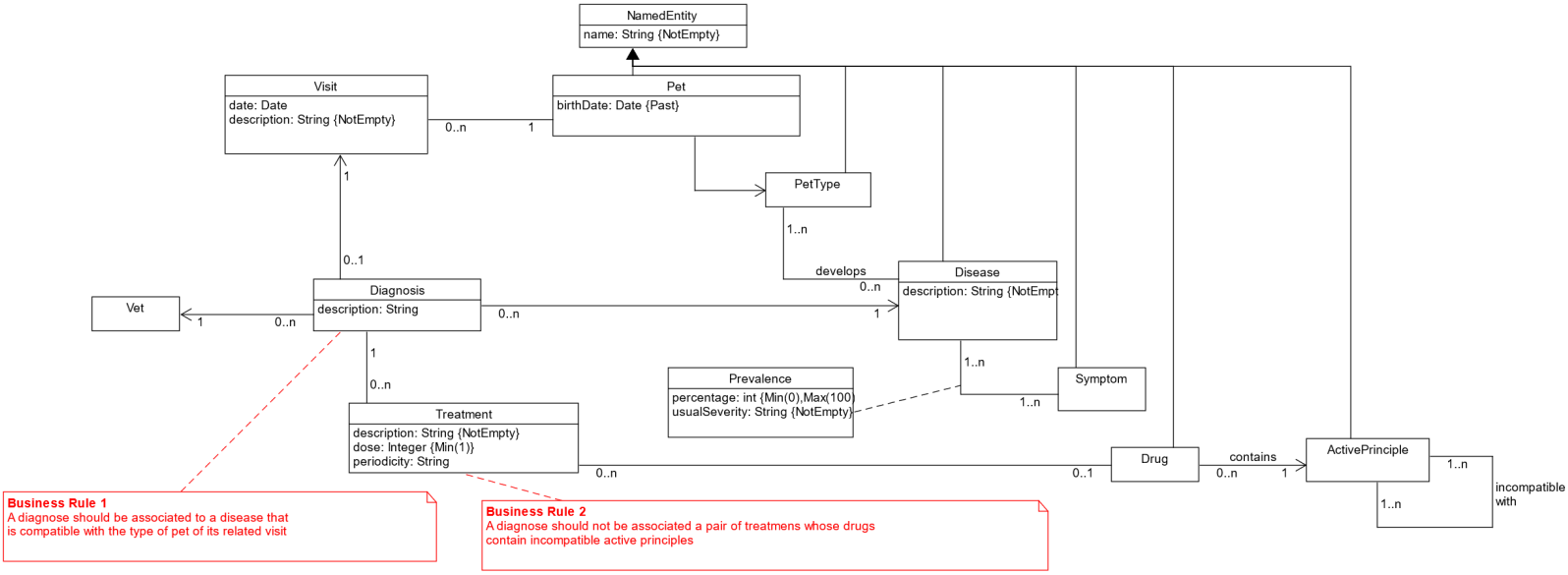
# Diagrama(s) UML:

## Diagrama de Dominio/Diseño

*En esta sección debe proporcionar un diagrama UML de clases que describa el modelo de dominio, recuerda que debe estar basado en el diagrama conceptual del documento de análisis de requisitos del sistema pero que debe:*

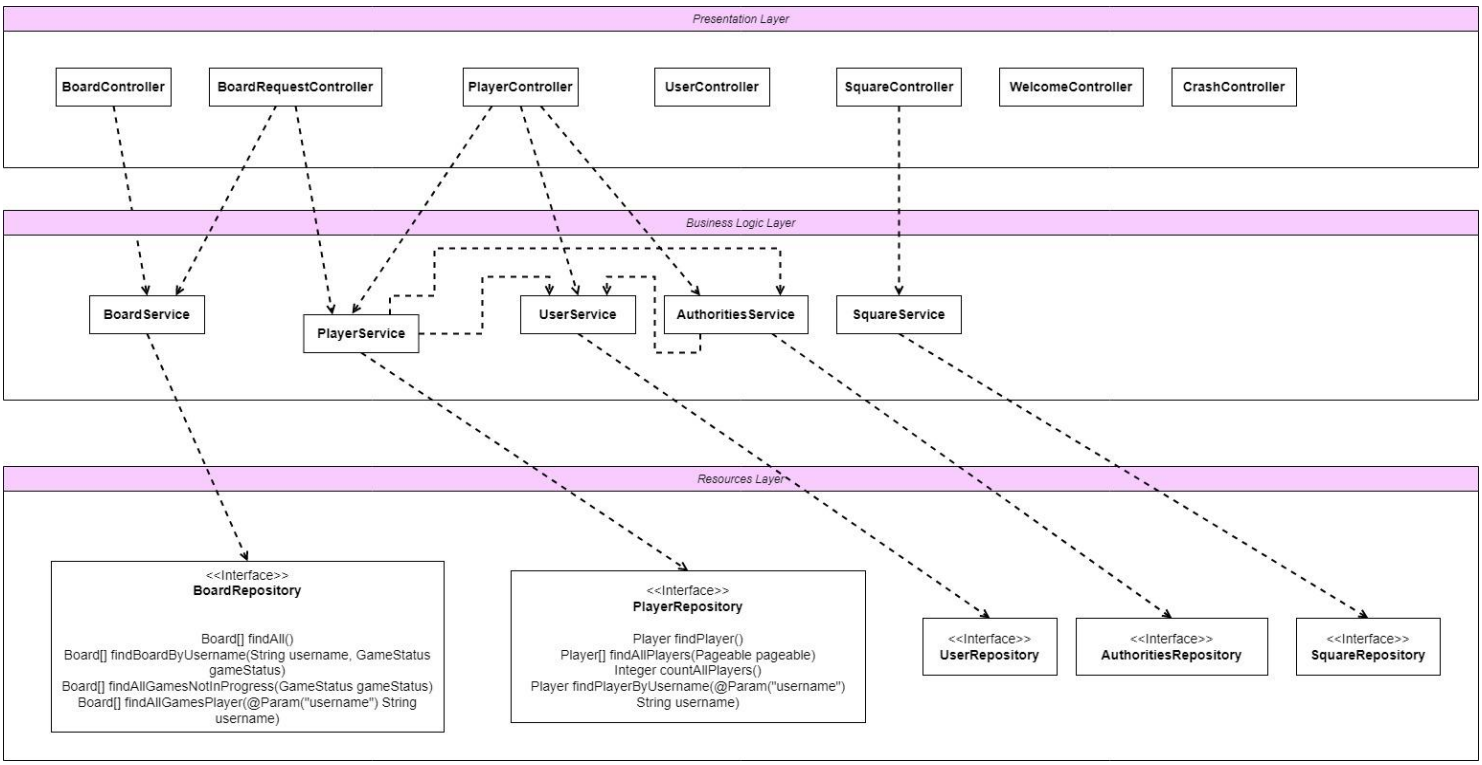
- *Especificar la direccionalidad de las relaciones (a no ser que sean bidireccionales)*
- *Especificar la cardinalidad de las relaciones*
- *Especificar el tipo de los atributos*
- *Especificar las restricciones simples aplicadas a cada atributo de cada clase de domino*
- *Incluir las clases específicas de la tecnología usada, como por ejemplo BaseEntity, NamedEntity, etc.*
- *Incluir los validadores específicos creados para las distintas clases de dominio (indicando en su caso una relación de uso con el estereotipo <<validates>>.*

*Un ejemplo de diagrama para los ejercicios planteados en los boletines de laboratorio sería (hemos omitido las generalizaciones hacia BaseEntity para simplificar el diagrama):*



## Diagrama de Capas (incluyendo Controladores, Servicios y Repositorios)

En esta sección debe proporcionar un diagrama UML de clases que describa el conjunto de controladores, servicios, y repositorios implementados, incluya la división en capas del sistema como paquetes horizontales tal y como se muestra en el siguiente ejemplo:




El diagrama debe especificar además las relaciones de uso entre controladores y servicios, entre servicios y servicios, y entre servicios y repositorios.

Tal y como se muestra en el diagrama de ejemplo, para el caso de los repositorios se deben especificar las consultas personalizadas creadas (usando la signatura de su método asociado).

# Patrones de diseño y arquitectónicos aplicados

En esta sección se especifica el conjunto de patrones de diseño y arquitectónicos aplicados durante el proyecto. Para especificar la aplicación de cada patrón puede usar la siguiente plantilla:

## Patrón: <Nombre del patrón>

## Tipo: Arquitectónico | de Diseño

## Contexto de Aplicación

Describir las partes de la aplicación donde se ha aplicado el patrón. Si se considera oportuno especificar el paquete donde se han incluido los elementos asociados a la aplicación del patrón.

### Clases o paquetes creados
Indicar las clases o paquetes creados como resultado de la aplicación del patrón.

### Ventajas alcanzadas al aplicar el patrón
Describir porqué era interesante aplicar el patrón.

## 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, Game.java, BaseEntity.java, NamedEntity.java, Person.java, Player.java, Square.java, Authorities.java y User.java .
- Servicios: BoardService.java, GameService.java, PlayerService.java, SquareService.java, AuthoritiesService.java y UserService.java .
- Repository: BoardRepository.java, GameRepository.java, PlayerRepository.java, SquareRepository.java, AuthoritiesRepository.java y UserRepository.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:
- games:
- pets:
- players:
- squares:
- users:
- Y exception.jsp y welcome.jsp.

Por último, en cuanto a los controladores implementados son: BoardController.java, GameController.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.petclinic.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 proporcionado 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, como 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.petclinic.model. También serían todas las clases .java 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

Todas las clase de Servicio

### 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

Todas las clase de Repositorios

### 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 y PlayerService.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.

### 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 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 “Square.java” por ejemplo.

### Clases o paquetes creados

Los paquetes y sus clases creados con este patrón son:

- paquetes: “org.springframework.samples.petclinic.model”.
- clases: “BaseEntity.java”, “NamedEntity.java” en “org.springframework.samples.petclinic.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 X

#### Descripción del problema:

Describir el problema de diseño que se detectó, o el porqué era necesario plantearse las posibilidades de diseño disponibles para implementar la funcionalidad asociada a esta decisión de diseño.

#### Alternativas de solución evaluadas:

Especificar las distintas alternativas que se evaluaron antes de seleccionar el diseño concreto implementado finalmente en el sistema. Si se considera oportuno se puden incluir las vestajas e inconvenientes de cada alternativa

#### Justificación de la solución adoptada

Describir porqué se escogió la solución adoptada. Si se considera oportuno puede hacerse en función de qué ventajas/inconvenientes de cada una de las soluciones consideramos más importantes.

Ejemplo:

### Decisión 1: Importación de datos reales para demostración

#### Descripción del problema:

Como grupo nos gustaría poder hacer pruebas con un conjunto de datos reales suficientes, porque resulta más motivador. El problema es al incluir todos esos datos como parte del script de inicialización de la base de datos, el arranque del sistema para desarrollo y pruebas resulta muy tedioso.

#### Alternativas de solución evaluadas:

*Alternativa 1.a:* Incluir los datos en el propio script de inicialización de la BD (data.sql).

**Ventajas**:

- Simple, no require nada más que escribir el SQL que genere los datos.

**Inconvenientes**:

- Ralentiza todo el trabajo con el sistema para el desarrollo.
- Tenemos que buscar nosotros los datos reales

*Alternativa 1.b:* Crear un script con los datos adicionales a incluir (extra-data.sql) y un controlador que se encargue de leerlo y lanzar las consultas a petición cuando queramos tener más datos para mostrar.

**Ventajas:**

- Podemos reutilizar parte de los datos que ya tenemos especificados en (data.sql).
- No afecta al trabajo diario de desarrollo y pruebas de la aplicación

**Inconvenientes:**

- Puede suponer saltarnos hasta cierto punto la división en capas si no creamos un servicio de carga de datos.
- Tenemos que buscar nosotros los datos reales adicionales

*Alternativa 1.c:* Crear un controlador que llame a un servicio de importación de datos, que a su vez invoca a un cliente REST de la API de datos oficiales de XXXX para traerse los datos, procesarlos y poder grabarlos desde el servicio de importación.

**Ventajas:**

- No necesitamos inventarnos ni buscar nosotros lo datos.
- Cumple 100% con la división en capas de la aplicación.
- No afecta al trabajo diario de desarrollo y pruebas de la aplicación

**Inconvenientes:**

- Supone mucho más trabajo.
- Añade cierta complejidad al proyecto

## Justificación de la solución adoptada

Como consideramos que la división es capas es fundamental y no queremos renunciar a un trabajo ágil durante el desarrollo de la aplicación, seleccionamos la alternativa de diseño 1.c.

Decisión 1: Hemos decidido reutilizar clases del proyecto de petclinic de spring para seguir la misma estructura del proyecto, es decir, reutilizando clases como user, person, …

Hemos decidido que no se pueda eliminar un juego

Decisión 2: 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