# PROJECTE PROGRAMACIÓ

# KENKEN

GRUPO 14.2
VERSIÓN 1.0 - 22/04/2024

- DESCRIPCIÓN ESTRUCTURAS DE DATOS Y ALGORITMOS

David Cañadas López
*david.canadas*

Raúl Gilabert Gámez
*raul.gilabert*

Guillem Nieto Ribó
*guillem.nieto.ribo*

Pau Zaragoza Gallardo
*pau.zaragoza*

# ÍNDICE

## Decisiones de diseño principales

**Clase Kenken**

- ➔ La asociación con las casillas (`Box`) la hemos implementado como una matriz de casillas para poder acceder eficientemente a una casilla concreta mediante una coordenada vertical y una horizontal. Esta matriz es un atributo de la clase y es de tipo final, ya que al crearse un kenken (y las casillas que le corresponden) de un cierto tamaño, éste no puede cambiarse luego.
- ➔ La asociación con un conjunto de regiones (`Region`) la hemos implementado como una lista (`List`) de referencias a las instancias, ya que como queremos que las regiones se puedan crear y eliminar dinámicamente en el modo de edición, la implementación de tipo lista encadenada ofrece operaciones eficientes.
- ➔ El algoritmo que hace la comprobación sencilla del tablero (`checkBoard`) se implementa comprobando las filas y columnas de todas las casillas de la diagonal principal, no sin antes verificar que todas las casillas del tablero tengan un valor asignado. Luego, y sólo en caso de que no hubiera errores, se itera por todas las regiones del tablero comprobando si el resultado de la operación sobre las casillas es correcto. De esta forma se puede verificar si una solución dada es correcta de forma eficiente.

**Clase Region**

- ➔ Hemos decidido tener las casillas (`Box`) asociadas a la región agrupadas en un array porque, como no necesitaremos añadir ni quitar casillas de una región, es más simple usar un array que otras estructuras como una lista.

**Clase Box**

- ➔ Hemos decidido usar un set en el caso de `possibleValues` ya que esto nos facilita añadir y borrar los posibles valores asociados a una casilla. Además, así evitamos tener problemas con valores repetidos

**Clase Operation (y sus subclases)**

- ➔ Decidimos hacer que cada operación tuviera una subclase de una clase operación (`Operation`) de la que heredaran un método `calculate()` en el que cada subclase ejecutara su respectiva operación.
- ➔ Optamos también por, en vez de que la operación tuviera acceso a la región (`Region`) a la cual pertenece y que desde allí accediera a las casillas (Box) de la región, que la región le pasara sus casillas como parámetro al método `calculate()`. Esto lo decidimos porque teníamos un dilema a la hora de crear regiones y operaciones, donde una región necesitaba una operación en su constructora y una operación necesitaba una región en su constructora.

**Clase User**

- ➔ Decidimos que hubiera una única clase usuario en lugar de una de usuario y otra de usuario registrado para evitar complicaciones en la programación.
- ➔ Se controla que el usuario esté registrado en base a si tiene nombre de usuario o está en blanco.
- ➔ Para comprobar la contraseña en lugar de pasarla al controlador de dominio, por temas de seguridad, decimos que mejor se hiciera a partir de un método propio de la clase al que se le pasa la contraseña que se quiere comprobar y devuelve un valor booleano de la correctitud de esta.

**Clase Ranking**

- ➔ Por simplicidad de la interacción entre las clases y el domaincontroller decidimos que el ranking trabajara únicamente con strings identificadores del usuario y enteros que guardan la puntuación de estos.
- ➔ La estructura interna de almacenamiento de los datos es un ArrayList ya que no es una estructura organizada en base a una clave que, para permitir la ordenación correcta, en este caso debería de ser la puntuación pero al tener la situación de varios usuarios con los mismos puntos se impediría el uso correcto de la estructura y también por ser este un valor que se estará modificando.
- ➔ A la hora de ordenar el ranking se realiza en base a los puntos de estos usuarios.

**Clase PairSI**

- ➔ Debido a no encontrar una clase Pair que se pudiera usar sin importar librerías externas decidimos crear una propia con los elementos requeridos por la clase Ranking.

**Clase PresentationController**

- ➔ Para leer un kenken, iremos leyendo línea a línea y lo organizaremos en un vector de Strings, así nos es más fácil organizar la entrada por regiones. La primera posición del vector estará ocupada por el tamaño del kenken y el número de regiones, y las siguientes posiciones estarán ocupadas por las diferentes regiones.

**Clase DomainController**

- ➔ Decidimos hacer las cosas simples trabajando con un sólo usuario (`User`) y kenken (`Kenken`) al mismo tiempo. Es por ello que la clase DomainController sólo tiene un atributo User y uno Kenken, en el que se guarda el usuario cuya sesión está abierta (o un usuario no registrado si es el caso) y el kenken sobre el que se está trabajando (o jugando).
- ➔ A la hora de crear un kenken (`Kenken`) a través de texto hemos decidido utilizar un array de strings (`String[]`) para transmitir la información entre capas por diversos motivos. Al principio intentamos utilizar un array de enteros (`int[]`) pero al ver que el prompt podía tener números entre

corchetes (‘[‘ y ‘]’) vimos que esta opción no era viable. El segundo motivo es que debido a que es más fácil de leerlo por consola en la capa de presentación. El tercero es que al controlador del dominio (`DomainController`) no le supone un gran esfuerzo parsear la información de cadenas de texto a números, siendo incluso de ayuda el hecho de que cada región esté en una cadena diferente. Por último, hemos visto claro que este método será fácilmente exportable a la capa de persistencia en un futuro.

➔ En cuanto al algoritmo para solucionar un kenken optamos por utilizar un algoritmo de fuerza bruta, en el que vamos probando casilla a casilla si un número podría encajar y devuelve ‘true’ si encuentra una solución. Además guarda la solución en el kenken actual del controlador de dominio (`DomainController`).