PROJECTE PROGRAMACIÓ

KENKEN

GRUPO 14.2
VERSIÓN 2.0 - 27/05/2024

- DESCRIPCIÓN ESTRUCTURAS DE DATOS Y ALGORITMOS // Decisiones de diseño

David Cañadas López
david.canadas

Raúl Gilabert Gámez
raul.gilabert

Guillem Nieto Ribó
guillem.nieto.ribo

Pau Zaragoza Gallardo
pau.zaragoza

ÍNDICE

Capa Dominio....2
Clase Kenken....2
Clase Region....2
Clase Box....2
Clase Operation (y sus subclases)....2
Clase User....3
Clase Ranking....3
Clase PairSI....3
Clase DomainController....3
Clase KenkenController....3
Clase UserController....4
Capa Presentación....5
Clase PresentationController....5
Clase MainViewController....5
Clase MainView....7
Clase KButton....8
Clase RankingView....8
Clase RegistrationView....8
Clase KenkenSelectionView....8
Clase ParameterSelectionView....8
Clase ImportKenkenView....8
Capa Persistencia....9
Clase PersistanceController....9

Decisiones de diseño principales:

Capa Dominio

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 DomainController

- ➔ Este es el encargado de comunicar el controlador de presentación (`PresentationController`) con los controladores de kenkens (`KenkenController`) y usuarios (`UserController`).
- ➔ Decidimos separar los métodos relacionados con los kenkens y usuarios en dos controladores separados para mejorar la legibilidad y no complicar demasiado las cosas.
- ➔ Prácticamente cada método del controlador de dominio equivale a un caso de uso.

Clase KenkenController

- ➔ A la hora de crear un kenken, decidimos simplificar mucho el flujo de trabajo para evitar así el máximo de errores posibles. Es por eso que dividimos el flujo en tres partes: primero crear el kenken de un cierto

tamaño especificado por el usuario, segundo rellenar el kenken con los valores válidos que proporciona el usuario, por último definir las regiones con sus operaciones y calcular el resultado de cada región. Gracias a esto nos aseguramos de que el usuario cree kenkens válidos sin necesidad de mucha corrección de errores ni potencia de cálculo para detectar si un kenken es válido o no.

- ➔ A la hora de crear un kenken a través de texto hemos decidido utilizar un array de strings para transmitir la información entre capas por diversos motivos. Al principio intentamos utilizar un array de enteros 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 leer la información de un archivo por líneas que por palabras. 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.
- ➔ A la hora de generar un kenken optamos por hacer dos métodos: uno que generase le kenken a partir de un tamaño y unas operaciones válidas y otro a partir del día actual. Esto facilita el uso de estas funciones a la hora de crear tanto kenkens aleatorios con un cierto tamaño y operaciones como para crear el kenken del día.
- ➔ 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.
- ➔ Para devolver la información del kenken actual a la capa de presentación decidimos utilizar un array de `Object`. Esto principalmente debido a que necesitábamos pasar muchas matrices y vectores con los distintos datos al controlador de presentación. Si no hubiéramos utilizado este método habríamos tenido que hacer varias funciones, en las que cada una devolviera una parte de la información. Utilizando este array podemos devolver toda la información de una forma más compacta y fácil de adquirir por parte de la capa de presentación.

Clase UserController

- ➔ Para el control de usuarios decidimos no guardar todos los usuarios en memoria. En cambio, sólo guardamos el usuario que está activo actualmente. Esto nos simplifica mucho el control y las operaciones con el usuario.
- ➔ Utilizamos la misma clase `User` para un usuario registrado que para uno que no lo está. Es por eso que cuando se cierra la sesión de un usuario, en vez de hacer que la variable `user` apunte a `null`, creamos un nuevo usuario no registrado, es decir, que no tiene nombre ni contraseña.

Capa Presentación

A nivel de presentación hemos decidido que tendremos una vista principal, donde encontraremos el menú y el tablero y donde se realizarán las funciones principales (seleccionar acción a realizar y jugar), y otras vistas secundarias que realizarán las demás funciones (iniciar sesión, mostrar el ranking, etc).

La clase PresentationController hará de intermediaria entre los controladores de las diferentes vistas, y también entre estos y el controlador de dominio. Hemos decidido tener un controlador para cada vista para tener mejor organizado el proyecto, ya que cada vista va a necesitar estructuras de datos diferentes y hemos pensado que sería mejor tenerlos separados.

Clase PresentationController

- ➔ Como ya hemos dicho antes, esta será la encargada de instanciar y controlar el controlador de dominio y los controladores de las vistas. De primeras parece que será muy lioso pero el resultado es que queda una capa de presentación bien estructurada.

Sólo pararemos a explicar la decisiones de diseño tomadas en la clase MainViewController porque en los otros controladores, encargados de las vistas más secundarias, prácticamente las únicas funciones que hay son de recibir el input del usuario y mandarlo al controlador de presentación para que haga lo que crea conveniente con la información, y devolver la respuesta a la vista si es necesario.

Clase MainViewController

- ➔ Es la clase encargada de controlar los distintos menús, donde el usuario podrá escoger las distintas opciones que ofrecemos, y también de controlar el tablero donde el usuario podrá crear y resolver kenkens. Debido a sus distintas funciones, la clase tiene un atributo que nos permite saber en qué estado se encuentra. Las posibles opciones son:
  - IN_MENU: el usuario no se encuentra ni creando ni resolviendo.
  - PLAYING: el usuario está resolviendo un kenken.
  - PLAYING_RANKED: el usuario está resolviendo un kenken en modo ranking. Nos es útil establecer esta diferenciación ya que hay ciertas ayudas que en modo ranked no están disponibles, por ejemplo avisar de un error al introducir un valor.
  - CREATING_NUMBERS: el usuario está creando un kenken y se encuentra en la fase de rellenar el tablero con los valores de las casillas.
  - CREATING_REGIONS: el usuario está creando un kenken y se encuentra en la fase de delimitar las diferentes regiones y decidir sus operaciones.

- ➔ Todo lo relacionado con la creación de kenkens es lo que nos ha llevado más trabajo. Como se puede ver arriba, la creación la hemos dividido en dos etapas. En la primera, el usuario se dedicara a rellenar, disponiendo los valores como el quiera, el tablero (se le avisará si algún valor es incorrecto). En la segunda etapa, el usuario podrá delimitar las diferentes regiones. Decidimos dividir esto en dos etapas ya que nos ofrece más control sobre el usuario, y así sabemos cuando va a introducir un valor o una nueva región.
  Los otros estados no tienen mucho misterio.
- ➔ Cuando el usuario esté jugando una partida no clasificatoria, estará en el estado PLAYING, y podrá rellenar el tablero y pedir la solución cuando quiera. Estando en este estado podrá recibir ayudas del sistema.
- ➔ Cuando el usuario esté jugando una partida clasificatoria, estará en el estado PLAYING_RANKED. Estando en este estado no podrá recibir ayudas del sistema. El kenken se resolverá cuando el usuario decida que ha terminado.
- ➔ Cuando no esté jugando ni creando, estará en el estado IN_MENU, donde decidirá su siguiente acción.
- ➔ Para que el usuario pueda crear un kenken a su gusto, la clase dispone de las siguientes estructuras:
  - selectedX y selectedY: coordenadas de la última casilla seleccionada.
  - regions: número de regiones creadas (sirve para identificar a cada región creada con un número distinto).
  - valueMatrix: matriz de enteros donde se irán guardando los valores que vaya introduciendo el usuario.
  - isSelected: matriz de booleanos que nos indica que casillas están seleccionadas para formar una nueva región.
  - inRegionMatrix: matriz de enteros que nos indicará a que región pertenece cada casilla.
  - boxesSelected: número de casillas seleccionadas.
  - boxesInRegion: indica cuántas casillas perteneces ya a una región (nos ayudará a saber cuando se ha acabado de crear un kenken).
  - opRegions: un Map<Integer,Integer>, con clave primaria el número de región y la clave secundaria el código de operación, que nos indica para cada región qué operación tiene asignada (decidimos usar un Map ya que es una estructura dinámica y porque así buscar la operación asociada a una región tiene coste constante).
  - sizeRegions: un Map<Integer,Integer>, con clave primaria el número de región y la clave secundaria el tamaño de la región). Guardar el tamaño de cada región nos facilita muchísimo el trabajo a la hora de preparar los datos para enviarselos al controlador de dominio.
- ➔ Todos los datos relacionados con la creación se recogen en la capa de presentación, y sólo cuando el usuario ha acabado se envían a la capa de dominio. Hemos decidido hacerlo así para no tener que comunicarnos constantemente con el dominio y porque las comprobaciones que se

realizan sobre los datos se pueden hacer en presentación. Los diferentes datos los recogemos y procesamos de las siguientes maneras:

- Los valores de las casillas se irán introduciendo en la matriz de valores a medida que el usuario los vaya añadiendo. Si introduce un valor que ya está registrado en esa misma fila o columna no se le dejará añadirlo.
- Para las regiones, se le pide al usuario seleccionar las casillas que desee y darle al botón correspondiente de añadir región con la operación que desee. De esta forma, el usuario puede sobrescribir regiones, seleccionando casillas que ya formaban parte de una.
  Cuando el controlador recibe que se ha pulsado un botón, mira de qué operación se trata y la asocia a un nuevo número de región en `opRegions`. Finalmente, el controlador mira que casillas había seleccionadas y las marca con el número de región pertinente en la matriz de regiones.
- Cuando se finaliza la creación, el controlador analiza todos los datos recogidos y los organiza en un vector de strings para que lo reciba el controlador de dominio (se organiza de forma similar al formato estándar).
  Para ello, el controlador recorrerá la matriz de regiones e irá añadiendo las coordenadas de las casillas a un Map<Integer,String>, donde la clave principal es el número de región y la secundaria será un String que contenga el código de operación de la región, seguido del número de casillas y de sus coordenadas. Cuando se haya recorrido la matriz, se irán recogiendo los Strings que forman el Map y se irán añadiendo al vector de Strings que se enviará al dominio.
  Hemos decido hacerlo así ya que de esta forma sólo recorremos la matriz de regiones una vez, y nos ahorramos recorrerla una vez por región. Aumentamos la complejidad pero reducimos bastante el tiempo para generar la información a mandar al dominio.
  Finalmente, el usuario decidirá si quiere simplemente jugar este kenken, guardarlo en la base de datos o guardarlo y jugarlo. Esta decisión también se la notificaremos al controlador de dominio, para que haga lo que le parezca necesario con los datos en cada caso.

Clase MainView

- ➔ En esta vista encontraremos el menú principal en forma de barra en la parte superior, que dispondrá de diferentes pestañas donde el usuario podrá acceder a todas las opciones y realizar la acción que desee.
  Adicionalmente, aquí también se generará el tablero sobre el que el usuario podrá resolver o crear kenkens.
- ➔ Esta vista tiene funcionalidades sencillas que permiten al controlador de la vista implementar diferentes respuestas dependiendo de estados y eventos anteriores. Por ejemplo, la vista llama al controlador cada vez que se pulsa un botón (una casilla del Kenken actual), indicando las

coordenadas de ésta en la matriz. Éste decide si guardar la posición para cálculos posteriores o hacer comprobaciones con su posición, o si indicar a la vista que marque o desmarque dicha casilla u otra diferente.

- ➔ La vista también ofrece otros métodos para que el controlador modifique el estado de la interfaz de usuario para reflejar el estado del Kenken, como (entre otras):
  - Modificar el indicador de operación y resultado de una región en una casilla particular.
  - Modificar el grosor de los bordes de una casilla para poder delimitar regiones en el tablero.
  - Borrar el Kenken actual de la pantalla, o bien sustituirlo por uno distinto.

Clase KButton

- ➔ Esta clase que extiende JButton es necesaria para poder mostrar el indicador de operación y resultado para las regiones. Dado que cualquier casilla del tablero puede tener este indicador, vimos necesario hacer una clase nueva que pudiera mostrarlo dinámicamente.
- ➔ También guarda el estado del tamaño de sus bordes, para poder modificarlos con mayor granularidad (a nivel de casilla, cuando sea necesario) a la hora de definir regiones.

Clase RankingView

- ➔ En esta vista se generará un listado donde aparecerán los usuarios con sus respectivos puntos, ordenados de mayor a menor puntuación.

Clase RegistrationView

- ➔ En esta vista aparecerán dos recuadros de texto donde el usuario insertará su nombre de usuario y su contraseña.
- ➔ Utilizaremos la misma vista para la función de registrar nuevo usuario y para la de iniciar sesión.

Clase KenkenSelectionView

- ➔ En esta vista aparecerá un listado de kenkens, donde se mostrará su id y el nombre del usuario que lo creó (si es un kenken que no pertenece a ningún usuario también se especificará). Dispondrá también de un botón que servirá para cargar el kenken seleccionado por el usuario.
- ➔ Al principio de la lista aparecerá la partida guardada por el usuario, si este anteriormente guardó la partida para poder continuar después.

Clase ParameterSelectionView

- ➔ Esta vista dispondrá de distintos botones que permitirán al usuario seleccionar:
  - Si quiere crear el kenken manualmente o quiere que se lo genere el sistema.

- Tamaño del tablero. Estos botones serán excluyentes (sólo se puede seleccionar uno a la vez).
- Operaciones permitidas (sólo disponible si se selecciona generar por el sistema)

Clase ImportKenkenView

➔ En esta vista se podrá acceder al sistema de ficheros del usuario desde donde el usuario podrá seleccionar un archivo que contenga un kenken en formato estándar para obtener su ubicación y poder abrirlo en el sistema.

Capa Persistencia

En cuanto a la capa de persistencia hay poca cosa que comentar. Maneja la lectura y escritura de datos tanto de kenkens como de información de usuarios, controlando la existencia de esta información y lanzando las excepciones pertinentes si no es posible acceder a esta o crearla.

Clase PersistanceController

➔ Es la única clase de la capa y controla la creación y acceso de datos. La creadora de la clase gestiona la creación de los directorios necesarios para el almacenamiento de la información en caso de que no existan.

➔ Hemos decidido que toda esta información se guarde en un directorio con el nombre *data* que contendrá a su vez un directorio *users* y uno *kenkens* que tendrán la información de estas clases almacenadas.

➔ Desde esta clase, por tanto se pueden crear y leer usuarios y kenkens, comprobar la existencia de un usuario y obtener una lista de los usuarios como también obtener una lista de todos los kenkens base y los kenkens en juego por un usuario en específico.