Documentación Segunda Entrega

PROP QT 2023-2024

Equipo 33.1

Luis Jesús Valverde Zabaleta (luis.jesus.valverde)

Juan José Torredemer Pueyo (juan.jose.torredemer)

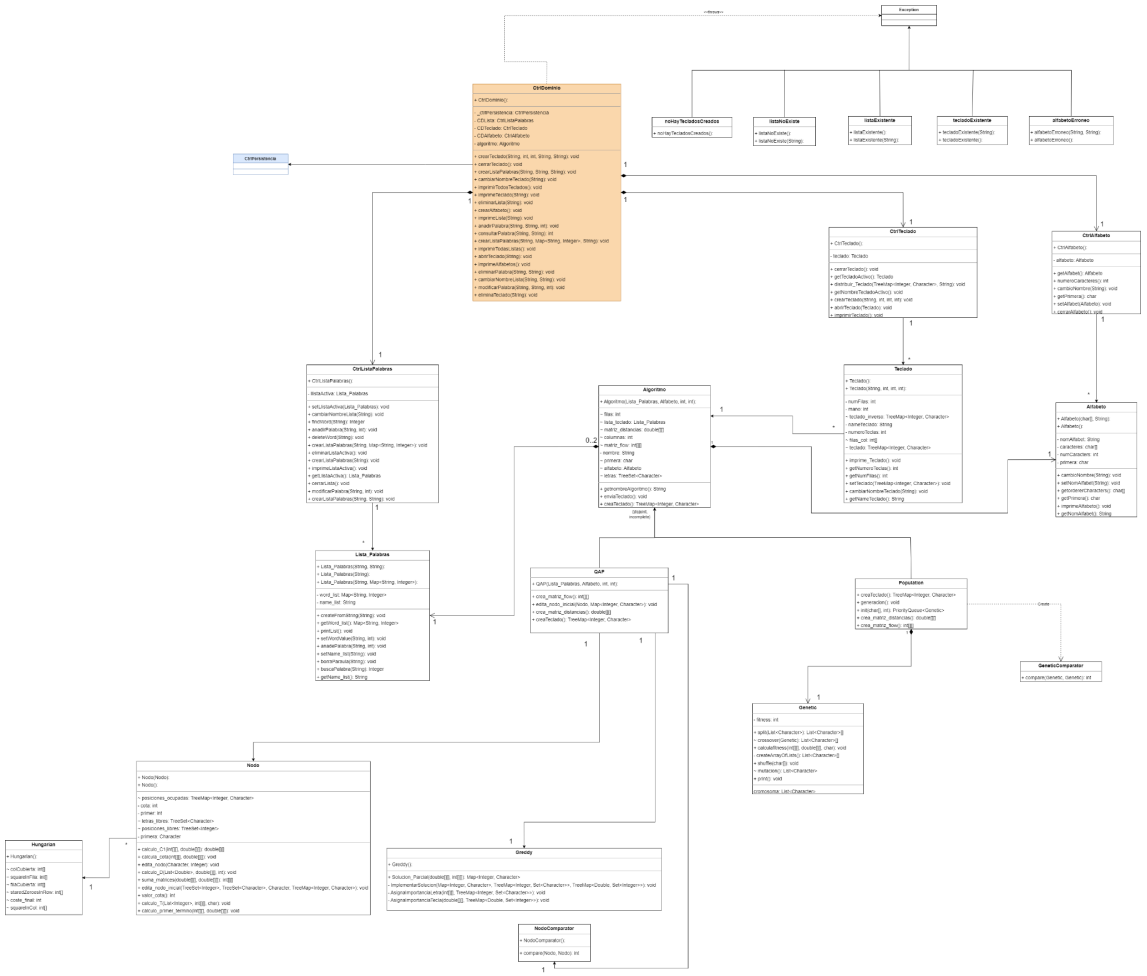
Iván Parreño Benítez (ivan.parreno)

Rafael Ibáñez Rodríguez (rafael.ibanez.rodriguez)

Índice

Capa de Dominio

Diagrama



**En el SVG de la misma carpeta se ve en alta definición*

Descripción de las clases

Teclado

La clase teclado es la encargada de definir una instancia de teclado, donde tendremos los siguientes atributos:

```
private String nameTeclado;
private int numeroTeclas;
private int numFilas;
int[] filas_col= new int[numFilas];
TreeMap<Integer,Character> teclado;
TreeMap<Integer,Character> teclado_inverso;
private int mano;
```

nameTeclado definirá el nombre del teclado, numeroTeclas definirá el número de teclas del teclado, numFilas definirá el número de filas que tendrá el teclado, y filas_col será un vector que nos dirá cuántas columnas tendrá cada fila.

Guardaremos la relación letra y tecla en teclado que será un TreeMap<Integer, Character> que nos permitirá tener ordenado de forma creciente según el número de tecla la letra que le corresponde. teclado_inverso será lo mismo pero invertido según si es para zurdo o diestro que se especificara con la variable mano.

Lista de Frecuencia de Palabras

Esta clase la usamos para instanciar una lista de frecuencia de palabras para posteriormente poder crear un teclado mediante su algoritmo. La lista de palabras estará compuesta por un mapa, en un apartado posterior comentaremos la razón detrás de nuestra elección. Los atributos son los siguientes:

```
private String name_list;
private Map<String, Integer> word_list;
```

En nuestro caso el nombre de la lista un String que es la palabra como clave y un número entero que representa su frecuencia.

En cuanto a las constructoras de la función tenemos varios tipos de maneras de crear la lista:

```
public Lista_Palabras (String name) {
        this.name_list = name;
}

public Lista_Palabras (String name, String text) {
        this.name_list = name;
        createFromString(text);
}

public Lista_Palabras (String name, Map<String, Integer> freq) {
        this.name_list = name;
        this.word_list = freq;
}
```

La primera crea una instancia de la lista de palabras, pero no la inicializa, la segundo crea la instancia y la inicializa en una función que crea la lista a partir de un texto. Por último, la tercera recibe el nombre y un mapa con la lista como parámetro para crear la instancia.

Para acabar en cuanto a los métodos tenemos uno para crear la lista a partir del texto, para imprimir la lista, buscar una palabra, y los getters y setters.

Alfabeto

La clase alfabeto es la que se encarga de gestionar un vector de caracteres y un nombre, que son sus atributos. No hemos considerado la creación de nuevos alfabetos o la modificación de estos.

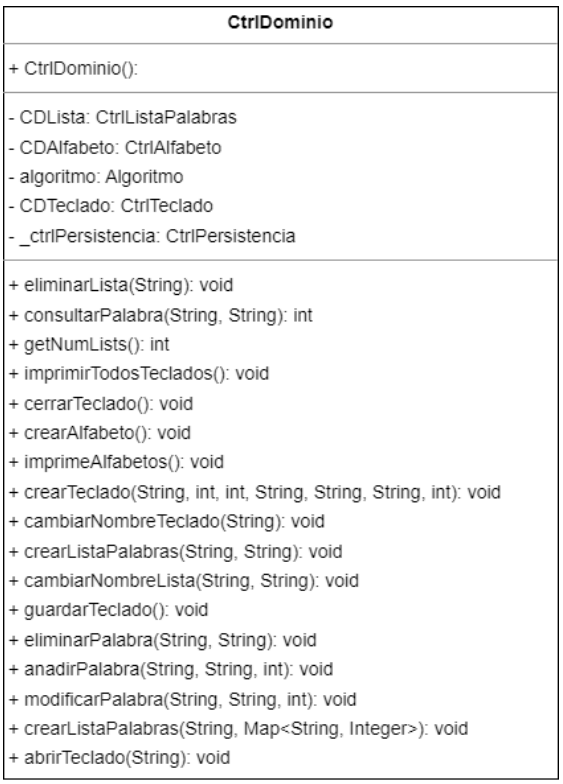
```
private char[] caracteres;
private String nomAlfabet;
private int numCaracters;
private char primera;
```

Los métodos de esta clase se encargan de imprimir el alfabeto, y el resto no dejan de ser getters y setters de la clase del modelo.

Controlador del Dominio

El controlador del dominio es el encargado de gestionar las instancias de los controladores y de comunicarse con las capas de persistencia y de presentación. Es el núcleo que lo une todo. Todas las funcionalidades que se lanzan desde la capa de presentación se gestionan desde el controlador del dominio.

Por lo tanto los campos del controlador del dominio son los controladores de las diferentes clases del modelo (lista de frecuencias, alfabeto, teclado), y la clase algoritmo que no va a necesitar un controlador crearemos la distribución del teclado mediante la clase algoritmo directamente desde el dominio.



Aquí podemos observar la clase en el diagrama del modelo, podemos observar los atributos, que comentaremos más abajo, y los métodos que son bastante auto explicativos sobre la función que realizan

Controlador de las Listas de Frecuencias

El controlador de las listas de frecuencia controla las instancias de la clase lista de palabras, toda función que se vaya a ejecutar sobre una lista se lanza desde aquí.

En el controlador hemos introducido un mapa con clave String y valor Lista para poder reducir tiempos en las llamadas a la capa de persistencia, en este mapa del controlador almacenamos las listas que ya hemos cargado. Esto realmente solo nos reduce tiempos

si hacemos una consulta y ya tenemos instanciada la lista, o para crear un teclado si la lista ya la hemos instanciado antes.

```
private Lista_Palabras llistaActiva;
private Map<String, Lista_Palabras> listasInstanciadas;
```

Estos son los campos del controlador, la lista sobre la que estamos realizando las operaciones, y las listas que hemos cargado desde la capa de persistencia.

Los métodos son variados, tenemos un ejecutor de las funciones básicas de la lista activa, así como funciones encargadas de buscar listas en el mapa y cambiar la lista activa. Es decir, getters y setters.

Controlador del Alfabeto

El controlador del alfabeto gestiona las instancias de la clase alfabeto, y todas las funciones relacionadas con un alfabeto se ejecutan a través de este controlador

En este controlador también hemos incorporado un mapa, con clave String y valor Alfabeto, que permite una mejora significativa en la eficiencia sobretodo cuando se realiza consulta o se necesita la instancia de un alfabeto ya cargado, ya que al no permitir la creación, eliminación o modificación de los alfabetos, estos los cargaremos al principio y ya no volveremos a hacerlo.

```
private Alfabeto alfabeto;
private static Map<String, Alfabeto> alfabetoMap;
```

Los campos del controlador, que representan la instancia actual de la clase alfabeto con la que realizamos las operaciones y el mapa compartido por todas las instancias del controlador, que almacena las instancias de los alfabetos asociados a un nombre.

Los métodos proporcionan funcionalidades para obtener, establecer y cerrar alfabetos, así como acceder a información del alfabeto activo.

Controlador del Teclado

El controlador del teclado controla las instancias de la clase teclado, centraliza todas las operaciones relacionadas con los teclados.

En este controlador también implementaremos el mapa con el objetivo de reducir tiempos de ejecución, por ejemplo en el caso de que el teclado lo creas, y luego lo quieres abrir, así no el programa no tendrá que pedirle el teclado a la capa de datos.

```
private Teclado teclado;
```

Este campo representa la instancia de la clase teclado y sus métodos permiten la creación, modificación y la consulta de información de los teclados, además de abrir teclados existentes y obtener el teclado activo.

Greedy

La clase Greedy se encargará de generar una primera solución, que nos permitirá ahorrar el recorrido de diferentes nodos y empezar desde allí. Nuestro algoritmo Greedy se basa en colocar 4 letras en nuestro teclado. La letra que más peso tiene en la matriz de Flow la colocaremos en la tecla con menor sumatorio de distancias con las otras teclas (habitualmente esta tecla suele ser la del medio). Después de colocar la primera letra, colocaremos las tres que menos influencia tienen en nuestra matriz de Flow, y estas como son las menos utilizadas las colocaremos en las teclas que tienen los mayores costes en el sumatorio de distancias.

La clase Greedy solo tendrá una función pública que se encargará de devolver la solución parcial e inicial, y que de parámetros tendrá la matriz de Distancias, la matriz de Flow y el primer carácter del alfabeto en que se base el teclado, esta es la cabecera de la función solución Parcial:

```
public static Map<Integer, Character> Solucion_Parcial(double[][]
distancia, int[][] flow, char primera)
```

Las estructuras de datos que utilizaremos en esta clase son las siguientes:

```
Map<Integer, Character> solucion
TreeMap<Double, Set<Integer>>  mejores_teclas
TreeMap<Integer, Set<Character>> mejores_letras
```

El Map de clave Integer y valor Character es un HashMap donde colocaremos la primera solución y que será lo que retorne la clase Greedy al llamarlo. Integer será el numero de la tecla y el Character la letra que se colocará en esa tecla.

El TreeMap de Double y Set<Integer> es un Map **basado en árbol y** es donde guardaremos las teclas ordenadas de forma descendente. Double representa el sumatorio de las distancias de una tecla determinada con el resto y es la clave del TreeMap, y el valor es el Set<Integer> que representa donde guardamos las teclas con ese mismo sumatorio de distancias ja que diversas teclas pueden tener el mismo sumatorio de distancias.

El TreeMap de Integer y Set<Character> es un Map **basado en árbol** y es donde guardaremos las letras ordenadas en forma ascendente. Integer representara el sumatorio de la letra con su relación con las otras y será la clave del TreeMap, y el valor es el Set<Character> que representa donde guardaremos las letras con el mismo sumatorio de relación con las otras letras, usamos el Set porque puede presentarse el caso de que haya letras con la misma clave.

Inicializo las anteriores tres estructuras y a mejores_teclas la paso como parámetro junto la matriz de Distancias a la función ***AsignaImportanciaTecla*** que se encargara de hacer el sumatorio de distancias de cada tecla con respecto al resto, y la acumulare en mejores_teclas, recorreré la matriz de Distancias y por cada fila que representa una tecla, sumare en una variable llamada result todas sus columnas. Después a mejores_letras hago algo parecido a lo anterior, la paso como parámetro junto a la matriz de Flow y el carácter primera a AsignaImportanciaLetra y se encarga de calcular la influencia de cada letra respecto a las otras, haciendo un sumatorio de sus relaciones, literalmente es lo mismo que la de teclas solo que en vez de recorrer la matriz de Distancias recorro la de Flow.

Luego cuando ya tengo mejores_teclas y mejores_letras construidas, llamo a ***ImplementarSolucion*** que se encargara de añadir las teclas con su correspondiente letra en el Map solución. La letra y tecla más influyentes estarán en el último elemento de sus respectivos TreeMaps y accedemos y conseguimos su valor, y la introducimos en solución. Después mediante iteradores recorro los dos TreeMaps y con un contador inicializado a 0, entro e un bucle while que le meteré como condición que haya next en los iteradores y que contador < 3, después voy relacionando teclas con letras y añado hasta tres parejas de estas.

Hungarian

La clase Hungarian principalmente solo se basará en calcular el coste de la matriz C3 mediante un conjunto de operaciones que serán descritas a continuación. La uncia función pública será Coste_Hungarian a la que pasaremos la matriz C3 como parámetro, esta es la cabecera de la función:

```
public int Coste_Hungarian(int[][] matriz)
```

Tendremos como atributos de clase los siguientes vectores de int[] que nos servirán como indicadores para realizar los pasos para el cálculo del coste final:

```
int[] marcaInFila, marcaInCol, filaCubierta, colCubierta,
zerosInFila;
```

Si la matriz que pasamos no es cuadrada, nos encargamos de ampliarla mediante la función ***normalizar*** que le pasaremos matriz, final_size (será el máximo entre num_filas y num_cols), num_filas y num_cols.

Para evitar las molestias de copias me genero una matriz_costes que será una copia de matriz sin modificar y que no compartirán referencia porque hare una Deep Copy.

El siguiente paso será inicializar los atributos de la clase, y poner zerosInFila, marcaInFila y marcaInCol con -1.

Ahora tocara restar el máximo de cada fila a esta misma, y el máximo de cada columna a esta misma también mediante la función ***restar*** y pasándole como parámetro solo la matriz.

Después haremos ***marcar_zeros*** y ***marcar_columnas*** que se encargaran respectivamente de marcar los ceros cuando no compartan columna y fila y se indicaran en *marcaInFila y marcaInCol*, y marcar_columnas que me marcara las columnas que tienen cero asignado y se les dará valor 1, y sino 0.

Después entrare en un bucle while que tendrá de condición la función ***columnasCubiertas*** que devuelve un booleano y que se encargará de iterar por colCubierta y si hay algún elemento con valor 0 retornamos false, y si no hay ningún 0 retornamos true.

Me genero un vector llamado ***mainZero*** que lo inicializaremos con la función ***zero_sin_cubrir***, que se encargara de localizar los ceros no cubiertos y meterlos en el vector las coordenadas i, j, y marcarlo en zerosInFila.

Si ***mainZero*** es null calculamos el mínimo que de lo no cubierto y lo restamos a los no cubierto y lo sumamos en las posiciones que tienen columnas y filas marcadas, y volvemos a llamar a zero_sin_cubrir.

Después si marcaInFila en la posición mainZero[0] es -1, hacemos el ***marcarRestantes*** que se encargara de modificar macarInCol para posteriormente volver a hacer ***marcarColumnas.***

Si no es -1, cubrimos la fila de mainZero, y desmarcamos la columna de mainZero, y volvemos a calcular el mínimo que de lo no cubierto y lo restamos a los no cubierto y lo sumamos en las posiciones que tienen columnas y filas marcadas.

Cuando acaba el bucle, en marcaInCol tendremos las filas que esta el 0 y la i del bucle nos indicará la columna, lo que nos permitirá ir acumulando en coste los valores respectivos. Después devolvemos el coste y acaba la función.

QAP/Nodo

El problema por resolver en esta primera entrega es el denominado QAP. La idea de este problema es ubicar n instalaciones, en este caso letras, en n instalaciones que serán las posiciones del teclado.

Para resolver este problema, hemos utilizado el algoritmo denominado Branch&Bound. Cabe destacar que aunque la clase la hemos llamado QAP, el algoritmo que se implementa es el anteriormente mencionado.

La idea general sobre este algoritmo es que se crea un árbol iterativo donde vamos almacenando diferentes soluciones parciales a las que hemos denominado Nodo. Cada nodo tiene un coste al que hemos llamado cota. Además también contiene diferentes TreeSet de Integer y Caracteres donde guardaremos las letras no ubicadas y las posiciones sin asignar. Y por último un TreeMap<Integer,Character> que definiría la estructura del teclado.

Durante la ejecución del árbol iterativo, se crea un Nodo inicial padre proveniente del Greedy, para agilizar el tiempo de computación. Está solución inicial se añade a una priority queue de nodos que será el árbol. A medida que vamos añadiendo letras en posiciones y evaluamos cotas, definimos con cuales queremos o no quedarnos. La idea

general es que dado una rama con un nodo padre con valor de cota x, y un primer hijo con valor de cota y, vayamos almacenando los hijos con cota z tal que:(z < x || z < y). En caso de que todos los hijos tengan peor cota que el padre, nos quedaremos con el hijo con mejor cota y desechamos el resto. Sino, iremos almacenando los mejores hijos. Si hay dos hijos con la misma cota, nos quedaremos con el primero encontrado.

Finalmente, para conseguir la solución, iremos consultando si el nodo que "desencolamos", el nodo con cota más baja, tiene ya todas las letras asignadas, si es así, hemos encontrado una solución y crearemos el teclado.

Genetic (Pendiente de Finalización)

El algoritmo Genético es el segundo algoritmo implementado en este proyecto. Este algoritmo es definido como un método de búsqueda heurística. Tiene diferentes versiones y posibles desarrollos, sobre todo en la parte de "crossover", pero nos centraremos únicamente en la que hemos escogido.

Este algoritmo consta de X partes durante toda su ejecución. La primera es la generación de una población inicial aleatoria entre 60 y 100 aproximadamente. Cada miembro de esta generación tendrá un fitness según la posición de las letras en un char[ ] que representará el teclado. Este conjunto de listas es guardado en un PriorityQueue, dónde se ordenarán de menor a mayor según su fitness, y a partir de aquí es dónde empieza verdaderamente el algoritmo.

De esta generación se escogen a los mejores individuos, en este caso nos quedamos con el primer tercio y desechamos todo el resto. Hemos almacenado todos los individuos buenos en otra PriorityQueue y iteramos sobre esta n/2 veces, siendo n el tamaño de la cola. El motivo de esta iteración es que crearemos dos nuevos hijos cada 2 miembros de la generación mediante un crossover.

El crossover es un método donde mezclamos la información de dos miembros padres para crear dos nuevos miembros hijos. El tamaño de la información que se cruza entre padres (letras), además de las posiciones que se cruzan(teclas) es determinado aleatoriamente aunque como mínimo se intercambiarán 4 letras para que el crossover tenga un mínimo de sentido.

Una vez finalizado el crossover, llegamos a la etapa de mutación que consiste en intercambiar dos o más posiciones aleatorias de un miembro hijo, con tal de conseguir soluciones parciales más precisas que con el crossover serían más difíciles. Se hacen pequeños cambios con el objetivo de cambiar el fitness un poco, ya que si empezamos a intercambiar muchas posiciones estaríamos generando un nuevo hijo, en vez de crear una mutación.

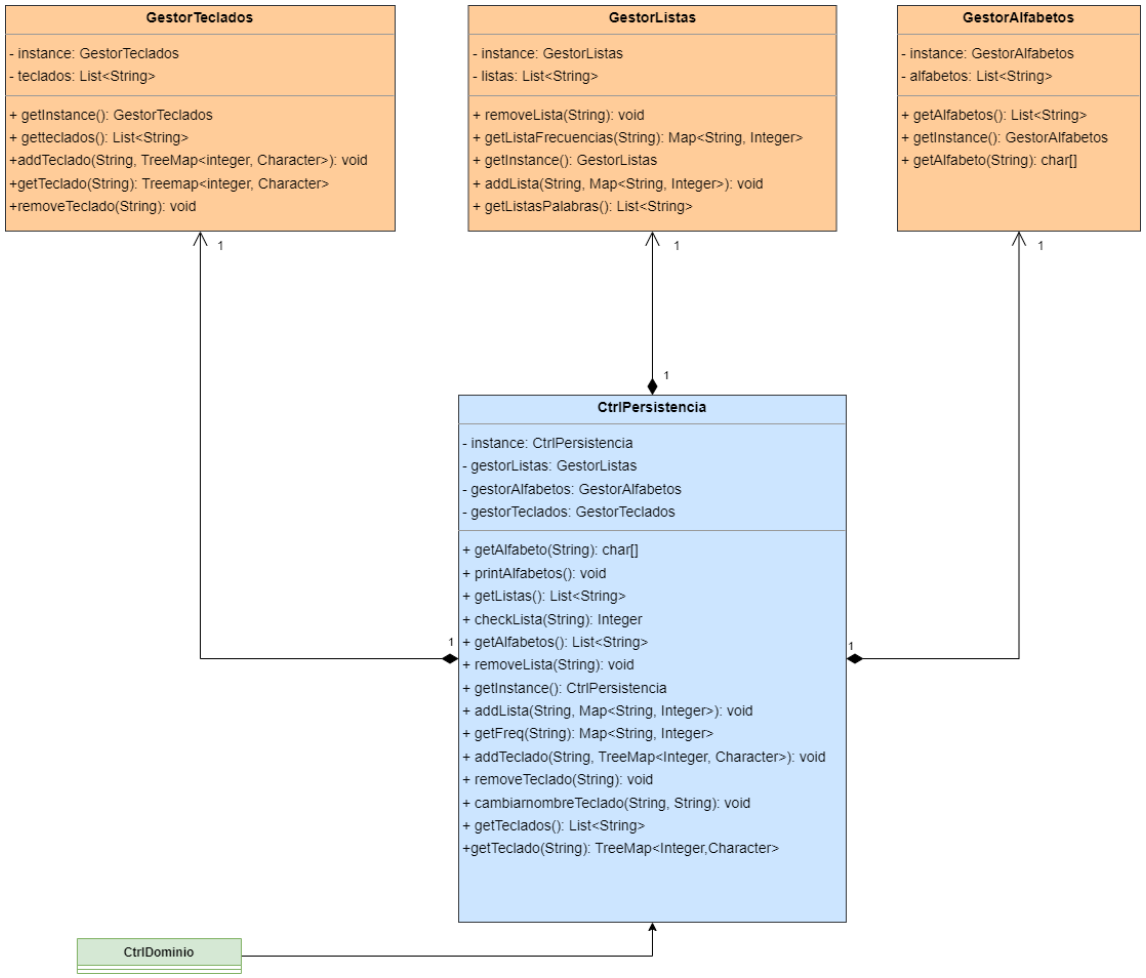
Finalizada esta etapa, hemos creado una nueva generación y volveremos a repetir el proceso. Por tanto todas estas fases están contenidas en un bucle de tamaño X, que va creando y eliminando soluciones. Este proceso se repite X veces y una vez acabado el bucle, nos quedamos con el primer miembro de la lista o, si una solución lleva repitiendo durante 100 generaciones, ya nos quedamos con esta.

Es importante entender que la solución obtenida durante el genético no tiene porque ser la óptima. La mejor solución puede distar muy poco de la encontrada, pero quizá para dar con ella se requiere de un mayor tiempo de computación, además de cierta aleatoriedad, que en este caso no vale tanto la pena. Por ese motivo, hemos pensado que cuándo una solución se repetía durante 100 generaciones, ya se podía imprimir.

Por último, queríamos informar, que la optimización del algoritmo no se encuentra al 100% finalizada y que es probable que hayan diferencias entre la versión presentada en esta entrega y la versión final. Quizá se añade alguna forma nueva de mutación o algún cambio en el crossover, pero informaremos de dicho cambio en la implementación en la última entrega.

Capa de Persistencia

Diagrama



**En el SVG de la misma carpeta se ve en alta definición*

Descripción Capa de Persistencia

Para llevar a cabo el proyecto mediante una programación orientada a objetos usamos una arquitectura de tres capas. Una de esas capas es la capa de persistencia, esta capa también se le puede llamar capa de datos, y es la encargada de almacenar con los datos pero no necesita saber cómo tratarlos. Hemos optado por usar archivos de texto llano, o txt, para almacenar las listas, los alfabetos, y los teclados. En los controladores de la capa de dominio hemos creado un mapa con los objetos ya instanciados para reducir el tiempo que se necesita cada vez que se carga un objeto. Dicho esto cada vez que hay un cambio en un objeto se actualiza en la capa de persistencia.

Controlador de Persistencia

Para empezar hemos creado un controlador de la capa de persistencia que se dedica a gestionar la comunicación de esta capa con la capa de dominio. Todas las peticiones de

la capa de dominio llegarán al controlador y este se comunicará con los gestores de los diferentes tipos de archivos que tenemos.

```
private static CtrlPersistencia instance = null;
private GestorAlfabetos gestorAlfabetos;
private GestorListas gestorListas;
private GestorTeclados gestorTeclados;

private CtrlPersistencia() {

    gestorListas = GestorListas.getInstance();
    gestorAlfabetos = GestorAlfabetos.getInstance();
    gestorTeclados = GestorTeclados.getInstance();
}

public static CtrlPersistencia getInstance() {
    if (instance == null) instance = new CtrlPersistencia();
    return instance;
}
```

En la figura que se muestra arriba podemos observar los campos, la constructora, y la búsqueda de la instancia singleton. Este controlador, y los gestores los hemos declarado como singletons ya que consideramos que solo es necesaria una instancia, no tiene sentido que haya más de un gestor de las listas de la capa de datos.

Clases de la capa de persistencia

Hemos optado por crear tres clases para gestionar los datos (puede que en la tercera entrega añadamos alguno más) cada una de estas gestiona las principales clases del modelo del dominio. Tenemos las siguientes:

GestorListas

Esta clase gestiona los datos de las listas. En la capa de datos guardamos un índice con todos los nombres de las listas, y en la misma carpeta llamada DATA tendremos una carpeta listas en las que se encuentra un .txt con el nombre que aparece en el índice en los cuales tenemos la lista de frecuencias.

El gestor tiene solo unas pocas clases que tienen unas funcionalidades muy básicas. Estas funciones incluyen el almacenamiento, borrado, y listado de las listas presentes en la capa de persistencia.

```java
/**
* Función que añade una lista
* @param lista nombre de la lista
* @param freq frecuencias de la lista
*/
public void addLista(String lista, Map<String, Integer> freq){}
* Función que elimina una lista
* @param lista nombre de la lista a eliminar
*/
public void removeLista(String lista){}


* Función que devuelve las frecuencias de una lista
* @param lista nombre de la lista
* @return frecuencias de la lista
*/
public Map<String, Integer> getListaFrecuencias(String lista){}

/**
*  Función que devuelve una lista con el nombre de todas las
listas
* @return lista con todas las listas
*/
public List<String> getListasPalabras(){}
```

GestorAlfabetos

Esta clase que gestiona los alfabetos es quizás la menos usada de la capa de datos ya que por el momento no hemos optado por hacer funcionalidades de añadido/eliminación/modificación de los Alfabetos. Tendremos unos que están predefinidos en nuestra base de datos.

Por lo tanto, este gestor solo tendrá la necesidad de cargar los archivos a nuestra aplicación una vez ésta se inicie. En el controlador del dominio los almacenamos durante el transcurso de la actividad de la aplicación. Cada vez que iniciamos la aplicación los volvemos a cargar, esto lo hacemos para en un futuro dar lugar a aumentar la cambiabilidad del sistema y permitir las funcionalidades previamente mencionadas.

Esta inicialización se hace siempre desde la capa de dominio por lo tanto las clases de este gestor son muy parecidas al anterior, listamos los alfabetos almacenados, y podremos obtener los alfabetos siempre.

```
private GestorAlfabetos() {
    try {
       Scanner in = new Scanner(new FileReader
("../../DATA/alfabetos.txt"));
        while (in.hasNextLine()) {
            alfabetos.add(in.nextLine());
        }
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    }
}

/**
* Función que devuelve los alfabetos
* @return lista de todos los alfabetos
*/
public List<String> getAlfabetos() {}

/**
* Función que devuelve un alfabeto
* @param alfabeto nombre del alfabeto
* @return alfabeto
*/
public char[] getAlfabeto(String alfabeto){
}
```

Gestor Teclados

La clase gestorTeclados se encarga de almacenar la información de los teclados. Mediante una carpeta teclados en la carpeta DATA que guardará por cada teclado que sea creado en el programa, un .txt con su respectivo nombre y el orden de los caracteres.

Lee la información del teclado desde archivos, imprimir su distribución de teclas por consola y gestionar la creación y eliminación de los teclados

Obtenemos la distribución del teclado con el nombre escogido, leyendo la información desde un archivo y se retorna como un TreeMap que mapea la posición a un carácter.

Y obtener la lista de nombres de los teclados que tenemos en persistencia

```java
/**
* Obtiene la distribución de teclado asociada al nombre nom.
* @param nom Nombre del teclado.
* @return TreeMap que representa la distribución de teclas.
*/
public TreeMap<Integer, Character> getTeclado(String nom){
/**
* Obtiene la lista de nombres de todos los teclados existentes
en el sistema.
* @return Lista de nombres de teclados.
*/
public List<String> getTeclados(){}
```

Se agrega un nuevo teclado al sistema con el nombre y la distribución especificados, saltando excepción si ya existe y se elimina el teclado identificado por el nombre, realizando las actualizaciones en los archivos correspondientes

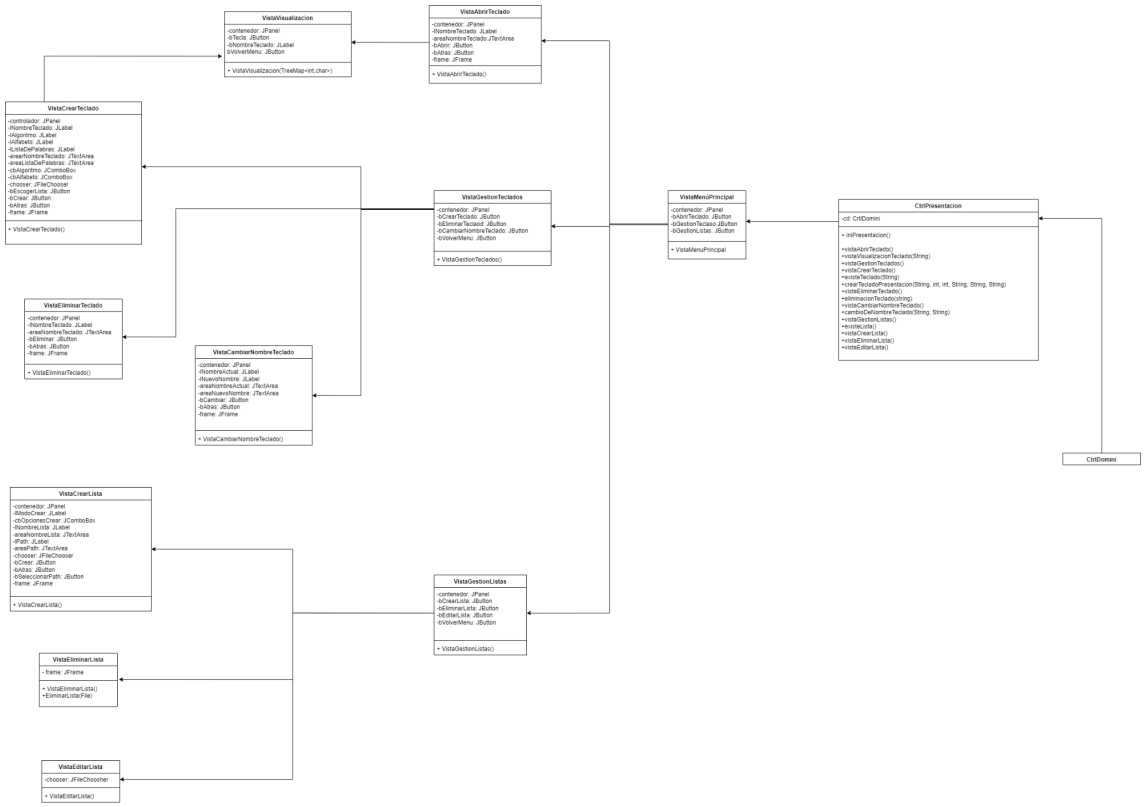
```java
/**
* Agrega un nuevo teclado con el nombre y la distribución
especificados
* @param nom_teclado Nombre del nuevo teclado
* @param tm Distribución de teclas representada como un TreeMap
* @throws tecladoExistente excepción lanzada si el teclado ya
existe
*/
public void addTeclado(String nom_teclado, TreeMap<Integer,
Character> tm) throws tecladoExistente{


/**
* Elimina un teclado existente por su nombre
* @param nom_Teclado nombre del teclado a eliminar
*/
public void removeTeclado(String nom_Teclado) {}
```

También proporciona métodos para cambiar el nombre de un teclado existente, lanzando excepción si ya existe un teclado con el nuevo nombre introducido y para imprimir en consola la distribución de las teclas de un determinado teclado, facilitando la visualización de la disposición de teclas del mismo.

Capa de Presentación

Diagrama



**En el SVG de la misma carpeta se ve en alta definición*

Descripción

VistaMenuPrincipal

Esta vista se encarga principalmente de ser la primera pantalla de la interfaz, y como dice su propio nombre es el menú principal de nuestra API. Esta interfaz consiste básicamente en un título de la ventana que se llamará “Menú Principal” y dispondremos de tres botones, donde cada uno se encargará de llevarnos a otra vista que tendrá su respectiva función, un botón no llevará a una vista para abrir un teclado, otro botón nos llevará a una vista donde gestionaremos nuestros teclados y el último nos llevará a una vista que gestionará las listas de palabras del sistema. En caso de querer cerrar la aplicación habrá en la esquina superior derecha el típico símbolo de “x” con la que se cerrará la interfaz, este aspecto estará presente en las diversas vistas que habrá en este proyecto por lo que todas se cerrarán de la misma forma y no lo repetiré para las siguientes que explicaré.

- Atributos:
  - **JPanel contenedor:** Panel que almacenará todos los elementos de la ventana.
  - **JButton bAbrirTeclado:** Botón para ir a la ventana donde seleccionaremos el nombre del teclado que queremos abrir.
  - **JButton bGestionTeclados:** Botón para ir a la ventana donde podremos gestionar nuestros teclados y donde profundizaremos más en los siguientes puntos.
  - **JButton bGestionListas:** Botón para ir a la ventana donde podremos gestionar nuestras listas de palabras que servirán de input para crear el teclado, y donde también se profundizará más de esta en los siguientes puntos.
- Métodos:
  - **VistaMenuPrincipal():** Constructora de la ventana menú principal.

VistaAbrirTeclado

Esta ventana se encargará en su totalidad de hacer la transacción de escoger el teclado que queremos y pasar a la siguiente ventana donde se visualizará el teclado escogido en esta. Esta ventana tendrá el nombre como título de “Escoger Teclado”. Principalmente los componentes que habrá son un texto refiriéndose al campo Nombre Teclado y un espacio que permite swing para escribir el nombre (JTextArea), además dispondremos de dos botones que uno será el de abrir una vez introducido el nombre del teclado y el otro botón hará la acción de volver al menú principal en caso de que cambiemos de idea y no queramos abrir un teclado sino otra acción. Nos encargaremos de las excepciones que pude haber que son la de que si no especificamos un nombre y le damos abrir lance un mensaje de “error: falta especificar el nombre” y otra excepción en caso de poner un nombre de un teclado que no exista en nuestro sistema lanzando un mensaje de “error: teclado inexistente”.

- Atributos:
  - **JPanel contenedor:** Panel que almacenará todos los elementos de la ventana.
  - **JLabel lNombreTeclado:** Etiqueta que indicará que el campo a rellenar tiene que ir un nombre de teclado.
  - **JTextArea areaNombreTeclado:** Área sobre la cual podremos escribir el nombre del teclado que queremos abrir para visualizar.
  - **JButton bAbrir:** Botón con el cual mediante se pasará a la ventana donde visualizaremos el teclado con el nombre especificado.
  - **JButton bAtras:** Botón con el cual podremos volver hacia el Menú Principal en caso de que queramos cancelar el proceso de visualizar un teclado.
  - **JFrame frame:** Frame que servirá para imprimir los mensajes de error que lanzaremos en caso de ejecutar una excepción.

- Métodos:
    - **VistaAbrirTeclado():** Constructora de la ventana de selección Escoger Teclado.

VistaVisualizacionTeclado

Vista fundamental que se encargará de la visualización del teclado por la interfaz, mostrará el objetivo principal de nuestro proyecto que es la creación de un teclado. Todavía no tenemos claro cómo será la implementación final de esta vista, pero hasta el momento nuestra estrategia más sólida será la de crear tantos botones como teclas tenga nuestra teclado, y asignar a cada botón el carácter correspondiente asignada por el algoritmo utilizado, siempre siguiendo la distribución de número de filas y columnas especificada al generarlo. El título de esta ventana será "Visualización Teclado" y además de los botones correspondientes al número de teclas, tendremos un texto que dirá "Este teclado es: <NombreDelTeclado>" un botón para volver al menú principal.

- Atributos:
    - **JPanel contenedor:** Panel que almacenará todos los elementos de la ventana.
    - **JButton bTecla:** Botón que corresponderá a la tecla correspondiente del teclado con su respectivo carácter, no serán botones funcionales que harán una acción al presionarlos, será solo visual.
    - **JLabel bNombreTeclado:** Etiqueta que indicara que el teclado que se visualizará por pantalla es el que tiene como nombre.
    - **JButton bVolverMenu:** Botón que se encargará de llevarnos a la ventana Menú Principal.
- Métodos:
    - **VistaVisualizaciónTeclado(TreeMap<Integer,Character> teclado):** Constructora de la ventana que nos mostrará el teclado.

VistaGestionTeclados

La siguiente vista representará un submenú con todas las opciones que existen en nuestro sistema para poder gestionar todos los teclados existentes. Este submenú principalmente nos ofrece la posibilidad de ejecutar 4 acciones, que son las de crear un nuevo teclado, la de eliminar un teclado existente de nuestro sistema, la de cambiarle el nombre a uno de nuestros teclados y la última función es la de volver al menú principal. Esta ventana estará compuesta por estas 4 acciones donde cada una tendrá su botón que la ejecutará y nos llevará a sus ventanas respectivas donde podremos llevar la acción a cabo. Esta ventana tendrá el título de "Gestión de Teclados" y como he mencionado antes estará compuesta de 4 botones y nada más.

- Atributos:
    - **JPanel contenedor:** Panel que almacenará todos los elementos de la ventana.

- **JButton bCrearTeclado:** Botón que se encargará de llevarnos a la ventana de creación del teclado donde especificaremos los atributos para su creación, se profundizará en los siguientes puntos.
- **JButton bEliminarTeclado:** Botón que se encargará de llevarnos a la ventana para eliminar un teclado.
- **JButton bCambiarNombreTeclado:** Botón que se encargará de llevarnos a la ventana que nos permitirá cambiar el nombre a un teclado.
- **JButton bVolverMenu:** Botón que se encargará de llevarnos a la ventana Menú Principal.

- Métodos:
  - **VistaGestionTeclados():** Constructora de la ventana de gestión de los teclados del sistema.

VistaCrearTeclado

Esta ventana es la encargada de la selección de los atributos que utilizaremos para la creación de nuestro teclado. Esta es una ventana de las más importantes ya que en esta decidimos mediante interfaz como queremos el teclado. En esta vista los atributos que podremos especificar será los de nombre del teclado a crear, algoritmo que se utilizará para su creación, el alfabeto que queremos representar en teclas y por último lo que decidirá la distribución de los caracteres del alfabeto en las respectivas teclas junto al algoritmo, la lista de palabras.

Esta ventana tendrá de título “Crear Teclado” y estará compuesta por diversos componentes, primero estarán las etiquetas que servirán para representar los campos a rellenar, luego estarán los propios campos a rellenar que serán de tres tipos, uno será un espacio en el que escribir (nombre) , otro una lista desplegable con diferentes opciones a elegir (algoritmo, alfabeto) y la última un buscador en nuestro directorio mediante el cual podremos dar con la lista que estará en formato .txt, esta opción también permite poner el nombre de la lista como tal escrita por teclado. Además de todo estos estarán presentes tres botones que serán los de dar a crear el teclado cuando estén todos los campos rellenados, el de volver para atrás y que te lleve a la gestión de teclados y por último el botón que te dirige al buscador de tu directorio para que puedas escoger el .txt que es la lista de palabras.

En esta ventana también dispondremos de varias excepciones que pueden saltar en diversos casos, habrá excepción por cada campo que no se rellene y se le dé al botón de crear estando vacío, otra que habrá es si introducimos un .txt que no sea válido y que no esté en el formato que permite la creación del teclado, otra excepción que habrá será la de crear un teclado con un nombre ya existente en el sistema, todas estas excepciones especificadas serán mostradas a través de una mini pantalla que lanzará un mensaje de “Error: <Mensaje del porque el error>”.

Las opciones de algoritmos que ofrecemos para la generación del teclado son el algoritmo QAP y el algoritmo Genético.

Para alfabetos las opciones disponibles que habrá para escoger serán las siguientes: Arabic, Cylliric, Greek, Hebrew, Latin, Latin_CAT, Latin_ESP y Thai (Aún no podemos confirmar la presencia de todos ellos).

- Atributos:
    - **JPanel contenedor:** Panel que almacenara todos los elementos de la ventana.
    - **JLabel lNombreTeclado:** Etiqueta que representara el campo donde hay que introducir el nombre del teclado.
    - **JLabel lAlgoritomo:** Etiqueta que representara el campo donde hay que introducir el algoritmo.
    - **JLabel lAlfabeto:** Etiqueta que representara el campo donde hay que introducir el alfabeto.
    - **JLabel lListaDePalabras:** Etiqueta que representara el campo donde hay que introducir la lista de palabras.
    - **JTextArea areaNombreTeclado:** Área donde podremos escribir el nombre del teclado a crear.
    - **JTextArea areaListaPalabras:** Área que obtendrá el valor del nombre del .txt que es la lista con la que se creará el teclado.
    - **JComboBox cbAlgoritmo:** Lista desplegable que nos permitirá elegir entre los dos algoritmos que hay.
    - **JComboBox cbAlfabeto:** Lista desplegable que nos permitirá elegir entre las diversas opciones de alfabetos del sistema.
    - **JFileChooser chooser:** Como dice el nombre herramienta que permite escoger archivos en nuestro caso un .txt que representará la lista de palabras.
    - **JButton bEscogerLista:** Botón que servirá para dar paso al FileChooser y poder elegir la lista de palabras que utilizaremos.
    - **JButton bCrear:** Botón que nos permitirá crear el teclado con los atributos especificados y que nos llevará a la ventana de visualización del nuevo teclado.
    - **JButton bAtras:** Botón que nos mandará de vuelta a la ventana de gestión de teclados.
    - **JFrame frame:** Frame que servirá para imprimir los mensajes de error que lanzaremos en caso de ejecutar una excepción.
- Métodos:
    - **VistaCrearTeclado():** Constructora de la ventana que nos permitirá crear el teclado.

VistaEliminarTeclado

Ahora trataremos la ventana que consistirá en eliminar un teclado. Esta ventana es muy simple ya que tendrá muy pocos componentes. Principalmente el proceso para eliminar un teclado será el de introducir el nombre del cual queremos eliminar del sistema y darle al botón eliminar. El título de esta ventana será "Eliminar Teclado" y los componentes que la formarán serán una etiqueta que especificará el campo donde introducir el teclado a eliminar y el campo que señala la etiqueta donde escribiremos el nombre de este, además una vez escrito para eliminarlo habrá un botón llamado eliminar que al pulsarlo lo borrará del sistema, por último habrá un botón para retornar al submenú de gestión de teclados.

También programaremos una excepción en dos casos, la primera es la de darle al botón de eliminar sin haber introducido ningún nombre donde saldrá una ventana que imprima mensaje por una mini ventana que diga "Error: No se ha especificado ningún nombre" y la segunda es la de el nombre del teclado introducido no existe en el sistema que pintará por la mini ventana "Error: El teclado especificado no existe en el sistema", y en caso de eliminación satisfactorio mostraremos por la mini ventana un mensaje de "Eliminación del teclado completada".

- Atributos:
  - **JPanel contenedor:** Panel que almacenará todos los elementos de la ventana.
  - **JLabel lNombreTeclado:** Etiqueta que indicara el campo donde introducir el nombre del teclado.
  - **JTextArea areaNombreTeclado:** Área donde escribiremos el teclado que queremos eliminar.
  - **JButton bEliminar:** Botón que se encargará de eliminar el teclado con nombre introducido de nuestro sistema.
  - **JButton bAtras:** Botón que se encargará de mandarnos de vuelta a la ventana de gestión de teclados.
  - **JFrame frame:** Frame que utilizaremos para lanzar una mini ventana imprimiendo los mensajes de error si salta una excepción y el mensaje de que se ha eliminado correctamente.
- Métodos:
  - **VistaEliminarTeclado():** Constructora de la ventana para eliminar un teclado

VistaCambiarNombreTeclado

Esta vista también no será de una complejidad muy elevada, ya que simplemente consistirá en seleccionar un teclado existente introduciendo su nombre, y después renombrarlo con el nuevo nombre que introduciremos. El título que adquirirá esta ventana será el de "Cambiar Nombre Teclado" y estará compuesta por pocos elementos como tal, habrá dos etiquetas que indicaran que nombre escribir en cada campo, la

primera indicará donde escribir el nombre actual del teclado y la otra donde escribir el nuevo nombre de este, después estarán los dos campos donde escribiremos el nombre antiguo y nuevo, y por último habrá dos botones, un botón se llamará cambiar que se pulsara para efectuar el renombramiento y el otro botón para volver al submenú de gestión de teclados. En esta ventana puede haber excepciones por lo que consecuentemente también habrá una mini ventana para imprimir cuando salten las excepciones.

Las excepciones que habrá en este caso pueden ser 4, una es si no introducimos nada en el campo de nombre actual y le damos a cambiar, otra es si no introducimos nada en nuevo nombre y le damos a cambiar, y las otras dos son en los casos que el nombre del teclado a renombrar no exista y que el nuevo nombre que le queremos poner ya esté existente para otro teclado. Los mensaje que imprimiremos serán para la excepción 1: “Error: No se ha especificado el teclado a renombrar”, para la 2: “Error: No se ha especificado el nuevo nombre”, la tercera será: “Error: El teclado a renombrar no existe en el sistema” y la última será: “Error: El nuevo nombre del teclado esta en uso”. Para finalizar si el cambio de nombre es satisfactorio se imprimirá por la mini ventana: “Cambio ejecutado”.

- Atributos:
  - **JPanel contenedor:** Panel que almacenará todos los elementos de la ventana.
  - **JLabel lNombreActual:** Etiqueta que indicará donde introducir el nombre actual del teclado.
  - **JLabel lNuevoNombre:** Etiqueta que indicará donde introducir el nuevo nombre del teclado.
  - **JTextArea areaNombreActual:** Campo en el que escribiremos el nombre actual del teclado.
  - **JTextArea areaNuevoNombre:** Campo en el que escribiremos el nuevo nombre del teclado.
  - **JButton bCambiar:** Botón que pulsaremos con el que cambiaremos el nombre del teclado.
  - **JButton bAtras:** Botón que nos permitirá volver al submenú de gestión de teclados.
  - **JFrame frame:** Frame que utilizaremos para imprimir los mensajes lanzados por las excepciones o por que se ha ejecutado bien el cambio.
- Métodos:
  - **VistaCambiarNombreTeclado():** Constructora de la ventana que se encargará de cambiar de nombre a un teclado.

VistaGestionListas

La siguiente vista representará un submenú con todas las opciones que existen en nuestro sistema para poder gestionar todas las listas de palabras existentes. Este submenú principalmente nos ofrece la posibilidad de ejecutar 4 acciones, que son las de

crear una nueva lista de palabras, la de eliminar una lista de palabras existente de nuestro sistema, la de editar una lista de palabras pudiéndose añadir más palabras o cambiar frecuencias o eliminar palabras o renombrar a la lista, y la última función es la de volver al menú principal. Esta ventana estará compuesta por estas 4 acciones donde cada una tendrá su botón que la ejecutará y nos llevará a sus ventanas respectivas donde podremos llevar la acción a cabo. Esta ventana tendrá el título de "Gestión de Listas" y como he mencionado antes estará compuesta de 4 botones y nada más.

- Atributos:
  - **JPanel contenedor:** Panel que almacenará todos los elementos de la ventana.
  - **JButton bCrearLista:** Botón que nos llevará a la ventana donde crearemos una lista de palabras y en los siguientes puntos profundizaremos más sobre este aspecto.
  - **JButton bEliminarLista:** Botón que nos llevará a la ventana donde eliminaremos una lista de palabras.
  - **JButton bEditarLista:** Botón que nos llevará a la ventana de edición de listas.
  - **JButton bVolverMenu:** Botón que se encargará de llevarnos a la ventana Menú Principal.
- Métodos:
  - **VistaGestionListas():** Constructora de la ventana de gestión de las listas del sistema.

VistaCrearLista

Esta es de las interfaces más complicadas de diseñar, y por eso la implementación pensada todavía no está del todo definidas y puede que no se la implementación final que haremos, pero de momento tenemos pensado hacer la creaciones de listas de dos formas, la primera es introducir a mano las palabras y frecuencias en un campo y guardarlas con extensión .txt, y la otra opción es hacer un copy paste de un texto y que nuestro software lo convierta al final en una lista de palabras con extensión .txt, como he comentado es un punto que no está del todo definido y genera debate dentro del grupo, así que volvemos a decir que podría cambiar bastante como es la gestión de los nuevos inputs que es lo mismo que crear nuevas listas para que sean procesadas por el algoritmo.

Nuestra idea es tener una etiqueta que indique la opción de cómo se genera la lista, mediante introducción a mano o copy paste de un texto, donde se podrá elegir entre una de estas dos opciones será una lista desplegable, después existirá otra etiqueta que indicara el campo donde introduciremos el nombre de la lista final, este campo indicado es donde podremos el nombre, también indicaremos el path donde guardar la lista, y habrá tres botones que serán los de crear cuando estén todos los campos rellenados, el botón que permitirá selecciona el path del nuevo archivo y un botón para volver al submenú de gestión de listas. El título de esta ventana será "Crear Lista", y puede haber

excepciones en caso de que no se cumplan requisitos como que haya campos vacíos al pulsar el botón de crear o que creemos una lista con un nombre ya existente.

- Atributos:
  - **JPanel contenedor:** Panel que almacenará todos los elementos de la ventana.
  - **JLabel lModoCrear:** Etiqueta que te indicará de qué forma crear una lista de palabras.
  - **JComboBox cbOpcionesCrear:** Una lista desplegable con las diferentes opciones para crear una lista.
  - **JLabel lNombreLista:** Etiqueta que te indicará el campo donde introducir el nombre de la lista.
  - **JTextArea areNombreLista:** Área donde escribiremos el nombre de la lista de palabras.
  - **JLabel lPath:** Etiqueta que indicará dónde introducir el Path donde se guardará la lista de palabras.
  - **JTextArea areaPath:** Espacio donde escribiremos el Path donde se guardará o se escribirá automáticamente al selecciona el Path con el JFileChooser.
  - **JFileChooser chooser:** Ventana de selección del directorio donde se quiere crear la lista
  - **JButton bCrear:** Botón con el que crearemos la lista.
  - **JButton bAtras:** Botón con el que volveremos a la ventana de submenú de gestión de listas.
  - **JButton bSeleccionarPath:** Botón con el que abriremos un explorador de archivos y podremos seleccionar el directorio donde guarda el documento.
  - **JFrame frame:** Mini ventana que mostraremos en caso de que salte una excepción o si la creación del archivo es satisfactoria.
- Métodos:
  - **VistaCrearLista():** Constructora de la ventana que nos permitirá crear la lista.

VistaEliminarLista

Esta vista solo se encargará de explorar nuestro explorador de documentos y donde podremos seleccionar el fichero .txt que representará una lista de palabras. Para eliminarlo hay que seleccionar el fichero y darle a borrar, de otra manera podemos cancelar y volveremos al submenú de gestión de listas.

- Atributos:
  - **JFrame frame:** Ventana que indicará si se ha borrado o no el fichero.
- Métodos:
  - **VistaEliminarLista():** Constructora de la ventana para eliminar un fichero

- **EliminarLista(File lista):** Borra del sistema el fichero lista y aparece un mensaje si se ha borrado correctamente.

VistaEditarLista

Esta ventana se encargará de editar una lista que seleccionaremos en nuestro buscador de ficheros, y se abrirá y la podremos editar a nuestro gusto. La idea es que cuando le demos al botón de Editar lista se abra el fichero .txt y se pueda editar a nuestro gusto, pudiendo quitar y añadir palabras, modificar frecuencias y renombrar el archivo .txt.

- Atributos:
  - **JFileChooser chooser:** Ventana de selección del archivo que se quiere editar.
- Métodos:
  - **VistaEditaLista():** Constructora de la ventana donde cargaremos el archivo

Explicación del controlador de Presentación

El controlador de presentación tiene una función clave en el proyecto ya que es el que hará de intermediario por así decirlo entre el controlador de dominio y las vistas que forman nuestra interfaz y de la comunicación entre las diversas vistas.

- Atributos:
  - **CtrlDomini cd:** Instancia del controlador de dominio.
- Métodos:
  - **iniPresentacion():** Muestra por pantalla la ventana de menú principal.
  - **vistaAbrirTeclado():** Muestra por pantalla la ventana de Escoger Teclado relacionado con la vistaAbrirTeclado.
  - **vistaVisualizacionTeclado(String):** Muestra por pantalla el teclado final con nombre String en la ventana “Visualización Teclado”.
  - **vistaGestionTeclados():** Muestra por pantalla la ventana de “Gestión Teclados”.
  - **vistaCrearTeclado():** Muestra por pantalla la ventana de “Crear Teclado”.
  - **existeTeclado(String):** Se encarga de devolver un bool si existe o no el teclado con nombre string.
  - **crearTecladoPresentacion(String, int, int, String, String, String):** Función encargada de iniciar el proceso para la creación de un teclado con los parámetros especificados.
  - **vistaEliminarTeclado():** Muestra por pantalla la ventana de “Eliminar Teclado”.
  - **eliminacionTeclado(string):** Se encarga de iniciar el proceso de eliminación de nuestro teclado con nombre String.
  - **vistaCambiarNombreTeclado():** Muestra por pantalla la ventana de “Cambiar Nombre Teclado”.

- **cambioDeNombreTeclado(String, String):** Inicia el proceso de cambio de nombre del teclado en el sistema.
- **vistaGestionListas():** Muestra por pantalla la ventana de “Gestión Listas”.
- **existeLista():** Devuelve un bool si existe o no la lista en el sistema.
- **vistaCrearLista():** Muestra por pantalla la ventana de “Crear Lista”.
- **vistaEliminarLista():** Muestra por pantalla la ventana para seleccionar un archivo que representa una lista para eliminar.
- **vistaEditarLista():** Muestra por pantalla la ventana que muestra los ficheros disponibles para editar.

Estructuras de Datos y Algoritmos

En esta sección del documento vamos a comentar y justificar el porqué hemos usado cada estructura de datos en cada clase del modelo. También comentaremos y explicaremos cómo hemos desarrollado el primer algoritmo para solucionar el problema planteado.

Estructuras de Datos

Alfabeto

Un alfabeto siempre es un conjunto de caracteres, para ordenarlo no había muchas maneras, por lo tanto escogimos hacer un vector de chars ya que el coste de recorrerlo no era muy grande, $\Theta(n)$.

```
public static final char[] Hebrew = {
        'י' ,'ט' ,'ח' ,'ז' ,'ו' ,'ה' ,'ד' ,'ג' ,'ב' ,'א'
,'ר' ,'ק' ,'צ' ,'פ' ,'ע' ,'ס' ,'נ' ,'מ' ,'ל' ,'כ'
ת' ,'ש'         ',
};


public static final char[] Latin = {
            'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j',
            'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't',
            'u', 'v', 'w', 'x', 'y', 'z',
};
```

En este fragmento de código es un ejemplo de como hemos representado el alfabeto hebreo y el alfabeto latín. Como se puede observar lo hemos hecho mediante un vector de caracteres.

Listas de Frecuencias

Para la clase Lista_Palabras que representa una lista de frecuencias teníamos varias opciones para representarla, un mapa convencional, o un HashMap entre otras opciones. Resumidamente explicamos cual usaremos y por qué.

- **Sorted Map**

  Un sorted map o un mapa ordenado, como indica su nombre es un mapa en el cual las claves están ordenadas por su valor. En nuestro caso las palabras estarían ordenadas según su frecuencia. El Sorted Map también se conoce en Java como Tree Map y la clase se implementa mediante un árbol binario.

  Por lo tanto, los costes de búsqueda y recorrido serán de $\Theta(\log(n))$.

- **HashMap**

  El HashMap sería una representación de palabra i su frecuencia por tanto el mapa será de string – entero. Para recorrer el mapa en busca de una palabra el

tiempo de acceso medio seria de $\Theta(1)$ y en el pero de los caso sería un coste lineal $\Theta(n)$. Como no necesitamos que nuestro mapa este ordenado ya que para el algoritmo implementado es irrelevante no nos hace falta.

Nos hemos decantado por el HashMap ya que es el que tiene un menor coste de búsqueda y de recorrido de toda la estructura de datos.

Teclados

La distribución del teclado que crea un algoritmo, en esta primera entrega un TreeMap ya que en el caso del teclado nos interesa mucho que se ordenen las letras de una cierta manera mediante el algoritmo. Por lo tanto el coste asintótico que tiene recorrer un teclado es $\Theta(\log(n))$.

QAP

Hemos utilizado dos matrices, una de enteros y otra de doubles. En la de enteros almacenamos la matriz de flow y en la de doubles la distancia heurística. Aunque buscamos información respecto a la ley de Fitts y cómo implementarla para el cálculo de esta matriz de distancias, finalmente decidimos dejarlo para futuras entregas, ya que, en esta primera solución, conseguimos resultados correctos.

Utilizamos matrices para almacenar la información de frecuencias y distancias ya que son mucho más eficientes a la hora de acceder a su información cuándo estamos calculando la cota.

Nodo

Cómo se ha mencionado anteriormente, nodo tiene dos TreeSet y un TreeMap almacenados en su clase.

TreeSet<Integer> posiciones_libres: La idea principal de este Set es ir recorriendo las posiciones libres que quedan por emplazar en orden.

TreeSet<Character> letras_libres: TreeSet ordenado de las letras que faltan por ubicar.

TreeMap<Integer, Character> posiciones_ocupadas: A la hora de imprimir el TreeMap, queremos tenerlo ordenado para que escriba las letras en el orden que queremos. Además, durante la ejecución del algoritmo, el tiempo de recorrido del TreeMap es de log n y en todo momento accedemos a valores, en caso de implementar un vector (estructura que se valoró al inicio), pueden existir posiciones que todavía no se ha añadido y por tanto recorrer un vector semi_vacio.

Nodo, también hace el cálculo de su cota. Primero se calcula el primer término y posteriormente C1 y C2 y se envía su suma C3 = C1+C2 a Hungarian que nos devuelve el coste. Una vez calculado podemos elegir si desechar o no el nodo.

Population

Para almacenar el conjunto de datos de los cromosomas, hemos utilizado una Priority Queue con un CromosomaComparator para ordenarla de menor a mayor.

El motivo principal del uso de esta estructura es para almacenar los datos de forma ordenada y eliminarlos con un .poll(). Sólo se recorre una vez por generación y es para seleccionar a los mejores individuos y para desechar a los peores, este coste es de O(n)

Genetic

Cada gen es una solución parcial del teclado, además de un miembro de la población. En esta versión de la implementación se ha optado por utilizar un char[ ] dónde la posición del vector indica implícitamente la posición del Character en el teclado.