EasyType

Proyecto de Programación
Curso 2023/24 Q1
Grupo 42.1

Rozhina Ahmadi
Jiahao Liu
Esther Lozano
Bruno Ruano

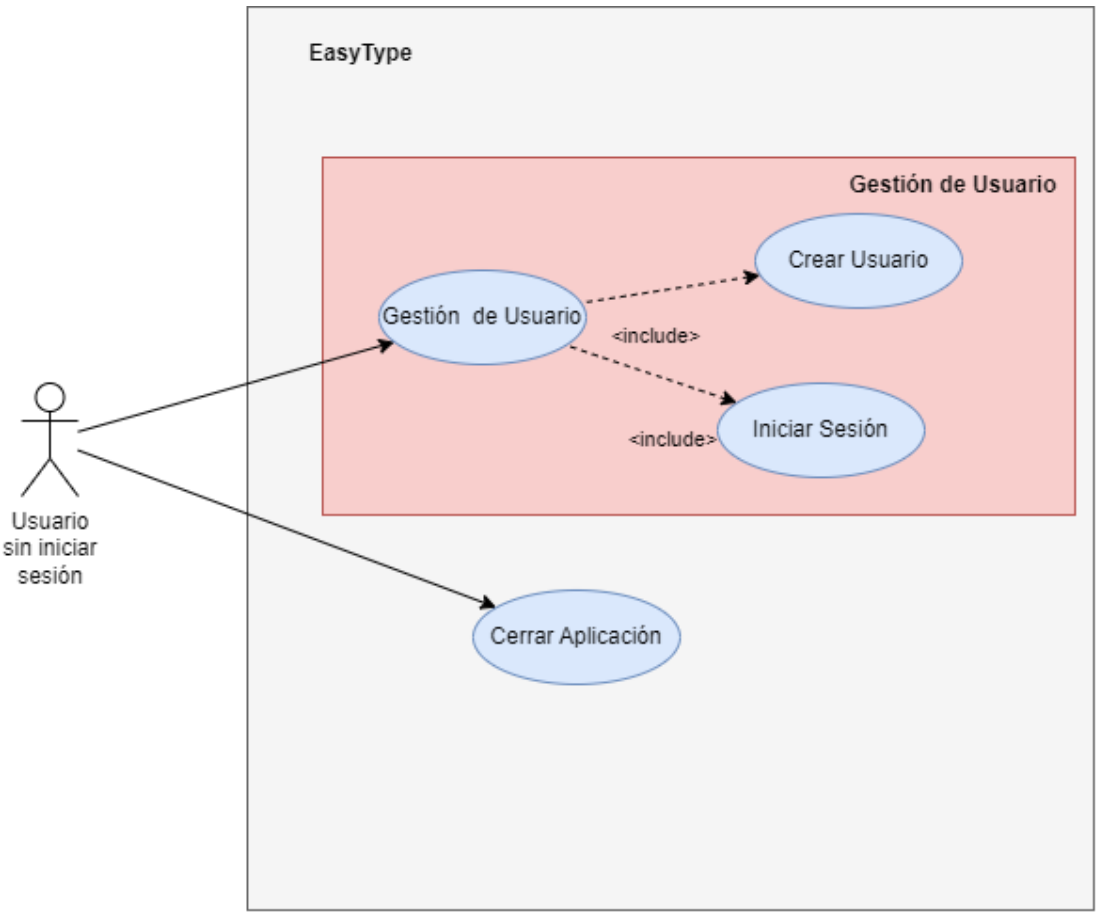
Entregable 3
23 de Diciembre 2023

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

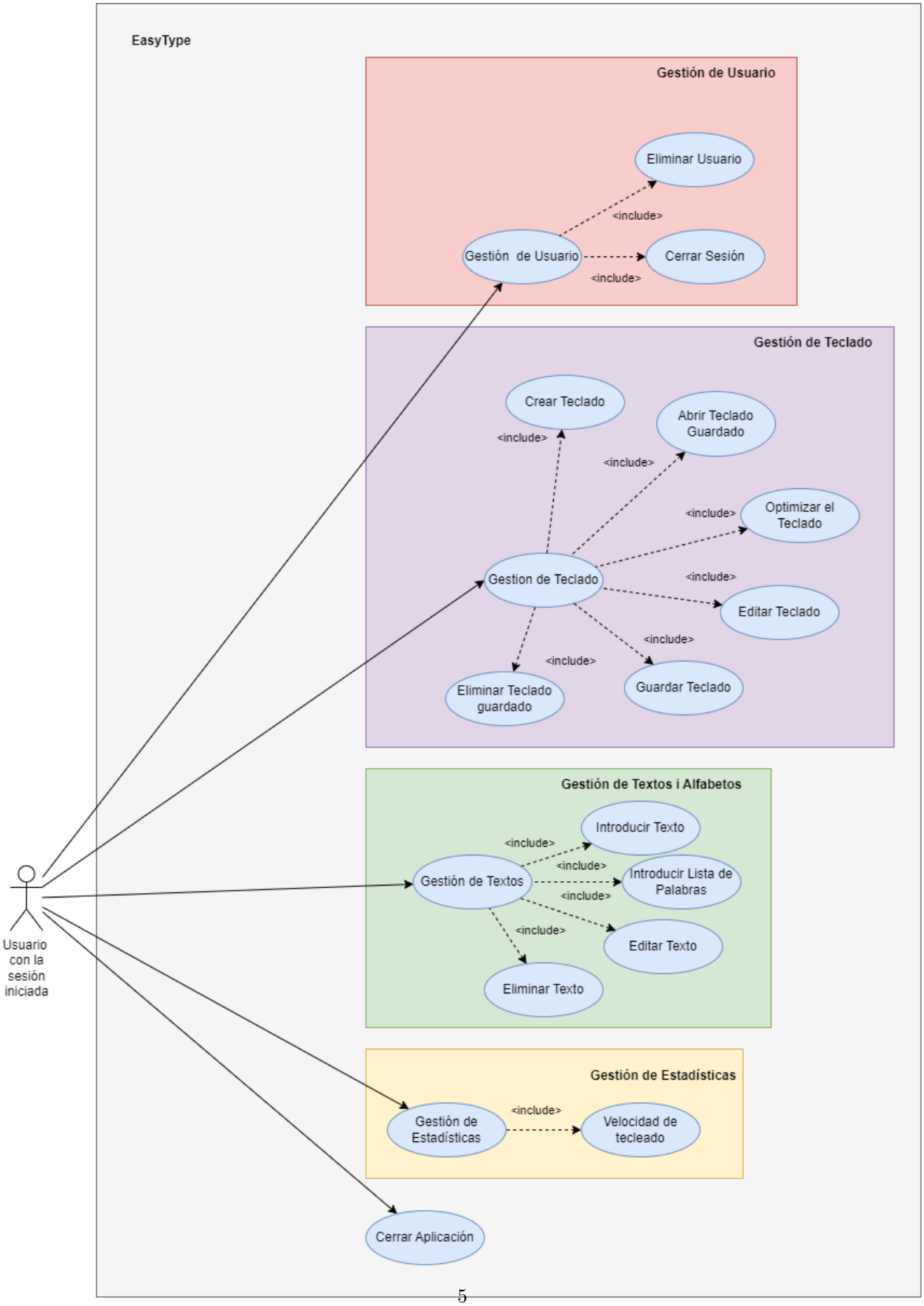
Índice

1 Diagrama de Casos de Uso

1.1 Usuarios sin iniciar sesión



1.2 Usuarios con sesión iniciada



2 Descripción de Casos de Uso

2.1 Gestión de usuario

Actor: Usuario
Comportamiento:

1. El usuario acaba de iniciar la aplicación y quiere, crear un usuario (caso de uso 2.1.1) o iniciar una sesión (caso de uso 2.1.2)
2. El usuario tiene la sesión iniciada y quiere, eliminar un usuario (caso de uso 2.1.3) o cerrar sesión (caso de uso 2.1.4)

Errores posibles y alternativas: -

2.1.1 Crear usuario

Actor: Usuario con la sesión no iniciada
Comportamiento:

1. El usuario introduce el nombre de usuario, la contraseña y le da a crear cuenta.
2. El sistema crea y guarda el usuario y la contraseña.

Errores posibles y alternativas:

2a. El usuario ya existe, el sistema informa de que el nombre de usuario ya está en uso.

2.1.2 Iniciar sesión (Login)

Actor: Usuario con la sesión no iniciada
Comportamiento:

1. El usuario introduce el nombre de usuario y la contraseña y le da a iniciar sesión.
2. El sistema valida los valores introducidos para dejar acceder al usuario.

Errores posibles y alternativas:

2a. Si el nombre de usuario no existe, el sistema informa del error.

2b. Si la contraseña es incorrecta, el sistema informa del error.

2.1.3 Eliminar usuario

Actor: Usuario con la sesión iniciada
Comportamiento:

1. El usuario con la sesión iniciada quiere borrar su cuenta.
2. El sistema pide al usuario confirmar la petición.
3. El usuario confirma o cancela la operación:
   - 3.1 El sistema borra la cuenta si este ha confirmado y el usuario pasa a no tener la sesión iniciada.
   - 3.2 El sistema cancela la operación.

Errores posibles y alternativas: -

2.1.4 Cerrar sesión

Actor: Usuario con la sesión iniciada
Comportamiento:

1. El usuario quiere cerrar la sesión.
2. El sistema cierra la sesión y el usuario pasa a no tener la sesión iniciada.

Errores posibles y alternativas: -

2.2 Gestión de teclados

Actor: Usuario con la sesión iniciada
Comportamiento:

1. El usuario quiere crear un teclado (caso de uso 2.2.1), abrir un teclado guardado (caso de uso 2.2.2), optimizar el teclado (caso de uso 2.2.4), editar el teclado (caso de uso 2.2.5), eliminar un teclado guardado (caso de uso 2.2.6) o guardar un teclado (caso de uso 2.2.6).

Errores posibles y alternativas: -

2.2.1 Crear teclado

Actor: Usuario con la sesión iniciada
Comportamiento:

1. El usuario quiere crear un nuevo teclado.
2. El sistema le muestra los nombres de los alfabetos guardados en su cuenta.
3. El usuario escoge un alfabeto de los listados.
4. El sistema asigna el alfabeto al teclado.
5. El sistema le pregunta cuántas filas y columnas quiere para el teclado.
6. El usuario introduce las cantidades.
7. El sistema le pregunta si quiere teclas pequeñas, medianas o grandes para el teclado.
8. El usuario contesta con una de las tres opciones.
9. El sistema acaba de crear el teclado y lo inicializa con letras en posiciones aleatorias en sus teclas.

Errores posibles y alternativas:

5a. El usuario no ha introducido valores correctos para crear el teclado, ya que con esos valores hay menos teclas que letras en el teclado.

2.2.2 Escoger teclado guardado

Actor: Usuario con la sesión iniciada
Comportamiento:

1. El usuario quiere escoger un teclado de los que tiene guardados.
2. El sistema le muestra la lista de teclados guardados.
3. El usuario escoge el que quiere.
4. El sistema carga el teclado guardado.

Errores posibles y alternativas:

2a. El sistema no tiene teclados guardados.

2.2.3 Optimizar el teclado

Actor: Usuario con la sesión iniciada
Comportamiento:

1. El usuario elige cargar una distribución para el teclado actual.
2. El sistema le muestra una lista con los nombres de los archivos guardados en su cuenta, y le pregunta con qué archivo quiere calcular la distribución del teclado.
3. El usuario escoge el archivo que quiere que se aplique.
4. El sistema le pregunta al usuario cual de los dos algoritmos quiere usar para implementar la distribución.
5. El usuario elige el algoritmo que prefiere.
6. El sistema ejecuta el algoritmo y muestra en el teclado la nueva distribución. Si el teclado estaba guardado, se crea una copia que no estará guardada.

Errores posibles y alternativas:

1a. El usuario no tiene ningún teclado abierto actualmente.

2a. El sistema no tiene textos para aplicar con el algoritmo.

2.2.4 Editar teclado

Actor: Usuario con la sesión iniciada
Comportamiento:

1. El usuario quiere modificar la distribución del teclado actual de forma manual.
2. El sistema le permite modificar el teclado seleccionando las dos teclas que quiere cambiar de posición.
3. El usuario indica las dos teclas que quiere intercambiar.
4. El sistema aplica las modificaciones.

Errores posibles y alternativas:

1a. El usuario no tiene ningún teclado abierto actualmente.

2.2.5 Eliminar teclado guardado

Actor: Usuario con la sesión iniciada
Comportamiento:

1. El usuario quiere borrar un teclado guardado.
2. El sistema le muestra los nombres de los teclados guardados, y le pregunta que teclado quiere borrar.
3. El usuario escoge el teclado a borrar.
4. El sistema borra el teclado y las estadísticas del teclado.

Errores posibles y alternativas:

2a. El sistema no tiene teclados guardados.

2.2.6 Guardar teclado

Actor: Usuario con la sesión iniciada
Comportamiento:

1. El usuario quiere guardar el teclado actual.
2. El sistema le pregunta el nombre con el que lo quiere guardar.
3. El usuario entra el nombre del teclado.
4. El sistema guarda el teclado.

Errores posibles y alternativas:

1a. El usuario no tiene ningún teclado abierto actualmente.

3a. El usuario ya tiene un teclado guardado con ese nombre.

2.3 Gestión de textos y alfabetos

Actor: Usuario con la sesión iniciada
Comportamiento:

1. El usuario quiere introducir un texto (caso de uso 2.3.1) o una lista de palabras (caso de uso 2.3.2), eliminar un texto (caso de uso 2.3.3) o editar un texto (caso de uso 2.3.4).

Errores posibles y alternativas: -

2.3.1 Introducir texto

Actor: Usuario con la sesión iniciada
Comportamiento:

1. El usuario quiere introducir un texto.
2. El sistema pregunta por un nombre para el archivo.
3. El sistema pregunta cómo quiere hacerlo:
   - 3.1 Introducirlo manualmente.
     - 3.1.1 El usuario introduce el texto.
   - 3.2 Mediante un archivo.
     - 3.2.1 El sistema le pide el fichero.
     - 3.2.2 El usuario le proporciona el fichero.
4. El sistema guarda el texto del fichero.
5. El sistema pide la asignación de un alfabeto para este texto.
6. El sistema pregunta cómo quiere hacerlo:
   - 6.1 Crearlo de forma automática.
     - 6.1.1 El usuario introduce el nombre de alfabeto.
   - 6.2 Escoger un alfabeto ya existente.
     - 6.2.1 El sistema le proporciona una lista de alfabetos compatibles con el texto.
     - 6.2.2 El usuario escoge el alfabeto
7. El sistema lo guarda.

Errores posibles y alternativas:

4a. Si el fichero está vacío, se informa del error.

7a. El sistema ya tiene ese nombre guardado.

2.3.2 Introducir lista de palabras

Actor: Usuario con la sesión iniciada
Comportamiento:

1. El usuario quiere introducir un lista de palabras.
2. El sistema pregunta por un nombre para el archivo.
3. El sistema pregunta cómo quiere hacerlo:
   - 3.1 Introducirlo manualmente.
     - 3.1.1 El usuario introduce la lista de palabras.
   - 3.2 Mediante un archivo.
     - 3.2.1 El sistema le pide el fichero.
     - 3.2.2 El usuario le proporciona el fichero.
4. El sistema guarda los datos introducidos.
5. El sistema pide la asignación de un alfabeto para esta lista de palabras.
6. El sistema pregunta cómo quiere hacerlo:
   - 6.1 Crearlo de forma automática.
     - 6.1.1 El usuario introduce el nombre de alfabeto.
   - 6.2 Escoger un alfabeto ya existente.
     - 6.2.1 El sistema le proporciona una lista de alfabetos compatibles con el texto.
     - 6.2.2 El usuario escoge el alfabeto
7. El sistema lo guarda.

Errores posibles y alternativas:

4a. Si el fichero está vacío, se informa del error.

7a. El sistema ya tiene ese nombre guardado.

2.3.3 Eliminar texto

Actor: Usuario con la sesión iniciada
Comportamiento:

1. El usuario quiere eliminar uno de los textos que tiene guardados.
2. El sistema le hace elegir cuál quiere eliminar.
3. El usuario indica qué texto quiere eliminar.
4. El sistema elimina el texto.

Errores posibles y alternativas:

2a. No hay textos guardados.

2.3.4 Editar texto

Actor: Usuario con la sesión iniciada
Comportamiento:

1. El usuario elige un texto de los que están guardados en el sistema.
2. El sistema le da permiso para editar el texto seleccionado.
3. El usuario edita el texto seleccionado, modificando su contenido.
4. El sistema comprueba si el nuevo texto es compatible con el alfabeto asignado.
5. El sistema pregunta un nombre para el nuevo alfabeto compatible con el texto en caso de que el antiguo deje de serlo.

Errores posibles y alternativas:

1a. Si no hay textos que se puedan editar, el sistema informará del error.

2.4 Gestión de Estadísticas

Actor: Usuario con la sesión iniciada
Comportamiento:

1. El usuario quiere probar la velocidad (caso de uso 2.4.1).

Errores posibles y alternativas: -

2.4.1 Velocidad de teclado

Actor: Usuario con la sesión iniciada
Comportamiento:

1. El sistema muestra una lista de textos y pide al usuario que escoja con cual quiere probar la velocidad del teclado abierto actualmente.
2. El usuario elige el texto con el que quiere probar el teclado.
3. El sistema calcula la velocidad de escritura del texto para el teclado abierto.

Errores posibles y alternativas:

1. No existe ningún texto.
2. No hay un teclado abierto con el que ver estadísticas.

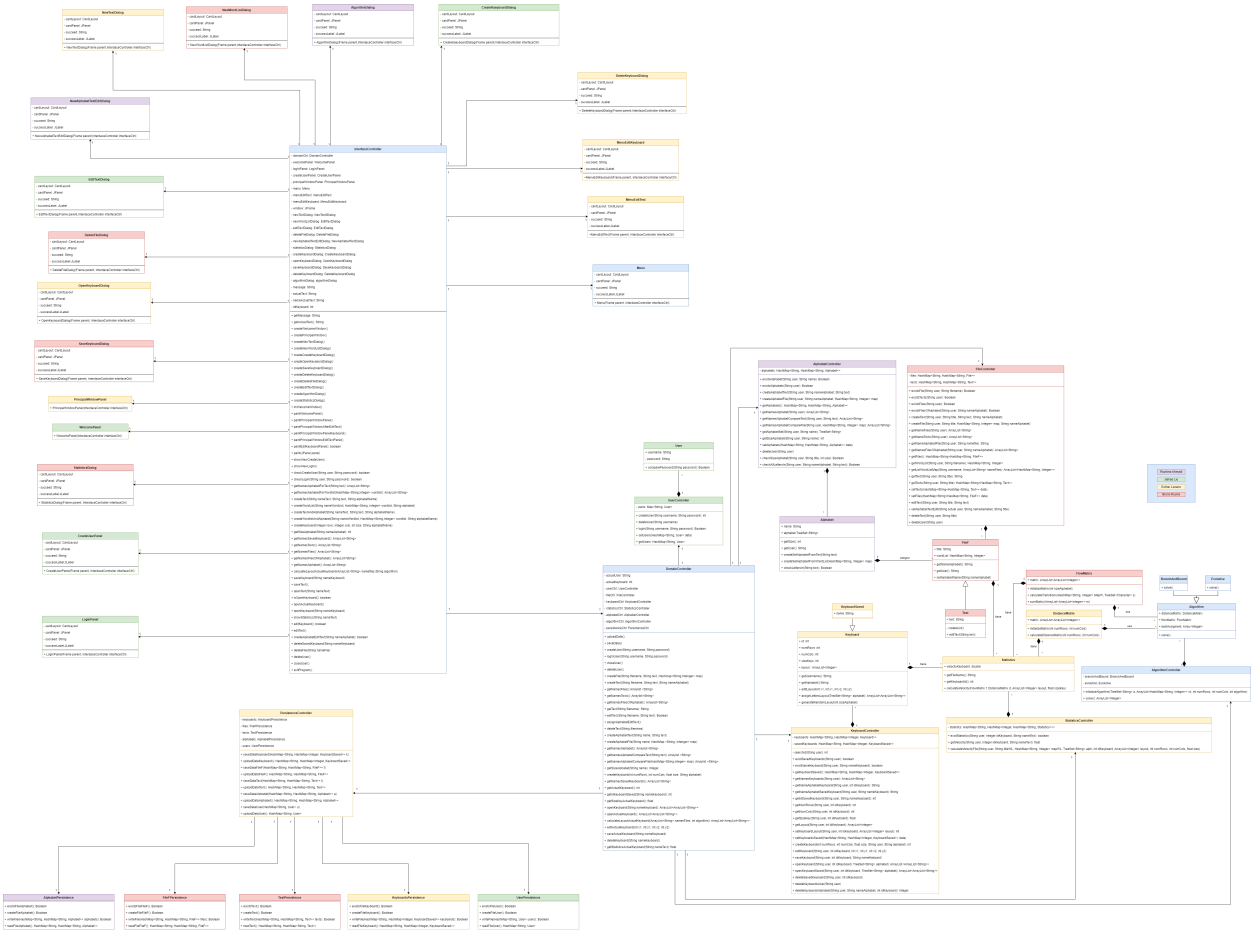
2.5 Cerrar aplicación

Actor: Usuario
Comportamiento: El sistema cierra la aplicación cuando lo indica el usuario.

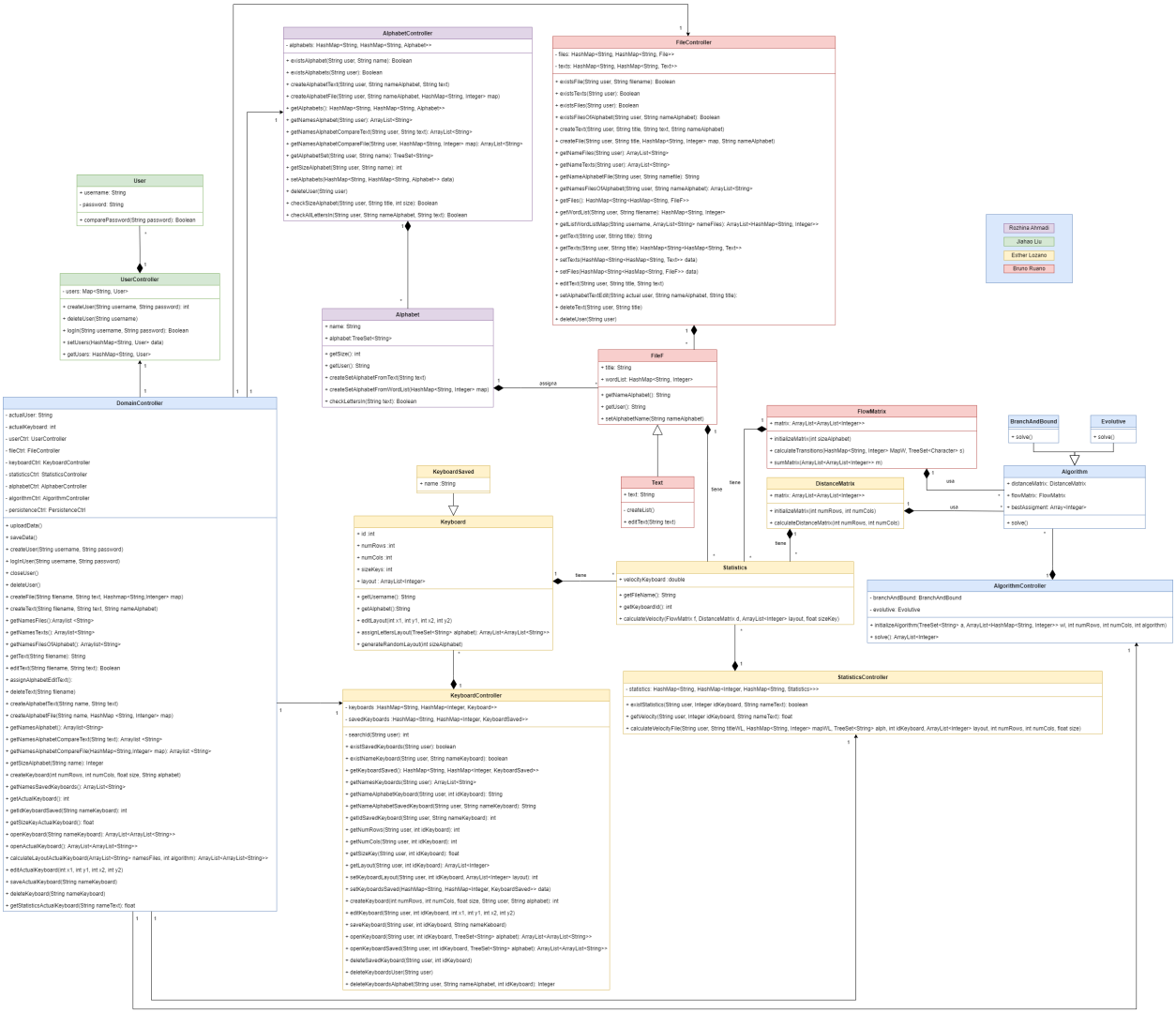
1. El usuario quiere salir de la aplicación.
2. El sistema lo cierra

Errores posibles y alternativas: -

3 Diagrama de Clases



4 Diagrama de Clases de la Capa de Dominio



5 Descripción de Clases de Dominio

5.1 Domain

5.1.1 Algorithm

La clase `Algorithm` es la clase base para los algoritmos que se encargan de asignar teclas en un teclado. Define la estructura y métodos comunes para los diferentes algoritmos de asignación.

5.1.2 Alphabet

La clase`Alphabet` es una clase que define un conjunto de caracteres que pueden ser utilizados en un texto. Esta clase se compone de un conjunto ordenado de cadenas de caracteres (strings), representando los caracteres disponibles en el alfabeto. Cada instancia de `Alphabet` tiene un nombre único y está asociada a un usuario específico.

5.1.3 BranchAndBound

La clase `BranchAndBound` implementa el algoritmo de Branch and Bound para la asignación óptima de teclas en un teclado. Es una subclase de `Algorithm` y adapta sus métodos para aplicar este algoritmo específico.

5.1.4 DistanceMatrix

La clase `DistanceMatrix` es una clase que representa una matriz de distancias, ofreciendo operaciones para calcular y manipular esta matriz. La matriz se calcula en base a las dimensiones del teclado, como el número de filas y columnas.

5.1.5 Evolutive

La clase `Evolutive` implementa un algoritmo evolutivo para la asignación óptima de teclas en un teclado. Deriva de la clase `Algorithm` y proporciona una implementación específica basada en técnicas evolutivas.

5.1.6 FileF

La clase `FileF` representa un archivo que contiene un título, un nombre de usuario asociado, una lista de palabras con sus respectivas frecuencias y un nombre de alfabeto.

5.1.7 FlowMatrix

La clase `FlowMatrix` gestiona una matriz de flujo, que representa las relaciones de uso entre las teclas de un teclado. Proporciona operaciones para manipular y entender estas relaciones.

5.1.8 Keyboard

La clase `Keyboard` esta destinada a representar un teclado, incluyendo su diseño, tamaño y otros atributos. El diseño se representa como una lista de enteros, indicando la disposición de las teclas.

5.1.9 KeyboardSaved

La clase `KeyboardSaved` es una subclase de `Keyboard` y representa un teclado que ha sido guardado con su diseño y otros atributos específicos.

5.1.10 Statistics

La clase `Statistics` se encarga de recolectar y procesar estadísticas relacionadas con la eficiencia de uso de un teclado. Incluye métodos para calcular la velocidad de tipeo, basándose en matrices de flujo y distancia.

5.1.11 Text

La clase `Text` extiende la clase `FileF` y representa un archivo de texto con título, contenido, nombre de usuario asociado y alfabeto.

5.1.12 User

La clase `User` modela a un usuario del sistema, incluyendo su nombre de usuario y contraseña.

5.2 DomainController

5.2.1 AlgorithmController

El `AlgorithmController` es un controlador encargado de la ejecución de algoritmos de asignación de teclas en teclados.

5.2.2 AlphabetController

Este controlador se ocupa de la creación, gestión y verificación de alfabetos asociados a usuarios. Organiza los alfabetos según el usuario y su nombre. Incluye métodos para crear alfabetos a partir de textos o listas de palabras (*wordlists*) y para verificar la correspondencia de letras en un texto con los alfabetos.

5.2.3 FileController

El `FileController` gestiona la creación, eliminación y manipulación de archivos y textos asociados a un usuario.

5.2.4 KeyboardController

El `KeyboardController` se encarga de la gestión de teclados y de teclados guardados.

5.2.5 StatisticsController

Este controlador se encarga de gestionar las estadísticas relacionadas con el uso y eficiencia de los teclados.

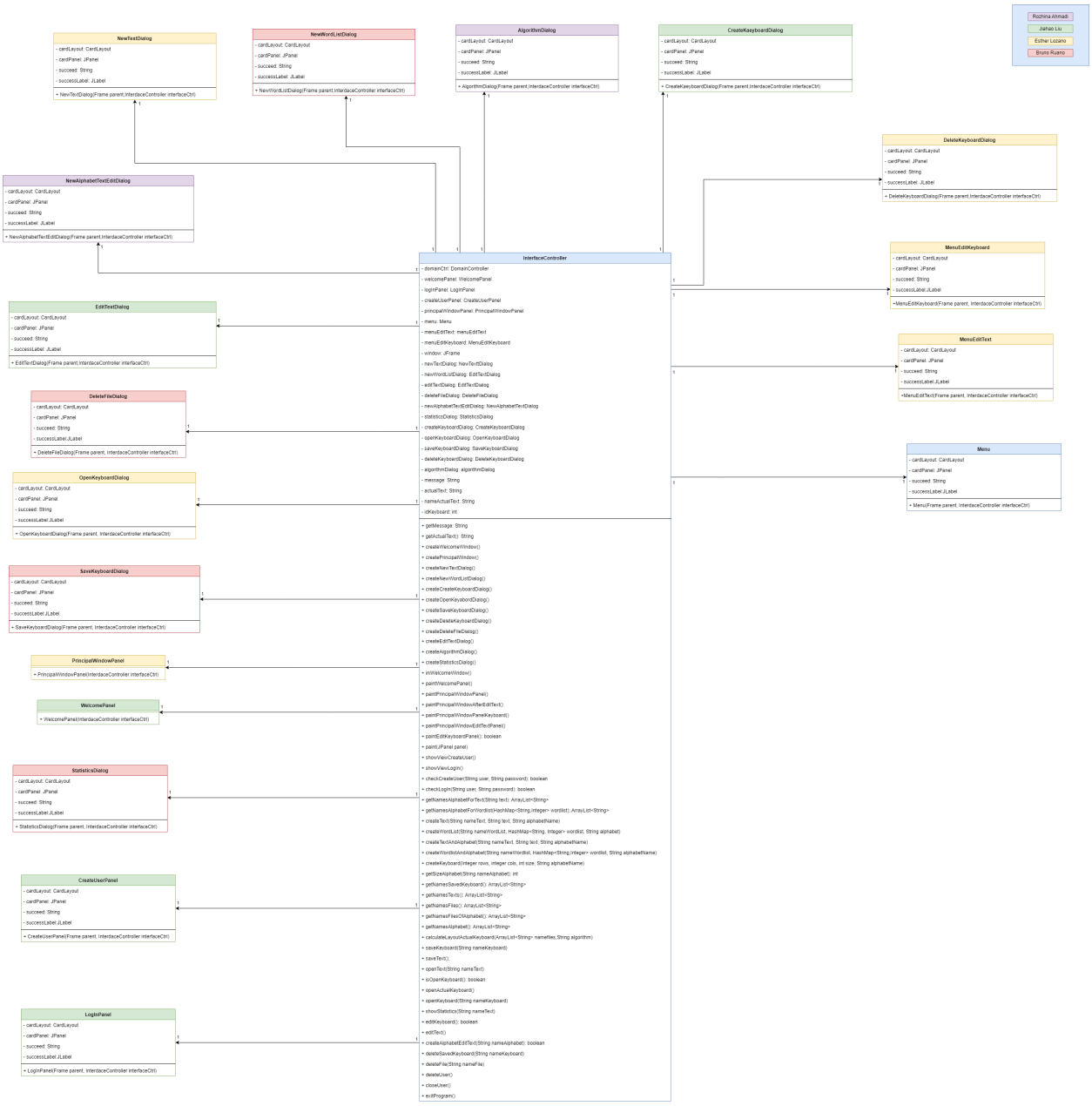
5.2.6 UserController

El `UserController` es un controlador para la gestión de la creación, eliminación, inicio de sesión y cierre de sesión de usuarios.

5.2.7 DomainController

La clase `DomainController` gestiona la interacción de operaciones en la aplicación. Coordina la gestión de usuarios, archivos, alfabetos, teclados y estadísticas. Incluye métodos para cargar y guardar datos, gestionar usuarios, manipular archivos de texto, trabajar con alfabetos, gestionar teclados y calcular estadísticas utilizando otros controladores. En general, actúa como la unidad de control principal para coordinar acciones dentro de la aplicación.

6 Diagrama de Clases de la Capa de Presentación



7 Descripción de Clases de Presentación

7.1 InterfaceController

El `InterfaceController` gestiona los datos, estructuras y vistas para la capa de presentación. Funciona como un singleton, garantizando que solo exista una instancia en todo el programa que devuelve siempre la misma instancia. Además, tiene una instancia del `DomainController` para conectar y pasar funcionalidades y datos del programa.

7.2 AlgorithmDialog

`AlgorithmDialog` permite elegir un algoritmo para optimizar el teclado. Ofrece opciones entre `Branch and Bound` y `Evolutive`. Tras seleccionar y aceptar, el teclado optimizado se muestra en el `Menu`.

7.3 CreateKeyboardDialog

`CreateKeyboardDialog` muestra el proceso de creación de un teclado. Incluye selección de texto, configuración de filas, columnas y tamaño de teclas. Al aceptar, se muestra el teclado creado en el `Menu`.

7.4 DeleteFileDialog

La vista `DeleteFileDialog` muestra una lista de archivos guardados en el sistema. Permite seleccionar un archivo para eliminarlo. Una vez seleccionado y aceptada la elección, el archivo se borrará y la vista desaparecerá.

7.5 DeleteKeyboardDialog

`DeleteKeyboardDialog` permite eliminar un teclado del sistema. Muestra una lista de teclados guardados para seleccionar y borrar.

7.6 EditTextDialog

`EditTextDialog` muestra una lista de textos guardados en el sistema. El usuario selecciona el texto que desea modificar y, al aceptar, la vista `Menu` cambia a `EditTextMenu`, donde puede modificar el texto.

7.7 NewAlphabetTextEditDialog

`NewAlphabetTextEditDialog` se activa al finalizar la edición de un texto que contiene caracteres nuevos. Solicita al usuario nombrar el nuevo alfabeto y, tras aceptar, se guarda el alfabeto en el sistema.

7.8 NewTextDialog

`NewTextDialog` guía al usuario en el proceso de introducción de un texto nuevo. Primero, se elige el título del texto y la fuente del contenido (escrito en la vista `Menu` o un archivo del sistema). Luego, se decide si crear un nuevo alfabeto a partir del texto o seleccionar uno existente. Finalmente, al aceptar, el texto se guarda en el sistema.

7.9 NewWordListDialog

En `NewWordListDialog`, el usuario introduce una nueva wordlist. Las opciones incluyen escribir la wordlist manualmente o seleccionar un archivo del sistema. En la segunda vista, se elige entre crear un nuevo alfabeto a partir de la wordlist o seleccionar uno existente. Al aceptar, la wordlist se guarda en el sistema.

7.10 OpenKeyboardDialog

`OpenKeyboardDialog` facilita la apertura de un teclado guardado. Muestra una lista de teclados y, al seleccionar y aceptar, abre el teclado en la vista `Menu`.

7.11 SaveKeyboardDialog

`SaveKeyboardDialog` maneja el proceso de guardar un teclado. El usuario escribe el nombre del teclado y al aceptar, se guarda el teclado actualmente mostrado en `Menu`.

7.12 StatisticsDialog

`StatisticsDialog` muestra la puntuación de un teclado, calculando la distancia entre teclas y el esfuerzo para escribir un texto. Una puntuación alta indica un peor rendimiento. El usuario selecciona un texto para evaluar y el sistema muestra la puntuación.

7.13 CreateUserPanel

`CreateUserPanel` maneja la creación de un usuario. Tras introducir y validar usuario y contraseña, el sistema lleva al usuario a la pantalla principal.

7.14 LogInPanel

`LogInPanel` gestiona el acceso a cuentas de usuario existentes. Tras introducir usuario y contraseña y validarlos, el sistema permite el acceso a la cuenta y redirige a la pantalla principal.

7.15 PrincipalWindowPanel

`PrincipalWindowPanel` es la pantalla principal del programa. Muestra la estructura `Menu` y un área de texto central para la introducción y edición de teclados. La parte inferior muestra el teclado abierto.

7.16 WelcomePanel

`WelcomePanel` es la primera vista al iniciar el programa. Ofrece opciones para crear un nuevo usuario o acceder a una cuenta existente.

7.17 Menu

`Menu` es una barra que incorpora funcionalidades para `File`, `Keyboard` y `User`. Se encuentra en las vistas principales del programa.

7.18 MenuEditText

`MenuEditText` adapta la pantalla principal para el modo `EditText`. Permite modificar el texto elegido y guardar o deshacer cambios realizados.

7.19 MenuEditKeyboard

`MenuEditKeyboard` adapta la pantalla principal para el modo `EditKeyboard`. Permite modificar la posición de las teclas y guardar o deshacer cambios realizados.

8 Diagrama de Clases de la Capa de Persistencia

9 Descripción de Clases de Persistencia

9.1 AlphabetPersistence

Esta clase gestiona la persistencia de datos relacionados con alfabetos. Sus funciones principales incluyen la verificación de la existencia, así como la creación de un archivo de almacenamiento, y la lectura y escritura de datos de alfabetos en dicho archivo. La clase está diseñada para interactuar con un objeto *Alphabet* y utiliza un archivo de texto para almacenar la información de forma persistente.

9.2 FileFPersistence

La clase *FileFPersistence* se encarga de la persistencia de datos relacionados con archivos de tipo *FileF*. Ofrece funciones para comprobar la existencia y crear un archivo de almacenamiento, así como para leer y escribir datos de estos archivos. Interactúa con objetos *FileF* y emplea un archivo de texto para el almacenamiento persistente de información.

9.3 KeyboardPersistence

KeyboardPersistence maneja la persistencia de datos relacionados con teclados, incluyendo los teclados guardados. Provee funciones para la verificación de existencia y creación de un archivo de almacenamiento, además de la lectura y escritura de datos de teclados en este archivo. La clase trabaja con objetos *KeyboardSaved* y usa un archivo de texto para el almacenamiento persistente.

9.4 TextPersistence

TextPersistence se ocupa de la persistencia de datos relacionados con textos. Incluye funciones para comprobar la existencia y crear un archivo de almacenamiento, así como para la lectura y escritura de datos de textos en dicho archivo. La clase interactúa con objetos *Text* y utiliza un archivo de texto para la persistencia de la información.

9.5 UserPersistence

La clase *UserPersistence* gestiona la persistencia de datos relacionados con usuarios. Proporciona funciones para verificar la existencia de un archivo de almacenamiento, crearlo y realizar operaciones de lectura y escritura de datos de usuarios. Interactúa con objetos *User* y utiliza un archivo de texto para el almacenamiento persistente.

9.6 PersistenceController

El *PersistenceController* coordina la interacción entre las clases de persistencia y la lógica del dominio. Ofrece métodos para guardar y cargar datos relacionados con teclados, archivos (*files*), textos, alfabetos y usuarios. Utiliza instancias de las clases de persistencia pertinentes para realizar estas operaciones, asegurando así la conservación de los datos entre sesiones de la aplicación.

10 Algoritmos

10.1 Introducción

El primer algoritmo implementado en este proyecto es el algoritmo de ramificación y acotación; también conocido en inglés como *Branch and Bound*. Este es un algoritmo para resolver problemas QAP, como puede ser el caso de este proyecto que implica asignar teclas en un teclado optimizando las distancias entre teclas al escribir un texto.

El objetivo, como se ha dicho, es encontrar una disposición de teclas en un teclado, minimizando el coste asociado con las distancias entre teclas y el flujo que hay entre las letras del texto.

10.2 Estructura de datos

10.2.1 Matriz de distancias (`DistanceMatrix`)

Se ha desarrollado la clase `DistanceMatrix` con el objetivo de implementar de manera eficiente una matriz que represente las distancias entre las teclas de un teclado. En el contexto de este proyecto, la clase `Keyboard` ha sido diseñada con un atributo denominado `layout`, que es un vector del tipo `ArrayList<Integer>`. Este vector se organiza de manera específica: si el teclado tiene n filas y m columnas, la primera fila ocupa las primeras m posiciones, la segunda fila las siguientes m posiciones, y así sucesivamente. La dimensión total del teclado es de $n \times m$, donde n representa el número de filas y m el número de columnas.

Para calcular las distancias en la matriz de distancias, se utiliza una matriz implementada como `Array<ArrayList<Double>> matrix`, con un tamaño de $t \times t$, siendo t el número total de teclas en el teclado. Para cualquier posición i y j, donde $0 < i, j < t$, `matrix[i][j]` representa la distancia entre la tecla en la posición i del vector del teclado y la posición j en el vector del teclado, considerando la distribución del teclado de $n \times m$.

A continuación se puede ver un pequeño ejemplo de cómo funciona el vector y la distribución que tomaría en el teclado, teniendo 2 filas y 5 columnas, lo que nos deja un total de 10 teclas:

0	1	2	3	4	5	6	7	8	9

Table 1: Vector que representa el teclado en el código

0	1	2	3	4
5	6	7	8	9

Table 2: Representación del teclado

La finalidad de la matriz de distancias es almacenar la distancia entre cada par de teclas. Por ejemplo, entre la tecla 0 y 1 existe una distancia de 1 unidad; entre la tecla 0 y 6, la distancia es la raíz cuadrada de 2; y entre la tecla 0 y 7, la distancia es la raíz cuadrada de 5 unidades. Este cálculo se aplica para todas las teclas. La matriz de distancias resultante de este teclado, considerando que debe ser una matriz de 10×10, se presenta a continuación:

Todas estas distancias se calculan gracias a la función `calculateDistanceMatrix(int numRows, int numCols)`. Esta función se encarga de determinar la distancia en el teclado entre una tecla y otra, tomando como referencia el número de filas y columnas del teclado.

El tiempo necesario para construir esta estructura de datos se debe principalmente a la creación de la matriz y al cálculo de las distancias. Dado que la matriz tiene dimensiones $t \times t$, donde t es el número total de teclas, podemos afirmar que el coste de crear y calcular esta matriz es de $O(t^2)$.

10.2.2 Matriz de transiciones (`FlowMatrix`)

La clase `FlowMatrix` ha sido diseñada con el propósito de analizar la frecuencia de ocurrencia de caracteres consecutivos en un texto, utilizando un alfabeto específico. La construcción de esta matriz implica la necesidad de un alfabeto que incluya n caracteres diferentes. Para representar las transiciones entre cada par de

0	1	$\sqrt{2}$	$\sqrt{3}$	$2\sqrt{1}$	1	$\sqrt{2}$	$\sqrt{2\sqrt{2}}$	$\sqrt{3\sqrt{1}}$	$2\sqrt{\sqrt{3}}$
1	0	1	$\sqrt{2}$	$\sqrt{3}$	$\sqrt{2}$	1	$\sqrt{2}$	$\sqrt{2\sqrt{2}}$	$\sqrt{3\sqrt{1}}$
$\sqrt{2}$	1	0	1	$\sqrt{2}$	$\sqrt{2\sqrt{2}}$	$\sqrt{2}$	1	$\sqrt{2}$	$\sqrt{2\sqrt{2}}$
$\sqrt{3}$	$\sqrt{2}$	1	0	1	$\sqrt{3\sqrt{1}}$	$\sqrt{2}$	1	$\sqrt{2}$	$\sqrt{2\sqrt{2}}$
$2\sqrt{1}$	$\sqrt{3}$	$\sqrt{2}$	1	0	$2\sqrt{\sqrt{3}}$	$\sqrt{2\sqrt{2}}$	$\sqrt{2}$	1	$\sqrt{2}$
1	$\sqrt{2}$	$\sqrt{2\sqrt{2}}$	$\sqrt{3\sqrt{1}}$	$2\sqrt{\sqrt{3}}$	0	1	$\sqrt{2}$	$\sqrt{3}$	$2\sqrt{1}$
$\sqrt{2}$	1	$\sqrt{2}$	$\sqrt{2\sqrt{2}}$	$\sqrt{3\sqrt{1}}$	1	0	1	$\sqrt{2}$	$\sqrt{3}$
$\sqrt{2\sqrt{2}}$	$\sqrt{2}$	1	$\sqrt{2}$	$\sqrt{2\sqrt{2}}$	$\sqrt{2}$	1	0	1	$\sqrt{2}$
$\sqrt{3\sqrt{1}}$	$\sqrt{2\sqrt{2}}$	$\sqrt{2}$	1	$\sqrt{2}$	$\sqrt{3}$	$\sqrt{2}$	1	0	1
$2\sqrt{\sqrt{3}}$	$\sqrt{3}$	$\sqrt{2\sqrt{2}}$	$\sqrt{2\sqrt{2}}$	1	$2\sqrt{1}$	$\sqrt{3}$	$\sqrt{2}$	1	0

Table 3: Representación de la matriz de distancias

caracteres, se utiliza una matriz implementada con la estructura `ArrayList<ArrayList<Integer>>`, con un tamaño de $n \times n$, donde n es el número de caracteres en el alfabeto.

En cada posición $[i][j]$ de la matriz, donde $0 < i, j < n$, se almacena la cantidad de veces que los caracteres en las posiciones i y j aparecen consecutivamente en el texto, respetando el orden y sin invertirlos. Como ejemplo, consideremos un alfabeto $[b, c, f, i, p, u]$ y el texto "upcfibfibupc". La matriz resultante reflejará la frecuencia de ocurrencia de las secuencias de caracteres en las posiciones i y j.

A modo de ejemplo, consideremos un alfabeto con las letras $[b, c, f, i, p, u]$ y analicemos el siguiente texto: "upcfibfibupc". La matriz de transición correspondiente se construirá de manera que cada elemento `matrix[i][j]` refleje la frecuencia de ocurrencia de la secuencia de caracteres en las posiciones i y j.

0	0	1	0	0	1
0	0	1	0	0	0
0	0	0	2	0	0
2	0	0	0	0	0
0	2	0	0	0	0
0	0	0	0	2	0

Table 4: Representación de la matriz de transiciones

La función central para determinar estas frecuencias es `calculateTransitions(HashMap<String, Integer> MapW, TreeSet<String> s)`. Esta función utiliza una lista de palabras con sus frecuencias calculadas a partir del texto y analiza las transiciones entre caracteres consecutivos. Cada par de caracteres se evalúa como la fila y columna respectivamente, registrando la frecuencia de la secuencia específica. Cabe destacar que se considera la existencia del carácter espacio en el alfabeto y, por ende, en el teclado.

Adicionalmente, se ha implementado una función que permite la suma de esta matriz de transiciones con otra. Esta capacidad facilita la combinación de múltiples textos para calcular la distribución de un teclado, posibilitando la integración de conjuntos de datos diversos en el análisis de la disposición óptima de las teclas.

El tiempo requerido para instanciar esta clase está determinado por la inicialización de la matriz de transición. Si el tamaño del alfabeto es n, el coste de esta inicialización es $O(n^2)$. Por otro lado, la aplicación de la función `calculateTransitions` puede tener un coste diferente. Suponiendo que hay m caracteres en la lista de palabras utilizada por la función, el coste computacional de esta función sería recorrer el texto para cada par de caracteres, resultando en un coste de $O(m)$. Es crucial destacar que hemos permitido a las matrices de transiciones utilizar una función `sumMatrix` para combinar dos matrices y así unir varios textos. Esto tiene un coste de $O(n^2)$, ya que implica sumar las dos matrices de tamaño $n \times n$.

10.2.3 Mejor asignación del teclado (`bestAssigment`)

Al ejecutar los algoritmos, se encontrarán diversas asignaciones de teclado, algunas de las cuales serán mejores que otras. Gracias a este atributo, que se presenta como un vector implementado mediante `ArrayList<Integer>`, podemos mantener un registro de cuál es la distribución óptima del teclado conforme ejecutamos los algoritmos.

10.3 Funcionamiento de la clase Algoritmo

Para hacer funcionar esta clase, se ha desarrollado un controlador que facilita la ejecución de los dos algoritmos diseñados. Este controlador se encarga de la inicialización de la matriz de distancias y la matriz de transiciones, así como de calcular cada matriz mediante la provisión de datos relacionados con el teclado, el alfabeto y los textos pertinentes.

Una vez que se ha inicializado el algoritmo destinado a calcular la mejor asignación para un teclado, el controlador procede a aplicar dicho algoritmo. En este proceso, el controlador invoca la función `solve` de la clase `Algorithm`, la cual resolverá de manera específica dependiendo de que subclase sea ese algoritmo.

11 Algoritmo de Ramificación y Acotación (`Branch and Bound`)

11.1 Introducción

El primer algoritmo implementado en este proyecto es el algoritmo de ramificación y acotación; también conocido en inglés como *Branch and Bound*. Este es un algoritmo para resolver problemas QAP, como puede ser el caso de este proyecto que implica asignar teclas en un teclado optimizando las distancias entre teclas al escribir un texto.

El objetivo, como se ha dicho, es encontrar una disposición de teclas en un teclado, minimizando el coste asociado con las distancias entre teclas y el flujo que hay entre las letras del texto.

11.2 Estructura de datos

Teniendo en cuenta que este algoritmo, al ser una subclase de la clase `Algorithm`, cuenta con los atributos de esta mencionados anteriormente. Además de estas estructuras de datos, también ha sido necesario crear ciertas estructuras de datos dentro del propio algoritmo.

11.2.1 Mejor Coste (`bestCost`)

El atributo `bestCost` está implementado como un `double` y se utiliza para realizar un seguimiento del mejor coste encontrado durante la ejecución del algoritmo. Se inicializa con el valor de `Integer.MAX_VALUE` al comienzo del método `solve`. A medida que avanza la ejecución y se exploran las posibles soluciones, el atributo se actualiza cuando se encuentra una asignación de teclas completa que tiene un coste menor que el actual.

11.2.2 Cola de Nodos (`queue`)

El atributo `queue` es una cola de prioridad implementada como una `PriorityQueue<Node>`. Se encarga de almacenar los nodos vivos ordenados por su coste, de menor a mayor. Se refiere a nodos vivos a aquellos que aún no han sido completamente explorados y pueden generar más descendientes en el árbol de búsqueda.

El algoritmo usa esta cola de prioridad para seleccionar el próximo nodo a explorar, eligiendo aquel con menor coste. La cola de prioridad nos permite seleccionar de manera óptima las ramas del árbol que tienen más probabilidad de contener soluciones óptimas. El coste de insertar y extraer nodos de la cola de prioridad es de $O(\log n)$, donde n es el número de elementos en la cola de prioridad.

11.2.3 Clase `Nodo` (`Node`)

La clase `Node` en este código representa un nodo en el árbol de búsqueda utilizado por la cola de prioridad. Cada nodo contiene información importante que ayuda en la exploración eficiente del espacio de búsqueda. Aquí hay una descripción de los atributos de la clase `Node`:

- `level`: Representa el nivel del nodo en el árbol de búsqueda. Inicialmente, el nivel es 0 para el nodo raíz y se incrementa a medida que el algoritmo explora más profundamente en el árbol.
- `assignment`: Es un `ArrayList` que almacena la asignación de teclas asociada a ese nodo en particular. Cada posición en el `ArrayList` corresponde a una tecla en el teclado, y el valor almacenado es el índice de la letra asignada a esa tecla. Inicialmente, este `ArrayList` se llena con valores -1 para indicar que ninguna tecla ha sido asignada.
- `cost`: Representa el coste estimado del nodo. Este coste se utiliza para ordenar los nodos en la cola de prioridad (`PriorityQueue`). La idea es explorar primero los nodos con menores costes para realizar podas más efectivas y mejorar la eficiencia del algoritmo.

En resumen, la clase `Node` encapsula la información necesaria para representar un estado parcial en el árbol de búsqueda, lo que incluye la asignación de teclas actual, el nivel en el árbol y el coste estimado asociado con ese estado parcial.

11.3 Métodos de la clase

11.3.1 `solve`

Este método inicia el proceso de resolución del problema utilizando el algoritmo. Inicializa las estructuras necesarias y comienza a explorar el árbol de búsqueda.

11.3.2 `branchAndBound`

Este método es una implementación recursiva del algoritmo de backtracking. Explora las ramas del árbol de búsqueda y realiza podas para evitar la exploración innecesaria de ciertas ramas.

11.3.3 `calculateLowerBound`

Calcula una cota inferior del coste de las soluciones alcanzables desde la asignación actual utilizando una heurística greedy.

11.3.4 `completeAssignment`

Completa la asignación de teclas faltantes utilizando el algoritmo greedy. El algoritmo greedy que se está usando para calcular la cota inferior consiste en asignar las teclas restantes de manera eficiente, minimizando el coste acumulado de la asignación. Este proceso se lleva a cabo iterando sobre las teclas del teclado que aún no tienen asignados. Para cada tecla no asignada, se evalúa cada letra disponible sin asignar cuál es la mejor para que no dé un coste muy elevado.

11.3.5 `calculateCost`

Calcula el coste de una asignación dada.

11.4 Funcionamiento del Algoritmo

Primero de todo se inicializan las estructuras de datos, se establece el mejor coste como infinito y se añade el nodo raíz a la cola. Seguidamente se entra en el bucle principal que se encarga de extraer el nodo con mejor coste de la cola mientras esta no está vacía. De este nodo extraído se expande generando nodos sucesores. Para cada sucesor generado se calcula una cota inferior y se verifica si es menor que el mejor coste actual. Si es así, se agrega el sucesor a la cola de prioridad.

El algoritmo al mismo tiempo también se encarga de explorar el árbol y hacer podas de este. La poda se realiza si hay alguna cota inferior que es mayor que el coste encontrado hasta el momento.

En cada nodo completo del árbol, se verifica si la asignación actual tiene un coste menor que el mejor coste encontrado hasta el momento. Si es así, se actualiza el mejor resultado con la nueva asignación y coste.

Esto pasa hasta que la cola de prioridad queda vacía. Al final, en el atributo `bestAssigment`, encontraremos la mejor distribución encontrada por el algoritmo.

A continuación, hay un pseudocódigo de lo que hace el algoritmo:

Algorithm 1 Branch and Bound

```
1: procedure BranchAndBound(distanceMatrix, flowMatrix)
2:     bestCost ← ∞
3:     bestAssignment ← []
4:     queue ← PriorityQueue()
5:     procedure solve
6:         Initialize bestCost and bestAssignment
7:         Create root node with level = 0, currentAssignment = [], and cost = 0
8:         Add root node to the queue
9:         while queue is not empty do
10:            node ← queue.poll()                    ▷ Extract node with the lowest cost
11:            branchAndBound(node.level, node.assignment, node.cost)
12:        end while
13:    end procedure
14:    procedure branchAndBound(level, currentAssignment, currentCost)
15:        count ← Count keys already assigned in currentAssignment
16:        if level == size of distanceMatrix then
17:            if currentCost < bestCost then
18:                Update bestCost and bestAssignment
19:            end if
20:            return
21:        else if count == size of flowMatrix then
22:            if currentCost < bestCost then
23:                Update bestCost and bestAssignment
24:            end if
25:            return
26:        end if
27:        for i = 0 to size of flowMatrix do
28:            if i not in currentAssignment then
29:                currentAssignment[level] ← i
30:                lowerBound ← calculateLowerBound(currentAssignment)
31:                if lowerBound < bestCost then
32:                    Add new Node(level + 1, currentAssignment, lowerBound) to the queue
33:                end if
34:                currentAssignment[level] ← −1
35:            end if
36:        end for
37:    end procedure
38:    function calculateLowerBound(assignment)
39:        greedyAssignment ← copy of assignment
40:        completeAssignment(greedyAssignment)
41:        return calculateCost(greedyAssignment)
42:    end function
```

Algorithm 2 Branch and Bound

```
43:     procedure completeAssignment(assignment)
44:         for each t1 in distanceMatrix do
45:             if assignment[t1] == −1 then
46:                 minCost ← ∞
47:                 bestKey ← −1
48:                 for each k in flowMatrix do
49:                     if k not in assignment then
50:                         cost ← calculateCostWithNewAssignment(t1, k, assignment)
51:                         if cost < minCost then
52:                             minCost ← cost
53:                             bestKey ← k
54:                         end if
55:                     end if
56:                 end for
57:                 assignment[t1] ← bestKey
58:             end if
59:         end for
60:     end procedure
61:     function calculateCostWithNewAssignment(t1, k, assignment)
62:         cost ← 0
63:         for each t2 in distanceMatrix do
64:             x ← assignment[t1]
65:             y ← assignment[t2]
66:             if x ≠ −1 and y ≠ −1 then
67:                 cost ← cost + distanceMatrix[t1][t2] × flowMatrix[k][x]
68:             end if
69:         end for
70:         return cost
71:     end function
72:     function calculateCost(assignment)
73:         cost ← 0
74:         for each t1 in distanceMatrix do
75:             for each t2 in distanceMatrix do
76:                 x ← assignment[t1]
77:                 y ← assignment[t2]
78:                 if x ≠ −1 and y ≠ −1 then
79:                     cost ← cost + distanceMatrix[t1][t2] × flowMatrix[x][y]
80:                 end if
81:             end for
82:         end for
83:         return cost
84:     end function
85: end procedure
```

11.4.1 Coste computacional

En el peor caso, la cola de prioridad puede contener todos los nodos posibles en el árbol de búsqueda, y cada nodo puede generar un número significativo de sucesores. Suponiendo que el árbol de búsqueda es completamente explorado, la complejidad del método `solve()` es del orden de $O(b^d)$, donde b es el factor de ramificación promedio y d es la profundidad máxima del árbol.

En cada llamada recursiva del método `branchAndBound`, se realizan operaciones de bucle sobre las teclas y las letras, y se llevan a cabo operaciones adicionales para calcular cotas y realizar podas. Suponiendo que el número de teclas y letras es n, la complejidad de cada llamada recursiva podría ser del orden de $O(n^2)$.

12 Algoritmo Evolutivo (`Evolutive`)

12.1 Introducción

El segundo algoritmo implementado en este proyecto es un algoritmo evolutivo. Este algoritmo intenta resolver el problema QAP que hemos visto usando un algoritmo de la categoría de algoritmos de optimización y búsqueda heurística, inspirado en los principios de la evolución biológica.

El objetivo, como se ha mencionado en el algoritmo anterior, es encontrar una disposición de teclas en un teclado, minimizando el costo asociado con las distancias entre teclas y el flujo entre las letras del texto.

12.2 Estructura de datos

Teniendo en cuenta que este algoritmo es una subclase de la clase `Algorithm`, hereda los atributos previamente mencionados de esta clase. Además de utilizar estas estructuras de datos, ha sido necesario introducir nuevas estructuras dentro del propio algoritmo.

Antes de abordar las estructuras de datos específicas utilizadas en este algoritmo, es relevante mencionar una serie de constantes. Estas constantes se utilizan para definir las probabilidades de que ciertos eventos ocurran al generar una nueva generación de la población actual. Además, se establecen límites poblacionales para determinar cuándo el algoritmo debe finalizar su ejecución.

12.2.1 Población (`population`)

El atributo `population` sirve para guardar la población actual que usará el algoritmo para calcular la siguiente población durante la ejecución del algoritmo.

12.2.2 Clase Individuo (`Individual`)

La clase llamada `Individual` representa a un individuo en la población. Aquí hay una descripción de los atributos:

- `layout`: Es un `ArrayList` de enteros que representa la disposición de las letras del teclado o genes en el caso de imaginarnos un individuo.
- `numKeys`: Es un entero que nos indica el número total de teclas o genes del individuo.
- `numLetters`: Es un entero que representa el número total de letras que tiene el teclado, o genes que aportan información útil del individuo.
- `fitness`: Es un número real que representa lo bueno que es ese individuo con su código genético o, más bien, lo bueno que es ese teclado para teclear el texto.

En resumen, la clase `Individual` encapsula la información necesaria para representar toda la información de un individuo, el cual es un teclado.

12.3 Métodos de la clase

12.3.1 `solve`

Este método inicia el proceso de resolución del problema utilizando el algoritmo. Inicializa la población y evoluciona a través de las generaciones para encontrar la mejor asignación de teclas.

12.3.2 `iniPopulation`

Este método inicializa la población de individuos dando una asignación aleatoria de las teclas.

12.3.3 `generateLayout`

Genera una asignación aleatoria de teclas.

12.3.4 `evolutePopulation`

Realiza la evolución de la población, aplicando los operadores genéticos de cruce y mutaciones diferentes.

12.4 Funcionamiento del Algoritmo

En primer lugar, se inicia la población con individuos aleatorios. Una vez que se tiene una población inicial, se procede a evolucionarla para obtener una nueva población. La evolución implica el cruce de algunos individuos entre sí, así como la introducción de mutaciones en sus genes o teclas.

Una vez que se ha generado una nueva población compuesta únicamente por descendientes que han experimentado mutaciones, se procede a seleccionar los mejores individuos entre los hijos y los padres. Estos individuos formarán el conjunto que constituirá la población de la siguiente generación. Este proceso se repite durante tantas generaciones como se haya definido previamente. Al llegar a la última generación, el mejor individuo será aquel que proporcione la asignación óptima.

A continuación, hay un pseudocódigo de lo que hace el algoritmo:

Algorithm 3 Algoritmo Evolutivo para Asignación Óptima de Teclas

```
1: Constantes:
2:     CROSSOVER_PROBABILITY = 0.5
3:     CROSSOVER_THEN_MUTATION_PROBABILITY = 0.7
4:     PERMUTATION_PROBABILITY = 0.4
5:     INVERSION_PROBABILITY = 0.2
6:     MAXIMUM_POPULATION = 100
7:     MAXIMUM_GENERATION = 300
8: procedure SOLVE
9:     iniPopulation
10:    h ← 0
11:    while h ≤ MAXIMUM_GENERATION do
12:        evolutePopulation
13:        h ← h + 1
14:    end while
15:    bestAssignment ← population[0].layout
16: end procedure
17: function iniPopulation
18:    population ← ArrayList¡Individual¿()
19:    for i ← 0 to MAXIMUM_POPULATION do
20:        layout ← generateLayout(distanceMatrix.matrix.size())
21:        individual ← Individual(layout, flowMatrix.matrix.size(), distanceMatrix.matrix.size())
22:        population.add(individual)
23:    end for
24: end function
25: function generateLayout(size)
26:    layout ← ArrayList¡Integer¿()
27:    for i ← 0 to flowMatrix.matrix.size() do
28:        layout.add(i)
29:    end for
30:    for i ← flowMatrix.matrix.size() to size do
31:        layout.add(−1)
32:    end for
33:    Collections.shuffle(layout)
34:    return layout
35: end function
```

Algorithm 4 Algoritmo Evolutivo para Asignación Óptima de Teclas

```
36: function evolutePopulation
37:     newPopulation ← ArrayList¡Individual¿()
38:     while newPopulation.size() ≤ MAXIMUM_POPULATION do
39:         for i ← 0 to population.size() do
40:             n ← random.nextInt(MAXIMUM_POPULATION)
41:             while n = i do
42:                 n ← random.nextInt(MAXIMUM_POPULATION)
43:             end while
44:             son ← population[i].crossover(population[n])
45:             if random.nextDouble() ≤ PERMUTATION_PROBABILITY then
46:                 son.swapPositions()
47:             end if
48:             if random.nextDouble() ≤ INVERSION_PROBABILITY then
49:                 son.inversionLayout()
50:             end if
51:             son.calculateFitness(distanceMatrix, flowMatrix)
52:             newPopulation.add(son)
53:         end for
54:     end while
55:     allPopulation ← ArrayList¡Individual¿()
56:     allPopulation.addAll(newPopulation)
57:     allPopulation.addAll(population)
58:     allPopulation.sort(Individual::compareTo)
59:     population ← ArrayList¡Individual¿(allPopulation.subList(0, MAXIMUM_POPULATION))
60: end function
```

12.4.1 Coste computacional

El algoritmo evolutivo implementado presenta una complejidad del orden de $O(\text{MAXIMUM\_GENERATION} \times \text{MAXIMUM\_POPULATION} \times n^2 \times m^2)$, donde n es la cantidad de teclas, m es el número de letras. El método `solve` itera a lo largo de un máximo de `MAXIMUM_GENERATION` generaciones, cada una implicando la evolución de la población en el método `evolutePopulation`. La complejidad de esta evolución depende de las operaciones realizadas en la creación de nuevos individuos y la actualización de la población.