Universitat Politècnica de Catalunya - FIB/UPC

PROP - Projectes de Programació

Generador de teclados - tercera entrega

Momin Miah Begum (momin.miah) , Muhammad Yasin Khokhar Jalil (muhammad.yasin.khokhar) , Jianing Xu (jianing.xu), Rubén Catalán Rua (ruben.catalan)

Cluster 23.5

Versión 1.0

Q1 2023-24

Índice

1. Casos de uso

1.1. Diagrama

Figura 1: Diagrama de casos de uso

1.2. Descripción de los casos de uso

Nombre	Seleccionar tipo de interfaz
Actores	Usuario
Dependencias	
Precondición	
Descripción	El usuario tiene la opción de escoger que interfaz quiere utilizar para ejecutar el programa. En concreto tiene dos opciones, interfaz de gráfica y interfaz linea de comandos (terminal).
Secuencia normal	1. El usuario inicia la ejecución del programa. 2. El sistema le pide al usuario qué tipo de interfaz quiere utilizar. 3. El usuario le proporciona la opción que desea. 4. El sistema despega una interfaz basado en la opción elegida por el usuario.
Postcondición	Se ha lanzado la interfaz con el que se llevará acabo la interacción con el programa.
Excepciones	

Nombre	Crear Perfil
Actores	Usuario
Dependencias	
Precondición	No existe un perfil con el mismo nombre.
Descripción	Crea un perfil con un nombre de usuario y una contraseña.
Secuencia normal	1. El usuario inicia el programa y pide al sistema crear un perfil. 2. El sistema le pide al usuario un nombre y una contraseña. 3. El usuario proporciona un nombre de usuario y una contraseña. 4. El sistema crea un perfil con los datos proporcionados por el usuario.
Postcondición	Se ha creado un perfil en el sistema.
Excepciones	▪ Ya existe un usuario con el nombre proporcionado. ▪ El formato del nombre no es correcto. ▪ La contraseña debe tener al menos 8 caracteres y un símbolo especial.

Nombre	Iniciar Sesión
Actores	Usuario
Dependencias	
Precondición	Existe un perfil con el mismo nombre creado.
Descripción	Carga todos los datos necesarios de un archivo .prop para el perfil sobre el cual se inicia sesión.
Secuencia normal	1. El usuario inicia el programa y pide al sistema iniciar la sesión. 2. El sistema le pide al usuario un nombre y una contraseña. 3. El usuario proporciona un nombre de usuario y una contraseña. 4. El sistema carga los datos y la información relacionada al perfil sobre el cual se inicia la sesión.
Postcondición	Se ha iniciado la sesión de un perfil en el sistema.
Excepciones	▪ No existe ningún perfil con los datos del usuario. ▪ Los datos del archivo .prop no se han cargado correctamente.

Nombre	Cambiar Contraseña
Actores	Usuario
Dependencias	
Precondición	Existe un perfil con el nombre de usuario.
Descripción	Permite modificar cambiar la contraseña que contiene el perfil.
Secuencia normal	1. El usuario inicia el programa e inicia con sus datos de usuario (usuario + contraseña). 2. El usuario selecciona cambiar contraseña. 3. El sistema le pide que vuelva a ingresar el usuario y la contraseña antigua y la nueva. 4. El sistema evalúa los datos que el usuario ha facilitado, y realiza el cambio de contraseña.
Postcondición	Se ha cambiado la contraseña del usuario.
Excepciones	▪ No existe un usuario con el nombre proporcionado o la contraseña es incorrecta. ▪ La contraseña nueva no es válida, la longitud de la contraseña es menor que 8.

Nombre	Borrar Perfil
Actores	Usuario
Dependencias	
Precondición	No existe un perfil con el mismo nombre.
Descripción	Borra un perfil existente del sistema.
Secuencia normal	1. El usuario inicia el programa y pide al sistema borrar un perfil. 2. El sistema le pide al usuario un nombre y la contraseña del perfil a borrar. 3. El usuario proporciona un nombre de usuario y la contraseña. 4. El sistema borra el perfil con los datos proporcionados por el usuario. 5. El sistema elimina la información del fichero .prop relacionado al usuario.
Postcondición	Se ha borrado un perfil en el sistema.
Excepciones	▪ No existe un usuario con el nombre proporcionado.

Nombre	Crear teclado
Actores	Usuario
Dependencias	
Precondición	Existe al menos un alfabeto en el sistema.
Descripción	Crea un teclado con un nombre, alfabeto disponible, texto, lista de palabras y algoritmo.
Secuencia normal	1. El usuario escribe el nombre del teclado a crear y el de uno de los alfabetos disponibles. 2. Posteriormente, se escribe un texto y una lista de palabras con frecuencias (formato <palabra> <entero>) y uno de los dos algoritmos disponibles (B&B o SA).
Postcondición	Se ha creado un teclado con los parámetros proporcionados.
Excepciones	▪ Ya existe un teclado con el nombre proporcionado. ▪ No existe un alfabeto con el nombre proporcionado. ▪ Formato incorrecto para lista de palabras. ▪ Nombre de algoritmo incorrecto. ▪ El nombre de teclado a crear no es válido (espacios, o nombre vacío).

Nombre	Modificar teclado
Actores	Usuario
Dependencias	
Precondición	Existe al menos un teclado en el sistema.
Descripción	Modifica alguno de los atributos de un teclado existente.
Secuencia normal	1. Se escoge uno de los teclados creados. 2. Posteriormente, se escoge una de las opciones mostradas para realizar la modificación: Cambio del alfabeto, texto, lista de palabras o algoritmo.
Postcondición	Se ha modificado el teclado especificado y se ha actualizado su distribución.
Excepciones	▪ No existe un teclado con el nombre proporcionado.

Nombre	Borrar teclado
Actores	Usuario
Dependencias	
Precondición	Existe al menos un teclado en el sistema.
Descripción	Borra alguno de los teclados existentes.
Secuencia normal	1. El usuario escribe el nombre de un teclado. 2. Si este existe, se borrará del conjunto de teclados del usuario. Si no, no se hará nada.
Postcondición	Se ha borrado el teclado especificado.
Excepciones	▪ No existe un teclado con el nombre proporcionado.

Nombre	Mostrar distribución de teclado
Actores	Usuario
Dependencias	
Precondición	Existe al menos un teclado en el sistema.
Descripción	Muestra por pantalla la distribución de un teclado.
Secuencia normal	1. El usuario escribe el nombre de uno de sus teclados creados. 2. Se muestra por pantalla la distribución eficiente de los símbolos de su alfabeto generada a partir de uno de los algoritmos.
Postcondición	
Excepciones	▪ No existe un teclado con el nombre proporcionado.

Nombre	Consultar teclados existentes
Actores	Usuario
Dependencias	
Precondición	
Descripción	Se escribe el nombre de todos los teclados existentes.
Secuencia normal	1. Se escoge la opción çonsultar teclados disponibles". 2. Se muestran por pantalla el nombre de todos los teclados creados.
Postcondición	
Excepciones	▪ No hay ningún teclado creado.

Nombre	Intercambiar teclas (Modificar teclado)
Actores	Usuario
Dependencias	
Precondición	Existe al menos un teclado en el sistema.
Descripción	Intercambia la posición de dos teclas del teclado.
Secuencia normal	1. El usuario selecciona la opción Ïntercambiar teclas". 2. El sistema pide al usuario el nombre del teclado objetivo. 3. El usuario introduce el nombre del teclado. 4. El sistema pide al usuario la fila y columna de los dos caracteres a intercambiar. 5. El usuario proporciona los datos pedidos. 6. El sistema intercambia la posición de las teclas especificadas.
Postcondición	Se ha actualizado la distribución del teclado especificado intercambiado las posiciones de las teclas especificadas.
Excepciones	▪ No existe un teclado con el nombre proporcionado. ▪ El rango de las teclas a intercambiar no es válido.

Nombre	Crear Alfabeto
Actores	Usuario
Dependencias	
Precondición	
Descripción	Crea un nuevo alfabeto con un nombre único con sus caracteres correspondientes.
Secuencia normal	1. El usuario pide al sistema crear un alfabeto. 2. El sistema pide al usuario un nombre para el alfabeto. 3. El sistema pide al usuario los caracteres que tiene el alfabeto. 4. El sistema crea el alfabeto con los datos proporcionados.
Postcondición	Se ha creado un alfabeto con los datos proporcionados.
Excepciones	▪ Ya existe un alfabeto con el nombre proporcionado. ▪ El nombre del alfabeto no es válido (espacios, o nombre vacío). ▪ Los caracteres no pueden ser vacíos.

Nombre	Modificar Alfabeto
Actores	Usuario
Dependencias	
Precondición	Existe al menos un alfabeto en el sistema.
Descripción	Modifica los caracteres de un alfabeto existente en el sistema.
Secuencia normal	1. El usuario pide al sistema modificar un alfabeto. 2. El sistema le pide al usuario el nombre del alfabeto que desea modificar. 3. El usuario le proporciona el nombre del alfabeto a modificar. 4. El sistema le pide al usuario los caracteres. 5. El sistema modifica los caracteres del alfabeto.
Postcondición	Se ha modificado los caracteres del alfabeto especificado.
Excepciones	▪ No existe un alfabeto con el nombre proporcionado en el sistema. ▪ Los nuevos caracteres no pueden ser vacíos.

Nombre	Borrar Alfabeto
Actores	Usuario
Dependencias	
Precondición	Existe al menos un alfabeto en el sistema.
Descripción	Borra un alfabeto del sistema.
Secuencia normal	1. El usuario pide al sistema borrar un alfabeto. 2. El sistema le pide al usuario el nombre del alfabeto que desea borrar. 3. El usuario le proporciona el nombre del alfabeto a borrar. 4. El sistema borra el alfabeto.
Postcondición	Se ha borrado el alfabeto especificado.
Excepciones	▪ No existe un alfabeto con el nombre proporcionado en el sistema.

Nombre	Mostrar caracteres de alfabeto
Actores	Usuario
Dependencias	
Precondición	Hay al menos un alfabeto creado en el sistema.
Descripción	Muestra los caracteres del alfabeto especificado.
Secuencia normal	1. El usuario pide al sistema mostrar los caracteres de un alfabeto. 2. El sistema pide al usuario el nombre del alfabeto. 3. El usuario proporciona el nombre del alfabeto. 4. El sistema muestra los caracteres del alfabeto deseado.
Postcondición	
Excepciones	▪ No existe un alfabeto con el nombre proporcionado en el sistema.

Nombre	Consultar alfabetos existentes
Actores	Usuario
Dependencias	
Precondición	
Descripción	Muestra todos los alfabetos existentes.
Secuencia normal	1. El usuario pide al sistema mostrar los alfabetos existentes. 2. El sistema muestra los nombres de los alfabetos existentes.
Postcondición	
Excepciones	▪ No hay alfabetos creados.

2. Diagrama de clases y explicaciones

En esta tercera entrega, respecto a la segunda , hemos hecho algunos cambios importantes en la forma en que se presentan las cosas. Decidimos quitar algunas partes que, después de pensarlo bien, no eran realmente necesarias o útiles. Ahora, en lugar de tener un montón de vistas complicadas, solo mantuve las más esenciales y útiles, como Iniciar Sesión, Menú Principal, Gestionar Teclado, Gestionar Alfabeto, Crear Cuenta, Bienvenida y Cambiar Contraseña. Esto implica cambios de las funciones en CtrlPresentacion, los cuales volveremos a definir.
En la entrega anterior, cometimos un error al confundir el diagrama de clases con el diagrama de flujo de ejecución de las vistas. Para corregir este problema, presentamos una nueva versión más precisa, donde todas las vistas se comunican directamente con el controlador de presentación. Este enfoque asegura una representación más adecuada de la arquitectura del sistema, mejorando la claridad en la interacción entre las vistas y el controlador de presentación.

2.1. Clases y controladores de la capa de presentación

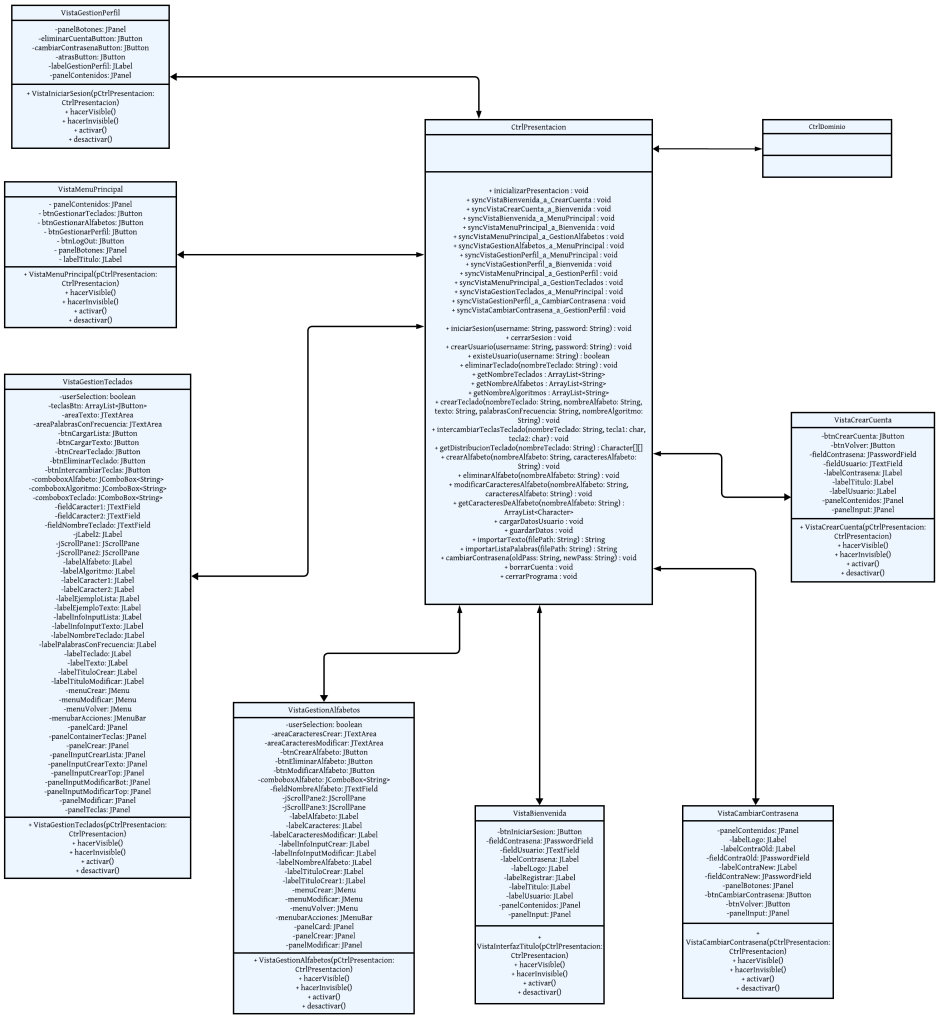

Figura 2: Diagrama de clases de la capa de presentación

2.1.1. VistaBienvenida

La clase `VistaBienvenida` gestiona la interfaz de usuario para la bienvenida al programa. Permite al usuario iniciar sesión, registrarse y acceder a otras funcionalidades del programa.

Sus atributos son los siguientes:

- `iCtrlPresentacion`: Controlador de presentación asociado a esta vista.

- `btnIniciarSesion`: Botón para iniciar sesión.
- `fieldContrasena`: Campo de contraseña para ingresar la contraseña del usuario.
- `fieldUsuario`: Campo de texto para ingresar el nombre de usuario.
- `labelContrasena`: Etiqueta que indica el campo de entrada para la contraseña.
- `labelLogo`: Etiqueta que representa el logo del programa.
- `labelRegistrar`: Etiqueta que proporciona la opción para registrar una nueva cuenta.
- `labelTitulo`: Etiqueta que muestra el título de la vista.
- `labelUsuario`: Etiqueta que indica el campo de entrada para el nombre de usuario.
- `panelContenidos`: Panel que contiene todos los elementos visuales de la interfaz.
- `panelInput`: Panel que agrupa los campos de entrada de usuario y contraseña.

La clase también presenta métodos públicos esenciales, entre ellos:

- `hacerVisible()`: Muestra la interfaz gráfica al usuario.
- `hacerInvisible()`: Oculta la interfaz gráfica.
- `activar()`: Activa la interfaz para interacciones.
- `desactivar()`: Desactiva la interfaz para prevenir interacciones no deseadas.

Ya que el resto de clases también tienen estos últimos cuatro métodos públicos, estos no se mencionarán a partir de ahora.

2.1.2. VistaGestionPerfil

La clase `VistaGestionPerfil` gestiona la interfaz de usuario relacionada con la gestión del perfil. Proporciona funcionalidades para eliminar la cuenta, cambiar la contraseña y regresar a la pantalla anterior.

Sus atributos son los siguientes:

- `iCtrlPresentacion`: Referencia al controlador de presentación, encargado de gestionar la lógica de la interfaz.
- `panelBotones`: Panel que contiene los botones de la interfaz.
- `eliminarCuentaButton`: Botón para eliminar la cuenta del usuario.
- `cambiarContrasenaButton`: Botón para cambiar la contraseña del usuario.
- `atrasButton`: Botón para regresar a la pantalla anterior.

- `labelGestionPerfil`: Etiqueta que indica la sección de gestión de perfil.
- `panelContenidos`: Panel que contiene todos los elementos visuales de la interfaz.

2.1.3. VistaCrearCuenta

La clase **VistaCrearCuenta** representa la interfaz gráfica encargada de gestionar la creación de nuevas cuentas en el programa de generación de teclados. Los usuarios deben ingresar un nombre de usuario y una contraseña para crear su cuenta.
Sus atributos son los siguientes :

- `iCtrlPresentacion`: Referencia al controlador de presentación, encargado de gestionar la lógica de la interfaz.
- `panelContenidos`: Panel que contiene todos los elementos visuales de la interfaz.
- `labelLogo`: Etiqueta que representa el logo del programa.
- `labelUsuario`: Etiqueta que indica el campo de entrada para el nombre de usuario.
- `fieldUsuario`: Campo de entrada de texto para el nombre de usuario.
- `labelContra`: Etiqueta que indica el campo de entrada para la contraseña.
- `fieldContra`: Campo de entrada de contraseña.
- `panelBotones`: Panel que contiene los botones de la interfaz.
- `btnCrearCuenta`: Botón para crear una nueva cuenta.
- `btnVolver`: Botón para volver a la pantalla de bienvenida.
- `panelInput`: Panel que agrupa los campos de entrada de usuario y contraseña.

2.1.4. VistaMenuPrincipal

La clase `VistaMenuPrincipal` gestiona la interfaz de usuario del menú principal. Proporciona funcionalidades para transicionar a la gestión de teclados, la gestión de alfabetos, la gestión de perfil y cerrar sesión.

Sus atributos son los siguientes:

- `iCtrlPresentacion`: Controlador de presentación asociado a esta vista.
- `panelContenidos`: Panel que contiene todos los elementos visuales del menú principal.
- `btnGestionarTeclados`: Botón para transicionar a la gestión de teclados en el menú principal.

- `btnGestionarAlfabetos`: Botón para transicionar a la gestión de alfabetos en el menú principal.
- `btnGestionarPerfil`: Botón para transicionar a la gestión de perfil en el menú principal.
- `btnLogOut`: Botón para cerrar sesión en el menú principal.
- `panelBotones`: Panel que contiene los botones del menú principal.
- `labelTitulo`: Etiqueta de título del menú principal.

2.1.5. VistaGestionarAlfabeto

La clase `VistaGestionarAlfabeto` gestiona la interfaz de usuario para la creación, modificación y eliminación de alfabetos. Permite al usuario interactuar con diferentes elementos visuales para llevar a cabo estas acciones.

Sus atributos son los siguientes:

- `iCtrlPresentacion`: Controlador de presentación asociado a esta vista.
- `userSelection`: Booleano que indica si el usuario puede seleccionar un elemento del combobox.
- `areaCaracteresCrear`: Área de texto para ingresar caracteres al crear un alfabeto.
- `areaCaracteresModificar`: Área de texto para ingresar caracteres al modificar un alfabeto.
- `btnCrearAlfabeto`: Botón para crear un alfabeto.
- `btnEliminarAlfabeto`: Botón para eliminar un alfabeto.
- `btnModificarAlfabeto`: Botón para modificar un alfabeto.
- `comboboxAlfabeto`: Cuadro desplegable para seleccionar un alfabeto.
- `fieldNombreAlfabeto`: Campo de texto para ingresar el nombre de un alfabeto.
- `jScrollPane2`: Barra de desplazamiento para el área de caracteres al crear un alfabeto.
- `jScrollPane3`: Barra de desplazamiento para el área de caracteres al modificar un alfabeto.
- `labelAlfabeto`: Etiqueta que representa el texto .ªlfabeto".
- `labelCaracteres`: Etiqueta que representa el texto Çaracteres".
- `labelCaracteresModificar`: Etiqueta que representa el texto Çaracteres del alfabeto seleccionado".

- `labelInfoInputCrear`: Etiqueta de información para la entrada de caracteres al crear un alfabeto.
- `labelInfoInputModificar`: Etiqueta de información para la entrada de caracteres al modificar un alfabeto.
- `labelNombreAlfabeto`: Etiqueta que representa el texto "Nombre del alfabeto".
- `labelTituloCrear`: Etiqueta que representa el título Çreación de alfabeto".
- `labelTituloCrear1`: Etiqueta que representa el título Çonsulta de alfabeto".
- `menuCrear`: Menú de opciones para crear un alfabeto.
- `menuModificar`: Menú de opciones para consultar o modificar un alfabeto.
- `menuVolver`: Menú de opciones para volver al menú principal.
- `menubarAcciones`: Barra de menú para las acciones de la ventana.
- `panelCard`: Panel principal que contiene las tarjetas de creación y modificación.
- `panelCrear`: Panel para la creación de alfabetos.
- `panelModificar`: Panel para la modificación de alfabetos.

2.1.6. VistaGestionarTeclados

La clase `VistaGestionarTeclados` gestiona la interfaz de usuario para la creación, modificación y eliminación de teclados. Permite al usuario interactuar con diferentes elementos visuales para llevar a cabo estas acciones.

Sus atributos son los siguientes:

- `iCtrlPresentacion`: Controlador de presentación asociado a esta vista.
- `userSelection`: Booleano que indica si el usuario puede seleccionar un elemento del combobox.
- `teclasBtn`: Lista de botones que contiene los botones que representan las teclas del teclado seleccionado.
- `areaTexto`: Área de texto para ingresar o mostrar el contenido de un archivo de texto.
- `areaPalabrasConFrecuencia`: Área de texto para mostrar palabras con sus frecuencias.
- `btnCargarLista`: Botón para cargar una lista de palabras desde un archivo.
- `btnCargarTexto`: Botón para cargar un archivo de texto.
- `btnCrearTeclado`: Botón para crear un nuevo teclado.

- `btnEliminarTeclado`: Botón para eliminar un teclado existente.
- `btnIntercambiarTeclas`: Botón para intercambiar dos teclas en un teclado.
- `comboboxAlfabeto`: Cuadro desplegable para seleccionar un alfabeto.
- `comboboxAlgoritmo`: Cuadro desplegable para seleccionar un algoritmo.
- `comboboxTeclado`: Cuadro desplegable para seleccionar un teclado.
- `fieldCaracter1`: Campo de texto para introducir el primer caracter a intercambiar.
- `fieldCaracter2`: Campo de texto para introducir el segundo caracter a intercambiar.
- `fieldNombreTeclado`: Campo de texto para ingresar el nombre de un nuevo teclado.
- `jLabel2`: Etiqueta de texto.
- `jScrollPane1`: Panel de desplazamiento para el área de texto de entrada.
- `jScrollPane2`: Panel de desplazamiento para el área de palabras con frecuencias.
- `labelAlfabeto`: Etiqueta para el campo de selección del alfabeto.
- `labelAlgoritmo`: Etiqueta para el campo de selección del algoritmo.
- `labelCaracter1`: Etiqueta para el campo del primer caracter a intercambiar.
- `labelCaracter2`: Etiqueta para el campo del segundo caracter a intercambiar.
- `labelEjemploLista`: Etiqueta de ejemplo para la lista de palabras.
- `labelEjemploTexto`: Etiqueta de ejemplo para el texto.
- `labelInfoInputLista`: Etiqueta de información para la entrada de la lista de palabras.
- `labelInfoInputTexto`: Etiqueta de información para la entrada de texto.
- `labelNombreTeclado`: Etiqueta para el campo del nombre del teclado.
- `labelPalabrasConFrecuencia`: Etiqueta para el área de palabras con frecuencias.
- `labelTeclado`: Etiqueta para el campo de selección del teclado.
- `labelTexto`: Etiqueta para el área de texto.
- `labelTituloCrear`: Etiqueta de título para la creación.
- `labelTituloModificar`: Etiqueta de título para la modificación.
- `menuCrear`: Menú para la acción de crear.
- `menuModificar`: Menú para la acción de modificar.

- `menuVolver`: Menú para la acción de volver.
- `menubarAcciones`: Barra de menú que contiene los menús de acciones.
- `panelCard`: Panel que contiene las tarjetas (paneles) para mostrar diferentes vistas.
- `panelContainerTeclas`: Panel que contiene los botones para las teclas.
- `panelCrear`: Panel para la creación de teclados.
- `panelInputCrearLista`: Panel de entrada para crear una lista de palabras.
- `panelInputCrearTexto`: Panel de entrada para crear un texto.
- `panelInputCrearTop`: Panel de entrada superior para la creación.
- `panelInputModificarBot`: Panel de entrada inferior para la modificación.
- `panelInputModificarTop`: Panel de entrada superior para la modificación.
- `panelModificar`: Panel para la modificación de teclados.
- `panelTeclas`: Panel que contiene los botones de las teclas.

2.1.7. VistaCambiarContraseña

La clase **VistaCambiarContraseña:** Interfaz para cambiar la contraseña de un usuario. Incluye campos para la contraseña antigua y nueva, así como botones para realizar la acción de cambio y volver al menú principal.
Sus atributos son los siguientes:

- `iCtrlPresentacion`: Referencia al controlador de presentación, encargado de gestionar la lógica de la interfaz.
- `panelContenidos`: Panel que contiene todos los elementos visuales de la interfaz.
- `labelLogo`: Etiqueta que muestra el logo o identificador visual de la interfaz.
- `labelContraOld`: Etiqueta que indica el campo de la contraseña antigua.
- `fieldContraOld`: Campo de contraseña para introducir la contraseña antigua.
- `labelContraNew`: Etiqueta que indica el campo de la nueva contraseña.
- `fieldContraNew`: Campo de contraseña para introducir la nueva contraseña.
- `panelBotones`: Panel que contiene los botones de la interfaz.
- `btnCambiarContrasena`: Botón para realizar el cambio de contraseña.
- `btnVolver`: Botón para regresar al menú principal.
- `panelInput`: Panel que agrupa los campos de entrada de contraseña.

2.1.8. Vistas Eliminadas

VistaConsultarTeclado, VistaModificarTeclado, VistaEliminarTeclado, VistaCrearTeclado, VistaConsultarAlfabeto, VistaModificarAlfabeto, VistaEliminarAlfabeto, VistaCrearAlfabeto.

2.1.9. CtrlPresentacion

La clase **CtrlPresentacion** representa el controlador de presentación del programa, encargado de gestionar las interacciones entre las vistas y el controlador de dominio. Vamos a mostrar los atributos y métodos :

Atributos:

- `ctrlDominio`: Referencia al controlador de dominio, responsable de gestionar la lógica del programa.
- `vistaBienvenida, vistaCrearCuenta, vistaMenuPrincipal, vistaGestionAlfabetos, vistaGestionTeclados, vistaGestionPerfil, vistaCambiarContrasena`: Instancias de las distintas vistas del programa.

Métodos:

- `syncVistaBienvenida_a_CrearCuenta()` - Sincroniza la vista de bienvenida con la vista de crear cuenta
- `syncVistaCrearCuenta_a_Bienvenida()` - Sincroniza la vista de crear cuenta con la vista de bienvenida
- `syncVistaBienvenida_a_MenuPrincipal()` - Sincroniza la vista de bienvenida con la vista de menú principal
- `syncVistaMenuPrincipal_a_Bienvenida()` - Sincroniza la vista de menú principal con la vista de bienvenida
- `syncVistaGestionAlfabetos_a_MenuPrincipal()` - Sincroniza la vista de gestión de alfabetos con la vista de menú principal
- `syncVistaMenuPrincipal_a_GestionAlfabetos()` - Sincroniza la vista de menú principal con la vista de gestión de alfabetos
- `syncVistaGestionPerfil_a_MenuPrincipal()` - Sincroniza la vista de gestión de perfil con la vista de menú principal
- `syncVistaGestionPerfil_a_Bienvenida()` - Sincroniza la vista de gestión de perfil con la vista de bienvenida
- `syncVistaMenuPrincipal_a_GestionPerfil()` - Sincroniza la vista de menú principal con la vista de gestión de perfil

- `syncVistaMenuPrincipal_a_GestionTeclados()` - Sincroniza la vista de menú principal con la vista de gestión de teclados
- `syncVistaGestionTeclados_a_MenuPrincipal()` - Sincroniza la vista de gestión de teclados con la vista de menú principal
- `syncVistaGestionPerfil_a_CambiarContrasena()` - Sincroniza la vista de gestión perfil a la vista de cambiar contraseña
- `syncVistaCambiarContrasena_a_GestionPerfil()` - Sincroniza la vista de cambiar contraseña a la vista de gestión perfil
- `iniciarSesion(String username, String password)`: Inicia sesión para el usuario con el nombre de usuario y contraseña proporcionados.
- `cerrarSesion()`: Cierra la sesión del usuario actual.
- `crearUsuario(String username, String password)`: Crea un nuevo usuario con el nombre de usuario y contraseña proporcionados.
- `boolean existeUsuario(String username)`: Verifica si un usuario con el nombre de usuario proporcionado existe.
- `eliminarTeclado(String nombreTeclado)`: Elimina un teclado asociado al usuario actual.
- `ArrayList<String>getNombreTeclados()`: Obtiene la lista de nombres de los teclados asociados al usuario actual.
- `ArrayList<String>getNombreAlfabetos()`: Obtiene la lista de nombres de los alfabetos asociados al usuario actual.
- `ArrayList<String>getNombreAlgoritmos()`: Obtiene la lista de nombres de los algoritmos asociados al usuario actual.
- `crearTeclado(String nombreTeclado, String nombreAlfabeto, String texto, String palabrasConFrecuencia, String nombreAlgoritmo)`: Crea un nuevo teclado con los parámetros proporcionados y lo asocia al usuario actual.
- `intercambiarTeclasTeclado(String nombreTeclado, char tecla1, char tecla2)`: Intercambia dos teclas en el teclado especificado por nombre.
- `Character[][] getDistribucionTeclado(String nombreTeclado)`: Obtiene la distribución del teclado especificado por nombre.
- `crearAlfabeto(String nombreAlfabeto, String caracteresAlfabeto)`: Crea un nuevo alfabeto con el nombre y caracteres especificados.
- `eliminarAlfabeto(String nombreAlfabeto)`: Elimina el alfabeto especificado por nombre.
- `modificarCaracteresAlfabeto(String nombreAlfabeto, String caracteresAlfabeto)`: Modifica los caracteres del alfabeto especificado por nombre.

- `ArrayList<Character>getCaracteresDeAlfabeto(String nombreAlfabeto)`: Obtiene los caracteres asociados al alfabeto especificado por nombre.
- `cargarDatosUsuario()`: Carga los teclados y alfabetos asociados al usuario actual.
- `guardarDatos()`: Guarda los teclados, alfabetos y usuarios asociados al usuario actual.
- `String importarTexto(String filePath)`: Importa un texto desde un archivo.
- `String importarListaPalabras(String filePath)`: Importa una lista de palabras desde un archivo.
- `cambiarContrasena(String oldPass, String newPass)`: Cambia la contraseña del usuario actual.
- `borrarCuenta()`: Elimina la cuenta del usuario actual.
- `cerrarPrograma()`: Cierra el programa.

2.1.10. Diseño de la interfaz

A continuación, se presenta el prototipo creado para llevar a cabo el diseño de la interfaz gráfica. Es de importancia destacar que es un prototipo inicial, es por ello que hay ciertas características/funciones que podrían cambiar según la posibilidad y viabilidad de alcanzar esos objetivos con el método facilitado para implementar. Cabe destacar, que hay ciertos puntos que se encuentran fase de desarrollo como puede ser el nombre, logotipo y portada, como se podrá observar, en algunas de las vistas de la interfaz gráfica salen marcado pero no implementados.

Figura 3: Vista Inicial, Crear Cuenta, Iniciar Sesión y Menú Principal

En la figura anterior hay cuatro vistas, la primera vista es la que usuario encontrará por el simple hecho de ejecutar el programa. Seguidamente, podrá seleccionar la opción de Iniciar Sesión o Crear Cuenta. Una vez dentro de Iniciar Sesión o Crear Cuenta puede volver a Iniciar Sesión, por el hecho de recordar que tenía una cuenta previa, o de Iniciar Sesión volver a Crear Cuenta si recuerda que no tenía una cuenta creada. Una vez que haya logrado Iniciar Sesión, pasa al Menú Principal dónde podrá gestionar alfabetos o teclados.

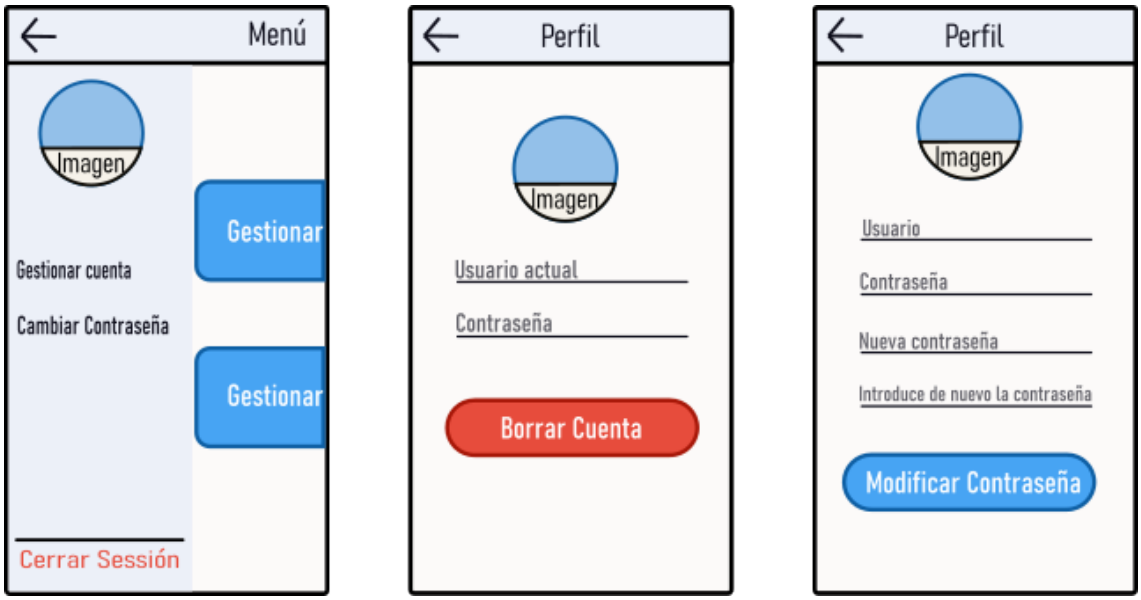

Figura 4: Vista Gestionar Perfil, Gestionar Cuenta y Cambiar Contraseña

En la figura 3, se muestra el diseño de la gestión del usuario, donde puede realizar la gestión de cuenta y cambio de contraseña, tal como muestran las últimas dos vistas de la figura. Cabe recalcar que hemos decidido separar, gestionar cuenta respecto a cambiar contraseña debido a que tenemos un caso de uso que es borrar cuenta y este es un tipo de acción que no se puede deshacer y es importante que el usuario sea informado de esta acción y su implicación en la cuenta. Tampoco se descarta la viabilidad de añadir esa opción en una vista que englobe gestionar cuenta y dentro de esta hubiera la función de cambiar contraseña y eliminar cuenta.

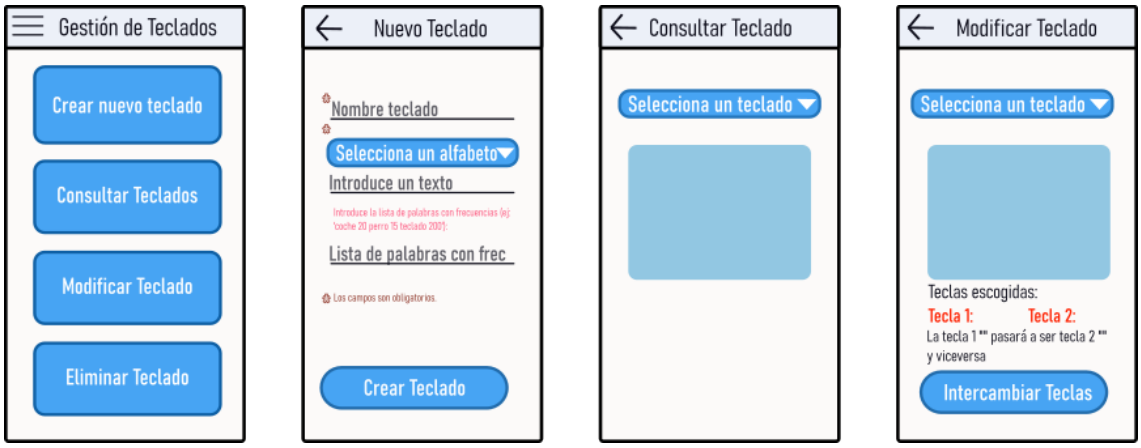

Figura 5: Vista Gestionar Teclados, incluye la vista de Crear Teclado Nuevo, Consultar Teclado y Modificar Teclado

Figura 6: Vista Eliminar Teclado

En cuanto a la parte de gestión de teclados, se ha decidido añadir por conveniencia propia los 4 botones principales, crear nuevo teclado, consultar teclado existente, modificar teclado existentes y eliminar teclado existente (Figura 5). Hay un caso de uso que no se ha implementado en la parte de diseño gráfico, es el de consultar todos los nombres del teclado. Dicho caso de uso es útil si el usuario quiere consultar solo los nombres, pero la misma función lo puede hacer mediante el desplegable de la vista de Consultar Teclados. El rectángulo que aparece del color azul saturado, indica que se realizará la impresión del teclado en esa región de la pantalla.

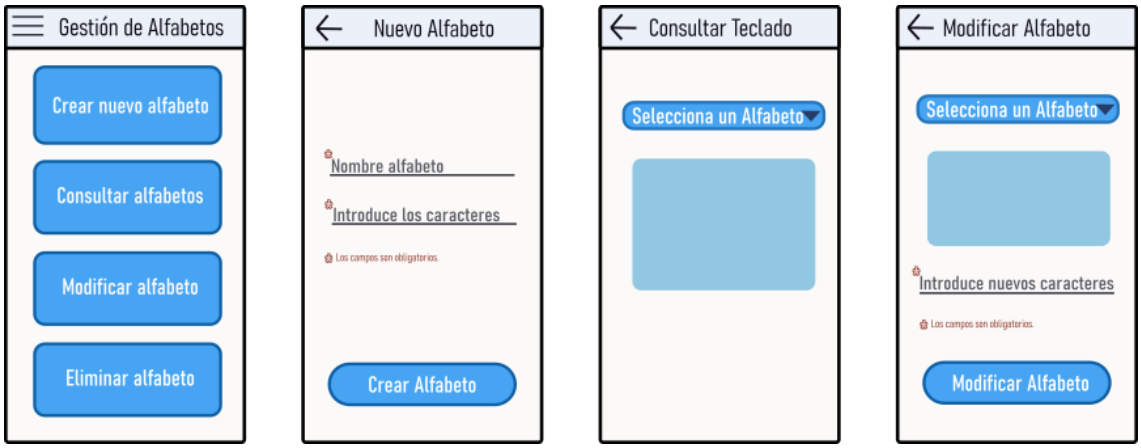

Figura 7: Vista Gestionar Alfabetos, Crear Nuevo Alfabeto, Consultar Alfabetos y Modificar Alfabetos

Figura 8: Vista Eliminar Alfabeto

Otra de las funciones importante que el programa tiene es la parte de gestionar alfabetos, son vistas casi idénticas a la del teclado, ya que los casos de uso que implementa éste son muy parecidos al de teclado pero aplicado a alfabetos.

Figura 9: Paleta de colores para la interfaz

En la figura anterior se puede observar conjunto de colores, estos son todos los colores que se han utilizado para implementar el diseño, los mas densos/fuertes como viene siendo el caso de la segunda fila, son los colores que se utilizan para el contorno de los botones. El valor que sale en la imagen corresponde al formato RGBA.

2.2. Clases y controladores de la capa de dominio

En la tercera entrega, se optó por suprimir la relación entre usuarios y sus conjuntos de teclados y alfabetos en la capa de dominio. Esta elección se sustentó en la observación de que, al restringir el programa a una única sesión de usuario activa, siempre se accedería a los ficheros de datos correspondientes a un único usuario al iniciar. Por lo tanto, la vinculación de usuarios con sus conjuntos de teclados y alfabetos resultó innecesaria, simplificando así el código y mejorando su eficiencia.

Cabe destacar que esta modificación introduce una falta de conexidad en el dominio, ya que la relación previa proporcionaba una estructura más completa pero no utilizada.

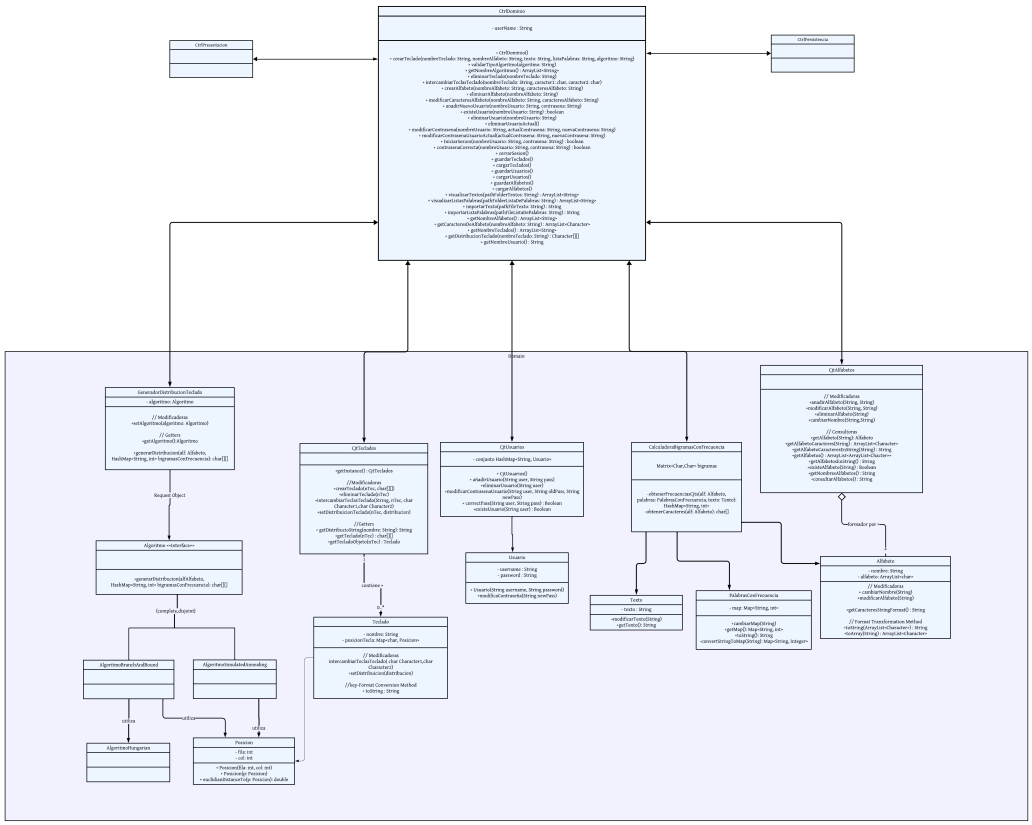

Figura 10: Diagrama de clases de la capa de dominio

Representa un teclado a partir de un String, que contiene su nombre identificador, y una matriz de caracteres que guarda la distribución de sus teclas. Además de los

getters y setters, contiene una función *intercambiarTeclas* que gira la posición de dos de las teclas del teclado. Por último, contiene un método redefinido *toString* que devuelve la matriz de caracteres en formato de String para pasar y mostrarlo a la capa de presentación.

2.2.1. CjtTeclados

Representa un conjunto de teclados, y proporciona funciones para gestionarlos. El conjunto de teclados está contenido en un HashMap<String,Teclado>, donde String es el identificador del teclado y Teclado el propio objeto. Además de sus getters y setters, iene una función modificadora *intercambiarTeclasTeclado* que, en base a dos índices de dos teclas del teclado, gira su posición.

2.2.2. Alfabeto

Representación de un alfabeto, con un nombre identificativo y una lista de caracteres. Ofrece métodos para cambiar el nombre del alfabeto, modificar su conjunto de caracteres y obtener información sobre él, como el nombre y la lista de caracteres en formatos de lista y cadena. Los métodos de conversión de formatos permiten transformar entre una cadena de caracteres y una lista, facilitando operaciones de entrada y salida.

2.2.3. CjtAlfabetos

Representa una colección de alfabetos en un sistema. Utiliza un mapa para almacenar objetos Alfabeto asociados con identificadores únicos (nombres). La clase proporciona métodos para añadir, eliminar, modificar y consultar alfabetos, así como obtener información detallada sobre ellos, como los caracteres que contienen. Además, incluye funcionalidades para verificar la existencia de un alfabeto y para obtener representaciones en cadena de los alfabetos contenidos en la colección.

2.2.4. Texto

La clase Texto es una clase diseñada para operar sobre el texto, uno de los parámetros en que se basa la generación del teclado. Contiene un atributo texto que almacena la cadena de palabras que un teclado utiliza, además incluye métodos para obtener este texto en formato de String y poder realizar modificaciones sobre este. Permite la construcción de sus instancias con un texto vacío o con uno específico.

2.2.5. PalabrasConFrecuencia

La clase PalabrasConFrecuencia representa una colección de palabras y sus frecuencias. Se puede construir con un texto vacío o con un texto que contiene una secuencia

se palabra y su frecuencia separados por espacio. El formato del texto para la creadora se puede proporcionar como "$< pal1 > < frec1 > < pal2 > < frec2 >$...".

Concretamente, para la implementación de esta colección se ha usado un LinkedHashMap porque 1. Linked: queremos mantener el orden específico de las palabras y sus frecuencias tal como se proporcionaron en el texto de entrada para que, si en el futuro, permitimos que el usuario pueda modificarlo, aparezcan en el mismo orden en el que las introdujo y 2. HashMap: para que podamos guardar pares String, Integer y que sus consultas sean constantes.

Además, la clase proporciona métodos para obtener el mapa de frecuencias, cambiar el mapa de frecuencias y obtener el texto en el que se basa.

2.2.6. Posición

La clase Posicion representa una posición en una matriz. Esta implementado mediante dos enteros como atributo, uno para representar la fila y la otra, la columna. Además, proporciona un método para calcular la distancia euclidiana entre dos posiciones.

2.2.7. CalculadoraBigramasConFrecuencia

La clase CalculadoraBigramasConFrecuencia representa una clase que permite calcular la frecuencia de bigramas de caracteres que hay tanto en el texto como en palabras con frecuencia, especificados para la generación del teclado. Posteriormente estos valores se utilizarán como entrada para el algoritmo con el fin de crear la distribución óptima. En cuanto a la estructura de la clase, se utiliza un HashMap¡String,Integer¿como atributo privado,para poder computar la frecuencia de pares de caracteres. La estructura elegida para la implementación es la óptima en este caso, ya que se requiere acceder a los valores varias veces, y esta estructura nos permite realizar consultas en tiempo casi constante, en peor de los casos es lineal en cuanto a los elementos que hay. Sin embargo, si se hubiera empleado una matriz con j=0 identificador y j=1 número de frecuencias, esta implementación hubiera llevado un coste temporal en número de filas por una consulta. Por otra parte, la escritura en matriz es constante y en el caso de HashMap pasa lo mismo, en peor de los casos es lineal al número de elementos.

La clase tiene un único método público ejecutar(Alfabeto alf, PalabrasConFrecuencia palabras, Texto texto), en primer lugar, esta función empieza inicializando el mapa de bigramas y su frecuencia iterando sobre el alfabeto, y para cada par de caracteres diferentes, se añade al mapa con frecuencia 0. Esto tiene un coste temporal de $O(|caracteres|^2)$, ya que por cada carácter lo empareja con su consecutivo. De forma que si hay tres caracteres,{a ,b ,c} la combinación será; {ab, ac, bc}. Es importante aclarar que para cualquier bigrama, consideramos en nuestra estructura de datos *ab* igual que *ba* por ejemplo. Seguidamente la función llama a dos funciones. La función contarBigramasTexto(String texto) para iterar sobre el texto y sacar las frecuencias.

Esto se consigue con un índice en la posición i y otro en la posición i+1, empezando desde i = 0. Para cada iteración, tendremos un bigrama del cual aumentaremos en uno la frecuencia en el mapa. Coste temporal de esta función es $O(|texto|)$. La función contarBigramasListaPalabras(palabras.getMap()), en esta función se itera sobre cada palabra con su frecuencia, y para cada palabra, lo tratamos como si fuera un texto, como en el paso anterior, pero ahora, en vez de sumar en 1 su frecuencia, sea x la frecuencia de la palabra, sumamos x a la frecuencia de cada bigrama de esa palabra. Coste temporal de la función $O\left(\sum_{\text{palabra}\in\text{PalabrasConFrecuencia}} |palabra|\right)$. Finalmente, la función devuelve el mapa de bigramas con frecuencias calculadas.

2.2.8. CjtUsuarios

La clase CjtUsuarios en Java representa un conjunto de usuarios identificados por nombres únicos y contraseñas. Su implementación incluye métodos para añadir, eliminar y modificar usuarios, así como verificar la validez de contraseñas y la existencia de usuarios en el conjunto. Utiliza un patrón Singleton para garantizar una única instancia de la clase y está diseñada para gestionar operaciones relacionadas con usuarios de manera eficiente, lanzando excepciones específicas en casos de errores.

2.2.9. Usuario

La clase Usuario en Java representa un usuario con su nombre de usuario y contraseña. Su constructor valida que el nombre de usuario no esté vacío, y la contraseña tenga al menos 4 caracteres. La clase ofrece métodos para modificar la contraseña, obtener la contraseña y el nombre de usuario.

2.2.10. GeneradorDistribucionTeclado

La clase GeneradorDistribucionTeclado se usa para crear distribuciones de teclado y aprovecha el patrón de diseño Estrategia. Mantiene un Algoritmo como atributo siendo este el algoritmo en específico que utiliza. Este patrón permite cambiar el algoritmo de generación de distribuciones de teclado en tiempo de ejecución mediante el método setAlgoritmo().

2.2.11. Algoritmo

La clase Algoritmo se utiliza para ejecutar el algoritmo de generación de distribuciones de teclado y devolverlas. Es una interfaz que define el método abstracto generarDistribucion(...), permitiendo la implementación de varios algoritmos siguiendo el patrón de diseño Estrategia. Las clases que heredan de ella deben implementar este método.

2.2.12. AlgoritmoBranchAndBound

La clase Algoritmo Branch and Bound es una subclase de Algoritmo que representa la implementación de un algoritmo Branch and Bound con cota Gilmore-Lawler para crear la distribución más óptima para un teclado dados un Alfabeto, Texto y PalabrasConFrecuencia.

Implementa el método generarDistribucion(...) heredado de Algoritmo para este propósito. Más específicamente, calcula una distribución óptima de caracteres en un teclado en función de las frecuencias entre caracteres y las distancias entre posiciones. La distribución resultante minimiza el coste total de asignar caracteres a posiciones en el teclado resolviendo un problema QAP.

Contiene un método privado que procesa el input para adaptarlo a un problema QAP. Los otros métodos privados que implementa son cada uno de los pasos necesarios para resolver el problema QAP y obtener la distribución óptima.

2.2.13. AlgoritmoHungarian

La clase AlgoritmoHungarian representa una implementación del algoritmo húngaro utilizado para resolver el problema de asignación, que busca encontrar la asignación óptima entre dos conjuntos de elementos minimizando el coste total. El algoritmo toma una matriz de costes como entrada y devuelve el coste mínimo de la asignación óptima. La clase incluye métodos para ejecutar el algoritmo, realizar preprocesamiento en la matriz de costes y realizar cálculos necesarios para encontrar la asignación óptima.

2.2.14. AlgoritmoSimulatedAnnealing

La clase AlgoritmoSimulatedAnnealing utiliza el algoritmo de "Simulated Annealing"para generar una solución optimizada (no óptima) del problema de generar un teclado en base a un alfabeto, texto y lista de palabras.

Al igual que el AlgoritmoBranchAndBound ya visto, utiliza un método privado $inicializarFrecuenciasCjts$ para crear el mapa de los bigramas y sus frecuencias, el resto de funciones se destina a inicializar las estructuras necesarias y el cálculo del coste de las soluciones hasta encontrar una solución que, si bien no será la óptima, será muy buena.

2.2.15. ControladorDominio

La clase controlador dominio es un intermediario entre la interfaz de usuario y el sistema. Con el fin gestionar diferentes solicitudes que el usuario ha pedido, la clase está equipada de tres atributos privados CjtAlfabetos, CjtTeclados y CjtUsuarios que se inicializan con el método de inicializarCtrlDominio().

Este controlador permite la gestión de teclados y alfabetos al comunicarse con diversas clases. Sus funciones permiten consultas, creación, modificación y eliminación de teclados y alfabetos. También incorpora una gestión de excepciones que permite una comunicación directa con el usuario, informándole sobre posibles errores durante la ejecución de sus peticiones.

También, para cargar y guardar información para usos posteriores, incorpora un mecanismo de persistencia a través de CtrlPersistencia para guardar y cargar datos del sistema. Además, cuenta con funcionalidades de autenticación de usuarios, permitiendo a los usuarios iniciar sesión, modificar sus contraseñas y gestionar sesiones.

2.3. Clases y controladores de la capa de persistencia

En esta sección, se han introducido modificaciones con respecto a la segunda entrega. Principalmente, se identificaron comportamientos similares en los Gestores de Ficheros, y se optó por aplicar el patrón de diseño estrategia para abordar esta cuestión. Esto conlleva la incorporación de una nueva clase denominada GestorEstrategia.

Además, se ha agregado la funcionalidad de Ïmportar/Cargar"textos y listas de palabras, algo que no estaba presente anteriormente. Para implementar esta funcionalidad, se ha añadido la clase CargarYVisualizar. Estos cambios están destinados a mejorar la modularidad y la flexibilidad del sistema, permitiendo una gestión más eficiente de los diferentes tipos de archivos y simplificando la incorporación de nuevas estrategias para la carga de datos.

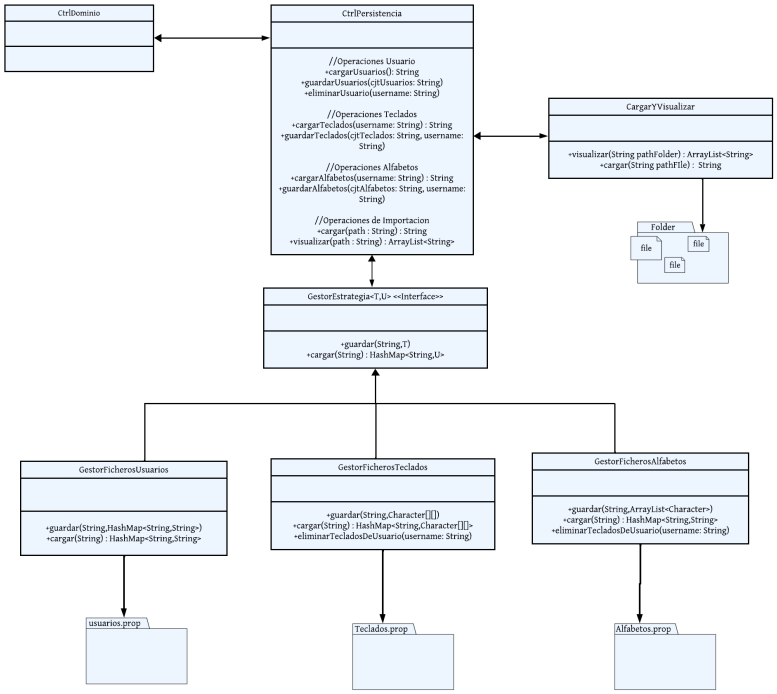

Figura 11: Diagrama de clases de la capa de persistencia

2.3.1. CargarYVisualizar

La clase `CargarYVisualizar` se encarga de proporcionar métodos para cargar el contenido de un archivo y visualizar los archivos en una carpeta. Su función principal es facilitar la lectura de archivos y la obtención de la lista de nombres de archivos en una carpeta. La metodología de la clase incluye:

- `cargar(String pathFile)`: Este método carga el contenido de un archivo ubicado en la ruta especificada y lo devuelve como una cadena de texto. Utiliza un `BufferedReader` para leer el archivo línea por línea y maneja la excepción `LecturaIncorrectaFicheroExcepcion` en caso de errores durante la lectura.
- `visualizar(String pathFolder)`: Visualiza los archivos en la carpeta especificada mediante la obtención de sus nombres y los devuelve en forma de lista. La función utiliza un `File` para explorar la carpeta y construye la lista con los nombres de los archivos encontrados.

La clase contribuye a la gestión de archivos al ofrecer funciones esenciales para cargar contenido y obtener información sobre los archivos presentes en una carpeta, lo que puede ser útil en diversos contextos de aplicación.

2.3.2. GestorEstrategia

La interfaz `GestorEstrategia` define un conjunto de métodos destinados a gestionar la escritura y lectura de datos en un fichero. Diseñada para trabajar con dos tipos de datos genéricos, `T` y `U`, representa un mecanismo flexible para la persistencia de información asociada a un usuario. La interfaz contiene los siguientes métodos fundamentales:

- `guardar(String username, HashMap<String, T>datos)`: Este método se encarga de guardar datos asociados a un usuario específico. Utiliza un mapa (`HashMap`) para almacenar información identificada por claves y maneja la excepción `EscrituraIncorrectaFicheroExcepcion` en caso de errores durante la escritura del fichero.
- `cargar(String username): HashMap<String, U>`: Realiza la carga de datos previamente guardados para un usuario dado. Devuelve un mapa (`HashMap`) con la información cargada y maneja la excepción `LecturaIncorrectaFicheroExcepcion` en caso de fallos durante la lectura del fichero.

Esta interfaz proporciona una abstracción general para las operaciones de persistencia de datos, permitiendo adaptarse a diferentes tipos de información y proporcionando una estructura común para la gestión de escritura y lectura de ficheros en el contexto de la aplicación.

2.3.3. GestorFicherosAlfabetos

La clase `GestorFicherosAlfabetos` se encarga de gestionar la lectura y escritura de archivos relacionados con alfabetos para un usuario específico. Implementa la interfaz `GestorEstrategia`, proporcionando métodos especializados para manejar colecciones de alfabetos representados por listas de caracteres. Algunas características clave de esta clase son:

- **Guardado de Alfabetos:** El método `guardar` almacena los alfabetos asociados a un usuario específico en un archivo. Cada alfabeto se guarda con su nombre y una representación de sus caracteres. Se utiliza una estrategia de persistencia que convierte la lista de caracteres en una cadena de texto, marcando los espacios con el símbolo "".
- **Carga de Alfabetos:** El método `cargar` recupera los alfabetos previamente guardados para un usuario dado. Realiza la inversa del proceso de guardado, convirtiendo la cadena de texto almacenada en el archivo de nuevo en una lista de caracteres.
- **Eliminación de Alfabetos:** El método `eliminarAlfabetosDeUsuario` elimina los archivos de alfabetos asociados a un usuario, proporcionando una funcionalidad para gestionar la persistencia de datos.

Esta clase forma parte del sistema de persistencia diseñado para garantizar la conservación y recuperación de información relevante para el funcionamiento del programa, proporcionando una capa de abstracción para la manipulación de archivos relacionados con alfabetos.

2.3.4. GestorFicherosTeclados

La clase `GestorFicherosTeclados` gestiona la escritura, lectura y eliminación de teclados en ficheros, implementando la interfaz `GestorEstrategia` para proporcionar métodos específicos para manejar colecciones de teclados representados por matrices de caracteres. Algunas características destacadas de esta clase son:

- **Guardado de Teclados:** El método `guardar` almacena los teclados asociados a un usuario en un archivo. Cada teclado se guarda con su nombre y una representación de su distribución en una matriz de caracteres. Se utiliza una estrategia de persistencia que convierte la distribución de teclado en una cadena de texto, marcando los espacios con el símbolo "".
- **Carga de Teclados:** El método `cargar` recupera los teclados previamente guardados para un usuario dado. Realiza la inversa del proceso de guardado, convirtiendo la cadena de texto almacenada en el archivo de nuevo en una matriz de caracteres que representa la distribución del teclado.
- **Eliminación de Teclados:** El método `eliminarTecladosDeUsuario` elimina los archivos de teclados asociados a un usuario, proporcionando una funcionalidad para gestionar la persistencia de datos.

Esta clase forma parte del sistema de persistencia diseñado para garantizar la conservación y recuperación de información relevante para el funcionamiento del programa, ofreciendo una capa de abstracción para la manipulación de archivos relacionados con teclados.

2.3.5. GestorFicherosUsuarios

La clase `GestorFicherosUsuarios` se encarga de gestionar la escritura y lectura de usuarios en un fichero, implementando la interfaz `GestorEstrategia`. Algunos aspectos destacados de esta clase son:

- **Guardado de Usuarios:** El método `guardar` almacena los usuarios en un archivo de usuarios. Cada usuario se guarda con su nombre y contraseña, utilizando el símbolo "ºçomo delimitador. La información se almacena en un formato que permite su posterior recuperación.
- **Carga de Usuarios:** El método `cargar` recupera los usuarios previamente guardados desde el archivo. Se realiza la lectura del archivo y se construye un conjunto de usuarios con los datos obtenidos. En caso de que el archivo no exista, se crea uno nuevo y se maneja la excepción correspondiente.

La gestión de usuarios es esencial para el funcionamiento del sistema, ya que proporciona una capa de persistencia para almacenar y recuperar la información de autenticación de los usuarios. La implementación de esta clase contribuye a la robustez y continuidad de la aplicación, asegurando que los datos de los usuarios estén disponibles incluso entre distintas sesiones de ejecución del programa.

3. Algoritmos y estructuras de datos

En esta sección, vamos a explicar los dos algoritmos implementados (y sus respectivos algoritmos internos, también importantes) para generar una distribución óptima de caracteres para un teclado, con la distribución siendo representada por una matriz de caracteres. Cualquiera de los dos algoritmos, parte de un alfabeto (implementado con un ArrayList de caractéres), un texto (String) y una lista de palabras con frecuencia ($HashMap < String, int >$).

3.1. Algoritmo Simulated Annealing

En primer lugar, tenemos el algoritmo de Simulated Annealing, un algoritmo de inteligencia artificial que, en base a una solución inicial y un procedimiento estocástico de generación de nuevas soluciones, busca una solución optimizada al problema (probablemente no la óptima).

Se escoge una temperatura T inicial, la cual decrecerá exponencialmente hasta cierto valor, y se escoge un número de iteraciones fijo por temperatura. En cada una de estas iteraciones, se realiza un swap de dos posiciones de la mejor solución actual (factor de ramificación: O(n^2), donde n es el número de teclas) y se verá si esta mejora la que tenemos guardada. Si es así, se guardará como nueva mejor solución. Si no es así, se escogerá igualmente con una probabilidad

$$e^{(mejorCosteTotal - costeActual)/T}$$

. Esto nos permite evitar máximos locales durante la búsqueda de soluciones.

El pseudocódigo del algoritmo es el siguiente:

Algoritmo 1 Simulated Annealing

```
Función SimulatedAnnealing():
    mejorActual ← array of size n
    for i ← 0 to n − 1 do
        mejorActual[i] ← i
    end
    costeActual ← calculoCoste(mejorActual)
    if costeActual < mejorCosteTotal then
        mejorCosteTotal ← costeActual
    end
    T ← 100
    iters ← 10000
    valActual ← array of size n
    rand ← new Random Number Generator  while T > 1.0 do
        for i ← 0 to iters − 1 do
            costeActual ← 0.0  (copia elementos de mejorActual a valActual)
            a ← rand.nextInt(n)  b ← rand.nextInt(n)
            while b = a do
                b ← rand.nextInt(n)
            end
            temp ← mejorActual[a]
            valActual[a] ← mejorActual[b]
            valActual[b] ← temp
            costeActual ← calculoCoste(valActual)
            if costeActual < mejorCosteTotal then
                mejorCosteTotal ← costeActual  (copia elementos de valActual a
                mejorActual)
            end
            else
                prob ← pow(e, (mejorCosteTotal − costeActual)/T)  if prob >
                Math.random() then
                    mejorCosteTotal ← costeActual  (copia elementos de valActual
                    a mejorActual)
                end
            end
        end
        T ← T × 0.9
    end
    for i ← 0 to n − 1 do
        mejorDistribucion[caracteres[i]] ← mejorActual[i]
    end
```

Sea x el factor de reducción de temperatura, T la temperatura usada, I el número de iteraciones por temperatura y n el número de caractéres del alfabeto. En el bucle más interno tenemos un coste $O(logn)$ debido a la copia del array de mejor coste. Este coste se encuentra en un for loop de I iteraciones, el cual, además, se

encuentra en un while loop ejecutado $|\log_x T|$ veces. En total, nos queda un coste de $O(|\log_x T| * I * n)$.

Posicion[] posiciones: array de las posiciones guardadas en el teclado

double[][] distancias: Por cada posición (i,j), se guarda la distancia euclidiana entre estas dos posiciones i y j del teclado.

Se ha elegido representarlo como una matriz debido a que la operación de consulta de la distancia entre ubicaciones es muy elevada en el algoritmo, y con esta estructura, el coste de esta consulta es constante. Además, mantenerlo guardado en la matriz nos evita el recalcular cada vez esta distancia cada vez que queramos consultarla, ya que una misma distancia es consultada muchas veces durante el algoritmo.

char[] caracteres: array que guarda los caractéres a utilizar.

int numCaracteres: El número de caractéres de nuestro teclado.

int rows, cols: Entero que almacena las filas y columnas de la distribución a general.

double mejorCosteTotal Double que almacenan el coste de la mejor distribución hasta ese momento.

Map<character, Integer> mejorDistribucion: Mapa que guarda la distribucion para luego transformarla en la matriz de retorno. La razón de usar un Map es por su facilidad para actualizar la posición en la que guardamos los caractéres y su coste de acceso (O(log numCaracteres)).

Random rand: Generador de números aleatorio utilizado para crear un algoritmo estocástico, se utiliza la semilla por defecto, que es el tiempo actual.

double costeIni: double utilizado unicamente en tests para observar que las distribuciones generadas son mejores que las iniciales.

Para este algoritmo, se requiere un número de inicializaciones, que se realizan en funciones separadas:

inicializarPosiciones: inicializa el array de posiciones y calcula las filas y columnas necesarias. Coste: $O(numCaracteres)$.

inicializarDistancias: inicializa la matriz de distancia entre posiciones. Coste: $O(numPosiciones)$.

inicializarCaracteres(Alfabeto alf): inicializa el array de caractéres a partir del alfabeto pasado por parámetro. Coste: $O(numCaracteres)$.

El resto de funciones tienen coste $O(lognumCaracteres)$, ya que requieren un acceso al mapa de bigramas con frecuencia.

calcularCosteEntreDosCaracteres(char c1, int posIndex1, char c2, int posIndex2): calcula el coste entre dos caracteres $c1$ y $c2$ en sus respectivas posiciones $posIndex1$ y $posIndex2$.

String calcularKey(char c1, char c2): Calcula la clave para la frecuencia de bigramas.

La función *calculoCoste* hace uso de la frecuencia de los bigramas calculados anteriormente. Para cada par de letras en el teclado, se calcula su coste como el producto entre su distancia euclidiana y la frecuencia de ese bigrama. El sumatorio de todos esos costes es el coste total que se devuelve.

HashMap<String, Integer> bigramasConFrecuencia: HashMap que contiene todos los bigramas posibles del alfabeto (exceptuando aquellos iguales invertidos, ej: .ªbƶ "ba") y sus frecuencias de aparación en el texto y lista de palabras de entrada. Se utiliza un HashMap para que la consulta de la frecuencia de un bigrama sea constante.

3.2. Algoritmo Branch and Bound

En esta sección, se explicará el funcionamiento de la clase AlgoritmoBranchAndBound para generar la distribución óptima de un teclado.

Antes de empezar, describimos como distribución óptima de un teclado aquella que minimiza la distancia entre dos caracteres en proporción al número de veces (frecuencia) que se teclean uno al lado de la otra.

Por lo tanto, el objetivo de esta clase es obtener una distribución de un teclado que optimice estas propiedades dados los caracteres y las frecuencias entre ellas.

El método principal de la clase es *generarDistribución*, que recibe como parámetros un Alfabeto (contiene los caracteres que tendrá el teclado) y un $HashMap < String, Int >$ de bigramas y sus frecuencias (siendo los bigramas aquellos formados a partir del Alfabeto dado) donde se ha utilizado la clase *CalculadoraBigramasConFrecuencia* para calcularla, y retorna la distribución óptima del teclado para estos parámetros.

Como el problema que nos concierne es equivalente a un QAP (Quadratic Assignment Problem), donde en vez de asignar instalaciones a ubicaciones según su tránsito, asignamos caracteres a posiciones en la distribución del teclado según su frecuencia, los procedimientos implementados tratarán de adaptarlo a la resolución de un QAP de forma general, la cual permitirá resolver un QAP cualquiera y que, además, resuelve nuestro problema en concreto.

Estos procedimientos son:

3.2.1. Procesamiento de los parámetros

El algoritmo empieza procesando el Alfabeto y bigramasConFrecuencia para adaptarlo a estructuras de datos que nos facilitan la resolución del problema.

En primer lugar, se inicializa un vector que contiene los n caracteres del Alfabeto.

A continuación, se crea una matriz de $frecuencias$ en la que cada elemento *frecuencias[i][j]* contiene la frecuencia entre el carácter i y el carácter j del vector de

caracteres, donde $0 \leq i < n$ y $0 \leq j < n$, consultando las frecuencias de bigramas anteriores.

A continuación, se inicializa un vector de Posiciones, que contiene las n posiciones del teclado, y con ello, se construye una matriz de distancias en la que cada elemento *distancias[i][j]*, con $0 \leq i < n$ y $0 \leq j < n$, contiene la distancia euclidiana entre la posición i y la posición j del vector de posiciones.

Con todo esto inicializado, disponemos de todos los elementos necesarios para abordar la resolución del problema.

A partir de esta sección, en vez de caracteres y posiciones, hablaremos de instalaciones y ubicaciones para dar una descripción más general de la resolución del problema QAP.

3.2.2. Branch and Bound

La función *calcularAsignacionOptima* calcula la asignación óptima de instalaciones a ubicaciones y determina su coste total en base a las matrices de frecuencias y distancias proporcionadas. Este método en concreto, se encarga de inicializar los parámetros necesarios para ejecutar la llamada a la función recursiva "backtracking" que resuelve el problema.
A continuación, se explican las estructuras de datos utilizadas y sus motivos:

- **int n:** es el número de ubicaciones, que es igual al número de instalaciones.
- **double[][] distancias:** matriz de double tal que para cada elemento distancias[i][j] donde $0 <= i < n$, $0 <= j < n$, contiene la distancia euclidiana entre la ubicación i y la ubicación j.

  La justificación es la misma que la expuesta en Algoritmo Simulated Annealing.
- **int[][] frecuencias:** matriz de enteros tal que para cada elemento frecuencias[i][j] donde $0 <= i < n$, $0 <= j < n$, contiene la frecuencia entre la instalación i y la instalación j.

  La justificación del uso de una matriz para guardar esta información es la misma que la de la matriz *distancias*.
- **HashMap<int, int> mapAsignacionActual:** mapa que guarda pares de índice de instalación e indice de ubicación, representando la asignación de la instalación a esa ubicación.

  Se ha decidido utilizar un HashMap porque como la asignación de instalaciones en el algoritmo es arbitraria, es necesario comprobar si una instalación ha sido asignada en la asignación actual antes de considerarla para la posible solución (se realiza $O(n)$ veces para cada llamada recursiva), y gracias a esta estructura de datos, esta consulta tiene coste constante. Además, nos permite iterar sobre las instalaciones asignadas, y las operaciones de inserción y borrado también son constantes, que son realizadas de manera frecuente en el algoritmo.

- **double costeActual:** representa el coste de la asignación actual contenida en *mapAsignacionActual*.
- **int ubicacionActual**: representa la ubicación a asignar.
- **HashMap<int, int> mapMejorAsignacion:** representa lo mismo que *mapMejorAsignacion* pero para guardar la mejor asignación, que es la que tiene el coste mínimo, que se ha encontrado hasta el momento.
- **double mejorCosteTotal:** representa el valor de coste de la mejor asignación encontrada.

A continuación, se muestran las funciones que implementan el algoritmo en pseudocodigo y luego, su explicación:

Algoritmo 2 calcularAsignaciónOptima

Input : Matriz de Frecuencias $frecuencias$, Matriz de Distancias *distancias*
Output: Coste total de la asignación óptima

Función *calcularAsignacionOptima(*$frecuencias$[][]*, distancias*[][]*)*:

```
this.n ← frecuencias.length
this.frecuencias ← frecuencias
this.distancias ← distancias
this.mapMejorAsignacion ← new HashMap<Integer, Integer>()
this.mejorCosteTotal ← Double.MAX_VALUE

mapAsignacionActual ← new HashMap<Integer, Integer>()
costeActual ← 0.0
ubicacionActual ← 0

calcularAsignacionOptimaRecursivo(mapAsignacionActual, costeActual,
 ubicacionActual)
return this.mejorCosteTotal
```

Algoritmo 3 calcularAsignacionOptimaRecursivo

Input : HashMap<Integer, Integer> *mapAsignacionActual*, double *costeActual*, int *ubicacionActual*

Output:

Función *calcularAsignacionOptimaRecursivo(mapAsignacionActual, costeActual, ubicacionActual)*:

```
if ubicacionActual = this.n then
    if costeActual < this.mejorCosteTotal then
        this.mapMejorAsignacion ← mapAsignacionActual
        this.mejorCosteTotal ← costeActual
    end
    return
end
for instalacion 0 to this.n − 1 do
    if mapAsignacionActual.containsKey(instalacion) then
        continue
    end
    // Calcular coste actual respecto entre las instalaciones ya emplazadas
     añadiendo la instalacion en la ubicacionActual
     newCosteActual ← (costeActual + calcularCosteDeAsig-
     nar(mapAsignacionActual, instalacion, ubicacionActual))
     mapAsignacionActual.put(instalacion, ubicacionActual)

    // Calcular la cota para decidir si hacer branch
     costeTotalAproximado ← newCosteActual + calcularCosteNoEmplaza-
     dos(mapAsignacionActual, ubicacionActual + 1)

    // Si la cota es peor, podamos
    if costeTotalAproximado ≥ this.mejorCosteTotal then
        mapAsignacionActual.remove(instalacion)
        continue
    end
    // Hacemos llamada recursiva para asignar la siguiente ubicacion
     calcularAsignacionOptimaRecursivo(mapAsignacionActual,
     newCosteActual, ubicacionActual + 1)

    mapAsignacionActual.remove(instalacion)
end
```

La función *calcularAsignacionOptimaRecursivo* implementa un algoritmo Branch and Bound que busca la asignación óptima de elementos a ubicaciones explorando exhaustivamente todas las posibilidades de asignar una instalación a una ubicación. Utiliza una estrategia de búsqueda en profundidad (lazy) para recorrer las opciones.

Para cada rama que se explora, se calcula una cota optimista del coste de la solución obtenible a partir de esta, específicamente, la cota Gilmore-Lawler explicada en el documento "Información adicional sobre QAP" en la carpeta DOCS. Dado que este coste es optimista (siempre menor que el mejor coste de solución para esa rama), si *costeTotalAproximado* es peor que el mejor coste encontrado hasta el momento, se poda esta rama (no se explora). Si exploráramos completamente esa rama, estaríamos seguros de que no obtendríamos una solución mejor.

En cada llamada a esta función, se asigna una instalación que aún no haya sido asignada previamente a la ubicaciónActual, y las llamadas recursivas que se ejecutan hacen lo mismo pero para asignar la siguiente ubicación. Siempre existe el mismo número de instalaciones y ubicaciones por asignar.

El caso base se alcanza cuando se han asignado todas las n ubicaciones a las n instalaciones, y se guarda la solución obtenida si su coste es menor que el coste de la mejor solución encontrada hasta ese momento.

Para el cálculo de la cota Gilmore-Lawler, se ha seguido las directrices del documento "Informació adicional sobre QAP", que explica detalladamente los pasos a hacer y el coste temporal de éste si se implementa siguiendo estos pasos, que es $O(n^3)$. Se han utilizado matrices ya que son necesarias para realizar las operaciones que se mencionan. Finalmente, se utiliza el Algoritmo Hungarian para el paso final que tiene un coste de $O(n^2)$ el cual explicaremos en la siguiente sección.

Esta implementación utiliza los atributos privados de la clase AlgoritmoBranchAndBound para guardar la mejor solución, además de la matriz de $frecuencias$ y la de *distancias* para simplificar el paso de parámetros.

En cuanto al coste temporal, la recurrencia que define la función recursiva tiene la forma: $T(n) = n \cdot T(n-1) + O(n^3)$, donde se hace una llamada recursiva para cada una de las n instalaciones tratando de explorar la solución en que se asigna esa instalación en la *ubicacionActual*, y donde el coste no recursivo viene dado por el cálculo de la cota de Gilmore-Lawler para calcular una cota optimista del coste de esa rama, que es de $O(n^3)$. Por el Teorema Maestro, el coste temporal en el peor caso es de $O(n^{n^3})$, aunque cabe aclarar que el coste en realidad es menor, gracias a las podas que realizamos con la cota, evitando explorar ramas innecesarias.

3.2.3. Algoritmo Hungarian

El Algoritmo Hungarian es un método para resolver problemas de asignación en matrices de costes, particularmente útil en la asignación de tareas o recursos en una matriz de $n \times n$ de manera que se minimice el coste total. En nuestro caso, lo utilizamos para encontrar la asignación óptima de la matriz $C1 + C2$ y así, obtener una cota optimista del coste de la asignación completa.

La implementación realizada sigue completamente las directrices del documento "Informació adicional sobre QAP" proporcionado por los profesores. Por lo tanto, nos enfocaremos en las estructuras de datos utilizadas y el coste temporal.

Estructuras de datos
Debido a la naturaleza inherente del algoritmo, donde cada paso realiza operaciones sobre una matriz de costes, resulta evidente la conveniencia de utilizar matrices, ya que otras estructuras de datos podrían complicar las operaciones que se llevan a cabo.

Las operaciones más comunes incluyen consultas de valores, especialmente para determinar si son ceros o encontrar el mínimo de una fila y/o columna, además de modificaciones en filas, columnas o en toda la matriz.

Para realizar modificaciones, hemos empleado matrices auxiliares, permitiéndonos hacer copias del original. Además, hemos creado versiones simplificadas, como matrices de booleanos, que resultan útiles para simplificar el código de muchos cálculos sin que suponga un coste espacial elevado.

Asimismo, hemos recurrido a vectores, especialmente eficaces para almacenar información adicional sobre las filas y/o columnas de la matriz de costes, como determinar si están cubiertas y/o marcadas (términos explicados en el documento del algoritmo), permitiendo consultas de coste constante.

Costes
Como espacio auxiliar, se generan un número constante de matrices de tamaño n x n o vectores de longitud n, lo que resulta en un coste espacial auxiliar de $O(n^2)$.

En la mayoría de las operaciones, se llevan a cabo un número constante de iteraciones completas sobre la matriz n x n, lo que conlleva un coste temporal de $O(n^2)$. Otra operación realizada es el backtracking para determinar la asignación óptima de ceros. La recurrencia que define su función recursiva adopta la forma: $T(n) = T(n-1) + O(n) = O(n^2)$ según el Teorema Maestro. Por lo tanto, el coste temporal también se establece en $O(n^2)$.

No obstante, dado que el algoritmo finaliza solo cuando el número mínimo de líneas que cubren todos los ceros es igual a n; de lo contrario, repite los cálculos en bucle hasta que se cumple esta condición. Si llamamos k al número de iteraciones necesarias para cumplir esta condición, el coste temporal total del Algoritmo Hungarian será $O(k \cdot n^2)$.

4. Relación clases/miembro

En este apartado se especificarán las clases programadas e implementadas por cada miembro del grupo. Se ha procurado que cada miembro realice una cantidad de trabajo similar, y aquellas clases creadas por más de un miembro han sido escogidas para conseguir esta división equitativa del proyecto.

4.1. Individuales

- **Jianing**: PalabrasConFrecuencia, AlgoritmoBranchAndBound, AlgoritmoHungarian, Posición y sus Tests Unitarios. Vistas relacionadas con teclado.
- **Yasin**: Texto y sus Tests Unitarios. Vistas relacionadas con usuario.
- **Rubén**: Teclado, CjtTeclados, AlgoritmoSimulatedAnnealing y CtrlPresentacion (interfaz) y sus Tests Unitarios. Vistas relacionadas con alfabeto.
- **Momin**: Alfabeto y CjtAlfabetos y sus Tests Unitarios. Vista de menú principal y miscelaneas.

4.2. Grupales

- **Jianing y Rubén**: Algoritmo, CalculadoraBigramasConFrecuencia
- **Momin y Yasin**: CtrlDominio
- **Momin y Yasin**: Conjunto de clases de excepciones. Usuario y clases de persistencia.
- **Rubén, Momin y Yasin**: Vista Terminal. Código implementado por Rubén y las excepciones añadidas posteriormente por Momin y Yasin