Universitat Politècnica de Catalunya - FIB/UPC

PROP - Projectes de Programació

# Generador de teclados - primera 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.<br>2. El sistema le pide al usuario qué tipo de interfaz quiere utilizar.<br>3. El usuario le proporciona la opción que desea.<br>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.<br>2. El sistema le pide al usuario un nombre y una contraseña.<br>3. El usuario proporciona un nombre de usuario y una contraseña.<br>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.<br>▪ El formato del nombre no es correcto.<br>▪ 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.<br>2. El sistema le pide al usuario un nombre y una contraseña.<br>3. El usuario proporciona un nombre de usuario y una contraseña.<br>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.<br>▪ 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).<br>2. El usuario selecciona cambiar contraseña.<br>3. El sistema le pide que vuelva a ingresar el usuario y la contraseña antigua y la nueva.<br>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.<br>▪ 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.<br>2. El sistema le pide al usuario un nombre y la contraseña del perfil a borrar.<br>3. El usuario proporciona un nombre de usuario y la contraseña.<br>4. El sistema borra el perfil con los datos proporcionados por el usuario.<br>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.<br>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.<br>▪ No existe un alfabeto con el nombre proporcionado.<br>▪ Formato incorrecto para lista de palabras.<br>▪ Nombre de algoritmo incorrecto.<br>▪ 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.<br>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.<br>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.<br>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". <br> 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". <br> 2. El sistema pide al usuario el nombre del teclado objetivo. <br> 3. El usuario introduce el nombre del teclado. <br> 4. El sistema pide al usuario la fila y columna de los dos caracteres a intercambiar. <br> 5. El usuario proporciona los datos pedidos. <br> 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. <br> ▪ 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.<br>2. El sistema pide al usuario un nombre para el alfabeto.<br>3. El sistema pide al usuario los caracteres que tiene el alfabeto.<br>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.<br>▪ El nombre del alfabeto no es válido (espacios, o nombre vacío).<br>▪ 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.<br>2. El sistema le pide al usuario el nombre del alfabeto que desea modificar.<br>3. El usuario le proporciona el nombre del alfabeto a modificar.<br>4. El sistema le pide al usuario los caracteres.<br>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.<br>▪ 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.<br>2. El sistema le pide al usuario el nombre del alfabeto que desea borrar.<br>3. El usuario le proporciona el nombre del alfabeto a borrar.<br>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.<br>2. El sistema pide al usuario el nombre del alfabeto.<br>3. El usuario proporciona el nombre del alfabeto.<br>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.<br>2. El sistema muestra los nombres de los alfabetos existentes. |
| **Postcondición** | |
| **Excepciones** | ▪ No hay alfabetos creados. |

# 2. Diagrama de clases y explicaciones

## 2.1. Clases y controladores de la capa de presentación

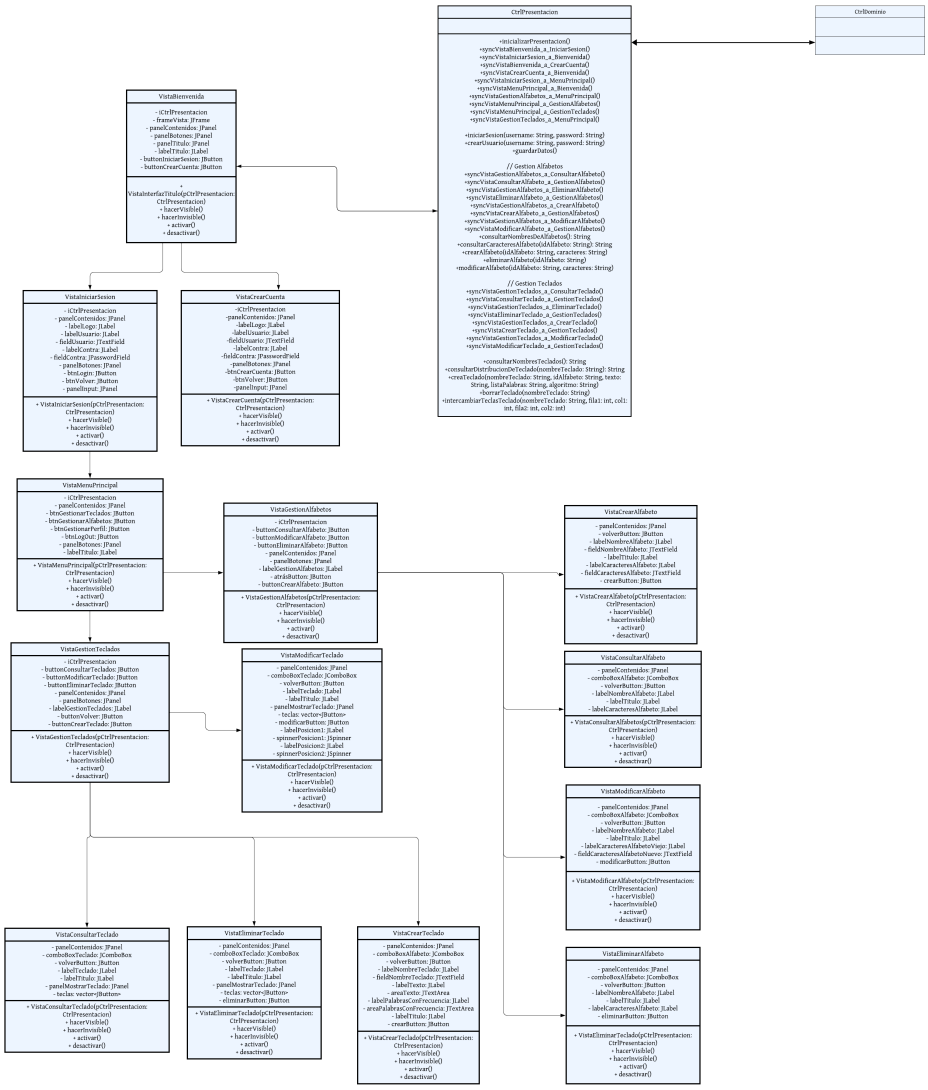


Figura 2: Diagrama de clases de la capa de presentación

### 2.1.1. VistaBienvenida

La clase `VistaBienvenida` sirve como pantalla de inicio. Al iniciarse, presenta un menú con el título del programa y botones para iniciar sesión o crear una cuenta. Sus atributos son los siguientes :

- `iCtrlPresentacion`: Referencia al controlador de presentación, responsable de gestionar la lógica de la interfaz.
- `frameVista`: Marco de la interfaz gráfica que contiene todos los elementos visuales.
- `panelContenidos`: Panel que alberga los componentes visuales de la interfaz.
- `panelBotones`: Panel que contiene los botones de la interfaz.
- `panelTitulo`: Panel que muestra el título del programa.
- `labelTitulo`: Etiqueta que presenta el título del programa.
- `buttonIniciarSesion`: Botón que permite iniciar sesión.
- `buttonCrearCuenta`: Botón que dirige a la creación de una nueva cuenta.

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. VistaIniciarSesion

La clase `VistaIniciarSesion` Gestiona el proceso de inicio de sesión al programa. Al iniciar sesión, el usuario debe proporcionar su nombre de usuario y contraseña a través de campos de entrada.
Sus atributos son los siguientes : a

- `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.

- `btnLogin`: Botón para iniciar sesión.
- `btnVolver`: Botón para volver a la pantalla de bienvenida.
- `panelInput`: Panel que agrupa los campos de entrada de usuario y contraseña.

### 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** proporciona una interfaz de menú que permite acceder a las diferentes funciones del programa. Incluye botones para acceder a la gestión de teclados, alfabetos, y el perfil del usuario, así como un botón de cierre de sesión.
Sus atributos son los siguientes :

- `iCtrlPresentacion`: Referencia al controlador de presentación, responsable de gestionar la lógica de la interfaz.
- `panelContenidos`: Panel que contiene todos los elementos visuales de la interfaz.
- `btnGestionarTeclados`: Botón para acceder a la gestión de teclados.

- `btnGestionarAlfabetos`: Botón para acceder a la gestión de alfabetos.
- `btnGestionarPerfil`: Botón para acceder a la gestión del perfil del usuario.
- `btnLogOut`: Botón para cerrar la sesión del usuario.
- `labelTitulo`: Etiqueta que muestra el título "Menú Principal".

### 2.1.5. VistaGestionarAlfabetos

La clase **VistaGestionAlfabetos:** Interfaz de gestión de alfabetos. Incluye botones para consultar, modificar, eliminar y crear nuevos alfabetos. Además, hay un botón para regresar al menú principal.
Sus atributos son los siguientes :

- `iCtrlPresentacion`: Referencia al controlador de presentación, responsable de gestionar la lógica de la interfaz.
- `panelContenidos`: Panel que contiene todos los elementos visuales de la interfaz.
- `buttonConsultarAlfabeto`: Botón para consultar alfabetos.
- `buttonModificarAlfabeto`: Botón para modificar alfabetos.
- `buttonEliminarAlfabeto`: Botón para eliminar alfabetos.
- `buttonCrearAlfabeto`: Botón para crear nuevos alfabetos.
- `atrásButton`: Botón para regresar al menú principal.
- `labelGestionAlfabetos`: Etiqueta que muestra el título "Gestión de alfabetos".

### 2.1.6. VistaGestionarTeclados

La clase **VistaGestionTeclados:** Interfaz de gestión de teclados. Incluye botones para consultar, modificar, eliminar y crear nuevos teclados. Además, hay un botón para regresar al menú principal.
Sus atributos son los siguientes :

- `iCtrlPresentacion`: Referencia al controlador de presentación, responsable de gestionar la lógica de la interfaz.
- `panelContenidos`: Panel que contiene todos los elementos visuales de la interfaz.
- `buttonConsultarTeclados`: Botón para consultar teclados.
- `buttonModificarTeclado`: Botón para modificar teclados.
- `buttonEliminarTeclado`: Botón para eliminar teclados.

- `buttonCrearTeclado`: Botón para crear nuevos teclados.
- `buttonVolver`: Botón para regresar al menú principal.
- `labelGestionTeclados`: Etiqueta que muestra el título "Gestión de teclados".

### 2.1.7. VistaConsultarTeclado

La clase **VistaConsultarTeclado** Permite la consulta de los diferentes teclados creados por un usuario. Incluye un menú desplegable (`comboBoxTeclado`) para seleccionar un teclado, un botón (`volverButton`) para regresar al menú principal, y un conjunto de botones (`teclas`) para representar el teclado. Además, hay etiquetas (`labelTeclado` y `labelTitulo`) que proporcionan información visual en la interfaz.
Sus atributos son los siguientes :

- `iCtrlPresentacion`: Referencia al controlador de presentación, responsable de gestionar la lógica de la interfaz.
- `panelContenidos`: Panel que contiene todos los elementos visuales de la interfaz.
- `comboBoxTeclado`: Menú desplegable para seleccionar un teclado.
- `volverButton`: Botón para regresar al menú principal.
- `labelTeclado`: Etiqueta que muestra información sobre la selección de teclado.
- `labelTitulo`: Etiqueta que muestra el título de la interfaz.
- `panelMostrarTeclado`: Panel que contiene la representación visual del teclado.
- `teclas`: Vector de botones que representan las teclas del teclado.

ola que ases

### 2.1.8. VistaModificarTeclado

La clase **VistaModificarTeclado** Permite la modificación de un teclado creado por el usuario. Contiene un menú desplegable (`comboBoxTeclado`) para seleccionar un teclado, un botón (`volverButton`) para regresar al menú principal, y un conjunto de botones (`teclas`) para representar el teclado. Además, hay etiquetas (`labelTeclado` y `labelTitulo`) que proporcionan información visual en la interfaz.
Sus atributos son los siguientes :

- `iCtrlPresentacion`: Referencia al controlador de presentación, responsable de gestionar la lógica de la interfaz.
- `panelContenidos`: Panel que contiene todos los elementos visuales de la interfaz.
- `comboBoxTeclado`: Menú desplegable para seleccionar un teclado.

- `volverButton`: Botón para regresar al menú principal.
- `labelTeclado`: Etiqueta que muestra información sobre la selección de teclado.
- `labelTitulo`: Etiqueta que muestra el título de la interfaz.
- `panelMostrarTeclado`: Panel que contiene la representación visual del teclado.
- `teclas`: Vector de botones que representan las teclas del teclado.
- `modificarButton`: Botón para iniciar la modificación del teclado.
- `labelPosicion1`: Etiqueta que muestra información sobre la primera posición a modificar.
- `spinnerPosicion1`: Selector numérico para establecer la primera posición.
- `labelPosicion2`: Etiqueta que muestra información sobre la segunda posición a modificar.
- `spinnerPosicion2`: Selector numérico para establecer la segunda posición.

### 2.1.9. VistaEliminarTeclado

La clase **VistaEliminarTeclado** Interfaz dedicada al borrado de uno de los teclados creados por el usuario. Se muestra un menú desplegable con los teclados disponibles, botones para visualizar y eliminar el teclado seleccionado, y un botón para regresar al menú principal.
Sus atributos son los siguientes :

- `iCtrlPresentacion`: Referencia al controlador de presentación, responsable de gestionar la lógica de la interfaz.
- `panelContenidos`: Panel que contiene todos los elementos visuales de la interfaz.
- `comboBoxTeclado`: Menú desplegable que muestra la lista de teclados disponibles.
- `volverButton`: Botón para regresar al menú principal.
- `labelTeclado`: Etiqueta que muestra información sobre el teclado seleccionado.
- `labelTitulo`: Etiqueta que muestra el título de la interfaz.
- `panelMostrarTeclado`: Panel que muestra visualmente el teclado seleccionado.
- `teclas`: Vector de botones que representan las teclas del teclado.
- `eliminarButton`: Botón para confirmar la eliminación del teclado.

### 2.1.10. VistaCrearTeclado

La clase **VistaCrearTeclado** representa la interfaz gráfica encargada de la creación de teclados. La interfaz incluye un menú desplegable con los alfabetos disponibles, campos para ingresar el nombre del teclado y su contenido textual, áreas para visualizar el texto ingresado y las palabras con frecuencia, y botones para crear el teclado y regresar al menú principal.
Sus atributos son los siguientes :

- `iCtrlPresentacion`: Referencia al controlador de presentación, responsable de gestionar la lógica de la interfaz.
- `panelContenidos`: Panel que contiene todos los elementos visuales de la interfaz.
- `comboBoxAlfabeto`: Menú desplegable que muestra la lista de alfabetos disponibles.
- `volverButton`: Botón para regresar al menú principal.
- `labelNombreTeclado`: Etiqueta que indica el campo para ingresar el nombre del teclado.
- `fieldNombreTeclado`: Campo de texto para ingresar el nombre del teclado.
- `labelTexto`: Etiqueta que indica el campo para ingresar el contenido textual del teclado.
- `areaTexto`: Área de texto para ingresar el contenido textual del teclado.
- `labelPalabrasConFrecuencia`: Etiqueta que indica el área para visualizar las palabras con frecuencia.
- `areaPalabrasConFrecuencia`: Área para visualizar las palabras con frecuencia.
- `labelTitulo`: Etiqueta que muestra el título de la interfaz.
- `crearButton`: Botón para confirmar la creación del teclado.

### 2.1.11. VistaConsultarAlfabeto

La clase **VistaConsultarAlfabeto** representa la interfaz gráfica encargada de la consulta de alfabetos. La interfaz incluye un menú desplegable con los alfabetos disponibles, un botón para regresar al menú principal y etiquetas que muestran el nombre del alfabeto y su lista de caracteres.
Sus atributos son los siguientes :

- `iCtrlPresentacion`: Referencia al controlador de presentación, responsable de gestionar la lógica de la interfaz.

- `panelContenidos`: Panel que contiene todos los elementos visuales de la interfaz.
- `comboBoxAlfabeto`: Menú desplegable que muestra la lista de alfabetos disponibles.
- `volverButton`: Botón para regresar al menú principal.
- `labelNombreAlfabeto`: Etiqueta que muestra el nombre del alfabeto seleccionado.
- `labelTitulo`: Etiqueta que muestra el título de la interfaz.
- `labelCaracteresAlfabeto`: Etiqueta que muestra la lista de caracteres del alfabeto.

### 2.1.12. VistaModificarAlfabeto

La clase **VistaModificarAlfabeto** representa la interfaz gráfica encargada de la modificación de alfabetos. La interfaz incluye un menú desplegable con los alfabetos disponibles, un botón para regresar al menú principal, etiquetas que muestran el nombre del alfabeto y su lista de caracteres, un campo de texto para ingresar los nuevos caracteres y un botón para realizar la modificación.
Sus atributos son los siguientes :

- `iCtrlPresentacion`: Referencia al controlador de presentación, responsable de gestionar la lógica de la interfaz.
- `panelContenidos`: Panel que contiene todos los elementos visuales de la interfaz.
- `comboBoxAlfabeto`: Menú desplegable que muestra la lista de alfabetos disponibles.
- `volverButton`: Botón para regresar al menú principal.
- `labelNombreAlfabeto`: Etiqueta que muestra el nombre del alfabeto seleccionado.
- `labelTitulo`: Etiqueta que muestra el título de la interfaz.
- `labelCaracteresAlfabetoViejo`: Etiqueta que muestra la lista de caracteres del alfabeto antes de la modificación.
- `fieldCaracteresAlfabetoNuevo`: Campo de texto para ingresar los nuevos caracteres del alfabeto.
- `modificarButton`: Botón para realizar la modificación del alfabeto.

### 2.1.13. VistaEliminarAlfabeto

La clase **VistaEliminarAlfabeto:** Vista encargada de permitir el borrado de uno de los alfabetos del usuario. La interfaz incluye un menú desplegable con los alfabetos disponibles, un botón para regresar al menú principal, etiquetas que muestran el nombre del alfabeto y su lista de caracteres, y un botón para realizar la eliminación. Sus atributos son los siguientes :

- `iCtrlPresentacion`: Referencia al controlador de presentación, responsable de gestionar la lógica de la interfaz.
- `panelContenidos`: Panel que contiene todos los elementos visuales de la interfaz.
- `comboBoxAlfabeto`: Menú desplegable que muestra la lista de alfabetos disponibles.
- `volverButton`: Botón para regresar al menú principal.
- `labelNombreAlfabeto`: Etiqueta que muestra el nombre del alfabeto seleccionado.
- `labelTitulo`: Etiqueta que muestra el título de la interfaz.
- `labelCaracteresAlfabeto`: Etiqueta que muestra la lista de caracteres del alfabeto a eliminar.
- `eliminarButton`: Botón para realizar la eliminación del alfabeto.

### 2.1.14. VistaCrearAlfabeto

La clase **VistaCrearAlfabeto** representa la interfaz gráfica encargada de la creación de nuevos alfabetos. La interfaz incluye un botón para regresar al menú principal, etiquetas y campos de texto para ingresar el nombre y los caracteres del nuevo alfabeto, y un botón para realizar la creación.
Sus atributos son los siguientes :

- `iCtrlPresentacion`: Referencia al controlador de presentación, responsable de gestionar la lógica de la interfaz.
- `panelContenidos`: Panel que contiene todos los elementos visuales de la interfaz.
- `volverButton`: Botón para regresar al menú principal.
- `labelNombreAlfabeto`: Etiqueta que indica el campo para ingresar el nombre del nuevo alfabeto.
- `fieldNombreAlfabeto`: Campo de texto para ingresar el nombre del nuevo alfabeto.
- `labelTitulo`: Etiqueta que muestra el título de la interfaz.

- `labelCaracteresAlfabeto`: Etiqueta que indica el campo para ingresar la lista de caracteres del nuevo alfabeto.
- `fieldCaracteresAlfabeto`: Campo de texto para ingresar la lista de caracteres del nuevo alfabeto.
- `crearButton`: Botón para realizar la creación del alfabeto.

### 2.1.15. CtrPresentacion

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, vistaIniciarSesion, vistaCrearCuenta, vistaMenuPrincipal, vistaGestionAlfabetos, vistaGestionTeclados, vistaConsultarTeclado`: Instancias de las distintas vistas del programa.

**Métodos:**

- `CtrlPresentacion()`: Constructor de la clase, inicializa el controlador de dominio y la vista de bienvenida.
- `inicializarPresentacion()`: Inicializa el controlador de dominio y carga los usuarios al inicio del programa.
- `syncVistaBienvenida_a_IniciarSesion()`, `syncVistaIniciarSesion_a_Bienvenida()`: Métodos de sincronización entre las vistas de bienvenida e iniciar sesión.
- `syncVistaBienvenida_a_CrearCuenta()`, `syncVistaCrearCuenta_a_Bienvenida()`: Métodos de sincronización entre las vistas de bienvenida y crear cuenta.
- `syncVistaIniciarSesion_a_MenuPrincipal()`: Sincroniza la transición desde la vista de iniciar sesión a la vista del menú principal.
- `syncVistaMenuPrincipal_a_Bienvenida()`: Sincroniza la transición desde la vista del menú principal a la vista de bienvenida.
- `syncVistaGestionAlfabetos_a_MenuPrincipal()`, `syncVistaMenuPrincipal_a_GestionAlfabetos()`: Métodos de sincronización entre la gestión de alfabetos y el menú principal.
- `syncVistaMenuPrincipal_a_GestionTeclados()`: Sincroniza la transición desde la vista del menú principal a la gestión de teclados.
- `syncVistaGestionTeclados_a_MenuPrincipal()`, `syncVistaMenuPrincipal_a_GestionTeclados()`: Métodos de sincronización entre la gestión de teclados y el menú principal.

- `syncVistaGestionTeclados_a_ConsultarTeclado()`, `syncVistaConsultarTeclado_a_GestionTeclados()`: Métodos de sincronización entre la consulta de teclado y la gestión de teclados.
- `iniciarSesion(String Username, String Password)`: Inicia sesión con el nombre de usuario y la contraseña proporcionados.
- `crearUsuario(String username, String password)`: Crea un nuevo usuario con el nombre de usuario y la contraseña proporcionados.
- `existeUsuario(String Username)`: Verifica si existe un usuario con el nombre de usuario proporcionado.
- `FuncCargarDatos()`: Realiza la carga de teclados y alfabetos, gestionando posibles excepciones durante el proceso.
- `guardarDatos()`: Guarda la información de teclados, alfabetos y usuarios.
- `cerrarPrograma()`: Cierra el programa.

### 2.1.16. 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.

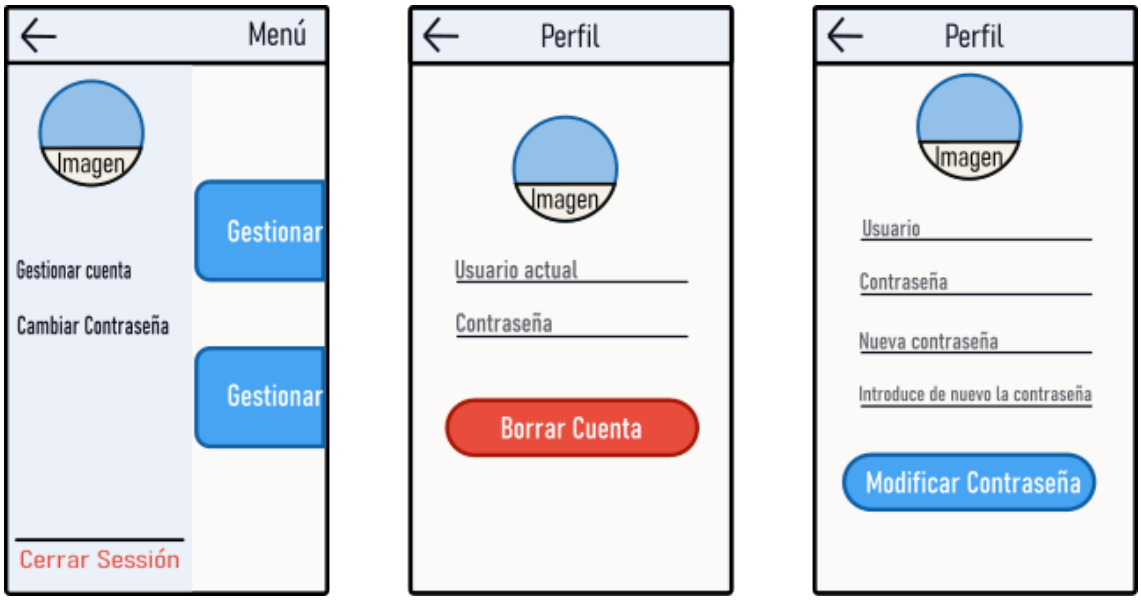


Figura 4: Vista Gestionar Perfil, Gestionar Cuenta y Cambiar Contraseña

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.

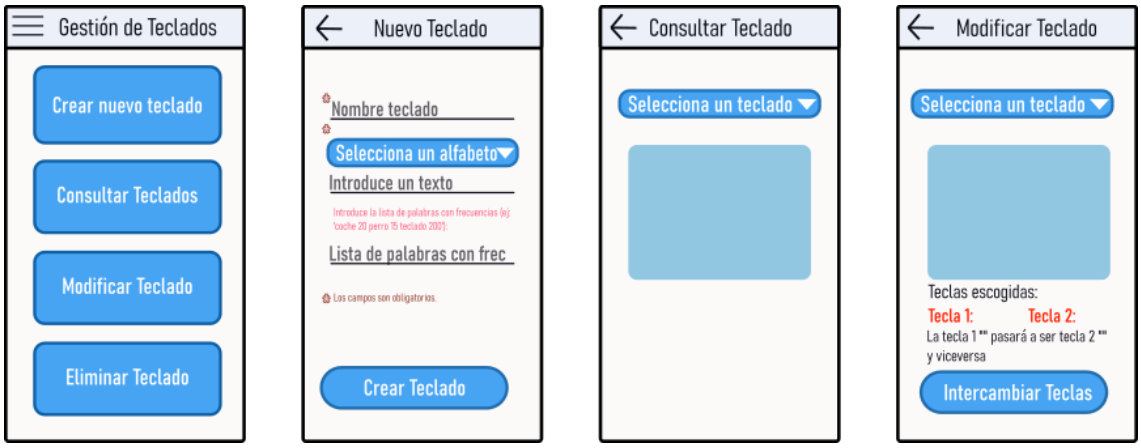


Figura 5: Vista Gestionar Teclados, incluye la vista de Crear Teclado Nuevo, Consultar Teclado y Modificar Teclado


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.

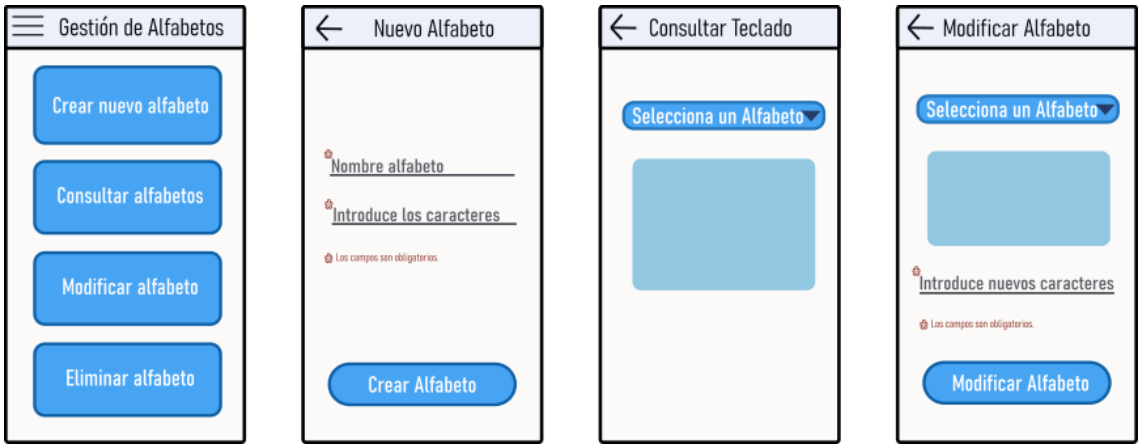


Figura 7: Vista Gestionar Alfabetos, Crear Nuevo Alfabeto, Consultar Alfabetos y Modificar Alfabetos


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 de 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 saturados como viene siendo el caso de la segunda fila, son los colores que se utilizan para el contorno de los botones.

## 2.2. Clases y controladores de la capa de dominio

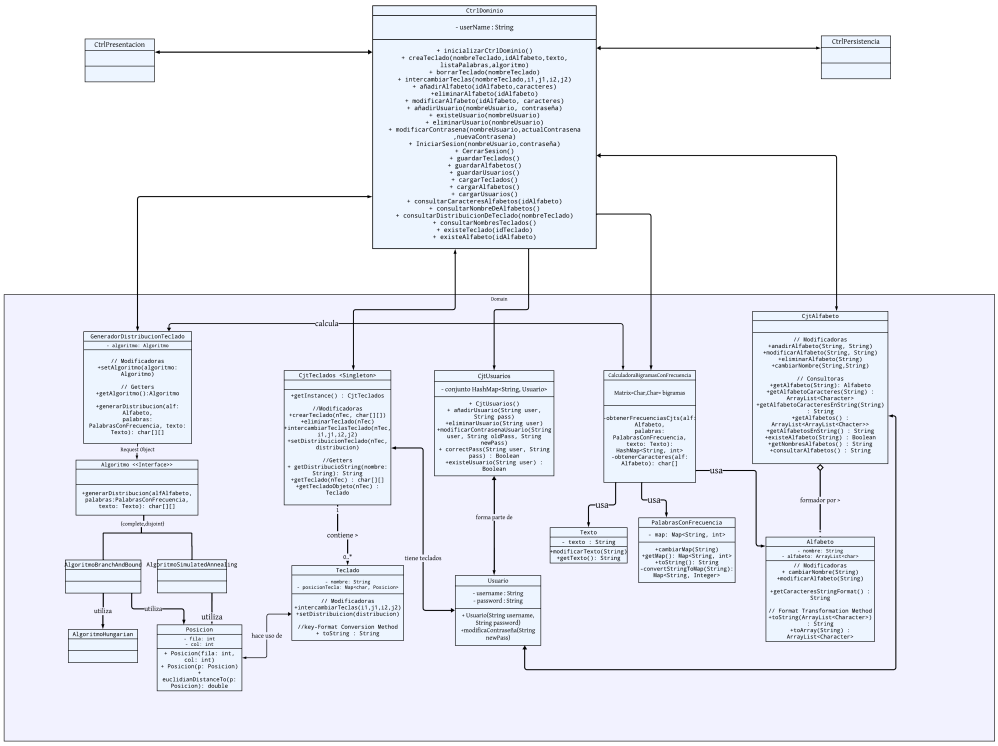


Figura 10: Diagrama de clases de la capa de dominio

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 8 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

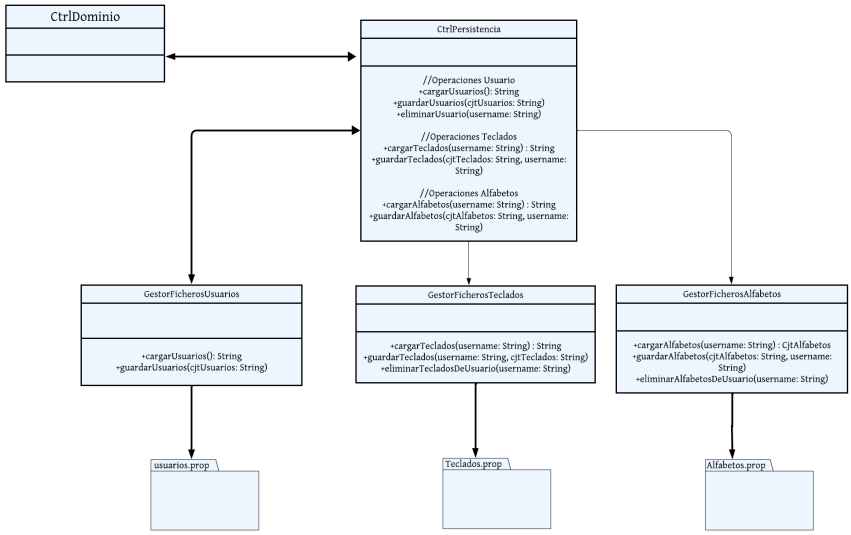


Figura 11: Diagrama de clases de la capa de persistencia

### 2.3.1. GestorFicheroUsuarios

La clase GestorFicherosUsuarios se especializa en la persistencia de conjuntos de usuarios en el sistema. Su método guardarUsuarios almacena los nombres de usuarios junto con sus contraseñas en un archivo específico (üsuarios.prop"). Por otro lado, cargarUsuarios recupera estos usuarios desde el archivo mencionado y retorna un conjunto de usuarios (CjtUsuarios).

### 2.3.2. GestorFicheroTeclados

La clase GestorFicherosTeclados se centra en la persistencia de conjuntos de teclados en el sistema. Permite guardar, cargar y eliminar teclados asociados a usuarios específicos. El método guardarTeclados almacena los teclados en un archivo específico del usuario, mientras que cargarTeclados recupera estos conjuntos, manejando excepciones. La función eliminarTecladosDeUsuario borra archivos asociados

a usuarios. La clase también incorpora una función privada para convertir strings en matrices de caracteres durante la carga desde el archivo, esto facilita la gestión eficiente de teclados en el sistema.

### 2.3.3. GestorFicheroAlfabetos

La clase GestorFicherosAlfabetos gestiona la escritura, lectura y eliminación de conjuntos de alfabetos en archivos. Permite guardar alfabetos asociados a usuarios, cargarlos cuando sea necesario y eliminar archivos vinculados a usuarios específicos. La clase también incluye una función interna para transformar strings en matrices, cosa que facilita de manera eficiente la manipulación de alfabetos en el sistema.

### 2.3.4. CtrlPersistencia

La clase CtrlPersistencia sirve para las operaciones de persistencia relacionadas con usuarios, alfabetos y teclados en el sistema. Las operaciones de lectura y escritura se delegan a gestores de archivos especializados. Los métodos de guardado (guardarUsuarios, guardarAlfabetos, y guardarTeclados) utilizan instancias de los gestores de archivos GestorFicherosUsuarios, GestorFicherosAlfabetos, y GestorFicherosTeclados, respectivamente. Del mismo modo, los métodos de carga (cargarUsuarios, cargarAlfabetos, y cargarTeclados) emplean estas instancias para cargar la información desde los archivos correspondientes.
La operación eliminarUsuario elimina tanto los teclados como los alfabetos asociados a un usuario específico, utilizando instancias de los gestores de teclados y alfabetos.

# 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