****

**Universidad Nacional de General Sarmiento**

**Programación III**

**Trabajo Práctico:**

**“El equipo ideal”**

**Profesores:**

* Patricia Bagnes.
* Javier Marenco.

**Grupo:**

* Andrés Geiser
  + 2018/42239150
  + geiserandres00@gmail.com
* Hernán Rodrigo Fernández
  + 2017/41584220
  + fernandezhr\_1998@hotmail.com

**Objetivo**

El objetivo que planteamos resolver en este trabajo práctico consiste en implementar una aplicación que muestre la conformación más grande de un equipo de desarrollo a partir de un conjunto de empleados, una cantidad mínima/máxima de cada puesto y los respectivos lazos de convivencia entre ellos, los cuales determinaran la posibilidad de generar un equipo optimo con las condiciones dadas.

**Responsabilidad de clases**

El proyecto cuenta con 6 paquetes en los que están distribuidas las clases según su funcionalidad:

* **Archivos**: se encuentran los archivos que contienen una serie de nombres y apellidos que se utilizan para hacer una combinación entre ellos y posteriormente generar empleados dinámicamente.
* **Controlador**: clases encargadas de comunicar los distintos componentes visuales con aquellas que implementan la lógica interna. También definen el comportamiento de los componentes de la interfaz.
* **Iconos**: archivos que se utilizan como iconos en los componentes de la interfaz visual.
* **Modelo**: se encuentran las clases encargadas de poner en funcionamiento la lógica interna del programa y la suite de Test correspondiente a las mismas.
* **Principal**: contiene a la clase principal que se encarga de inicializar las clases visuales y desencadenar el funcionamiento del programa.
* **Vista**: donde se encuentran las clases visuales, que contienen el código que implementa la interfaz correspondiente al MainFrame y a los distintos paneles.

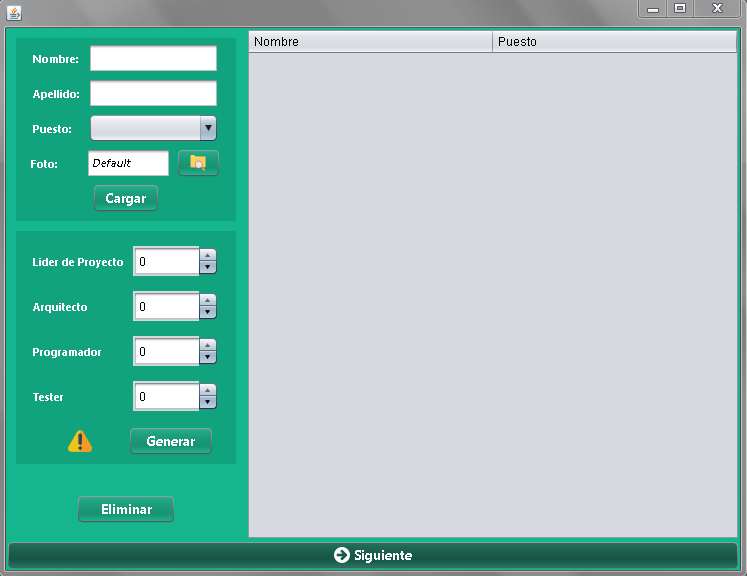
A continuación, presentaremos a cada una de las clases de los paquetes más importantes, describiendo sus variables internas, sus métodos y su rol general dentro del programa.

PAQUETE

**VISTA**

**Clase VentanaPrincipal**

Clase que implementa la interfaz principal de la aplicación:

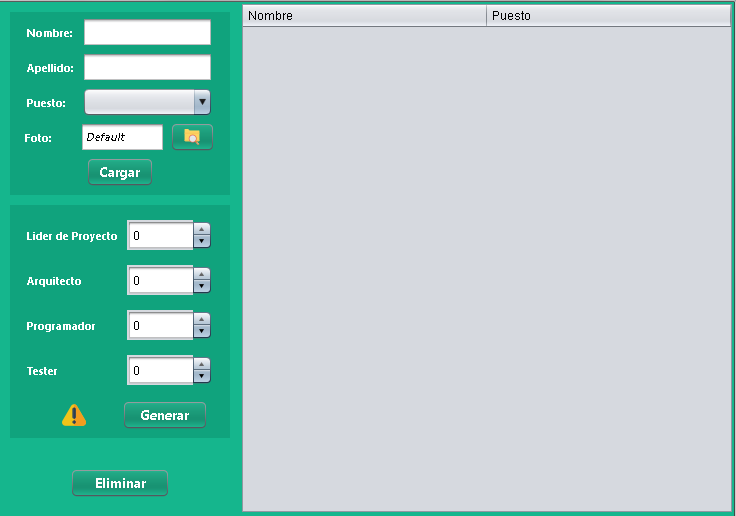


La ventana solo cuenta con un panel central (el cual cambiara en cuatro diferentes) y un botón en la parte inferior con el cual podremos cambiar el panel central para pasar hacia otras funcionalidades de la aplicación.

El panel central que se muestra en la imagen es el **CargarEmpleados**.

**Clase CargarEmpleado**

Implementa uno de los cuatro paneles centrales de la ventana principal. Tiene el siguiente aspecto:

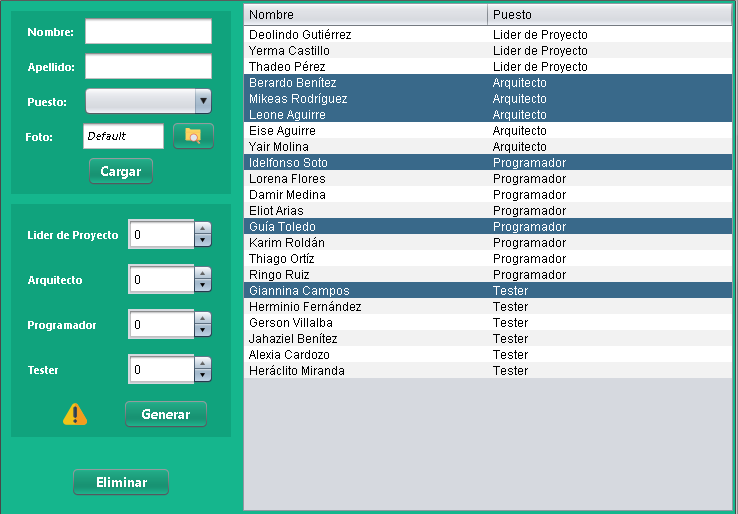


Es el primer panel que se visualiza y el cual usamos para cargar empleados y mostrarlos en pantalla.

Tiene un panel a la derecha con una tabla que mostrara la lista con los empleados que se irán cargando y a la izquieda otro panel con otros dos paneles más pequeños. El primero contiene dos JTextField para escribir el nombre del empleado, un JComboBox con el cual elegiremos el puesto del empleado, un botón que abrirá un selector de archivos para poder subir una foto al empleado y por último un botón “Cargar” que tomará esos datos y los colocará en la tabla. El segundo cuenta con cuatro JSpinner con los que podremos seleccionar una determinada cantidad de empleados y generarlos con un nombre que la aplicación le asignara, también tiene en la parte inferior izquierda un icono el cual al pasar el mouse por encima dice la cantidad de empleados que la aplicación puede generar.

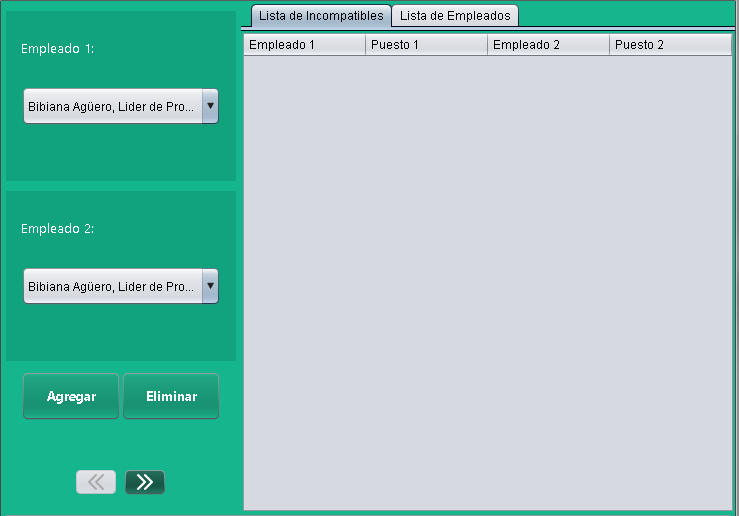
Por último, tiene un botón con el cual podremos eliminar los empleados. Esto se hace seleccionándolos en la tabla y luego presionando el botón “Eliminar”.

Panel con algunos empleados cargados:



**Clase CargarDatos**

Implementa otro de los cuatro paneles centrales de la ventana principal. Tiene los siguientes aspectos:



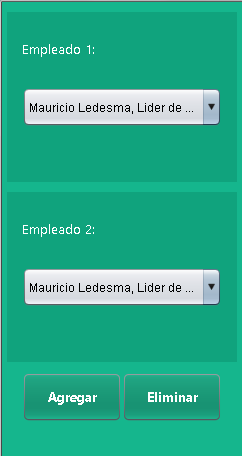


Es el panel que nos permitirá cargar la lista de empleados incompatibles y los requerimientos específicos del equipo.

Tiene un panel a la derecha que contiene una tabla que mostrara los empleados incompatibles entre sí y otra tabla con los empleados. En la parte izquierda hay otro panel que cuenta con un panel central (que será el **CargarIncompatibles** o **CargarCotas**) el cual podremos cambiar presionando los botones con las flechas ‘<<’ o ‘>>’que se encuentran abajo.

**Clase CargarIncompatibles**

Implementa el siguiente panel:



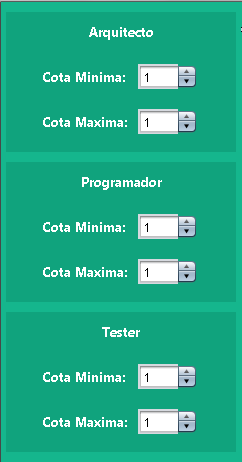
Este panel nos permitirá elegir dos de los empleados cargados mediante unos JComboBox y agregarlos a la tabla de empleados incompatibles, también tiene un botón “Eliminar” con el que podremos borrar de la tabla la relación.

Tabla de empleados incompatibles con algunos cargados:

****

**Clase CargarCotas**

Implementa el siguiente panel:

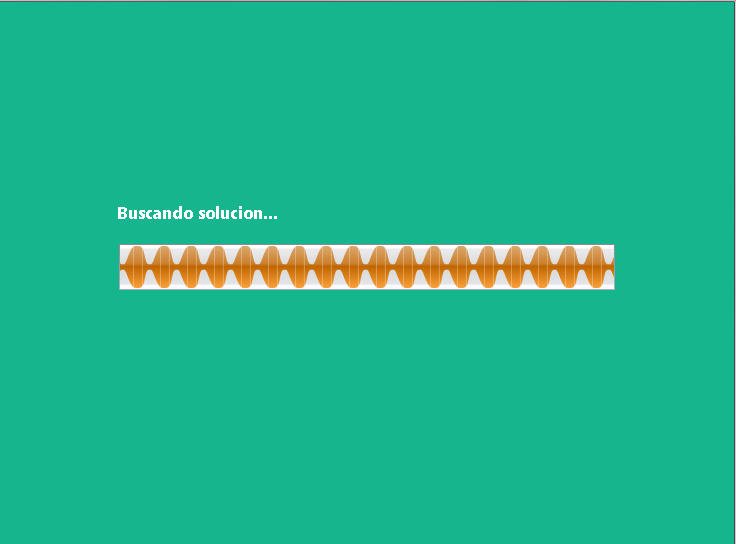


Este panel cumplirá la función de permitirnos elegir los requerimientos que deseamos para el equipo, es decir, una cota mínimamáxima para los arquitectos, programadores y testers.

Tiene dos JSpinner para elegir el mínimo y el máximo para cada puesto.

**Clase BuscarSolucion**

Implementa el tercer panel central de la ventana principal y se visualizara mientras se ejecuta el algoritmo que buscara el conjunto de empleados que cumpla con las condiciones:



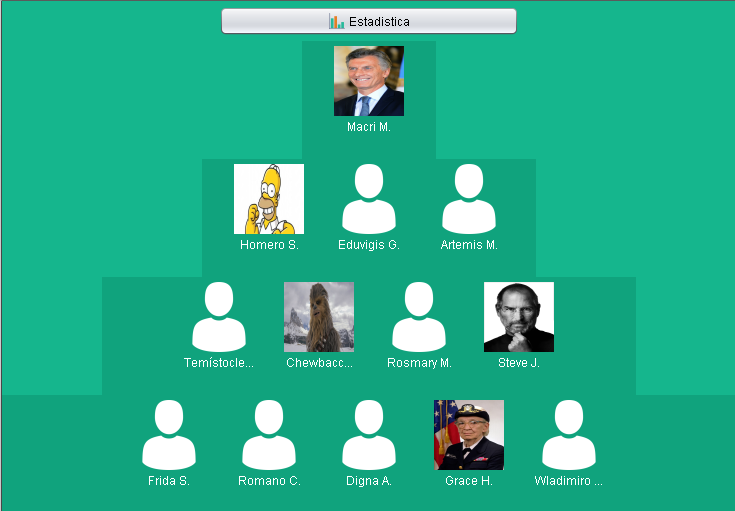
Su única funcionalidad es informar que se está buscando la solución a la problemática de este trabajo practico.

Los únicos componentes que contiene es un JLabel y un JProgressBar indeterminado.

**Clase MostrarSolucion**

Implementa el cuarto y último panel que se observara en la ventana principal. Puede tener 2 aspectos distintos:

1. Si se encontró un conjunto que cumpla las condiciones se mostrara de la siguiente forma:



Tiene un botón “Estadística” en la parte superior el cual mostrara el tiempo que ha tardado el algoritmo en encontrar la solución y la cantidad de veces que entro en el caso base.

Luego tiene una serie de 4 paneles uno abajo del otro en los cuales se visualizarán los empleados elegidos. El primero contendrá al líder de proyecto, el segundo a los arquitectos, el tercero a los programadores y el ultimo a los testers. Cada empleado se mostrará con su respectiva foto.

1. En caso de que no se haya encontrado una solución mostrara el siguiente panel:



PAQUETE

**MODELO**

**Clase Empleado**

Esta clase modelara a un empleado el cual tiene un nombre y un puesto dentro del equipo.

**Variables de clase**

* *String nombre*: representa el nombre del empleado.
* *String puesto*: representa el puesto del empleado.

**Métodos de clase**

Tiene métodos getNombre() y getPuesto() que son llamados por otras clases, también posee un método equals para verificar que dos empleados sean distintos.

**Clase Requerimiento**

Representará los requerimientos que debe cumplir el equipo de trabajo. Solo posee métodos getters y setters para cada una de sus variables.

**Variables de clase**

* *int minLideres, maxLideres*: representa la cantidad mínima y máxima de líderes de proyecto que debe tener el equipo. Por defecto es inicializada en 1.
* *int minArquitectos, maxArquitectos*: representa la cantidad mínima y máxima de arquitectos que debe tener el equipo. Por defecto es inicializada en 1.
* *int minProgramadores, maxProgramadores*: representa la cantidad mínima y máxima de programadores que debe tener el equipo. Por defecto es inicializada en 1.
* *int minTesters, maxTesters*: representa la cantidad mínima y máxima de testers que debe tener el equipo. Por defecto es inicializada en 1.

**Clase Grafo**

Esta clase será utilizada para representar las relaciones que existen entre los empleados.

**Variables de clase**

* *boolean [][] A*: representa la matriz de adyacencia interna del grafo.

**Métodos de clase**

* *void agregarArista(int i, int j)* : se encarga de agregar una arista entre los vértices **i** y **j** colocando en esa posición de la matriz true (para nuestro problema, true significara que tienen mala relación laboral).
* *void borrarArista(int i, int j)* : se encarga de eliminar la arista entre los vértices **i** y **j** volviendo false la posición correspondiente de la matriz **A**.
* *boolean existeArista(int i, int j):* verifica si existe una arista entre los vértices **i** y **j**. Es utilizado para verificar la relación entre dos empleados.
* *set<Integer> vecinos(int i):* retorna una colección con aquellos vértices tengan una arista con el vértice **i**.

**Clase Instancia**

Esta clase representa un estado actual del problema a resolver, contiene la lista de empleados, las relaciones entre ellos y los requerimientos de cada puesto.

**Variables de clase**

* *Grafo malasRelaciones:* contiene las relaciones entre los empleados.
* *ArrayList<Empleados> empleados*: contiene a todos los empleados generados por el usuario.
* *Requerimiento requerimiento*: contiene las especificaciones por puesto que fija el usuario.

**Métodos de clase**

Luego la clase contiene un método sonIncompatibles(int i, int j) que es utilizado para verificar la relación entre dos empleados de manera auxiliar y los demás métodos retornan la lista de empleados y las cotas especificadas en requerimientos.

**Clase Modelo**

Esta clase contiene la lógica interna principal del programa, organiza la carga de empleados, las relaciones entre los empleados y gestiona la solución del equipo de desarrollo final.

**Variables de clase**

* *ArrayList<Empleado> empleado*: lista en la cual se irán guardando los empleados.
* *Grafo malasRelaciones*: grafo que representa a los vértices como empleados y a las aristas como malas relaciones entre sí.
* *Requerimiento requerimiento*: contendrá la cantidad máxima y mínima de cada puesto que deberá cumplir el equipo final.
* *ArrayList<Empleado> solución*: contendrá al equipo final, es decir, al conjunto de empleados que cumple con las condiciones de la problemática.
* *Solver solver*: referencia a la clase que se utilizara para modelar el equipo óptimo.

**Métodos de clase**

* *void agregarEmpleado(String nombre, String puesto):* agrega un empleado a la lista con los parámetros recibidos (siempre y cuando sean validos).
* *void eliminarEmpleado(String nombre):*elimina de la lista el empleado con el nombre del parámetro recibido si es que existe*.*
* *void confirmarListaDeEmpleados()*: en caso de la lista contenga un empleado de cada puesto inicializa el grafo de malas relaciones con una cantidad de vértices igual a la cantidad de empleados de la lista. Llamar a esta función hará que después no sea posible agregar o eliminar empleados de la lista**.**
* *void agregarMalaRelacion(String nombreE1, String nombreE2) y eliminarMalaRelacion*: si la lista fue confirmada y los dos empleados existen agrega o elimina la arista en el grafo.
* *void resolver():* inicializa a *solver* y llama a .resolver con una nueva instancia armada con la lista de empleados, malasRelaciones y los requerimientos del equipo, almacenando en *solucion* a los empleados que integran el equipo final*.*
* *String getEstadisticas()*: devuelve una cadena con las estadísticas finales de la ejecución del solver.

**Clase Solver**

Esta clase implementa un algoritmo de fuerza bruta basado en clique máxima, pero con adiciones propias de nuestro problema. Dado un grafo G encuentra la clique (grupo de empleados que se llevan bien) que contiene el grupo con mayor cantidad de empleados que cumplen los requerimientos.

**Variables de clase**

* *Instancia instancia*: representa el conjunto de datos que recibe la clase para buscar la solución.
* *Set<Integer> conjuntoActual:* contendrá los índices de un subconjunto que el algoritmo de recursión ira construyendo.
* *ArrayList<Empleado> equipoMasOptimo:*contendrá los empleados que pertenezcan a la solución final, es decir al equipo óptimo con las condiciones dadas.
* *int totalIteraciones y double Tiempototal:*representan variables de estadística para controlar el tiempo total de ejecución y la cantidad de iteraciones del caso base.

**Métodos de clase**

* *ArrayList<Empleado> resolver():* pone en marcha la recursión y devuelve el equipo óptimo formado con los empleados de la *instancia*.
* *void recursión(int inicial):* implementa el algoritmo general recursivo de fuerza bruta que genera todos los subconjuntos de un conjunto. El caso base de esta recursión seria cuando el **inicial** sea igual a la cantidad de empleados de la instancia. Cuando se entra en el caso base se comprueba que en el **conjuntoActual** no existan incompatibilidades y que se cumplan con los requisitos mínimos de cada puesto indicados por la instancia. Si pasa estas condiciones, armamos el equipo máximo del **conjuntoActual (**es decir, que formamos un conjunto de empleados sin que la cantidad líderes, arquitectos, programadores o testers se pase de los requisitos máximos indicados por la instancia) y si la cantidad de empleados del equipo que se armó supera a la cantidad del **equipoMasOptimo,** estepasara a ser el más óptimo.

La cantidad de veces que se ejecutara el caso base corresponde a 2*n* con *n* igual a la cantidad de empleados del conjunto inicial.

Tiempo aproximado que tarda el algoritmo de recursión a partir de una cierta cantidad de empleados:

|  |  |
| --- | --- |
| Cantidad de empleados | Tiempo |
| de 4 a 14 | 0.001 seg a 0.013 seg |
| 15 | 0.3 seg |
| 16 | 0.6 |
| 17 | 1.3 |
| 18 | 3 seg |
| 19 | 6 seg |
| 20 | 13 seg |
| 21 | 28 seg |
| 22 | 58seg |
| 23 | 122 seg |
| 24 | 246 seg |

**Clase Assert, testModelo y testSolverEquipo**

Estas clases contienen la suite de test que controlan los métodos de modelo y solver que contienen 24 y 11 test respectivamente, luego la clase Assert es utilizada para probar si dos listas de empleados son iguales, es decir que representan el mismo equipo solución.

PAQUETE

**CONTROLADOR**

**Clase ListaDeNombres**

Clase que implementa un conjunto de nombres (nombre + apellido) que se obtienen a partir de los ficheros que se encuentran en el paquete **archivos**. Fue creada con el objetivo de poder crear empleados dinámicamente.

**Variables de clase**

* *ArrayList<String> listaDeNombres:* contendrá las combinaciones de nombres y apellidos.

**Métodos de clase**

* *void generarNombres():* pone en marcha la lectura de los archivos “nombres.txt” y “apellidos.txt”. Luego por cada nombre obtenido lo combina con cada uno de los apellidos.
* *List<String> leerFichero(String fichero):* devuelve una lista de String que se obtienen al leer cada una de las líneas del archivo pasado como parámetro.
* *String dameUno():*retorna un String aleatorio de la lista de nombres y lo elimina de la misma.
* *void eliminar(String n):* elimina de la lista de nombres el String que recibe como parámetro.
* *int cantidad():* retorna la cantidad de nombres de contiene la lista.

**Clase FotosDeEmpleados**

Clase en la cual guardaremos las fotos de los empleados que se irán cargando mediante variables y métodos estáticos.

**Variables de clase**

* *static HashMap<String, ImageIcon> fotosDeEmpleados:* almacenará las fotos de los empleados, donde la clave será el nombre del empleado.

**Métodos de clase**

* *static agregarFoto(String nombre, ImageIcon foto):* agrega la imagen que recibe como parámetro usando el String ‘nombre’ como clave.
* *static eliminar(String nombre):* elimina la foto correspondiente a la clave que se recibe como parámetro.
* *static ImageIcon getFoto(String nombre):*retorna la imagen asociada a la clave que se recibe como parámetro.

**Clase CtrlVentanaPrincipal**

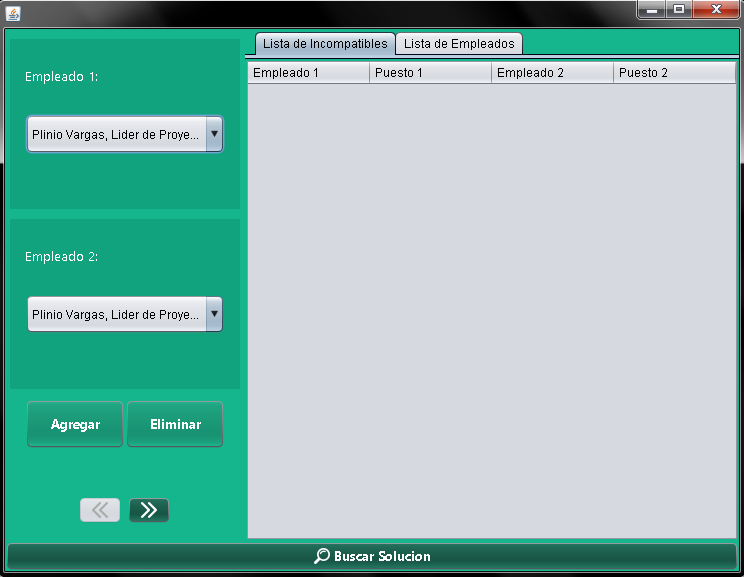
Esta clase se encarga de controlar el comportamiento de la interfaz principal y lógica de la aplicación haciendo uso también de los controladores de los distintos paneles.

De esta clase explicaremos el método principal que es el **cambiarDePanel()** al cual se lo llama al presionar el botón de la parte inferior de la aplicación. Lo que hace este método básicamente es cambiar el panel central de la ventana principal dependiendo de que panel central se encuentre actualmente e iniciar su respectivo controlador. Por ejemplo:



Presionamos botón “Siguiente”

Ventana principal con el panel CargarEmpleados como panel central.



Ventana principal con el panel CargarDatos como panel central.

**Clase CtrlCargarEmpleados**

Clase encargada de manejar la información que se obtiene en el panel CargarEmpleados que mencionamos anteriormente y pasársela a la lógica de la aplicación.

**Variables de clase**

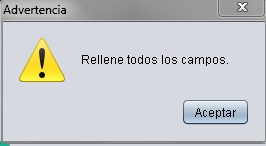
* *Modelo modelo:* lógica de la aplicación.
* *CargarEmpleado panelCargarEmpleado:*panel con el cual esta clase interactuara.
* *ListaDeNombres listaDeNombres:*lista la cual usaremos para generar empleados dinámicamente.

**Métodos de clase (más relevantes)**

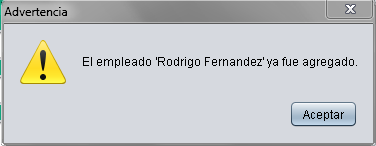
* *void elegirFoto():* abre un selector de archivos filtrando solo los archivos de tipo .png o .jpg.
* *void cargarEmpleado()*: obtiene el nombre y el apellido que se escribió en los JTextField, el puesto del JComboBox y se los pasa a la lógica. Además, guarda la foto del empleado en caso de que haya elegido alguna, sino se le asigna una imagen por defecto.
* *void generarEmpleados():* crea dinámicamente empleados dependiendo de la cantidad que halla en los JSpinner, asignándole un nombre el cual se obtiene de la listaDeNombres.
* *void eliminarEmpleados():* elimina tanto de la lógica como de la tabla los empleados seleccionados.
* *void actualizarVista():* actualiza la visualización de los distintos componentes del panel.
* *boolean esPosibleCargarEmpleado():* devuelve verdadero si los campos de información están llenos y el nombre no está repetido.
* *boolean sePuedeGenerar:* retorna verdadero si la cantidad de empleados que se desea crear no supera la cantidad de nombres que tiene la lista.
* *boolean tieneSuficientesDatos:* retorna verdadero si al menos se ha cargado un empleado de cada puesto.

Esta clase lanza ventanas emergentes como las siguientes:

En caso de que se intente cargar un empleado sin escribir el nombre o apellido o sin haber especificado un puesto.



En caso de que se intente cargar un empleado con nombre igual a uno que ya está cargado.

****

**Clase CtrlCargarDatos**

Esta clase se encarga de controlar el comportamiento del panel CargarDatos y lógica de la aplicación haciendo uso también de los controladores de los paneles CargarIncompatibles y CargarCotas.

Esta clase no hace mucho contacto directo con la lógica a diferencias de las demás.

El único método destacable de esta clase es el **actionPerformed()** el cual hará cambiar el panel izquierdo mediante los botones.

**Clase CtrlCargarIncompatibles**

Clase que se encarga de controlar el comportamiento del panel CargarIncompatibles y lógica.

**Variables de clase**

* *Modelo modelo: lógica de la aplicación.*
* *CargarIncompatibles panelCargarIncompatiles:* panel con el cual esta clase interactuara.
* *JTable tablaIncompatbiles:*tabla que esta clase usara para representar los empleados incompatibles.

**Métodos de clase**

* *void agregarIncompatibles():* toma los dos empleados seleccionados de los JComboBox y se los pasa a lógica para agregar la mala relación entre ellos.
* *void eliminaIncompatibles():* elimina la mala relación de los empleados que selecciono en la tabla.
* *void actualizarTablaDeIncompatibles():*actualiza la tabla agregando una fila mas con los datos de los dos JComboBox.

**Clase CtrlCargarCotas**

Esta clase se encarga de manejar la información que proveerá el panel CargarCotas y hacer la comunicación respectiva con la lógica.

**Variables de clase**

* *Modelo modelo: logica de la aplicación.*
* *CargarCotas panelCargarCotas:* panel con el cual esta clase interactuara.

**Métodos de clase**

* *void ejecutar():*toma los valores actuales delos JSpinner que marcan las cotas mínimas/máximas de los diferentes puesto y se los pasa a la lógica.
* *boolean tieneCotasValidas():* retorna verdadero si algunos de los valores de las cotas mínimas supera a la máxima(es decir, si la cota mínima de programadores supera a la cota máxima de programadores, etc.).

**Clase CtrlMostrarSolucion**

Clase encargada de mostrar visualmente en el panel MostrarSolucion el conjunto de empleados que la lógica encontró.

**Variables de clase**

* *Modelo modelo: lógica de la aplicación.*
* *MostrarSolucion panelMostrarSolucion:* panel con el cual esta clase interactuara.

**Métodos de clase**

* *void cargarEquipoElegido:* toma la solución de la logica y obtiene el nombre de cada empleado para buscar su foto en la clase FotosDeEmpleado y agregarla al panel correspondiente.

**Problemas encontrados**

A lo largo del desarrollo del programa nos topamos con algunas decisiones de implementación que requirieron reformular el funcionamiento interno del programa.

Una de ellas fue la implementación del algoritmo recursivo que posee el solver, que está basado en clique máxima, pero que al no tener condiciones sobre el problema para encontrar el conjunto mas grande posible que cumpla los requisitos, encontraba soluciones “casi optimas” sin tener en cuenta a algún empleado que podría haber sido incluido. Esto se arreglo re pensando el código y la estructura de datos.

Por otro lado, la ejecución del programa en un thread si bien no acarreo dificultades de implementación, trajo consigo conflictos de actualización del frame principal. Ante esto decidimos implementar una clase visual que actúe como pantalla de carga mientras el algoritmo implementado en solver encuentra la solución.

En último lugar cabe mencionar que según el seguimiento que realizamos sobre la cantidad de tiempo que tarda el algoritmo de fuerza bruta con la cantidad de empleados es razonable hasta 24 de los mismos. Luego el tiempo que tarda es demasiado tardío, ya que, como sabemos, la cantidad de veces que entra en el caso base es exponencial en un algoritmo que implemente fuerza bruta.