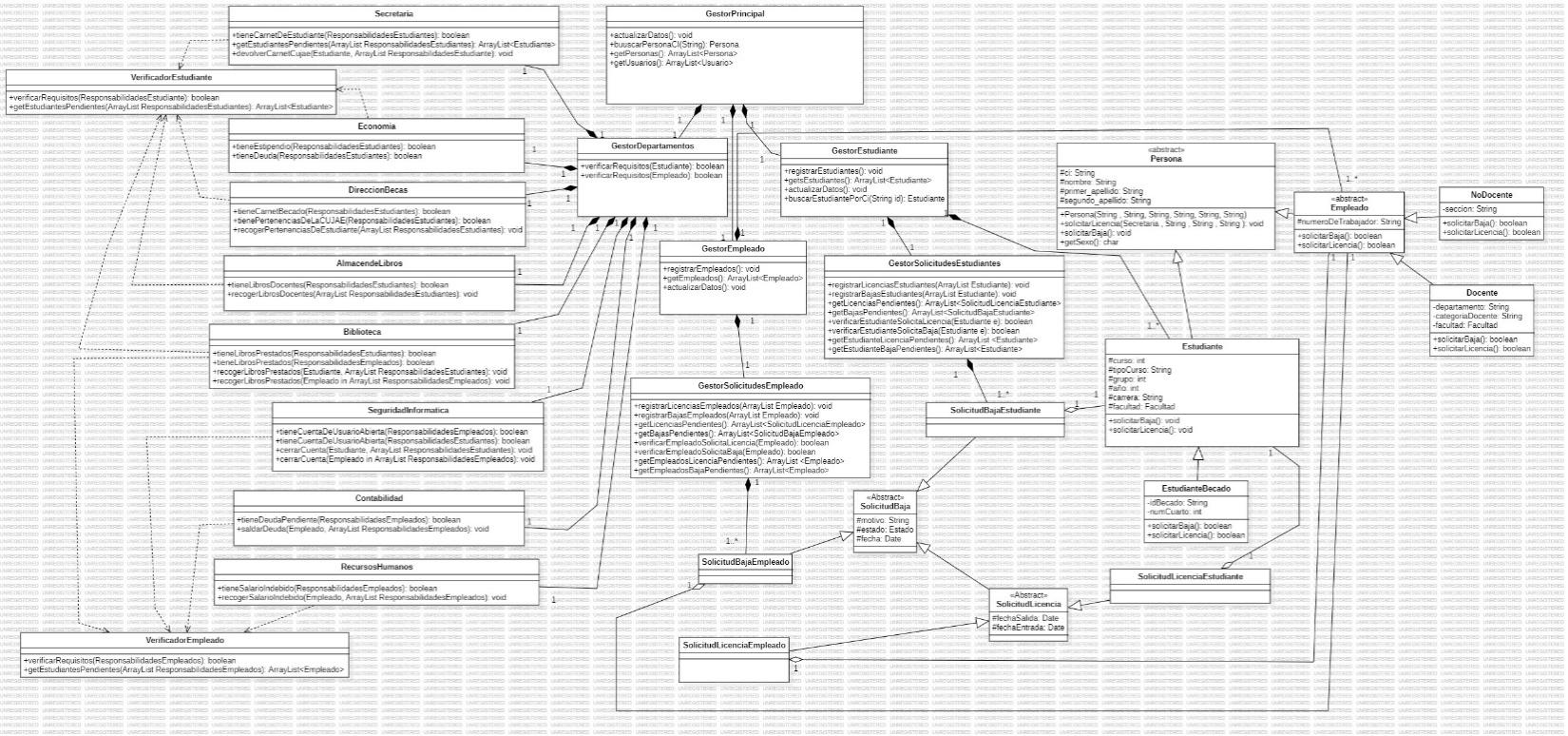
Entrega del primer corte Tarea Dpoo

Equipo 5 Grupo 13

* Tema: Correcaminos Cujae

Integrantes:

* Camilo Alfonso Gil #1
* Eduardo Fonseca Heredia #14
* Rubén Hernández Acevedo #18
* Roy Hernández Asin #19
* Yanisbe Hurtado Jiménez #21
* Marcel Pulido Fuentefría #28



1. **Explicación de cada clase (atributos y métodos) y las relaciones.**

**Clases:**

**1.Gestor Principal**

* Es laclase controladora encargada de manejar el flujo de control del sistema y la intermediaria entre las entidades que deben dar el visto bueno y el usuario.
* **Atributos:**

**private** GestorEstudiantes gestorEstudiantes;(clase encargada de registrar, obtener y actualizar los datos con respecto a los estudiantes y sus solicitudes)

**private** GestorEmpleados gestorEmpleados; ;(clase encargada de registrar, obtener y actualizar los datos con respecto a los empleados y sus solicitudes)

**private** GestorDepartamentos gestorDepartamentos; ;(clase encargada de crear las diferentes entidades que deben dar el visto bueno, encargada a su vez de verificar cada una de ellas para aceptar a un estudiante o empleado baja o licencia)

* **Relaciones:**

Gestor principal contiene atributos de los gestores mencionados anteriormente, por tanto tiene con cada uno de ellos una relación de composición al ser el encargado de crear las instancias de estos, los otros gestores serían la parte de las relaciones, y la cardinalidad de estas sería de uno a uno, debido a que el gestorPrincipal contiene 1 gestorEstudiantes, 1 gestorEmpleados y 1 gestorDepartamentos.

**2.GestorEstudiantes**

* **Atributos**

**private** GestorSolicitudesEstudiante gestorSolicitudesEstudiantes;(clase encargada de manejar las solicitudes de baja y licencia de los estudiantes)

**private** GestorResponsabilidadesEstudiantes gestorResponsabilidadesEstudiantes;(la clase responsabilidadesEstudiantes es una clase auxiliar que contiene a un estudiante y sus responsabilidades, aquello que debe entregar y no h entregado aún, de este modo GestorResponsabilidadesEstudiantes maneja las responsabilidades de los estudiantes)

**private** ArrayList<Estudiante> estudiantes;(un registro de los estudiantes de la universidad)

* **Métodos**

**private** **void** registrarEstudiantes()[Como su noombre indica se encarga de cargar los datos de los estudinates y registrarlos]

**public** Estudiante buscarEstudiantePorCi(String id)[Busca a un estudiante dado su carnet]

**public** ArrayList<Estudiante> getEstudiantes() [obtiene la lista de estudiantes de la Universidad]

public void actualizarDatos()[actualiza los datos, como puede ser eliminar un estudiante en la lista una vez dado de baja]

* **Relaciones**

Relación de todo-parte de tipo composición con Estudiante, donde Estudiante es la parte y la cardinalidad es de 1 GestorEstudiante a 1 o muchos estudiantes. Es de composición debido a que esta clase es la encargada de crear las instancias de estudiantes.

Relación de composición con GestorSolicitudesEstudiantes, porque GestorEstudiante se encarga de crear una instancia de GestorSolicitudesEstudiantes y la contiene a su vez como atributo, y la cardinlidad como se ve es de 1 a 1.

**3.GestorSolicitudesEstudiantes**

* **Atributos:**

son las diferentes listas de solicitudes tanto de baja como de licencia, las dividimos cada una en dos, donde las históricas son todas aquellas aprobadas o rechazadas ya y las pendientes son las que aún están esperando una respuesta.

Public ArrayList<SolicitudLicenciaEstudiante> solicitudesLicenciaHistoricas;

Public ArrayList<SolicitudBajaEstudiante> solicitudesBajaHistoricas;

Public ArrayList<SolicitudLicenciaEstudiante> solicitudesLicenciaPendientes;

Public ArrayList<SolicitudBajaEstudiante> solicitudesBajaPendientes;

* **Métodos:**

Registran las solicitudes de licencia y baja de los estudiantes

**private** **void** registrarLicenciasEstudiantes(ArrayList<Estudiante> estudiantes)

**private** **void** registrarBajasEstudiantes(ArrayList<Estudiante> estudiantes)

Retornan las solicitudes de baja o licencia de estudiantes que aun están pendientes de resultado

public ArrayList<SolicitudBajaEstudiante> getSolicitudesBajaPendientes()

public ArrayList<SolicitudLicenciaEstudiante> getSolicitudesLicenciaPendientes()

Verifican si un estudiante ha solicitado baja o licencia, esto sería útil para nuestra app para que una vez se sepa que el usuario ya ha solicitado en gestión solo se le muestre el estado de cumplimiento de los requisitos para obtener la baja o licencia y luego su resultado, de modo que alguien que ya solicitó la baja no vuelva a pedirla hasta que obtenga una respuesta y esta sea negativa.

**public** **boolean** verificarEstudianteSolicitaLicencia(Estudiante e)

**public** **boolean** verificarEstudianteSolicitaBaja(Estudiante e)

Devulven listas de aquellos estudiantes que han solicitado baja o licencia y aún no tienen respuesta

**public** ArrayList<Estudiante> getEstudianteLicenciaPendientes()

**public** ArrayList<Estudiante> getEstudianteBajaPendientes()

* Relaciones

GestorSolicitudesEstudiantes tiene listas de solicitudes, lo que se refleja en una relación de composición con la clase SolicitudBajaEstudiante puesto que es la encargada de crear las instancias de esta, donde SolicitudBajaEstudiante es la parte, la cardinalidad es de 1 a 1 o varias.

**4 y 5**.Las clases **GestorEmpleados** y **GestorSolicitudesEmpleados** tiene atributos y métodos semejantes a los de GestorEstudiantes y GestorSolicitudesEstudiantes la única diferencia es que estos manejan los datos y solicitudes de los empleados y no de los estudiantes.En cuanto a relaciones es semejante de igual manera que con los atributos y métodos, la diferencia es que en vez de ser la clase SolicitudBajaEstudiante seria SolicitudBajaEmpleado.

**6.GestorDepartamentos**

* **Atributos:** una instancia de cada entidad (Biblioteca, Economía, Almacén de libros…)
* **Métodos:** verificarRequisitos(Estudiante) y verificarRequisitos(Empleado) de tipo boolean que verifican en todas las entidades que se requieran dependiendo de si es estudiante o empleado, aquellos requisitos necesarios para recibir baja o licencia como entrega de libros en Biblioteca o cerrar cuenta en Seguridad Informática.
* **Relaciones:** tiene una relación de composición con cada una de las entidades, donde estas son la parte. Debido a que es el encargado de crear dichas clases. La cardinalidad de estas relaciones de 1 a 1, porque en la universidad solo existe una biblioteca, un departamento economía y así sucesivamente, solo debe existir una instancia de cada departamento (patrón singleton).

**7-15 Entidades**

* **Métodos**: todas poseen métodos para verificar requisitos que reciben como parámetros, un estudiante, un empleado, o las responsabilidades(clase auxiliar que contiene a un estudiante o empleado y sus requisitos pendientes), estos métodos son booleanos devuleven true si todo esta cumplido y ne caso cotrario false.
* **Relaciones**: las entidades implementan dos interfaces (relaciones de asociación):

VerificadorEstudiante(Biblioteca, Secretaría,Economía,Dirección de Becas, Almacén de Libros y Seguridad Informática) esta interface contiene los métodos para verificar todos los requisitos de un estudiante y obtener una lista de aquellos pendientes de algún requisito. Cuando las entidades implementen el método para verificar, será un método general que llame a cada método específico de la propia entidad por requisito a cumplir. En Economía por ejemplo, llamaría a tieneEstipendio() y tieneDeuda() para realizar la verificación.

VerificadorEmpleado(Biblioteca, Contabilidad, Recursos Humanos, Seguridad Informática) esta es similar a la anterior pero con empleados.

Las clases **Estudiante** y **Empleado** heredan de la clase padre **Persona** debido a que tiene propiedades y métodos en común con este, sin embargo traen a su vez atributos propios como Carrera, curso en Estudiante y número de trabajador en caso del Empleado. Persona sería la clase padre, la generalización, mientras que Estudiante y Empleado, serían clases hijas de esta que tienen los mismos atributos y métodos de Persona, pero con sus particularidades. De esta manera usuario sería una clase abstracta porque no es necesario instanciarla, la persona en este caso siempre va a ser o un estudiante o un empleado.

Un estudiante es una persona.

Un empleado es una persona.

Lo mismo ocurre con **Empleado** y las clases **Docente** y **Nodocente**, Empleado sería abstracta y Docente y Nodocente serían sus clases hijas, al poseer sus mismas características y métodos pero con algunas particularidades.

Un docente es un empleado.

Un no docente es de igual manera un empleado.

La clase **Estudiante becado** hereda de la clase **estudiante**, porque un estudiante becado es un estudiante, con algunas particularidades.

Todas estas clases poseen los **métodos** de solicitar baja y solicitar licencia (abstratos en Persona y Empleado) y la clase Persona posee además un método getSexo() de tipo char que a partir del CI obtiene el sexo de la persona.

Atributos:

**Persona**

**protected** String ci;

**protected** String nombre;

**protected** String primer\_apellido;

**protected** String segundo\_apellido;

**Estudiante**

protected int curso;

protected TipoCurso tipoCurso;

protected String carrera;

protected int grupo;

protected Facultad facultad;

**Estudiante Becado**

**private** String idBecado;

**private** **int** noCuarto;

**Empleado**

**protected** String numerotrabajador;

**Empleado Docente**

**private** String departamento;

**private** String categoriaDocente;

private Facultad facultad;

**Empleado No Docente**

**private** String seccion;

Clase abstracta SolicitudBaja

* **Atributos**

-**protected String motivo** (causa por la que se solicita baja o licencia en este caso dado que la clase SolicitudLicencia hereda de esta clase, ya que la única diferencia en los tramites de solicitar baja o licencia es que esta ultima tiene una fecha de salida y una fecha de regreso)

-**protected Estado estado** (es un enum porque las solicitudes pueden estar solo en 3 estados: aprobada, rechazadas o pendiente.

-**protected Date fecha** este atributo sería la fecha en que se realiza la solicitud

Clase abstracta **SolicitudLicencia** que hereda de **SolicitudBaja** por lo explicado anteriormente

* **Atributos**

Además de los de su clase padre, posee los atributos de tipo Date fechaSalida y fechaRegreso.

Las clases **SolicitudBajaEstudiante** y **SolicitudBajaEmpleado** heredan de la clase SolicitudBaja, poseen entonces todos los **atributos** de esta y a su vez un atributos de tipo Estudiante en la primera y Empleado en la segunda. Esto implica que exista una **relación** de agregación entre Estudiante y SolicitudBajaEstudiante, donde el estudiante es la parte y la cardinalidad es de 1 a 1, porque una SolicitudBajaEstudiante contiene a un estudiante, es de agregación porque el estudiante es creado ya por el gestorEstudiantes y una clase solo puede estar involucrada en una relación de parte y todo de tipo composición donde ella sea la parte. Ocurre lo mismo con la Clase Empleado y la clase SolicitudBajaEmpleado.

Las clases **SolicitudLicenciaEstudiante** y **SolicitudLicenciaEmpleado** heredan de la clase SolicitudLicencia, poseen entonces todos los **atributos** de esta y a su vez un atributo de tipo Estudiante en la primera y Empleado en la segunda. Esto implica que exista una **relación** de agregación entre Estudiante y SolicitudLicenciaEstudiante, donde el estudiante es la parte y la cardinalidad es de 1 a 1, porque una SolicitudLicenciaEstudiante contiene a un estudiante, es de agregación porque el estudiante es creado ya por el gestorEstudiantes y una clase solo puede estar involucrada en una relación de parte y todo de tipo composición donde ella sea la parte. Ocurre lo mismo con la Clase Empleado y la clase SolicitudLicenciaEmpleado.

1. Explicación de la división por capas de la solución

Este proyecto sigue el patrón de diseño de software Modelo-Vista-Controlador (MVC). Está organizado en diferentes paquetes según su funcionalidad, lo que facilita su ubicación y modificación:

* app: Contiene las clases que son responsables de iniciar la aplicación y configurar el entorno de ejecución. Esta capa es importante porque establece las condiciones iniciales para que la aplicación pueda funcionar correctamente y de forma segura.
* auth: Contiene las clases relacionadas con la autenticación, como Auth.java, Session.java y VerificarCredenciales.java. Estas clases son responsables de manejar el proceso de autenticación y autorización de los usuarios que utilizan la aplicación. Esto implica verificar las credenciales de los usuarios, crear y destruir sesiones, asignar roles y permisos, y controlar el acceso a las diferentes partes de la aplicación. Esta capa es importante porque garantiza la seguridad y la privacidad de los datos y evita el acceso no autorizado a la aplicación.
* controllers: Contiene los controladores del proyecto. Estas clases son responsables de manejar las solicitudes del usuario y actualizar el modelo y la vista según sea necesario. Los controladores reciben las solicitudes del usuario, validan los datos de entrada, invocan los métodos del modelo para obtener o modificar los datos, y envían los datos a la vista para que los muestre al usuario. Los controladores también pueden redirigir las solicitudes a otros controladores. Esta capa es importante porque actúa como intermediario entre el modelo y la vista, y coordina el flujo de la aplicación.
* data: Contiene las clases relacionadas con el acceso a datos. Estas clases son responsables de interactuar con fuentes de datos, en este caso archivos en formato JSON usando la librería Gson de Google para obtener o actualizar información. Esta capa es importante porque abstrae la lógica de acceso a datos de las demás capas y permite una mayor independencia y flexibilidad.
* diagramas: Contiene los diagramas utilizados en el proyecto, en formato de imágenes PNG y archivos de diagramas. Estos diagramas son utilizados para documentar la arquitectura o el diseño de la aplicación. Pueden ser de diferentes tipos, como diagramas de clases UML y de actividades. Esta capa es importante porque ayuda a visualizar y comprender mejor el funcionamiento y la estructura de la aplicación, y facilita la comunicación entre los desarrolladores y los clientes o usuarios.
* img: Contiene las imágenes utilizadas en el proyecto. Estas imágenes son utilizadas en la vista para mejorar la presentación de los datos. Pueden ser de diferentes tipos, como iconos, logos, gráficos, fotos, etc.
* models: Contiene los modelos de datos, organizados en diferentes paquetes según su funcionalidad, como departamentos, gestión, interfaces, responsabilidades, solicitudes y usuarios. Estas clases definen la estructura de los datos y su comportamiento, y son utilizadas por el controlador y la vista para mostrar o actualizar información. Algunas implementan patrones de diseño como Singleton. Esta capa es importante porque representa el dominio del problema y la lógica de negocio de la aplicación, y permite una mayor coherencia y consistencia de los datos.
* utiles: Contiene las clases de utilidad, como Colores, Facultades, Tipos de Responsabilidades las cuales proporcionan valores comunes que pueden ser utilizadas por diferentes partes del proyecto. Estas pueden tener diferentes tipos de datos, como constantes y enumeraciones. Esta capa es importante porque ayuda a evitar la duplicación de código y a simplificar el desarrollo de la aplicación.
* views: Contiene las vistas del proyecto. Estas clases son responsables de mostrar los datos al usuario y de recibir sus acciones. Las vistas tienen diferentes elementos, como etiquetas, botones, campos, tablas, formularios, etc. Esta capa es importante porque es la que interactúa directamente con el usuario y le permite usar la aplicación.

MVC es un buen patrón de diseño porque tiene las siguientes ventajas:

* Facilita el desarrollo y el mantenimiento de la aplicación, al separar las diferentes responsabilidades en capas independientes y bien definidas.
* Permite una mayor reutilización de código, al evitar la duplicación de lógica o datos entre las capas.
* Mejora la escalabilidad y el rendimiento de la aplicación, al permitir modificar o ampliar una capa sin afectar a las demás.
* Favorece la modularidad y la flexibilidad de la aplicación, al permitir cambiar o sustituir una capa por otra sin alterar el funcionamiento general.
* Mejora la calidad y la fiabilidad de la aplicación, al facilitar la realización de pruebas y la detección de errores en cada capa.

**Explicación general de los diagramas de actividades** porque todos realizan actividades similares, lo único que cambian son los requisitos.

El Gestor Solicitudes Estudiante si el usuario es un estudiante o el Gestor Solicitudes Empleados si es un empleado, inician este proceso al solicitar verificación de requisitos del usuario para saber si acepta o no la solicitud. Entonces envía a gestor departamentos quien manda entonces a cada departamento por separado a verificar si el usuario cumple con los requisitos necesarios, en caso de Biblioteca si no debe libros, en caso de economía si no tiene deuda o estipendio, si al revisar las responsabilidades del usuario se obtiene que debe algún requisito el departamento notifica a GestorDepartamentos incumplimiento de requisitos, en caso de que no deba nada notifica cumplimiento de requisitos , GestorDepartamentos notifica a Gestor Solicitudes y en caso de que estén todos los requisitos cumplidos este acepta la solicitud, en caso contrario la deja pendiente aún.

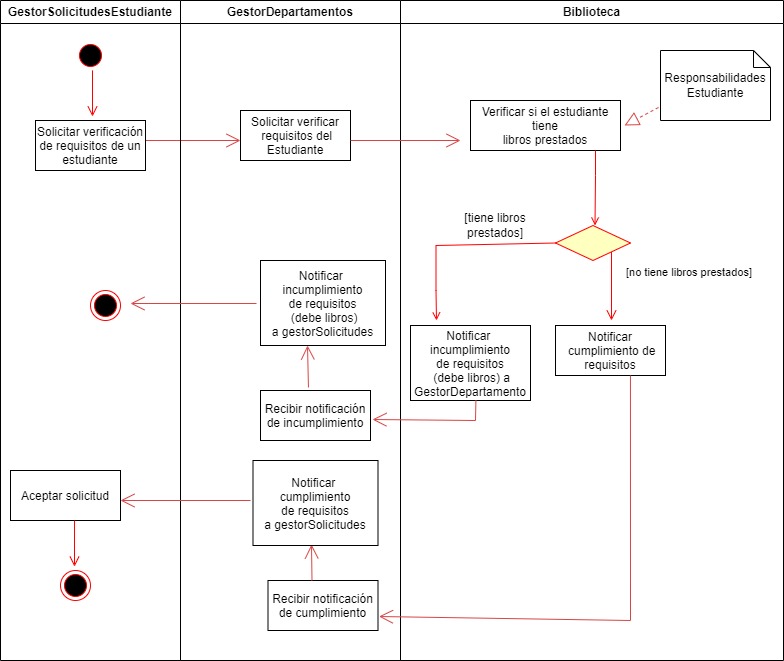
El único diagrama diferentes es el del login, que en este caso el usuario ingresa sus datos, el sistema verifica si se encuentra en la lista de usuarios registrados, si lo está le muestra su pantalla correspondiente y si no le pide que vuelva a ingresar sus datos.

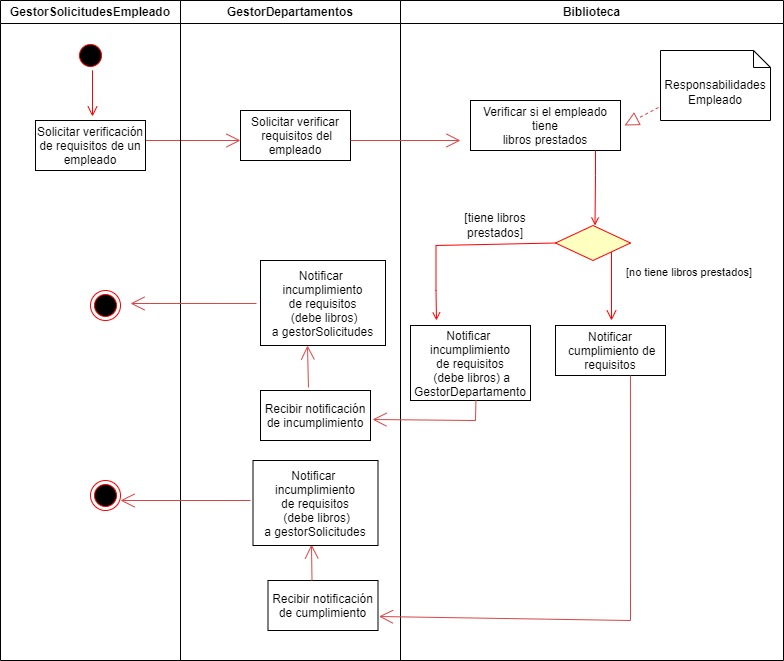
En algunos diagramas solo se evidencia la funcionalidad a partir de GestorDepartamentos.

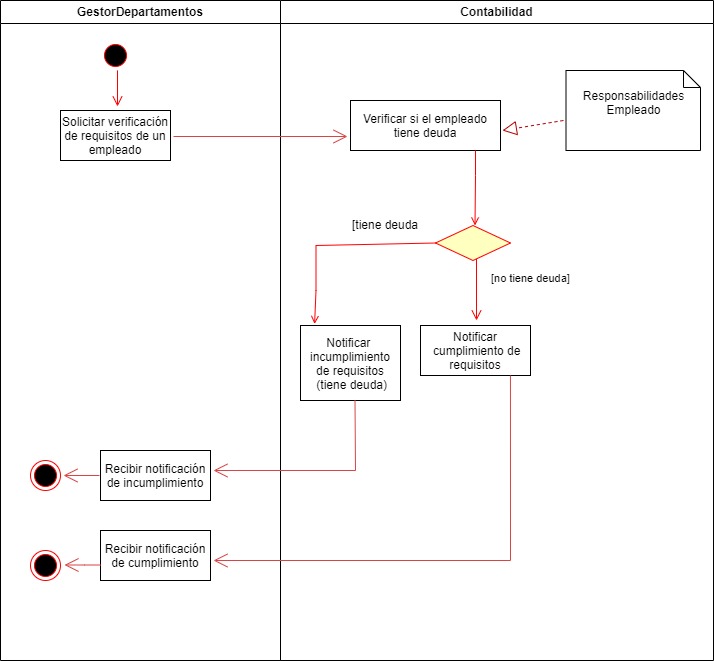
**Subequipo 1**

**Yanisbe Hurtado Jiménez**

**Roy Hernández Asin**





****

**Funcionalidad (Reporte)**

**//GestorSolicitudesEstudiante**

**public** ArrayList<BajasAceptadasPorFacultad> obtenerCantidadBajasAceptadasPorFacultad() { ArrayList<BajasAceptadasPorFacultad> bajas = **new** ArrayList<>();

**for** (SolicitudBajaEstudiante s : solicitudesBajaAceptadas) {

**boolean** encontrado = **false**;

**for** (**int** i = 0; i < bajas.size() && !encontrado; i++) {

**if** (bajas.get(i).getFacultad().equals(s.getEstudiante().getFacultad())) {

bajas.get(i).setCantidad(bajas.get(i).getCantidad() + 1);

encontrado = **true**;

}

}

**if** (!encontrado) {

bajas.add(**new** BajasAceptadasPorFacultad(s.getEstudiante().getFacultad(), 1));

}

} **return** bajas;

} **public** ArrayList<BajasAceptadasPorFacultad> ordenarFacultadesPorCantidadBajasAceptadas() {

ArrayList<BajasAceptadasPorFacultad> bajasOrdenadas = obtenerCantidadBajasAceptadasPorFacultad();

Collections.sort(bajasOrdenadas, **new** Comparator<BajasAceptadasPorFacultad>() {

@Override

**public** **int** compare(BajasAceptadasPorFacultad b1, BajasAceptadasPorFacultad b2) {

**return** Integer.compare(b1.getCantidad(), b2.getCantidad()); }

});

**return** bajasOrdenadas;

}

// controlador reportes

**public** **static** **void** facultadesConMasBajasAceptadas() {

ArrayList<BajasAceptadasPorFacultad> ordenado = GestorEstudiantes.gestorEstudiantes().getGestorSolicitudes() .ordenarFacultadesPorCantidadBaja sAceptadas();

Collections.reverse(ordenado);

ArrayList<Facultad> facMasBajas = **new** ArrayList<>();

**for** (**int** i = 0; i < ordenado.size(); i++) {

**if** (ordenado.get(i).getCantidad() == ordenado.get(0).getCantidad()) {

facMasBajas.add(ordenado.get(i).getFacultad());

} **else** {

i = ordenado.size();

}

}

String info = "Lista de facultades con mayor cantidad de bajas aceptadas:\n";

**for** (Facultad f : facMasBajas) {

info += "- " + f + "\n";

}

JTextArea textArea = crearTextArea(info);

JOptionPane.showMessageDialog(**null**, textArea);

}

**import** util.Facultad;

**public** **class** BajasAceptadasPorFacultad {

**private** Facultad facultad;

**private** **int** cantidad;

**public** BajasAceptadasPorFacultad(Facultad facultad, **int** cantidad) {

**this**.facultad = facultad;

**this**.cantidad = cantidad;

}

**public** Facultad getFacultad() {

**return** facultad;

}

**public** **void** setFacultad(Facultad facultad) {

**this**.facultad = facultad;

}

**public** **int** getCantidad() {

**return** cantidad;

}

**public** **void** setCantidad(**int** cantidad) {

**this**.cantidad = cantidad; }

}

La funcionalidad tiene como objetivo manejar solicitudes de bajas de estudiantes de una universidad y generar un reporte de las facultades con la mayor cantidad de bajas aceptadas.

El código consta de las siguientes partes:

1. Clase BajasAceptadasPorFacultad: Esta clase define un objeto que representa la cantidad de bajas aceptadas por cada facultad.

2. Método obtenerCantidadBajasAceptadasPorFacultad(): Este método recorre las solicitudes de baja aceptadas por estudiantes y cuenta la cantidad de bajas por facultad, devolviendo una lista de objetos BajasAceptadasPorFacultad.

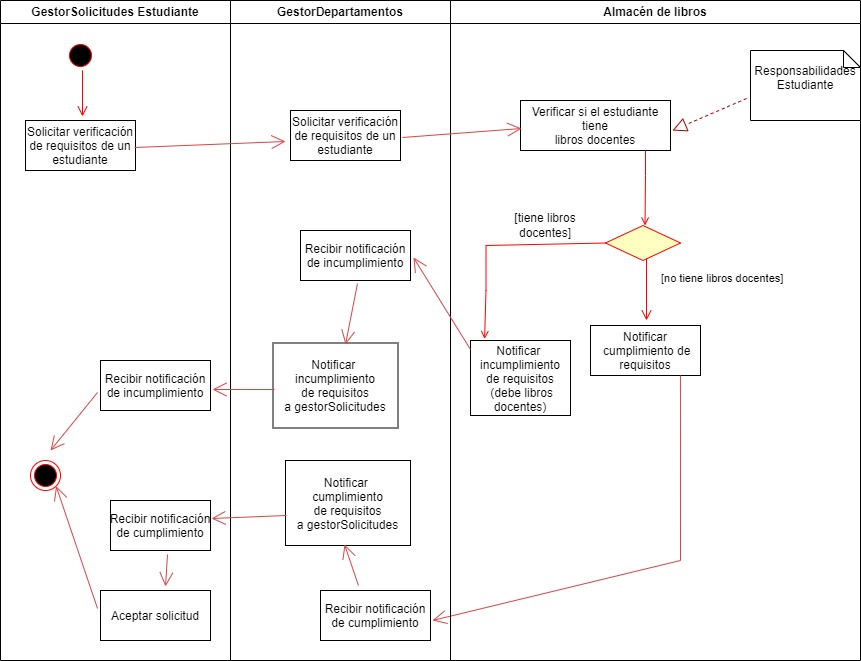
3. Método ordenarFacultadesPorCantidadBajasAceptadas(): Este método invoca obtenerCantidadBajasAceptadasPorFacultad, luego ordena la lista de bajas aceptadas por facultad en orden ascendente según la cantidad de bajas.

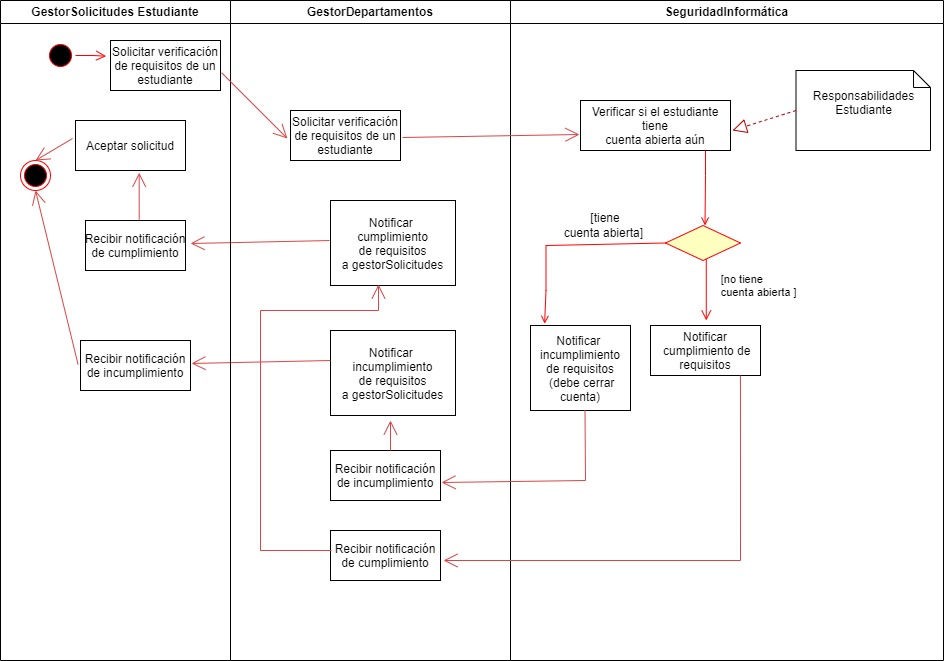
4. Método facultadesConMasBajasAceptadas(): Este método permite obtener las facultades con la mayor cantidad de bajas aceptadas. Primero obtiene la lista de facultades ordenadas por cantidad de bajas, luego selecciona aquellas con la mayor cantidad y las muestra en un cuadro de diálogo.

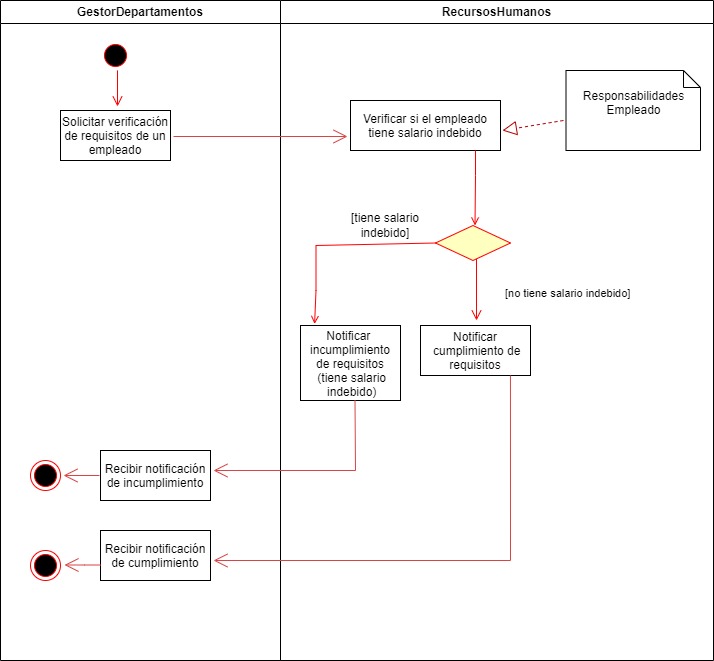
**Subequipo 2**

**Rubén Hernández Acevedo**

**Eduardo Enrique Fonseca**



****

****

**Funcionalidad**

Clase Auth:

public class Auth {

    private static Auth instancia;

    private Usuario usuario;

    private Auth() {

    }

    public static Usuario usuarioAutenticado() {

        if (instancia == null) {

            instancia = new Auth();

        }

        return instancia.getUsuario();

    }

    public static Auth obtenerInstancia() {

        if (instancia == null) {

            instancia = new Auth();

        }

        return instancia;

    }

    public static void iniciarSesion(Usuario usuario) {

        if (instancia == null) {

            instancia = new Auth();

        }

        instancia.login(usuario);

    }

    public static boolean hayUsuarioAutenticado() {

        if (instancia == null) {

            instancia = new Auth();

        }

       return instancia.existe();

    }

    public static void cerrarSesion() {

        if (instancia == null) {

            instancia = new Auth();

        }

        instancia.logout();

    }

    private Usuario getUsuario() {

        return usuario;

    }

    private void login(Usuario usuario) {

        this.usuario = usuario;

    }

    private boolean existe() {

        return usuario != null;

    }

    private void logout() {

        usuario = null;

    }

}

Clase Sessions:

public class Session {

    private String nombreUsuario;

    private String password;

    public Session(String nombreUsuario, String password) {

        this.nombreUsuario = nombreUsuario;

        this.password = password;

        Gson gson = new Gson();

        try (BufferedWriter writer = new BufferedWriter(new FileWriter("./session.json"))) {

            String json = gson.toJson(this);

            writer.write(json);

        } catch (IOException e) {

            e.printStackTrace();

        }

    }

    private static Session obtenerSession() {

        Gson gson = new Gson();

        File file = new File("./session.json");

        Session session = null;

        if (file.exists()) {

            try (BufferedReader reader = new BufferedReader(new FileReader(file))) {

                session = gson.fromJson(reader, Session.class);

            } catch (IOException e) {

                e.printStackTrace();

            }

        }

        else{

           session = new Session("null", "null");

        }

        return session;

    }

    public static Usuario ValidarSession() {

        boolean encontrado = false;

        ArrayList<Usuario> usuarios = GestorPrincipal.getGestorPrincipal().getUsuarios();

        Session session = obtenerSession();

        Usuario u = null;

        for (int i = 0; i < usuarios.size() && !encontrado; i++) {

            encontrado = session.getNombreUsuario().equals(usuarios.get(i).getNombreUsuario());

            if (encontrado && session.getPassword().equals(usuarios.get(i).getContrasena())) {

                u = usuarios.get(i);

            }

        }

        return u;

    }

    private String getNombreUsuario() {

        return nombreUsuario;

    }

    private String getPassword() {

        return password;

    }

}

Clase VerificarCredenciales:

public class VerificarCredenciales {

private boolean credencialesCorrectas;

private Usuario usuario;

    public VerificarCredenciales(String nombreUsuario, String contrasena){

        usuario= null;

        credencialesCorrectas =  autenticar(nombreUsuario, contrasena);

    }

    private boolean autenticar(String nombreUsuario, String contrasena) {

        boolean encontrado = false;

        boolean autenticado = false;

        ArrayList<Usuario> usuarios = GestorPrincipal.getGestorPrincipal().getUsuarios();

        for (int i = 0; i < usuarios.size() && !encontrado; i++) {

            encontrado = nombreUsuario.equals(usuarios.get(i).getNombreUsuario());

            if (encontrado && hashContrasena(contrasena).equals(usuarios.get(i).getContrasena())) {

                autenticado = true;

                this.usuario = usuarios.get(i);

            }

        }

        return autenticado;

    }

     private  String hashContrasena(String contrasena) {

        try {

            MessageDigest md = MessageDigest.getInstance("SHA-256");

            byte[] passwordBytes = contrasena.getBytes();

            byte[] hashBytes = md.digest(passwordBytes);

            StringBuilder hexString = new StringBuilder();

            for (byte hashByte : hashBytes) {

                hexString.append(String.format("%02x", hashByte));

            }

            return hexString.toString();

        } catch (NoSuchAlgorithmException e) {

            e.printStackTrace();

            return null;

        }

    }

    public boolean credencialesCorrectas() {

        return credencialesCorrectas;

    }

    public Usuario usuarioAlQuePertenecen() {

        return usuario;

    }

}

Métodos:

 private boolean autenticar(String nombreUsuario, String contrasena):

Se utiliza para autenticar las credenciales de un usuario. Primero, inicializa dos variables booleanas, encontrado y autenticado, en false. Luego, obtiene la lista de usuarios del GestorPrincipal. Recorre la lista de usuarios. Si encuentra un usuario cuyo nombre coincide con nombreUsuario, verifica si la contraseña proporcionada, después de ser procesada por el método hashContrasena, coincide con la contraseña del usuario encontrado. Si es así, establece autenticado en true y asigna el usuario a la variable usuario de la clase.Finalmente, el método devuelve el valor de autenticado. Si autenticado es true, significa que las credenciales son correctas. Si es false, las credenciales son incorrectas.

private String hashContrasena(String contrasena):

Se utiliza para convertir una contraseña en texto plano a una versión cifrada utilizando el algoritmo de hash SHA-256. Primero, se crea una instancia del algoritmo de hash SHA-256 utilizando MessageDigest.getInstance("SHA-256"). Luego, la contraseña en texto plano se convierte en bytes utilizando contrasena.getBytes(). Estos bytes se pasan al algoritmo de hash SHA-256 utilizando md.digest(passwordBytes), que devuelve un array de bytes que representa la contraseña cifrada. Finalmente, este array de bytes se convierte en una cadena hexadecimal, que es la representación en cadena de la contraseña cifrada. Este método es útil para almacenar contraseñas de manera segura, ya que incluso si alguien obtiene la versión cifrada de la contraseña, no puede convertirla fácilmente de nuevo a la versión en texto plano. Esto se debe a que SHA-256 es un algoritmo de hash unidireccional. Sin embargo, si dos personas tienen la misma contraseña, sus hashes serán iguales. Este método también maneja la excepción NoSuchAlgorithmException, que se produce si el algoritmo de hash SHA-256 no está disponible en el entorno. En este caso, el método imprime la traza de la pila de la excepción y devuelve null.

En la clase GestorSolicitudesEstudiante:

public int cantidadDeBajasDeUnAnio(int anio) {

        int total = 0;

        for (SolicitudBajaEstudiante s : solicitudesBajaAceptadas) {

            if (s.getAnioExpedicion() == anio) {

                total++;

            }

        }

        return total;

    }

Este método cantidadDeBajasDeUnAnio devuelve la cantidad de bajas aceptadas en un año especifico. Recorre el ArrayList de solicitudes de bajas aceptadas de estudiantes, verifica si el año de cada una coinside con el año suministrado, de ser asi incrementa en 1 la la cantidad total.

En la clase ControladorReportes:

public static void totalBajasAceptadasElAnioAnterior() {

        Calendar calendar = Calendar.getInstance();

        int anioActual = calendar.get(Calendar.YEAR);

        int anioAnterior = anioActual - 1;

        int total = GestorEstudiantes.gestorEstudiantes().getGestorSolicitudes().cantidadDeBajasDeUnAnio(anioAnterior);

        String info = "El total de licencias aceptadas el año anterior fue de: " + total;

        JTextArea textArea = crearTextArea(info);

        JOptionPane.showMessageDialog(null, textArea);

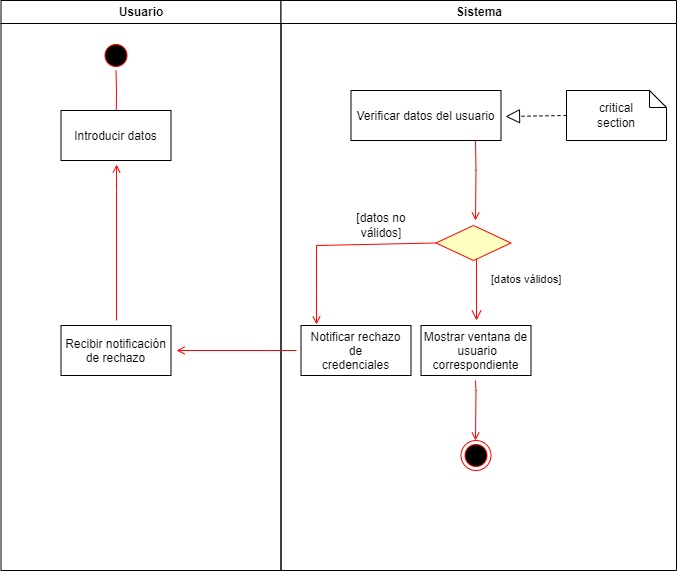
}

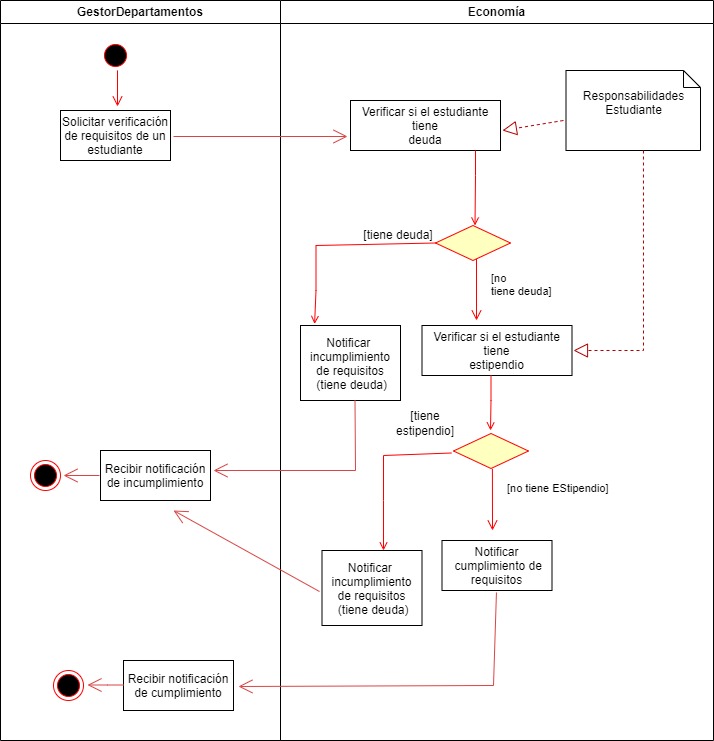
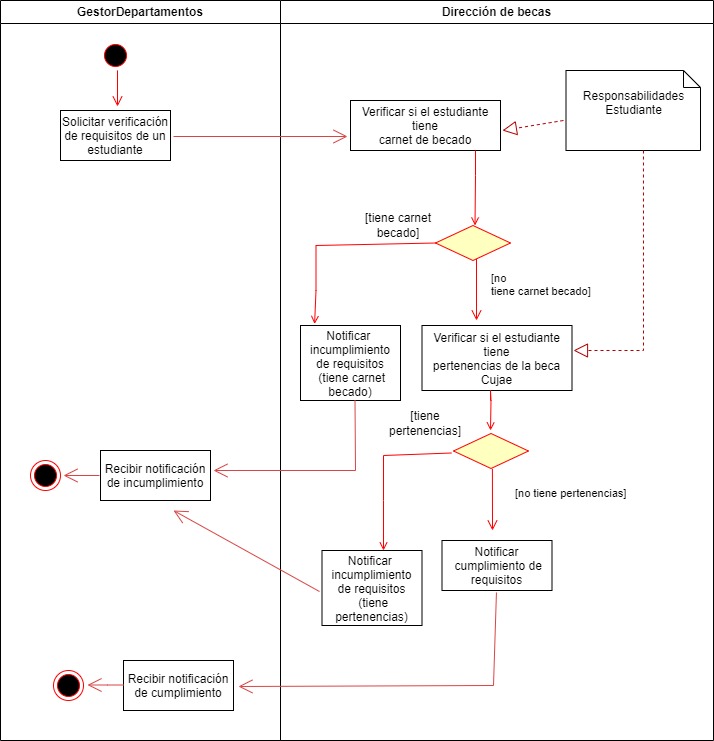
Este método se encarga de obtener y mostrar la cantidad de bajas aceptadas del año anterior, usando el método cantidadDeBajasDeUnAnio.

**Subequipo 3**

**Marcel Pulido FuenteFria**

**Camilo Enrique Alfonso**





//GESTOR SOLICITUDES ESTUDIANTES

public ArrayList<BajasAceptadasPorAnio> cantidadDeBajasPorAnio(int anioMinimo) {

ArrayList<BajasAceptadasPorAnio> solicitudes = new ArrayList<>();

for (SolicitudBajaEstudiante s : solicitudesBajaAceptadas) {

boolean encontrado = false;

if (s.getAnioExpedicion() >= anioMinimo) {

for (int i = 0; i < solicitudes.size() && !encontrado; i++) {

if (solicitudes.get(i).getAnio() == s.getAnioExpedicion()) {

solicitudes.get(i).setCantidad(solicitudes.get(i).getCantidad() + 1);

encontrado = true;

}

}

if (!encontrado) {

solicitudes.add(new BajasAceptadasPorAnio(s.getAnioExpedicion(), 1));

}

}

}

return solicitudes;

}

//CONTROLADOR REPORTES

public static void anioConMasBajasLicenciasAceptadasEnUltimos10() {

Calendar calendar = Calendar.getInstance();

int anioActual = calendar.get(Calendar.YEAR);

ArrayList<BajasAceptadasPorAnio> solicitudes = GestorEstudiantes.gestorEstudiantes().getGestorSolicitudes()

.cantidadDeBajasPorAnio(anioActual - 10);

ArrayList<Integer> aniosMasBajas = new ArrayList<>();

int maxBajas = 0;

for (BajasAceptadasPorAnio solicitud : solicitudes) {

int cantidadBajas = solicitud.getCantidad();

if (cantidadBajas > maxBajas) {

maxBajas = cantidadBajas;

aniosMasBajas.clear();

aniosMasBajas.add(solicitud.getAnio());

} else if (cantidadBajas == maxBajas) {

aniosMasBajas.add(solicitud.getAnio());

}

}

String info = "El anio(s) con mayor contidad de bajas y licencias aceptadas fue:\n";

for (int anio : aniosMasBajas) {

info = " - " + anio + "\n";

}

JTextArea textArea = crearTextArea(info);

JOptionPane.showMessageDialog(null, textArea);

}

//CLASE AUX

public class BajasAceptadasPorAnio {

private int anio;

private int cantidad;

public BajasAceptadasPorAnio(int anio, int cantidad) {

this.anio = anio;

this.cantidad = cantidad;

}

public int getAnio() {

return anio;

}

public void setAnio(int anio) {

this.anio = anio;

}

public int getCantidad() {

return cantidad;

}

public void setCantidad(int cantidad) {

this.cantidad = cantidad;

}

}

El código consta de las siguientes partes:

1. cantidadDeBajasPorAnio(): Este método devuelve una lista de objetos BajasAceptadasPorAnio, que representan la cantidad de solicitudes de baja aceptadas por año.
2. anioConMasBajasLicenciasAceptadasEnUltimos10(): Este método muestra un mensaje de diálogo con el año o años que tienen la mayor cantidad de bajas y licencias aceptadas en los últimos 10 años.
3. BajasAceptadasPorAnio:Este código define una clase llamada BajasAceptadasPorAnio que representa un objeto que contiene información sobre la cantidad de bajas aceptadas en un año específico.