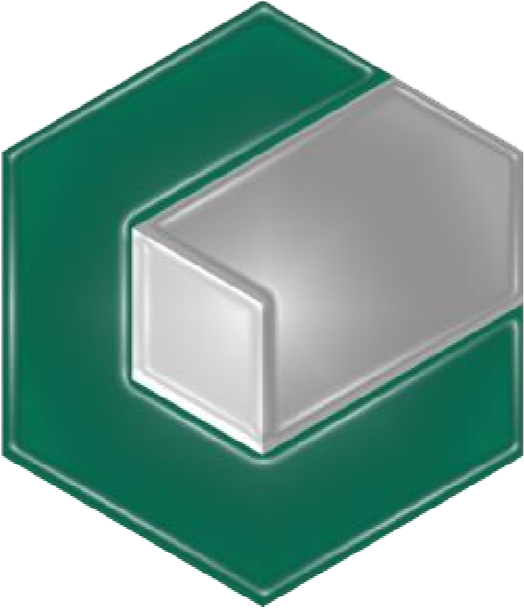
**Universidad Tecnológica de La Habana**

**“José Antonio Echeverría”**



**Proyecto Final**

**Tema 5**

**“Correcaminos Cujae”**

**Autores:**

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

**Asignatura:** Diseño y Programación Orientada a Objetos

**Año Académico:** 1ro.

**Carrera:** Ing. Informática

La Habana, 2023

## Documentación de las clases identificadas

### 1. Listado de Clases Identificadas

Tras el análisis del problema del trabajo de curso, se identificaron las siguientes clases necesarias para el modelado del programa en un diagrama de clases *UML*:

\*NOTA: el diagrama de clases al ser muy extenso no se distinguían bien los métodos y atributos de cada clase, por lo que lo incluimos en un archivo fuera del Word de nombre Correcaminos.mdzip.

**1.Correcaminos**: Es la clase 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:

1. private Secretaría secretaría: clase encargada de registrar, obtener y actualizar los datos con respecto a los estudiantes y sus solicitudes.
2. private GestorEmpleados gestorEmpleados: clase encargada de registrar, obtener y actualizar los datos con respecto a los empleados y sus solicitudes.
3. 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.

* Métodos

public Persona buscarPersonaporCi(String id)

public ArrayList<Persona> getPersonas()

* Relaciones:

Correcaminos contiene como 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 Correcaminos contiene 1 Secretaría, 1 gestorEmpleados y 1 gestorDepartamentos.

**2.Secretaría**

* Atributos

1. private GestorSolicitudesEstudiante gestorSolicitudesEstudiantes: clase encargada de manejar las solicitudes de baja y licencia de los estudiantes.
2. private ArrayList<Estudiante> estudiantes: un registro de los estudiantes de la universidad.

* Métodos

1. filtrar(Map<String,String>): ArrayList<Estudiante> este método es el de filtrado general que utiliza el método 3, devuelve la lista filtrada de estudiantes por facultad o nombre.
2. cargarInformacionEstudiantes (): void método que registra a todos los estudiantes de la universidad, cargando sus datos.
3. filtrar(String):ArrayList<Estudiante> método de filtrado de estudiantes dado un valor, puede ser facultad o nombre, mostrando solo aquellos que coincidan con el valor de entrada.
4. getEstudiantes(): ArrayList<Estudiante> método que retorna todos registrados.

* Relaciones
  1. Relación de todo-parte de tipo composición con Estudiante, donde Estudiante es la parte y la cardinalidad es de 1 Secretaría a 1 o muchos estudiantes. Es de composición debido a que esta clase es la encargada de crear las instancias de estudiantes.
  2. Relación de composición con GestorSolicitudesEstudiantes, porque Secretaría 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:

private HashMap<Estudiante, Set<Solicitud>> solicitudes : estudiantes y sus solicitudes.

* Métodos

1. private void registrarSolicitudes() método para registrar todas las solicitudes de estudiantes, cargándola desde los archivos.
2. public HashMap<Estudiante, Set<Solicitud>> getSolicitudes() método para obtener la lista de solicitudes relacionadas el estudiante que la solicita.
3. public boolean verificarEstudianteSolicitaAlgo(Estudiante e) método que permite verificar si un estudiante ha solicitado baja o licencia, retorna un valor lógico true en caso de que el estudiante se encuentre en alguna solicitud y de lo contrario retorna false.
4. public ArrayList<Estudiante> obtenerEstudiantesPendientes() método que permite obtener la lista de estudiantes que están pendientes, lo que quiere decir que su solicitud aún no ha sido aceptada o cancelada.
5. public void crearSolicitudBaja(MotivoBaja motivo, Estudiante e) crea una nueva solicitud de baja de un estudiante y la agrega a la lista de solicitudes.
6. public void crearSolicitudLiciencia(MotivoLicencia motivo, Estudiante e) crea una nueva solicitud de licencia de un estudiante y la agrega a la lista de solicitudes.

* Relaciones

1. GestorSolicitudesEstudiantes tiene listas de solicitudes, lo que se refleja en una relación de composición con la clase Solicitud puesto que es la encargada de crear las instancias de las solicitudes de los estudiantes, donde Solicitud es la parte, la cardinalidad es de 1 a 1 o varias.
2. A su vez está presente una relación de todo y parte del tipo agregación con la clase Estudiante. Al no ser GestorSolicitudesEstudiantes la clase encargada de crear a los estudiantes, su relación por tanto es de agregación porque contiene listas de estudiantes.

**4. GestorEmpleados**

* Atributos

1. private GestorSolicitudesEmpleados gestorSolicitudesEmpleados: clase encargada de manejar las solicitudes de baja y licencia de los empleados.
2. private ArrayList<Empleado> empleados: un registro de los empleados de la universidad.

* Métodos

1. public ArrayList<Empleado> getEmpleados() método que devuelve la lista de los empleados registrados
2. filtrar(Map<String,String>): ArrayList<Empleado> este método es el de filtrado general que utiliza el método 3, devuelve la lista filtrada de empleados por nombre u otro valor de entrada.
3. cargarInformacioneEmpleados (): void método que registra a todos los empleados de la universidad, cargando sus datos.
4. filtrar(String):ArrayList<Empleado> método de filtrado de empleados dado un valor como puede ser el nombre, mostrando solo aquellos que coincidan con el valor de entrada.

* Relaciones

1. Relación de todo-parte de tipo composición con Empleados, donde Empleado es la parte y la cardinalidad es de 1 GestorEmpleados a 1 o muchos empleados. Es de composición debido a que esta clase es la encargada de crear las instancias de empleados.
2. Relación de composición con GestorSolicitudesEmpleados, porque GestorEmpleados se encarga de crear una instancia de GestorSolicitudesEmpleados y la contiene a su vez como atributo, y la cardinalidad como se ve es de 1 a 1.

**5.GestorSolicitudesEmpleados**

* Atributos:

private HashMap<Empleado, Set<Solicitud>> solicitudesEmpleados listado de solicitudes y los empleados que la solicitan.

* Métodos:

1. private void cargarSolicitudesEmpleados() método que se encarga de cargar los datos de todas las solicitudes de los empleados de baja o licencia.
2. Public HashMap<Empleado, Set<Solicitud>> getSolicitudesPendientes() método que devuelve la lista de solicitudes con sus empleados que aún no han sido aprobadas ni canceladas.
3. public boolean verificarEmpleadoSolicitaAlgo(Empleado e) método que comprueba si un empleado ha solicitado baja o licencia, en ese caso retorna true, de lo contrario false.
4. public void crearSolicitudBaja(MotivoBaja motivo, Empleado e) método que crea una nueva solicitud de baja de un empleado y la agrega al listado ya existente.
5. public void crearSolicitudLicencia(MotivoLicencia motivo, Empleado e, Calendar salida, Calendar regreso) método que crea una nueva solicitud de licencia de un empleado y la agrega al listado ya existente.
6. public ArrayList<Empleado> obtenerEmpleadosPendientes() método que retorna la lista de empleados que sus solicitudes aún están en estado pendiente(ni aceptada ni cancelada).

* Relaciones

1. GestorSolicitudesEmpleados tiene listas de solicitudes, lo que se refleja en una relación de composición con la clase Solicitud puesto que es la encargada de crear las instancias de las solicitudes de los empleados, donde Solicitud es la parte, la cardinalidad es de 1 a 1 o varias.
2. A su vez está presente una relación de todo y parte del tipo agregación con la clase Estudiante. Al no ser GestorSolicitudesEstudiantes la clase encargada de crear a los estudiantes, su relación por tanto es de agregación porque contiene listas de estudiantes.

**6.GestorDepartamentos**

* Atributos: una instancia de cada entidad (Biblioteca, Economía, AlmacenDeLibros, DireccionDeBecas, SeguridadInformatica, Contabilidad, RecursosHumanos)
* Métodos:

* 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).

**7-13 Entidades** (Biblioteca, Economía, AlmacénDeLibros, DirecciónDeBecas, SeguridadInformática, Contabilidad, RecursosHumanos)

* A verificar en cada entidad:

Biblioteca: se verifica tanto para estudiantes como para empleados si debe libros.

Almacén de libros: se verifica si el estudiante debe libros docentes.

Economía: se verifica si el estudiante tiene estipendio.

Dirección de becas: se verifica si el estudiante tiene pertenencias de la beca y carnet de becado.

Contabilidad: se verifica si el empleado tiene deuda.

Seguridad Informática: se verifica si el estudiante o el empleado cerraron su cuenta.

Recursos Humanos: verifica si el empleado tiene salario indebido.

* Los atributos son private HashMap<String, TiposResponsabilidad> y el nombre varía dependiendo de la entidad. Este listado es de las personas que han pedido libros en el caso de Biblioteca, que tienen estipendio en el caso de Economía y así sucesivamente.La clase TiposResponsabilidad es auxiliar que contiene las diferentes responsabilidades que tiene un estudiante o empleado.
* Métodos:
  1. todas poseen métodos de **verificar requisitos,** puesto que implementan las interfaces VerificadorEstudiante o VerificadorEmpleado, incluso ambas cada una lo implementa de manera diferente dependiendo de lo que se debe entregar en cada entidad. Por ejemplo en biblioteca solo se comprueba que no deba libros, mientras que en SeguridadInformatica se comprueba que haya cerrado su cuenta. Estos métodos reciben como parámetros, un estudiante o un empleado y devuelven true si tienen todo cumplido y en caso contrario false.
  2. Todas poseen métodos de **cargarInformacionPersonas** con deuda ya sea libros, pertenecías de la beca, dinero, estipendio, cuenta abierta u otro. Estos métodos son utilizados para obtener la lista de personas pendientes por aceptar, denegar o dejar pendiente, si entrega su deuda, no la entrega, entrega deuda incompleta (en este caso se le actualiza la deuda).
  3. Además tienen métodos que devuelven la lista de estudiantes o empleados que tienen alguna deuda con ellos aún, o sea aquellos que aún están en pendiente. (obtenerEstudiantesPendientes u obtenerEmpleadosPendientes )
* Relaciones:

1. Las entidades implementan dos interfaces (relaciones de asociación):

**VerificadorEstudiante (14)** la implementan las entidades: Biblioteca, Secretaría, Economía, Dirección de Becas, Almacén de Libros y Seguridad Informática. Esta interface contiene un método 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 (15)** la implementan las entidadesBiblioteca, Contabilidad, Recursos Humanos, Seguridad Informática. Esta interface es similar a la anterior pero con empleados.

Las clases **Estudiante**(16) y **Empleado**(17) heredan de la superclase abstracta **Persona**(18) 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 es una clase abstracta porque no es necesario instanciarla, la persona en este caso siempre va a ser o un estudiante o un empleado; y es además 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:

Un estudiante es una persona.

Un empleado es una persona.

La clase **Empleado** (superclase abstracta) tiene una relación de herencia con **EmpleadoDocente (19)** y **EmpleadoNodocente (20),** estas serían las clases hijas al poseer sus mismas características y métodos pero con algunas particularidades.

Un empleado docente es un empleado.

Un empleado no docente es de igual manera un empleado.

La clase **EstudianteBecado (21)** hereda de la clase **Estudiante**, porque un estudiante becado es un estudiante, con algunas particularidades.

**Atributos:**

Persona

protected String ci;

protected String nombre;

protected String primer\_apellido;

protected String segundo\_apellido;

Estudiante (atributos de su clase padre)

protected int curso;

protected TipoCurso tipoCurso;

protected String carrera;

protected int grupo;

protected Facultad facultad;

Estudiante Becado (atributos de su clase padre)

private String idBecado;

private int noCuarto;

Empleado (atributos de su clase padre)

protected String numerotrabajador;

Empleado Docente (atributos de su clase padre)

private String departamento;

private String categoriaDocente;

private Facultad facultad;

Empleado No Docente (atributos de su clase padre)

private String seccion;

**22. Clase abstracta Solicitud**

* Atributos

1. protected Estado estado (es un enum porque las solicitudes pueden estar solo en 3 estados: aprobada, rechazadas o pendiente.
2. protected Date fecha este atributo sería la fecha en que se realiza la solicitud.

* Métodos

Los métodos que poseen son los de encapsulamiento.

* Relaciones

1. Además de la relación de composición explicada anteriormente. Tiene una relación de herencia con SolicitudLicencia y SolicitudBaja donde Solicitud es la superclase.

**23. SolicitudLicencia** clase concreta que hereda de **Solicitud** al poseer métodos y atributos semejantes, con sus particularidades, en este caso sería la fecha.

* Atributos

Además de los de su clase padre, posee los atributos de tipo Date fechaSalida y fechaRegreso. Y el atributo motivo de tipo MotivoLicencia (este tipo de dato es un enum puede ser vacaciones, enfermedad, accidente)

**24. SolicitudBaja** clase concreta que hereda de **Solicitud** al poseer métodos y atributos semejantes, con sus particularidades. Posee los mismos métodos y atributos que su clase padre y además el atributo motivo de tipo MotivoBaja (este tipo de dato es un enum puede ser enfermedad, accidente) y sus métodos de encapsulamiento.

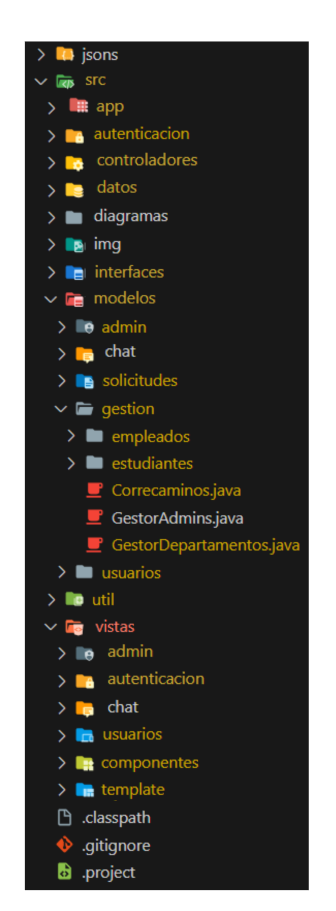
### 2. Explicación por capas de la solución

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.
* **autenticación:** 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.
* **controladores:** 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.
* **datos:** 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.
* **modelos:** Contiene los modelos de datos, organizados en diferentes paquetes según su funcionalidad, como admin, chat, solicitudes, gestión(dividido en empleados, estudiantes y departamentos), 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.
* **útil:** 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.
* **vistas**: 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.



### 3.Subequipo Yanisbe Hurtado y Roy Hernández

Responsabilidades a implementar:

1. Clases para los usuarios, explicadas ya anteriormente (Persona, Estudiante, EstudianteBecado).

2. Entidades: Biblioteca, Economía y DirecciónDeBecas.

Métodos:

//Biblioteca

* public boolean verificarRequisitos(Estudiante e) verifica si un estudiante está en la lista de personas que deben libros.
* public boolean verificarRequisitos(Empleado e) verifica si un empleado está en la lista de personas que deben libros.
* public void recogerLibrosPrestados(Persona p) actualiza la lista de personas que deben libros, si esa persona entrega todos los libros que debe se elimina de la lista, sino se le elimina ese libro de su deuda.
* public ArrayList<Estudiante> getEstudiantesPendientes(ArrayList<Estudiante> estudiantesSolicitudesPendientes) obtiene los estudiantes que deben libros aún.
* public ArrayList<Empleado> getEmpleadosPendientes(ArrayList<Empleado> empleadosSolicitudesPendientes) obtiene los empleados que deben libros aún.

//Economía

* public boolean verificarRequisitos(Estudiante e) verifica si un estudiante tiene estipendio.
* public void cancelarPagoEstipendio(Estudiante e) si un estudiante cancela su estipendio, se elimina de la lista de pendientes.
* public ArrayList<Estudiante> getEstudiantesPendientes(ArrayList<Estudiante> estudiantesSolicitudesPendientes) retorna los estudiantes que aun tienen estipendio.

// DirecciónDeBecas

* public boolean verificarRequisitos(Estudiante e) verifica si un estudiante ya entreg;o su carnet de becado y las pertenencias de la beca.
* public void recogerPertenenciasDeEstudiante(Estudiante e, Set<String> pertenencias) actualiza la lista de pertenencias pendientes si un estudiante entrega algo, en caso de que lo entregue todo se elimina de la lista de pendientes.
* public ArrayList<Estudiante> getEstudiantesPendientes(ArrayList<Estudiante> estudiantesSolicitudesPendientes) retorna la lista de estuidnates con pertenencias aún.
* public Set<String> obtenerPertenenciasBecado(Estudiante e) devuelve le listado de las pertenencias que posee un estduiante, esto es útil para la verificación.

//Correcaminos

* public Persona buscarPersonaporCi(String id) busca una persona por su carnet de identidad.

//Auth:

* usuarioAutenticado(): Devuelve el objeto del usuario que ha iniciado sesión.
* iniciarSesion(Autenticable user, String nombreUser): Inicia la sesión para el usuario proporcionado y establece el nombreUsuario.
* hayUsuarioAutenticado(): Verifica si hay un usuario autenticado actualmente o no.
* logout(): Cierra la sesión del usuario actual y borra el nombreUsuario.
* getNombreUsuario(): Devuelve el nombre del usuario que ha iniciado sesión.

Esta clase utiliza el patrón Singleton para la autenticación, lo que significa que solo puede haber un usuario autenticado a la vez en la aplicación. Esto se logra al hacer que las variables sean static, lo que significa que son compartidas por todas las instancias de la clase.

**4.Subequipo Rubén Hernández y Eduardo Enrique**

Responsabilidades a implementar:

1.Clases para las solicitudes, explicadas anteriormente (Solicitud, SolicitudBaja, SolicitudLicencia)

2.Entidades: AlmacenDeLibros, SeguridadInformatica.

//Clase VerificacionCredenciales :

Es una clase que maneja la autenticación de un usuario.

Método:

* autenticar(String nombreUsuario, String contrasena): Este método toma un nombre de usuario y una contraseña como parámetros, crea un objeto Credenciales con ellos, y luego verifica si el usuario existe y si la contraseña es correcta. Para hacer esto, primero verifica si el usuario existe en el sistema de SeguridadInformatica, y si no, verifica si el usuario es un administrador en el sistema de GestorAdmins. Si el usuario existe en cualquiera de los dos sistemas y la contraseña es correcta, devuelve el objeto del usuario. Si el usuario no existe o la contraseña es incorrecta, devuelve null.

Esta clase utiliza la autenticación basada en roles, lo que significa que verifica el rol del usuario (ya sea un usuario normal o un administrador) durante la autenticación. Esto se logra a través de la verificación de la existencia del usuario en los sistemas correspondientes.

//Clase SeguridadInformatica:

Es una clase que implementa las interfaces VerificadorEstudiante y VerificadorEmpleado, lo que significa que puede verificar los requisitos para estudiantes y empleados.

Métodos:

* cargarInformacionUsuariosCujae(): Este método carga la información de los usuarios a través de la clase ObtenerUsuariosCujae usando el método estatico cargarDesdeArchivo.
* tieneCuentaUsuarioAbierta(Persona c): Este método verifica si una persona tiene una cuenta abierta en el sistema.
* getUsuariosCujae(): Este método devuelve el HashMap usuariosCujae.
* verificarRequisitos(Estudiante e): Este método verifica si un estudiante tiene una cuenta abierta en el sistema.
* verificarRequisitos(Empleado e): Este método verifica si un empleado tiene una cuenta abierta en el sistema.
* cerrarCuenta(Credenciales c): Este método cierra la cuenta de un usuario en el sistema.
* getEstudiantesPendientes(ArrayList<Estudiante> estudiantesSolicitudesPendientes): Este método devuelve una lista de estudiantes que tienen solicitudes pendientes.
* getEmpleadosPendientes(ArrayList<Empleado> empleadosSolicitudesPendientes): Este método devuelve una lista de empleados que tienen solicitudes pendientes.

Esta clase utiliza la autenticación basada en roles y la persistencia de datos. Esto se logra a través de la verificación de la existencia del usuario en el sistema y la lectura y escritura de archivos.

//Clase ControladorLogin:

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.

//Clase Almacén de libros

* verificarRequisitos(Estudiante e) verifica si un estudiante debe libros docentes.
* recogerLibrosDocentes(Estudiante e, Set<String> librosDocentes) actualiza el listado de pendientes o las deudas de un estudiante, según vaya entregando.
* getEstudiantesPendientes(ArrayList<Estudiante>estudiantesSolicitudesPendientes) devuelve los estudiantes con solicitudes pendientes.
* obtenerEstudianteLibrosDocentesPendientes(Estudiante e) devuelve los estudiantes con deuda.

### 5.Subequipo Marcel Pulido y Camilo Alfonso

Responsabilidades a implementar:

1. Clases para los usuarios, explicadas ya anteriormente (Empleado, EmpleadoDocente, EmpleadoNoDocente)
2. Entidades de Recursos humanos y contabilidad.

//Clase Recursos Humanos

* public boolean verificarRequisitos(Empleado e) comprueba si el empleado tiene salario indebido, o sea si está en el listado de empleados con salario indebido.
* public ArrayList<Empleado> getEmpleadosPendientes(ArrayList<Empleado> empleadosSolicitudesPendientes) obtiene el listado de los empleados pendientes.

//Clase Contabilidad

* public void saldarDeuda(Empleado e) elimina a un empleado del listado de pendientes en caso de que pague toda la deuda, y sino reduce esta.
* public ArrayList<Empleado> getEmpleadosPendientes(ArrayList<Empleado> empleadosSolicitudesPendientes) obtiene la lista de los empleados con deudas.
* public boolean verificarRequisitos(Empleado e) verifica si un empleado ha pedido alguna deuda y aun no la ha pagado, revisando si el empleado se encuentra en la lista de personas con deuda por saldar.

//Clase Session:

Es una clase que maneja la sesión de un usuario a través de la persistencia de datos en un archivo JSON.

Constructor:

* Session(Credenciales c): Este constructor toma un objeto Credenciales como parámetro, lo convierte en formato JSON usando la biblioteca Gson, y luego escribe este JSON en un archivo llamado session.json. Si ocurre un error durante este proceso, se imprime la traza de la pila del error.

Métodos:

* obtenerSession(): Este método lee el archivo session.json, si existe, y convierte su contenido de formato JSON a un objeto Credenciales usando la biblioteca json. Si el archivo no existe, crea una nueva sesión con un objeto Credenciales nulo. Finalmente, devuelve el objeto Credenciales.

Esta clase utiliza la persistencia de datos para manejar la sesión del usuario, lo que significa que los datos de la sesión del usuario se guardan en un archivo y se pueden recuperar incluso después de que la aplicación se haya cerrado y vuelto a abrir. Esto se logra a través de la lectura y escritura de archivos y la conversión de objetos a formato JSON y viceversa.

//GestorSolicitudesEstudiantes

* public void crearSolicitudBaja(MotivoBaja motivo, Estudiante e) creación de una nueva solicitud de baja de un estudiante, a partir de la entrada de datos por parte de este (motivo por el que solicita baja) y la obtención de sus datos.
* public void crearSolicitudLiciencia(MotivoLicencia motivo, Estudiante e) semejante al anterior pero con licencia.
* public ArrayList<Estudiante> obtenerEstudiantesPendientes() obtiene los estudiantes cuyas solicitudes se encuentran aún en estado de pendientes.

//GestorSolicitudesEmpleados

* public void crearSolicitudBaja(MotivoBaja motivo, Empleado e) creación de una nueva solicitud de baja de un empleado, a partir de la entrada de datos por parte de este (motivo por el que solicita baja) y la obtención de sus datos.
* public void crearSolicitudLiciencia(MotivoLicencia motivo, Empleado e) semejante al anterior pero con licencia.
* public ArrayList< Empleado > obtener EmpleadosPendientes() obtiene los empleados cuyas solicitudes se encuentran aún en estado de pendientes.

### 6.Reportes

En cuanto a los reportes el equipo trabajó en una pantalla dinámica que de acuerdo a los criterios que el usuario seleccione filtra toda la información, mostrando específicamente la información buscada. Por lo que se considera esto como reportes, debido a que solo secretaria y gestorEmpleados tendrían estos reportes ya que son los que manejan los datos de las solicitudes de estudiantes y empleados y podrían necesitar estadísticas, búsquedas y otros datos. Se puede filtrar por nombre, año, estado de la solicitud, tipo de solicitud ya sea baja o licencia, todo conjuntamente, de modo que puedes buscar las solicitudes de baja aprobadas en un año dado, incluso las que tengan por nombre del solicitante uno dado. Permite obtener todas las licencias rechazadas que puede tener una persona, las licencias o bajas solicitadas por el mismo motivo y así sucesivamente se obtienen diferentes reportes.

* filtradoDinamicoSolicitudEstudiantes(Map<String, String> filtros): Este método toma un mapa de filtros como parámetro, donde cada filtro es un par clave-valor que representa un atributo y su valor correspondiente. El método recorre todas las solicitudes de estudiantes y verifica si cada solicitud cumple con todos los filtros. Si una solicitud cumple con todos los filtros, se agrega a un mapa de solicitudes filtradas que se devuelve al final.
* buscarPorNombreOCi(Estudiante e, String valor): Este método toma un estudiante y una cadena de texto como parámetros, y verifica si el nombre completo o el CI del estudiante contienen la cadena de texto. Devuelve true si es así, y false en caso contrario.

Acceso:

Para entrar como estudiante no becado:

usuario: estudiante1 hasta estudiante11

Para estudiante becado:

usuario: estudiante12 hasta estudiante22

contraseña: prueba (la misma para todos los usuarios de estudiante)

Para acceder como empleado docente:

usuario: empleado1 hasta empleado11

Para empleado no docente:

usuario: empleado12 hasta empleado22

contraseña: prueba (la misma para todos los usuarios de empleado)

Para entrar como administrador:

|  |  |  |
| --- | --- | --- |
| Departamento | Usuario | Contraseña |
| Biblioteca | Biblioteca | biblioteca |
| Economia | Economia | economia |
| SeguridadInformatica | SeguridadInformatica | seguridadinformatica |
| DireccionBecas | DireccionBecas | direccionbecas |
| Contabilidad | Contabilidad | contabilidad |
| Secretaria | Secretaria | secretaria |
| RecursosHumanos | RecursosHumanos | recursoshumanos |