

UNIVERSIDAD AMERICANA

Facultad de ingeniería y arquitectura



POO

Problemática SYSTECH

Integrantes:

- Franco Xavier Aguilera Ortiz
- David Joel Sanchez Acevedo
- Andrea Johanna Duarte Guerrero
- Solieth Valentina Trejos Perez
- Kimberly Maria Zapata Espinoza

Docente: Jose Duran Garcia

Managua, Nicaragua (10/09/25)

Índice

Índice.....	2
Problemática.....	3
Antecedentes.....	4
1. Resumen de cada clase.....	5
Clase Administrador.....	5
Clase Estudiante.....	6
Clase Evento.....	7
Clase Diploma.....	9
Clase AuthService.....	10
Clase ConsoleIO.....	12
Clase Main.....	13
2. Estructuras de datos utilizadas.....	16
3. Relaciones entre clases.....	18
4. Estados posibles de las clases.....	22
5. Ejemplos concretos de uso.....	25
6. Contexto general del sistema.....	29

Problemática

El evento SYSTECH de la Universidad Americana (UAM) se ha llevado a cabo como un espacio académico y tecnológico de gran relevancia para la Facultad de Ingeniería. Cada año, estudiantes, empresas tecnológicas, conferencistas nacionales e internacionales, y expertos en áreas como programación e inteligencia artificial se reúnen en un ambiente de innovación.

Uno de los elementos clave de este evento es la entrega de diplomas de participación, los cuales representan un reconocimiento formal a los estudiantes asistentes. Sin embargo, el proceso tradicional de entrega manual ha demostrado ser ineficiente. En la edición de este año (2025), los diplomas no se entregaron al finalizar el evento, como era costumbre, sino que se pospuso su entrega para la semana siguiente en la Facultad de Ingeniería de la Universidad Americana (UAM). Esto obliga a los estudiantes a trasladarse personalmente hasta las instalaciones, esperar y depender de la búsqueda manual de su diploma físico entre más de 400 documentos impresos.

En este contexto, se propone un sistema digital para la gestión de diplomas, cuyo objetivo es automatizar la generación y entrega de certificados digitales al correo institucional de los estudiantes, eliminando el proceso manual y brindando una experiencia moderna y eficiente.

Antecedentes

En ediciones pasadas de SYSTECH, los diplomas eran entregados en formato físico al finalizar el evento, en el mismo hotel donde se realizaba la actividad. Durante la edición 2024, este método se volvió problemático porque la entrega se realizaba de manera desordenada, con estudiantes aglomerados y empujándose para obtener su diploma, lo cual generaba incomodidad, caos logístico y falta de control en el proceso. Con base en estos antecedentes, se identificó la necesidad de un proceso digitalizado, lo cual se alinea a la visión de la Facultad de Ingeniería de promover la transformación digital y la eficiencia administrativa.

1. Resumen de cada clase

Clase Administrador

Propósito: Representa al usuario administrador del sistema, encargado de gestionar eventos y diplomas. Un administrador puede crear nuevos eventos, marcar la asistencia de estudiantes, emitir diplomas para estudiantes que cumplan requisitos y firmar esos diplomas.

Atributos (privados): - `idAdministrador (String)`: Identificador único del administrador.
- `nombre (String)`: Nombre del administrador. - `usuario (String)`: Nombre de usuario para iniciar sesión. - `password (String)`: Contraseña de acceso.

Métodos: - `Evento crearEvento(String id, String nombre, Date fecha, double creditosOtorgados)`: Crea un nuevo evento con el ID, nombre, fecha y créditos otorgados especificados, y retorna el objeto `Evento` creado. (El administrador **no** guarda el evento, solo lo crea; la relación es de asociación, manejada externamente). - `void marcarAsistencia(Evento evento, Estudiante estudiante, boolean asistio)`: Marca la asistencia de un estudiante en un evento dado. Internamente delega la operación al evento llamando `evento.registrarAsistencia(estudiante, asistio)` para registrar en el evento si el estudiante asistió (`true`) o no (`false`). - `Diploma emitirDiploma(Estudiante estudiante, Evento evento, int nuevoIdDiploma)`: Emite un diploma digital para un estudiante en un evento. Primero valida la asistencia del estudiante al evento usando `evento.validarAsistencia(estudiante)`. Si el estudiante **no asistió**, lanza una excepción que impide la emisión. Si sí asistió, crea un nuevo

objeto Diploma con un ID único (`nuevoIdDiploma`), la fecha actual, y referencias al evento y al estudiante. Además, suma los créditos del evento a los `creditosAcumulados` del estudiante. Devuelve el diploma creado. - `void firmarDiploma(Diploma diploma)`: Marca un diploma como firmado por la autoridad (decano). Cambia el estado interno del diploma llamando `diploma.setFirmaDecano(true)`, indicando que el diploma ha sido firmado. (En esta versión simplificada, solo se usa un booleano para representar la firma del decano). - (*Getters básicos*): `getIdAdministrador()`, `getNombre()`, `getUsuario()`, `getPassword()` retornan los atributos correspondientes del administrador.

Clase Estudiante

Propósito: Representa a un estudiante participante del evento. Los estudiantes pueden registrarse en el sistema, marcar su asistencia a eventos y consultar sus diplomas digitales.

Atributos (privados): - `cif (String)`: Código de identificación del estudiante (CIF), utilizado como clave única. - `nombre (String)`: Nombre completo del estudiante. - `password (String)`: Contraseña para autenticación del estudiante. - `creditosAcumulados (double)`: Total de créditos académicos acumulados por el estudiante, que aumentan al recibir diplomas por asistir a eventos (inicia en 0.0).

Métodos: - `void consultarDiploma(Diploma diploma)`: Permite al estudiante “ver” un diploma. Si el diploma pasado es `null` (no existe diploma disponible), muestra un mensaje indicando que no hay diploma. Si existe, imprime en consola la información

del diploma (usando `System.out.println(diploma)`, que invoca el `toString()` del diploma para mostrar detalles legibles). - `void registrarEvento(Evento evento, boolean asistio)`: Registra la participación del estudiante en un evento, indicando manualmente si asistió o no. Llama internamente a `evento.registrarAsistencia(this, asistio)` para actualizar la lista de asistencia del evento con el CIF del estudiante y el valor booleano proporcionado. - `void marcarAsistencia(Evento evento)`: Marca la asistencia del estudiante a un evento de manera interactiva. Si el evento es `null`, lanza error. Si el estudiante aún **no tiene registro** de asistencia en ese evento (verificado con `evento.consultarAsistencia(this)`), entonces registra su asistencia con valor `true` llamando a `evento.registrarAsistencia(this, true)`. Este método **evita duplicar registros**: solo marca la asistencia la primera vez. - (*Getters y Setters*): `getCif()`, `getNombre()`, `getPassword()`, `getCreditosAcumulados()` retornan los valores de los atributos del estudiante. `setCreditosAcumulados(double cred)` permite actualizar los créditos acumulados (usado al emitir un diploma para sumar créditos).

Clase Evento

Propósito: Modela un evento académico (como SYSTECH) donde los estudiantes pueden asistir y obtener créditos y diplomas. La clase Evento lleva control de la información del evento y la lista de estudiantes asistentes.

Atributos (privados): - `idEvento (String)`: Identificador único del evento, por ejemplo "SYSTECH2025". - `nombreEvento (String)`: Nombre oficial o título del evento. -

fechaEvento (Date): Fecha en que se realiza el evento. - creditosOtorgados (double): Cantidad de créditos académicos que el evento otorga a cada estudiante que asiste (estos créditos se sumarán al emitir el diploma). - asistencia (Map<String, Boolean>): Registro de asistencia de estudiantes, almacenado como un mapa que asocia el CIF de cada estudiante con un valor booleano. Un valor true indica que el estudiante asistió al evento; false indica que **no asistió** (o que se registró su ausencia).

Métodos: - boolean validarAsistencia(Estudiante est): Verifica si un estudiante en particular asistió al evento. Retorna true si en el mapa de asistencia el CIF del estudiante aparece con valor true. Si el estudiante no está en el registro o tiene false, retorna false. Este método se usa antes de emitir un diploma para asegurarse de que solo se emitan diplomas a asistentes. - void registrarAsistencia(Estudiante est, boolean asistio): Registra (o actualiza) la asistencia de un estudiante en el evento. Agrega una entrada en el mapa asistencia donde la llave es el CIF del estudiante (est.getCif()) y el valor es el booleano asistio. Si ya existía una entrada para ese estudiante, la sobrescribe. (*Nota:* En esta versión, se registra explícitamente tanto asistencias true como inasistencias false; típicamente solo se registrarían los asistentes con true). - Set<String> obtenerCifsAsistentes(): Devuelve un conjunto con todos los CIF de estudiantes que fueron marcados como asistentes (true) en este evento. Está implementado filtrando el mapa de asistencia (toma solo las entradas con valor true). Esto permite, por ejemplo, obtener la lista de todos los estudiantes que **sí asistieron**. - boolean consultarAsistencia(Estudiante estudiante): Consulta rápidamente si un estudiante está marcado como asistente en el evento. Retorna true

si el CIF del estudiante está en el mapa con valor true; de lo contrario false (es similar a validarAsistencia, funciona como alias semántico). - *(Getters)*: getIdEvento(), getNombreEvento(), getFechaEvento(), getCreditosOtorgados(), que devuelven los datos básicos del evento. *(No se implementan setters, ya que se asume que estos datos no cambian una vez creado el evento.)*

Clase Diploma

Propósito: Representa el diploma digital que se emite a un estudiante por haber asistido a un evento. Contiene los datos necesarios del certificado, incluyendo referencias al estudiante y al evento correspondientes.

Atributos (privados): - idDiploma (int): Identificador numérico único de cada diploma emitido (por simplicidad se puede usar un contador incremental para asignarlo). - fechaEmision (Date): Fecha en que el diploma fue generado/emisión. - firmaDecano (boolean): Indica si el diploma ha sido firmado digitalmente por el decano. Por defecto es false al crear el diploma; luego puede cambiar a true cuando se firma. - evento (Evento): Referencia al objeto Evento asociado, es decir, el evento en el que participó el estudiante para obtener este diploma. - estudiante (Estudiante): Referencia al objeto Estudiante que recibió el diploma.

Métodos: - *(Getters y Setters principales)*: getIdDiploma(), getFechaEmision(), isFirmaDecano(), setFirmaDecano(boolean), getEvento(), getEstudiante(). Permiten acceder a los datos del diploma y cambiar el estado de la firma del decano. - String toString(): Se ha sobrescrito el método toString() para mostrar de forma

legible la información del diploma. Incluye el ID, fecha de emisión, estado de firma (true/false) y el nombre del evento y del estudiante asociados. Esto facilita imprimir un diploma en consola con `System.out.println(diploma)`, mostrando sus detalles de manera clara.

Clase AuthService

Propósito: Provee servicios de autenticación y registro de usuarios (tanto administradores como estudiantes). Se encarga de la lógica de login (validar usuario/contraseña) y de crear nuevos usuarios con datos ingresados por consola, asegurando ciertos criterios (por ejemplo, nombres de usuario únicos y contraseñas confirmadas correctamente).

Atributos (privados): - `maxIntentos (int)`: Número máximo de intentos permitidos al iniciar sesión antes de bloquear el proceso (por defecto 3 intentos). Se establece en el constructor. - `seqAdmin, seqStudent (AtomicInteger)`: Contadores secuenciales usados para generar identificadores únicos automáticamente si se requiriera. *(En este código, realmente no se usan para el ID ya que el id del admin coincide con su usuario y el CIF del estudiante se ingresa manualmente; podrían emplearse en futuras ampliaciones.)*

Métodos de Login: - `Administrador loginAdmin(Map<String, Administrador> adminsByUser)`: Pide por consola las credenciales de un administrador (usuario y contraseña) y verifica si coinciden con algún objeto `Administrador` en el mapa proporcionado. Permite hasta `maxIntentos` intentos. Si las credenciales son correctas, retorna el objeto `Administrador` autenticado, mostrando un mensaje de éxito en

consola (✓ Autenticación de Administrador exitosa.). Si tras los intentos no se logra una coincidencia, lanza una excepción indicando que se excedió el número de intentos. - Estudiante loginEstudiante(Map<String, Estudiante> studentsByUser): Similar al anterior, pero para estudiantes. Solicita el CIF (como identificador de usuario del estudiante) y la contraseña, luego verifica contra el mapa de estudiantes proporcionado. Retorna el objeto Estudiante si la autenticación es exitosa (mensaje en consola: ✓ Autenticación de Estudiante exitosa.); si no, lanza excepción después de los intentos fallidos.

Métodos de Registro: - Administrador registrarAdmin(Map<String, Administrador> adminsByUser): Ejecuta el proceso de registro de un nuevo administrador. Pide ingresar nombre completo, luego un nombre de usuario (validando que no exista ya en el mapa adminsByUser y que cumpla un formato mínimo). Solicita una contraseña y su confirmación (asegurando que coincidan). Crea un objeto Administrador con un ID (en este caso utiliza el mismo valor del usuario como id) y los datos ingresados, lo almacena en el mapa y devuelve el nuevo Administrador. Muestra mensajes en consola indicando éxito (✓ Administrador registrado.) o problemas (por ejemplo, si el usuario ya existe). - Estudiante registrarEstudiante(Map<String, Estudiante> estudiantesPorCif): Registra un nuevo estudiante. Pide un CIF único (con validación similar de formato y duplicados en el mapa), nombre completo y contraseña (con confirmación). Crea un objeto Estudiante con esos datos y lo agrega al mapa estudiantesPorCif. Retorna el Estudiante creado y notifica en consola que el registro fue exitoso (✓ Estudiante registrado.).

(Métodos auxiliares privados): leerPasswordConConfirmacion(): Utilizado internamente para solicitar una contraseña dos veces y verificar que ambas entradas coincidan, cumpliendo requisitos mínimos (por ejemplo, al menos 4 caracteres). Reitera hasta obtener una confirmación válida, devolviendo la contraseña final ingresada correctamente.




Clase ConsoleIO

Propósito: Es una clase de utilidades para manejo de la entrada/salida en consola. Provee métodos estáticos para leer datos del usuario con validación, evitando tener código repetitivo de lectura y manejo de errores en otras clases.

Atributos: - SC (Scanner estático): Objeto Scanner para leer del System.in (entrada estándar de la consola). - ISO_DATE (DateTimeFormatter estático): Formato de fecha "yyyy-MM-dd" para interpretar correctamente las fechas ingresadas en ese formato.

(La clase es final y tiene un constructor privado para prevenir que se instancie; se usa directamente llamando a sus métodos estáticos.)

Métodos estáticos principales: - String readNonEmpty(String prompt): Muestra un mensaje o etiqueta (prompt) y lee una línea de texto ingresada. Si el usuario no ingresa nada (cadena vacía), advierte con ⚠ Entrada vacía. Intente nuevamente. y repite hasta que se ingresa algo. Retorna la cadena no vacía. - String readMatching(String prompt, String regex, String errorMsg): Solicita al usuario ingresar un texto que cumpla con un patrón especificado por una expresión regular (regex). Si el texto ingresado **no** coincide, muestra ⚠ seguido del mensaje de error proporcionado y

vuelve a pedir la entrada. Retorna la cadena que sí coincide. *(Se usa, por ejemplo, para validar el formato de usuario o CIF al registrarse.)* - `int readIntInRange(String prompt, int min, int max)`: Pide ingresar un número entero dentro de un rango [min, max]. Si el valor ingresado no es un entero válido o está fuera de rango, muestra un mensaje de error y repite la solicitud ( Ingrese un entero entre X y Y.). Devuelve el entero válido ingresado. - `double readDouble(String prompt)`: Solicita ingresar un número (puede ser decimal). Admite coma o punto decimal (internamente convierte coma a punto). Si el formato es inválido (no se puede parsear como número), muestra  Número inválido. Intente de nuevo. y repite. Retorna el valor en tipo double una vez válido. - `boolean readYesNo(String prompt)`: Formula una pregunta al usuario cuya respuesta debe ser sí o no (agrega automáticamente "(s/n)" al prompt). Lee la respuesta y acepta variantes de "sí" (s, si) o "no" (n, no). Devuelve true para sí y false para no. Repite hasta obtener una respuesta válida. - `LocalDate readIsoDate(String prompt)`: Pide una fecha con formato "yyyy-MM-dd". Si el usuario ingresa un formato incorrecto, muestra  Fecha inválida. Ejemplo: 2025-09-11 y solicita de nuevo. Devuelve un `LocalDate` válido cuando la entrada es correcta. - `void pressEnterToContinue()`: Realiza una pausa simple hasta que el usuario presione ENTER. Se utiliza para dar tiempo a leer mensajes antes de continuar con el flujo (por ejemplo, después de mostrar una lista de resultados o al finalizar una acción).

Clase Main

Propósito: Contiene el método `main` y coordina la ejecución de la aplicación en consola. Es el punto de entrada del programa, manejando el menú principal, las llamadas a `AuthService` y las acciones de Administrador/Estudiante. También mantiene las estructuras de datos globales en memoria que simulan la persistencia.



Atributos (estructuras de datos globales, estáticos):

- `ADMINS_BY_USER` (`Map<String, Administrador>`): Mapa que actúa como “base de datos” en memoria de administradores registrados, donde la clave es el nombre de usuario (`String`) y el valor es el objeto `Administrador`.
- `EST_BY_CIF` (`Map<String, Estudiante>`): Mapa de estudiantes registrados, usando el CIF (`String`) como clave y el objeto `Estudiante` como valor.
- `EVENTOS` (`List<Evento>`): Lista de todos los eventos creados en el sistema.
- `DIPLOMAS` (`List<Diploma>`): Lista de todos los diplomas emitidos hasta el momento.
- `AUTH` (`AuthService`): Instancia única de la clase de autenticación para manejar login/registro de usuarios.

Métodos principales:

- `public static void main(String[] args)`: Inicia la aplicación. Opcionalmente llama a `ejemplo()` para precargar un administrador y un estudiante de prueba, luego invoca el menú principal (`menu()`).
- `private static void ejemplo()`: Método auxiliar que crea un `Administrador` y un `Estudiante` de ejemplo, agregándolos a los mapas `ADMINS_BY_USER` y `EST_BY_CIF` respectivamente. *(Por ejemplo, crea un admin con usuario "admin001"/contraseña "1234" y un estudiante con CIF "CIF001"/contraseña "abcd").*
- `private static void menu()`: Muestra el menú

principal en consola con opciones numeradas para: registrar administrador, registrar estudiante, login administrador, login estudiante, o salir. Lee la opción usando `ConsoleIO.readIntInRange` y, según la elección, invoca los métodos correspondientes de `AuthService` o entra a los sub-menús específicos de admin/estudiante. Gestiona las excepciones mostrando mensajes de error si algo falla (por ejemplo, si se exceden los intentos de login, mostrará ✗ `Error: ...` en consola). - `private static void menuAdmin(Administrador admin):` Menú secundario que se muestra tras iniciar sesión un administrador. Muestra opciones para **crear evento**, **emitir diploma**, **firmar diploma**, **listar diplomas**, o **volver** al menú anterior. Cada opción llama a un método interno (descritos abajo) o realiza acciones directas utilizando las listas y mapas globales. - `private static void menuEstudiante(Estudiante estudiante):` Menú secundario para un estudiante autenticado. Ofrece opciones para **marcar asistencia** a un evento, **consultar asistencia** en un evento, **consultar diplomas** del estudiante, o **volver** al menú anterior. Las opciones invocan métodos del estudiante o realizan consultas sobre las listas globales. - *Métodos auxiliares de Main (operaciones concretas en los menús):* - `crearEvento(Administrador admin):` Solicita por consola los datos de un nuevo evento (ID, nombre, fecha, créditos) usando métodos de `ConsoleIO`. Luego invoca `admin.crearEvento(...)` para obtener un Evento y lo agrega a la lista `EVENTOS`. Finalmente notifica ✓ `Evento creado.` en la consola. - `emitirDiploma(Administrador admin):` Permite al admin emitir un diploma. Internamente pide elegir un evento existente (`pickEvento()`) y un estudiante registrado (`pickEstudianteByCif()`). Genera un nuevo ID de diploma (por ejemplo, usando el tamaño de la lista `DIPLOMAS + 1`), y llama `admin.emitirDiploma(est, ev, nuevoId)`. Si

la asistencia del estudiante es válida, se obtiene un objeto `Diploma` que se añade a `DIPLOMAS` y se muestra en consola  `Diploma emitido: ...` junto con los datos del diploma (usando su `toString()`). Si la asistencia **no** es válida, el método `emitirDiploma` lanzará una excepción informando que el estudiante no tiene asistencia registrada, lo cual se captura y muestra como error. - `firmarDiploma(Administrador admin)`: Permite firmar digitalmente un diploma existente. Muestra en consola todos los diplomas en `DIPLOMAS` con un índice numérico, solicita seleccionar uno, luego invoca `admin.firmarDiploma(diplomaSeleccionado)` para marcarlo como firmado. Finalmente muestra  `Diploma firmado.` en consola. - **Listar diplomas**: (Corresponde a la opción 4 del menú admin) Recorre la lista `DIPLOMAS` e imprime cada diploma usando `System.out.println`, lo que aprovecha el `toString()` de `Diploma` para mostrar cada diploma en una línea. - *Helpers privados de selección*: `pickEvento()` y `pickEstudianteByCif()`. Estos métodos muestran en consola una lista numerada de eventos o estudiantes disponibles, y permiten elegir uno ingresando el número correspondiente. Devuelven el Evento o Estudiante seleccionado, para su uso en las funciones anteriores. También verifican que las listas no estén vacías antes de listar (de lo contrario lanzan una excepción, por ejemplo "No hay eventos creados.", que es manejada en los menús mostrando un error).

2. Estructuras de datos utilizadas

El sistema emplea varias estructuras de datos en memoria para relacionar las clases y mantener información durante la ejecución (no existe una base de datos persistente en

esta versión). A continuación se describen las principales estructuras y su función, con ejemplos sacados del código:

- **Map<String, Boolean> asistencia** (en clase Evento): Es un mapa que relaciona el CIF de cada estudiante con un valor booleano que indica su asistencia. Por ejemplo, si un evento tiene un estudiante con CIF "A123" que asistió, internamente `asistencia` contendrá la entrada "A123" -> `true`. Si otro estudiante con CIF "B456" **no asistió** (o no fue marcado), podría aparecer como "B456" -> `false` (o no estar en el mapa). Esta estructura permite verificar rápidamente, dado un estudiante, si asistió o no al evento mediante llamadas como `asistencia.getOrDefault(cif, false)`.
- **Map<String, Administrador> ADMIN_BY_USER** (en Main): Mapa global que almacena los administradores registrados por su nombre de usuario. La clave es el usuario (String) y el valor es el objeto `Administrador`. Por ejemplo, después de registrar un admin con usuario "admin001", podríamos tener `ADMIN_BY_USER.get("admin001")` devolviendo ese objeto `Administrador`.
- **Map<String, Estudiante> EST_BY_CIF** (en Main): Mapa global de estudiantes registrados, indexado por el `cif` (String) de cada Estudiante. Ejemplo: tras registrar un estudiante con CIF "CIF001", el mapa contendrá "CIF001" -> `Estudiante("CIF001", nombre, ...)`. Este mapa se utiliza tanto para el registro como para el login de estudiantes (verificando la existencia del CIF y la contraseña), y también para buscar rápidamente un estudiante por su CIF cuando se va a emitir un diploma o marcar asistencia.

- **List<Evento> EVENTOS** (en Main): Lista global de eventos creados. Cada vez que un administrador crea un evento nuevo, se añade a esta lista. Sirve para presentar al usuario (admin o estudiante) los eventos disponibles (por ejemplo, en los menús se listan todos los eventos con su índice). También se recorre cuando se quiere obtener todos los eventos del sistema o seleccionar uno.
- **List<Diploma> DIPLOMAS** (en Main): Lista global de diplomas emitidos. Cada diploma nuevo que un administrador emite se agrega aquí. Esta lista permite, por ejemplo, listar todos los diplomas existentes, así como filtrar los diplomas correspondientes a un determinado estudiante. En el código, se utiliza para buscar diplomas por estudiante (cuando un estudiante consulta sus diplomas, se filtra esta lista por CIF) y para seleccionar un diploma a firmar.
- **Otras estructuras:** El sistema también utiliza contadores (`AtomicInteger` `seqAdmin/seqStudent`) para IDs secuenciales en `AuthService` (aunque no se aprovechan plenamente, como se comentó) y utilidades de formato (por ejemplo, `DateTimeFormatter` en `ConsoleIO` para manejar fechas). Sin embargo, las estructuras principales para la lógica del negocio son las mencionadas arriba, que actúan como una pseudo-base-de-datos en memoria.

3. Relaciones entre clases

Las clases del sistema interactúan entre sí de forma coordinada. A continuación se resumen las relaciones principales y cómo se comunican:

- **Administrador ↔ Evento:** Un administrador **crea** eventos (invocando al constructor de Evento a través de `crearEvento`), pero **no los contiene** en su interior; es la clase Main la que guarda el evento en la lista global `EVENTOS`. Asimismo, el administrador puede **marcar la asistencia** de un estudiante en un evento: para ello utiliza `marcarAsistencia(evento, estudiante, asistio)`, que a su vez invoca un método del Evento (`registrarAsistencia`). Por lo tanto, la clase Administrador **utiliza** a la clase Evento para estas operaciones, sin poseerla directamente (relación de asociación, no de composición). Un evento, por su parte, no conoce al administrador específico que lo creó (no guarda referencia a él).
- **Administrador ↔ Estudiante:** La interacción es principalmente **indirecta** a través de Evento y Diploma. Por ejemplo, cuando un admin marca asistencia, lo hace indicando un Estudiante específico en un Evento (llamando `evento.registrarAsistencia(estudiante, true)`). Al emitir un diploma, el administrador toma un Estudiante y un Evento: verifica en el Evento la asistencia del Estudiante y, si es válida, crea un Diploma para ese Estudiante. No hay un atributo que enlace directamente Administrador con Estudiante, pero en los flujos de la aplicación el administrador gestiona acciones que involucran objetos Estudiante (por ejemplo, seleccionándolos de la lista para emitir diplomas).
- **Administrador ↔ Diploma:** El administrador es quien crea (emite) el Diploma para un estudiante determinado y también quien puede “firmarlo”. Esto establece que el diploma queda asociado al administrador solo en el **proceso** (el objeto

Diploma guarda referencia al estudiante y evento, pero no al administrador que lo emitió). Sin embargo, conceptualmente el administrador es el actor que garantiza que el diploma sea válido al firmarlo. La firma del diploma se modela simplemente cambiando un booleano en el objeto Diploma (*firmaDecano*).

- **Evento ↔ Estudiante:** Existe una relación de **uno a muchos**: un Evento puede tener *muchos* estudiantes asociados a su lista de asistencia. En la implementación, el Evento contiene un mapa *asistencia* donde cada entrada representa a un estudiante (por su CIF) y su estado de asistencia. Los métodos *registrarAsistencia*, *validarAsistencia* y *consultarAsistencia* reflejan esta relación. Un Estudiante en la clase modelo **no tiene** una lista de eventos a los que asistió, pero puede registrarse o marcar asistencia a un Evento específico pasando la referencia del Evento a sus métodos (*registrarEvento* o *marcarAsistencia*). En otras palabras, **el Evento conoce a los estudiantes que asistieron** (a través de su CIF en el mapa), y el Estudiante “se inscribe” en eventos mediante llamadas a métodos de Evento (directa o indirectamente vía Admin).
- **Evento ↔ Diploma:** Cuando se crea un Diploma, se le asocia el Evento correspondiente. Podemos considerar que existe una relación de **uno a muchos** del Evento hacia Diplomas: un mismo Evento puede tener múltiples diplomas emitidos (uno por cada estudiante que asistió y obtuvo su diploma). En la clase Diploma hay un atributo *evento* que guarda la referencia al Evento en cuestión. Esto permite, por ejemplo, que al imprimir o inspeccionar un Diploma se pueda

saber a qué evento pertenece (`Diploma.toString()` muestra el nombre del evento).

- **Estudiante ↔ Diploma:** De modo similar, hay una relación de **uno a muchos** del Estudiante hacia Diplomas: un estudiante puede recibir varios diplomas (uno por cada evento distinto al que asista exitosamente). En la clase Diploma se almacena una referencia al Estudiante. El estudiante puede consultar sus diplomas recorriendo la lista global de diplomas y filtrando aquellos cuyo CIF coincide con el suyo, como se hace en `menuEstudiante`. No hay un enlace directo en la clase Estudiante hacia Diploma (por ejemplo, un estudiante no tiene un atributo lista de diplomas), pero la asociación se materializa a través de la estructura global.
- **AuthService ↔ (Administrador, Estudiante):** AuthService interactúa con ambas clases principalmente **creando instancias y validando credenciales**. No contiene a los usuarios en sí, sino que utiliza los mapas globales `ADMINS_BY_USER` y `EST_BY_CIF` para buscarlos o agregarlos. Por ejemplo, `registrarAdmin` crea un Administrador nuevo y lo inserta en `ADMINS_BY_USER`, mientras que `loginAdmin` busca en dicho mapa y retorna el Administrador existente si las credenciales coinciden. Esta es una relación típica de un servicio hacia clases de modelo (no es un vínculo bidireccional, sino una funcionalidad de enlace).
- **Main ↔ (AuthService, ConsoleIO, clases de modelo):** La clase Main funciona como **orquestador** del programa: conoce la instancia de AuthService y las

colecciones globales de Administradores, Estudiantes, Eventos y Diplomas.

Main pasa estas colecciones a AuthService para registrar o autenticar usuarios, y también recibe objetos Administrador o Estudiante tras un login exitoso para luego invocar los menús correspondientes. Además, Main utiliza los métodos de ConsoleIO para interactuar con el usuario (lectura de entradas y mostrar mensajes). En conjunto, Main conecta la capa de servicios (AuthService), la capa de utilidades (ConsoleIO) y las clases de modelo (Administrador, Estudiante, Evento, Diploma), actuando como controlador del flujo de la aplicación.

En resumen, las clases están relacionadas principalmente a través de las **estructuras de datos compartidas** (los mapas y listas en Main) y mediante el **paso de objetos** como parámetros en llamadas de métodos. No hay referencias circulares complejas: cada objeto conoce a otros solo en los momentos necesarios (por ejemplo, un Diploma conoce su Evento y Estudiante; un Evento conoce a los estudiantes vía el mapa de CIF; un Estudiante conoce un Evento solo al momento de marcar asistencia, etc.). El diseño mantiene las responsabilidades separadas: AuthService maneja autenticación, ConsoleIO la entrada/salida, Main la coordinación general, y Administrador/Estudiante/Evento/Diploma encapsulan el dominio del problema.

4. Estados posibles de las clases

Cada clase del modelo tiene ciertos estados o etapas en las que pueden encontrarse sus instancias a lo largo del flujo de la aplicación:

- **Diploma:** Un diploma recién creado está *emitido* pero inicialmente **no firmado** (`firmaDecano = false`). Este es el estado normal justo tras llamar a `emitirDiploma`. Más adelante, cuando un administrador invoca `firmarDiploma`, el diploma pasa al estado de **firmado** (`firmaDecano = true`). Por lo tanto, los estados clave por los que pasa un Diploma son: *no emitido* (inexistente hasta que se crea), *emitido (no firmado)*, y *emitido y firmado*. Mientras un diploma no es emitido, simplemente no existe en la lista `DIPLOMAS`. Una vez emitido, existe (queda en la lista) y puede estar pendiente de firma (`firmaDecano false`) o ya firmado (`firmaDecano true`).
- **Evento:** Al crear un evento, inicialmente **no tiene estudiantes** registrados en su lista de asistencia (el mapa `asistencia` está vacío). En ese estado podríamos decir que el evento está *sin asistentes*. Conforme se registran o marcan asistencias, el evento pasa a un estado *con asistentes registrados* (su mapa va teniendo entradas `true/false`). Un evento podría quedarse sin ningún asistente marcado (por ejemplo, si nadie asistió o aún no se ha tomado lista), o tener muchos asistentes. También podríamos considerar el estado de evento *concluido* tras su fecha, pero en la aplicación no hay un atributo explícito de finalización; no obstante, después de la fecha del evento se supone que ya no se marcan nuevas asistencias ni se emiten más diplomas para ese evento (esa lógica podría manejarse externamente).
- **Estudiante:** Un estudiante recién registrado comienza con 0 créditos acumulados y posiblemente *sin asistencias marcadas* en eventos. A medida que

participa en eventos, para cada evento podría estar en estado de *asistente* (si marcó asistencia) o *no asistente*. Esto no se refleja como un atributo dentro de Estudiante, sino a través de las entradas en los eventos. Desde la perspectiva del sistema, un estudiante puede estar *habilitado para diploma* (cuando asistió a un evento y cumple los requisitos para recibir diploma) o *no habilitado* (si no asistió o no califica, no recibe diploma de ese evento). Tras recibir diplomas, el estudiante incrementa su `creditosAcumulados` por cada diploma emitido; así, otro aspecto de estado es su nivel de créditos: inicia *sin créditos* (0) y luego pasa a *con créditos acumulados* (>0) una vez obtiene diplomas. Adicionalmente, en términos de uso del sistema, un estudiante puede estar *no autenticado* (antes de iniciar sesión) o *autenticado* (tras hacer login correctamente, mientras dura su sesión activa en el menú estudiante).

- **Administrador:** Un administrador, una vez creado, siempre tiene sus credenciales definidas y permanece en el sistema. Sus posibles estados durante la ejecución son similares al caso del estudiante en cuanto a la sesión: puede estar *no autenticado* (si aún no inicia sesión en la aplicación) o *autenticado* (una vez ingresa al sistema, navegando por el menú admin). En cuanto a acciones, un administrador podría no haber creado eventos ni diplomas aún (estado inicial justo tras su registro) y luego pasar a un estado “activo” tras crear eventos o emitir diplomas, pero esos son más bien estados del *sistema* por las acciones del admin, no estados internos del objeto Administrador (ya que sus atributos no cambian en función de esas acciones, salvo quizás su nombre de

usuario/contraseña si se consideraran cambios, lo cual no sucede en este prototipo).

- **AuthService:** Aunque no es un modelo de entidad del dominio, podemos mencionar que mantiene un estado interno relacionado con los intentos de login. Si un usuario excede el número máximo de intentos (`maxIntentos`), los métodos `loginAdmin` o `loginEstudiante` lanzan una excepción y de hecho **bloquean** la autenticación en ese momento (en la aplicación esto simplemente termina el proceso de login informando el error). Ese es un estado temporal de “bloqueo” de la operación de login (no del objeto en sí, que puede reutilizarse después para otro intento de login si se vuelve a llamar). Fuera de esto, AuthService no conserva información de sesiones ni de usuarios (solo utiliza los datos que se le pasan en cada llamada).
- **Main (y estructuras globales):** Las colecciones globales definen también estados generales del sistema. Por ejemplo, la lista de eventos puede estar *vacía* (ningún evento creado aún) o *con eventos*; la lista de diplomas puede estar vacía (ningún diploma emitido) o con diplomas emitidos. El programa maneja estos estados verificando antes de ciertas operaciones: por ejemplo, no se puede emitir un diploma si no hay eventos creados o estudiantes registrados (el código lanza excepciones "No hay eventos creados." o "No hay estudiantes." según el caso), ni se puede firmar un diploma si la lista `DIPLOMAS` está vacía. Estas condiciones reflejan estados del sistema que el usuario encuentra: *sin eventos*, *sin estudiantes* o *sin diplomas* disponibles, y se manejan

mostrando mensajes de error apropiados. Cuando se crean eventos, se registran estudiantes o se emiten diplomas, el estado del sistema cambia (por ejemplo, de “no hay eventos” a “hay X eventos disponibles”).

En general, la aplicación maneja los estados válidos antes de permitir ciertas acciones, usando condiciones y excepciones para guiar el flujo. Por ejemplo, un diploma solo puede emitirse cuando el estudiante tiene asistencia válida (estado del Evento para ese estudiante = asistió); una vez emitido el diploma, puede requerir firma (estado pendiente de firma) antes de considerarse “completo”. Estos estados garantizan la consistencia del proceso: solo quien asiste obtiene diploma, y los diplomas pueden distinguirse si están en espera de firma o ya firmados.

5. Ejemplos concretos de uso

A continuación se describen algunos escenarios de uso típicos, ilustrados con comportamientos vistos en el código proporcionado:

- **Registro de un nuevo estudiante:** Cuando en el menú principal se elige la opción de registrar estudiante (disponible sin necesidad de login previo), el sistema llama a `AuthService.registrarEstudiante(EST_BY_CIF)`. Supongamos que se ingresan los datos: CIF "CIF123", nombre "Juan Pérez", contraseña "pass123" (y se confirma correctamente). `AuthService` validará que "CIF123" no exista ya en `EST_BY_CIF` y que cumpla el formato requerido, luego creará `new Estudiante("CIF123", "Juan Pérez", "pass123")`. Este objeto se guarda en el mapa global de estudiantes bajo la clave "CIF123". En la consola se mostrará un

mensaje de éxito (✓ Estudiante registrado.) indicando que el registro se completó. A partir de entonces, Juan puede iniciar sesión usando su CIF y contraseña.

- **Marcado de asistencia a un evento:** Imaginemos que ya existe un evento creado (por el administrador) con ID "SYSTECH2025", y que hay estudiantes registrados en el sistema. Si Juan (CIF123) quiere marcar su asistencia, primero inicia sesión (opción "Login Estudiante" en el menú principal), y luego en su menú de estudiante elige la opción "Marcar asistencia a un evento". El sistema listará en la consola los eventos disponibles, por ejemplo:

1) SYSTECH2025 - NombreDelEvento

Juan ingresa el número correspondiente (en este caso 1). Internamente, `Main.pickEvento()` devuelve el objeto Evento seleccionado (`eventoSystech`). Luego se invoca `juan.marcarAsistencia(eventoSystech)`. Dentro de este método, como Juan **no estaba aún registrado** en la asistencia de ese evento, `eventoSystech.consultarAsistencia(juan)` devolverá `false`, entonces procede a registrar la asistencia: `eventoSystech.registrarAsistencia(juan, true)`. Esto añade la entrada "CIF123" -> `true` al mapa de asistencia del evento. En la consola el sistema confirma con ✓ Asistencia marcada.. Si posteriormente (durante la misma sesión, por ejemplo) Juan selecciona la opción "Consultar mi asistencia en un evento", el sistema nuevamente le pedirá que elija un evento y luego mostrará algo como:

Estado de asistencia en "NombreDelEvento": Sí

indicando que, para ese evento, su asistencia está registrada.

- **Emisión de un diploma:** Tras la realización del evento, el administrador puede proceder a emitir los diplomas para los estudiantes asistentes. El administrador inicia sesión (opción “Login Administrador”) y en su menú elige “Emitir diploma”. La aplicación le pedirá primero seleccionar un evento de la lista de eventos existentes (por ejemplo, SYSTECH2025) y luego seleccionar un estudiante de la lista de estudiantes registrados (por ejemplo, "CIF123 - Juan Pérez"). Una vez seleccionados, `Main.emitirDiploma(admin)` calcula un nuevo ID para el diploma (por ejemplo, si no hay diplomas previos, usará 1) y llama a `admin.emitirDiploma(juan, eventoSystech, 1)`. Dentro de este método, se verifica `eventoSystech.validarAsistencia(juan)`, que retornará true porque Juan marcó asistencia. Entonces se crea un objeto Diploma `d = new Diploma(1, fechaActual, eventoSystech, juan)`. Este diploma tendrá `firmaDecano = false` inicialmente. Además, el método suma los créditos del evento SYSTECH2025 a los créditos acumulados de Juan:

```
juan.setCreditosAcumulados( juan.getCreditosAcumulados() +  
eventoSystech.getCreditosOtorgados() );
```

El diploma `d` se añade a la lista global `DIPLOMAS`. En la consola se muestra algo como:

✓ Diploma emitido: `Diploma{id=1, fechaEmision=Thu Sep 11 15:37:32 CST 2025, firmaDecano=false, evento=SYSTECH2025, estudiante=Juan Pérez}`

(La representación exacta depende del formato `toString()` del Diploma). Ahora Juan tiene un diploma emitido a su nombre, y sus créditos acumulados han aumentado según lo definido por `creditosOtorgados` del evento.

- **Firma de un diploma:** Después de emitir los diplomas, el administrador puede proceder a firmarlos digitalmente (simulando la firma del decano en los certificados). En el menú del administrador, al elegir “Firmar diploma”, el sistema listará todos los diplomas emitidos hasta el momento, por ejemplo:

1) `Diploma{id=1, fechaEmision=Thu Sep 11 15:37:32 CST 2025, firmaDecano=false, evento=SYSTECH2025, estudiante=Juan Pérez}`

(cada diploma con su índice). El administrador ingresa el número del diploma que desea firmar (en este caso 1). Entonces `Main.firmarDiploma(admin)` obtiene ese diploma de la lista y llama `admin.firmarDiploma(diplomaJuan)`. Este método ejecuta `diplomaJuan.setFirmaDecano(true)`. La consola confirma la acción mostrando ✓ Diploma firmado.. Si ahora el administrador listara los diplomas, el diploma firmado aparecerá con `firmaDecano=true`, indicando que ya ha sido validado/firmado.

- **Consulta de diplomas por el estudiante:** Juan, como estudiante, puede revisar en cualquier momento qué diplomas tiene emitidos. En su menú de estudiante,

al elegir la opción “Consultar mis diplomas”, el programa filtrará la lista `DIPLOMAS` buscando aquellos cuyo estudiante asociado tenga el CIF de Juan:

```
DIPLOMAS.stream()  
    .filter(d -> d.getEstudiante().getCif().equals("CIF123"))  
    .forEach(juan::consultarDiploma);
```

Para cada diploma de Juan encontrado, se invoca `juan.consultarDiploma(d)`, que imprimirá los detalles del diploma en consola (utilizando `Diploma.toString()`). Así, Juan verá en pantalla la información de cada diploma que posee, por ejemplo:

```
Diploma{id=1, fechaEmision=Thu Sep 11 15:37:32 CST 2025,  
firmaDecano=true, evento=SYSTECH2025, estudiante=Juan Pérez}
```

(En este ejemplo, con `firmaDecano=true` porque el administrador ya firmó el diploma). Si Juan no tuviera ningún diploma emitido, el método `consultarDiploma` simplemente mostraría el mensaje “No hay diploma disponible.”.

Estos ejemplos reflejan casos de uso comunes del sistema: registro de usuarios, registro de asistencia a eventos, generación de diplomas y consultas, tal como está implementado en la aplicación de consola.



6. Contexto general del sistema

El sistema **Systech – Diplomas Digitales** se concibió para digitalizar el proceso de entrega de diplomas en eventos académicos como SYSTECH, alineado con la visión de modernización de la Facultad de Ingeniería. Es importante entender el contexto y alcance de esta aplicación:

- **Aplicación en consola sin persistencia:** Toda la interacción ocurre en la línea de comandos (consola), mediante menús de texto. **No se utiliza una base de datos** ni se guardan datos en archivos; la persistencia es únicamente en memoria mientras el programa se está ejecutando. Por ello, si se cierra la aplicación, se pierde la información registrada (usuarios, eventos, diplomas). Esta decisión simplifica el prototipo, aunque en un entorno real se integraría con una base de datos para almacenar la información de forma permanente, y posiblemente con servicios de correo para el envío automático de diplomas.
- **Digitalización del proceso de diplomas:** Antes, los diplomas se entregaban físicamente al final del evento de manera desordenada, lo que generaba inconvenientes (colas, confusión, retrasos). Este sistema propone un flujo digital: el administrador registra la asistencia de los estudiantes y el sistema genera automáticamente diplomas personalizados para cada asistente. Aunque en el código actual no se implementa el envío por correo, está pensado que estos diplomas digitales se envíen al correo institucional de cada estudiante, evitando las entregas físicas caóticas. Con esto se agiliza la distribución (por ejemplo, los

estudiantes podrían recibir su diploma por email al día siguiente del evento) y se garantiza que **solo** quienes realmente asistieron obtengan su certificado.

- **Roles de usuario – Administrador vs Estudiante:** La aplicación distingue entre dos tipos de usuarios con diferentes funcionalidades. Los **administradores** son responsables de la gestión: pueden crear eventos, marcar (o cargar) asistencias y emitir/firmar diplomas. Los **estudiantes**, por su parte, pueden registrarse a sí mismos, iniciar sesión para marcar su propia asistencia (en lugar de firmar en una hoja, lo hacen digitalmente) y consultar sus diplomas digitales. Esta separación asegura que solo personal autorizado (administradores) puedan crear eventos y emitir diplomas, mientras que cada estudiante tiene acceso únicamente a sus propias acreditaciones.
- **Alcance limitado del prototipo:** Cabe señalar que algunas funcionalidades están simplificadas o son supuestas. Por ejemplo, la “firma del decano” se modela solo con una bandera booleana en el objeto Diploma (no hay una firma digital real ni un certificado electrónico). Tampoco se genera un archivo PDF real para cada diploma en este prototipo; la clase Diploma podría ampliarse en el futuro para incluir, por ejemplo, un campo con la ruta del archivo PDF generado. Asimismo, no se envía ningún correo automáticamente en esta implementación. El enfoque estuvo en cubrir la lógica central: registro de asistencias y generación controlada de diplomas digitales, sentando las bases para que en una versión futura se integren la generación de PDF y el envío por email de forma automática.

- **Uso de la aplicación:** El uso típico sería el siguiente: antes o durante el evento, un administrador (por ejemplo, personal de la facultad) se registra o inicia sesión en el sistema y **crea un evento** en la aplicación. Durante el evento, los estudiantes pueden registrarse en el sistema si no lo estaban y **marcar su asistencia** digital a través del menú de estudiante (o el administrador podría marcar a todos los asistentes, según el caso de uso “cargar lista de asistencia”). Tras el evento, el administrador autenticado elige **generar/emitir diplomas**; el sistema automáticamente crea los diplomas para cada estudiante marcado como asistente. Luego el administrador puede **firmar** digitalmente esos diplomas. Finalmente, en una extensión real, el sistema enviaría los diplomas firmados al correo de cada estudiante y mantendría un **historial** (por ejemplo, accesible para consultas posteriores sobre quién recibió diploma). Todo esto se realiza en este prototipo mediante un menú de consola, con mensajes de confirmación (ej. “ Operación exitosa”) y avisos de error (ej. “ Error: descripción”).

En conclusión, esta aplicación de consola implementa una solución básica pero eficaz para la emisión digital de diplomas en el evento SYSTECH, reemplazando el proceso manual por un flujo controlado digitalmente. Si bien es un prototipo sin persistencia ni interfaz gráfica, ilustra claramente cómo las entidades del sistema (Administrador, Estudiante, Evento, Diploma) colaboran para automatizar la entrega de certificados, mejorando la organización y reduciendo errores en comparación con el antiguo proceso. Es un primer paso hacia un sistema completo que podría generar archivos

PDF de los diplomas y enviarlos automáticamente, integrándose con las herramientas institucionales existentes.