SISTEMA DE ADMINISTRACION UNIVERSITARIA DE ACTIVIDADES CULTURALES Y CIENTÍFICAS

CARRERA DE INFORMÁTICA

PROGRAMACIÓN II (INF – 121)

GRUPO 28 – PROYECTO 18

Jair Arias Flores

Alejandra Palmenia Bravo Iriarte

Gabriel Zeus Cutile Lopez

Alejandro Javier Farrachol Alcocer

Víctor Alberto Machaca Calle

Lic. Rosalía López Montalvo

24/07/2025

I. RESUMEN DEL PROYECTO

El sistema está diseñado para automatizar la gestión de clubes universitarios, facilitando el registro de estudiantes, actividades (culturales/científicas) y clubes; el control de certificados generados por participación en actividades; y la persistencia de datos para evitar pérdida de información. Asimismo, en cuanto a los problemas específicos abordamos la eliminación del manejo manual de archivos y listas en papel, la centralización de la información de estudiantes, actividades y recursos en un sistema digital y la garantía sobre la disponibilidad de datos mediante almacenamiento persistente.

El sistema de Gestión de Clubes Universitarios está diseñado para centralizar y automatizar procesos clave en la administración de actividades estudiantiles. Su principal funcionalidad es el registro y organización de clubes, permitiendo crear, modificar, eliminar y listar clubes universitarios, cada uno con sus datos básicos (nombre, ID) y listas de actividades asociadas. Adicionalmente, gestiona la inscripción de estudiantes a estos clubes, almacenando información como carrera, semestre y edad, lo que facilita el seguimiento de su participación en el ecosistema universitario.

Otra funcionalidad central es la gestión de actividades, diferenciando entre eventos culturales (ej: teatro, música) y científicos (ej: proyectos de investigación). Cada actividad registra detalles como tema, duración y certificado asociado, este último generado automáticamente al finalizar el evento. El sistema también incluye un módulo de recursos genéricos (GestorRecurso<T>), que permite asignar materiales o herramientas específicas a cada club, como equipos o espacios físicos, mediante un enfoque flexible y reutilizable.

Finalmente, el sistema destaca por su persistencia de datos robusta. Utiliza archivos binarios (.dat) y serialización para guardar toda la información, asegurando que los datos de clubes, estudiantes y actividades estén disponibles incluso después de reiniciar la aplicación. Aunque carece de interfaz gráfica o conexión a bases de datos, su arquitectura basada en clases y archivos lo hace eficiente para entornos académicos pequeños, con opciones claras de escalabilidad (ej: migrar a JSON o SQL en futuras versiones).

El sistema fue desarrollado en Java, aprovechando su enfoque en programación orientada a objetos (POO) para modelar entidades como clubes, estudiantes y actividades. Se utilizaron características clave de Java, como genéricos (ej: GestorRecurso<T>) para manejar recursos de forma flexible, serialización para persistencia de datos y clases abstractas (ej: Usuario) para compartir lógica común. El código se estructuró en paquetes modularizados (Clases, Archivos), siguiendo buenas prácticas de organización.

Para la persistencia, el sistema emplea archivos binarios (.dat) mediante las clases ObjectInputStream y ObjectOutputStream de Java, lo que permite guardar y recuperar objetos directamente. Aunque no se usaron librerías externas (como Gson o JDBC), la implementación nativa de serialización garantiza compatibilidad y simplicidad. El IDE inferido es NetBeans (por los comentarios autogenerados nbfs://nbhost), pero el código es compatible con cualquier entorno de desarrollo Java (Eclipse, IntelliJ). Como mejora futura, podría integrarse JavaFX para una interfaz gráfica o SQLite para gestión de bases de datos relacionales.

II. OBJETIVOS

Diseñar e implementar un sistema que aplique principios de programación, estructuras de datos, y patrones de diseño para la administración de actividades culturales y científicas de índole universitaria.

Objetivos específicos

- Modelar jerarquía de clases usando herencia y polimorfismo.
- Implementar manejo de errores con excepciones personalizadas.
- Usar estructuras genéricas para datos flexibles.
- Aplicar un patrón de diseño relevante.
- Implementar persistencia o interfaz funcional.
- Simular interacción funcional con usuarios o datos.

III. ANÁLISIS DEL PROBLEMA

3.1. Descripción del contexto

Dado el contexto actual, en un entorno universitario, la administración de actividades culturales y científicas enfrenta diferentes desafíos que afectan e impactan tanto a estudiantes como a administradores. En este sentido, a continuación, se presentan las problemáticas más relevantes y cómo el sistema propuesto las resuelve de manera eficiente.

3.1.1. Organización manual y propensa a errores

En la actualidad, todavía existen clubes universitarios que gestionan sus actividades mediante hojas de cálculo, documentos físicos o mensajes en grupos de WhatsApp o Telegram, sin embargo, este método manual genera problemas como:

- **Pérdida de datos:** La falta de un registro centralizado dificulta la actualización y recuperación de información.
- **Duplicación de registros:** Los estudiantes pueden aparecer inscritos múltiples veces en una misma actividad.

La solución para resolver esta problemática deriva de que el sistema implementará una base de datos estructurada que centralizará toda la información. Por ejemplo, por un lado, la clase *ClubUniversitario* gestiona automáticamente las actividades y participantes, eliminando la necesidad de registros manuales. Por otro lado, las validaciones en tiempo real evitan inscripciones duplicadas o inconsistentes.

Los resultados de la implementación serán: la obtención de información confiable y accesible en todo momento y la reducción de errores humanos en la gestión de datos.

3.1.2. Falta de trazabilidad en certificados

La generación manual de certificados conlleva riesgos como:

- **Falsificación:** Los certificados en Word o PDF pueden ser alterados con facilidad.
- **Inconsistencias:** Estudiantes pueden recibir certificados por actividades no completadas.

Para resolver esta problemática el sistema automatizará la generación de certificados bajo reglas estrictas. Por ejemplo, la clase *Certificado* se genera solo cuando actividad.estaCompletada() es verdadero. Además, cada certificado incluye un ID único (combinación de ID de actividad y estudiante) y sellos digitales para verificación. Los resultados serán: la obtención de certificados confiables y fácilmente verificables; y la eliminación de fraudes o errores en la acreditación.

3.1.3. Desbalance en la participación estudiantil

Algunos estudiantes participan en exceso, por ejemplo, mayor a diez mientras otros no se involucran. Este hecho genera:

- **Sobrecarga académica:** Estudiantes con múltiples actividades descuidan sus estudios.
- **Clubes invisibles:** Algunas actividades no logran atraer participantes.

La solución frente a esta problemática está en el hecho de que el sistema estará construido para establecer límites programados y fomentar la equidad, pues en este sentido, la clase *Estudiante* valida que ningún alumno exceda siete actividades por semestre. Los resultados serán: promover la distribución equilibrada de la participación estudiantil y la mayor visibilidad para todos los clubes.

3.1.4. Comunicación ineficiente

La falta de un canal centralizado para notificaciones provoca:

- **Eventos desatendidos:** Los estudiantes no reciben recordatorios de actividades.
- **Cambios no comunicados:** Cancelaciones o modificaciones no llegan a tiempo a los participantes.

Para resolver esta problemática el sistema implementará un servicio centralizado de notificaciones automáticas mediante: el patrón Singleton, que garantizará una única instancia global del servicio de notificaciones, evitando duplicación de mensajes y centralizando la gestión. Los resultados serán: la comunicación unificada y oportuna entre clubes y estudiantes y se obtendrá la reducción de eventos fallidos gracias al envio automatizado de recordatorios y alertas.

En este marco se desarrollará una aplicación de escritorio que cuente con un *backend* conformado por los patrones de diseño (Java: POO), un *frontend* conformado por una interfaz intuitiva (con Swing) y *persistencia de datos* que incorporará una base de datos por medio de serialización (.txt).

3.2. Requisitos funcionales

ID	Requisito	Descripción
RF – 01	Registrar actividades	Permitir a los administradores crear actividades científicas/culturales con ID, tema, duración y tipo.
RF – 02	Inscribir estudiantes	Los estudiantes pueden unirse a actividades si no exceden su límite personal (7) ni el cupo de la actividad (100).

RF – 03	Generar certificados	Automatizar la emisión de certificados al completar actividades, con ID único y datos del estudiante.
RF – 04	Buscar actividades por tipo	Filtrar actividades científicas/culturales por área de investigación o tipo de arte.
RF – 05	Actualizar datos de estudiantes	Modificar información académica de estudiantes (carrera, semestre).
RF – 06	Validar límites de clubes	Asegurar que un club no exceda 60 estudiantes ni una actividad 100 participantes.
RF – 07	Exportar reportes	Generar archivos PDF/CSV con listas de estudiantes por club o actividades por semestre.

3.3. Requisitos no funcionales

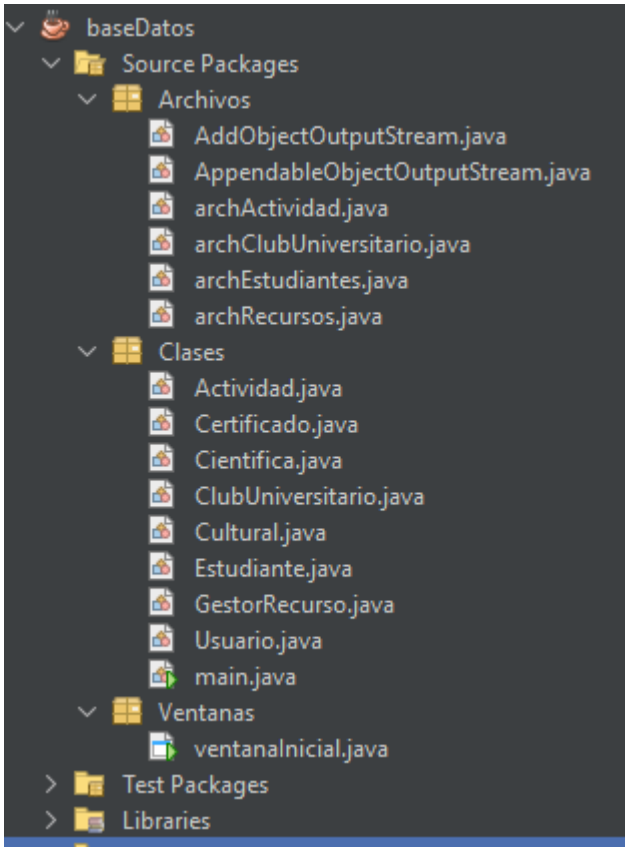
ID	Requisito	Descripción
RNF – 01	Rendimiento	El sistema debe responder en menos de 2 segundos para operaciones críticas como inscripciones.
RNF – 02	Seguridad	Todos los datos personales (correos, IDs) deben estar encriptados.
RNF – 03	Usabilidad	La interfaz debe permitir completar inscripciones en máximo 3 clics.
RNF – 04	Escalabilidad	El sistema debe soportar un crecimiento del 300% en actividades y usuarios.
RNF – 05	Disponibilidad	El sistema debe estar operativo el 99.5% del tiempo en horario académico.

3.4. Casos de uso

ID	Caso de uso	Actor	Descripción	Clases involucradas
UC – 01	Registrar actividad	Administrador	Crear/editar/eliminar actividades (culturales/científicas).	*ClubUniversitario*, *Actividad*
UC – 02	Inscribir estudiante	Estudiante	Unirse a una actividad (validando límites).	*Estudiante*, *Actividad*
UC – 03	Generar certificado	Sistema (automático)	Emitir certificado al completar actividad.	*Certificado*, *Actividad*
UC – 04	Buscar recursos	Coordinador	Filtrar actividades por tipo o fecha.	*GestorRecursos* <T>, *Actividad*
UC – 05	Modificar datos de estudiante	Administrador	Actualizar información académica (semestre, carrera).	*Estudiante*

IV. DISEÑO DEL SISTEMA

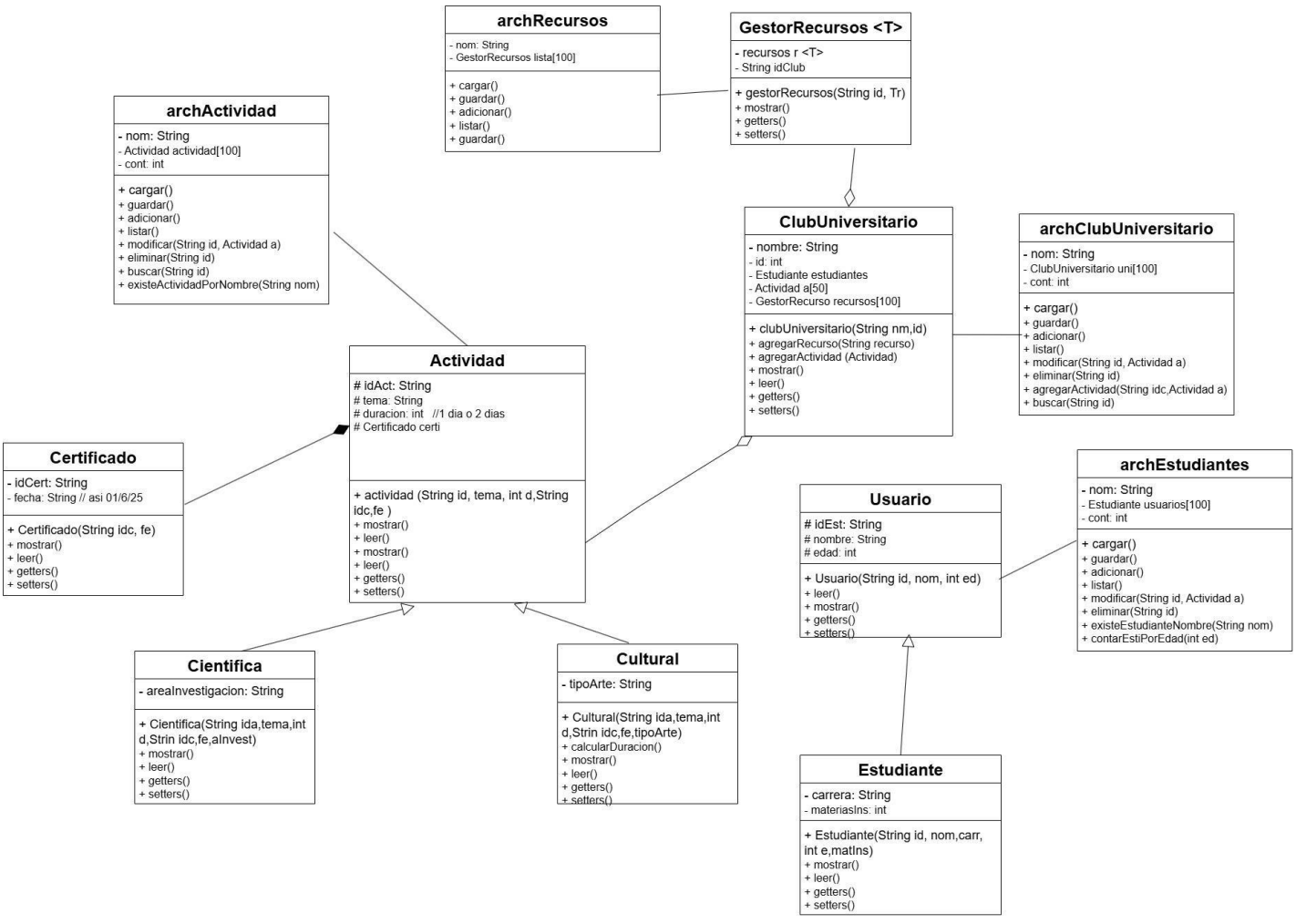
4.1. Clases y jerarquías




```java
package Clases;

import java.util.Scanner;
import java.io.*;
/**
 *
 * @author VictorAlbertoMachaca
 */
public class Actividad implements Serializable{
    protected String idAct;
    protected String tema;
    protected int duracion;
    protected Certificado certi;

    public Actividad(String idAct, String tema, int duracion,String idCert, String fecha) {
        this.idAct = idAct;
        this.tema = tema;
        this.duracion = duracion;
        certi = new Certificado(idCert,fecha);
    }
    public void mostrar(){
        System.out.println(idAct+"\t"+tema+"\t"+duracion);
        certi.mostrar();
    }
    public void leer(){
        Scanner sc =  new Scanner(System.in);
        idAct = sc.next();
        tema = sc.next();
        duracion = sc.nextInt();
        certi.leer();
    }
```

```java
import java.io.Serializable;
import java.util.Scanner;

/**
 *
 * @author VictorAlbertoMachaca
 */
public class Cultural extends Actividad implements Serializable{
    private String tipoArte;

    public Cultural(String idAct, String tema, int duracion ,String idCert,String fecha,String tipoArte){
        super(idAct, tema, duracion,idCert,fecha);
        this.tipoArte = tipoArte;
    }

    public void mostrar(){
        System.out.println(tipoArte);
        super.mostrar();
    }
    public void leer(){
        Scanner sc = new Scanner(System.in);
        super.leer();
        tipoArte = sc.next();
    }
```

4.1.1. Diagramas de clase – UML



4.1.2. Tabla de clases

Clase	Atributos principales	Métodos clave
ClubUniversitario	nombre, ID, estudiante, actividad	agregarEstudiante(), agregarActividad(), mostrar ()
Actividad	ID, tema, duración, participantes	agregarParticipantes(), estaCompletada(), mostrar()
Cultural	tipoArte	calcularDuracion(), mostrar()
Científica	areaInvestigacion	calcularDuracion(), mostrar()
Estudiante	edad, carrera, semestre, actividadInscrita	inscribirActividad(), mostrar()
Administrador	permisos	crearActividad(), eliminarActividad(), mostrar ()
Usuario	ID, nombre, correo, encriptado	registrar(), validarIdentidad(), mostrar()
Certificado	ID, fecha, actividad, estudiante	generarPDF(), mostrar()
GestorRecursos	recursos	agregarRecursos(), buscarRecursos(), mostrar()

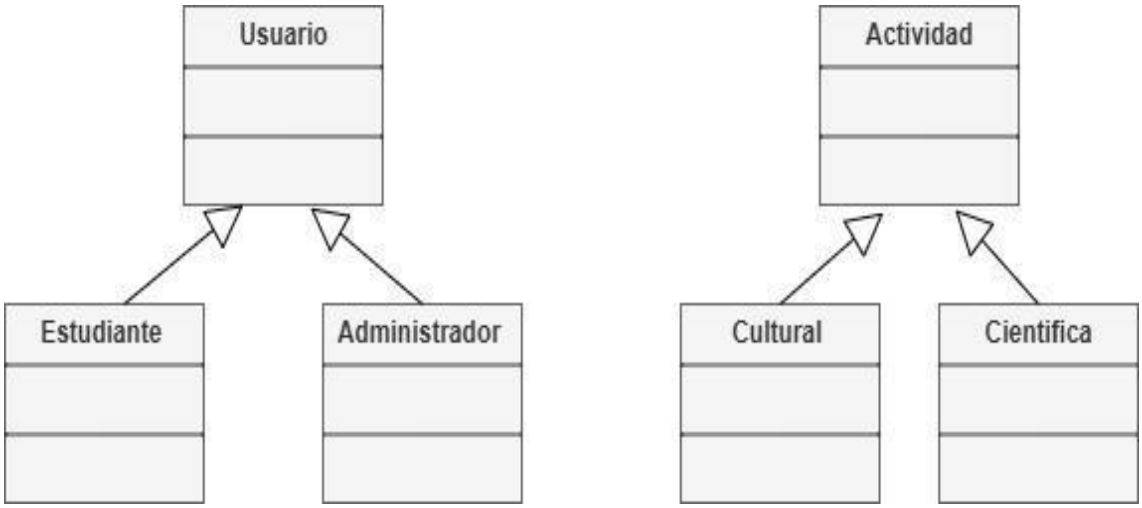
4.2. Relaciones entre clases

```java
public class Actividad implements Serializable{
    protected String idAct;
    protected String tema;
    protected int duracion;
    protected Certificado certi;

    public Actividad(String idAct, String tema, int duracion,String idCert, String fecha) {
        this.idAct = idAct;
        this.tema = tema;
        this.duracion = duracion;
        certi = new Certificado(idCert,fecha);
    }
    public void mostrar(){
        System.out.println(idAct+"\t"+tema+"\t"+duracion);
        certi.mostrar();
    }
    public void leer(){
        Scanner sc =  new Scanner(System.in);
        idAct = sc.next();
        tema = sc.next();
        duracion = sc.nextInt();
        certi.leer();
    }
```

```java
public class Certificado implements Serializable{
    protected String IdCert;
    protected String fecha;


    public Certificado(String IdCert, String fecha) {
        this.IdCert = IdCert;
        this.fecha = fecha;
    }
    public void mostrar(){
        System.out.println(IdCert+"\t"+fecha);
    }
    public void leer(){
        Scanner sc = new Scanner(System.in);
        IdCert = sc.next();
        fecha = sc.next();
    }
```

4.2.1. Herencia

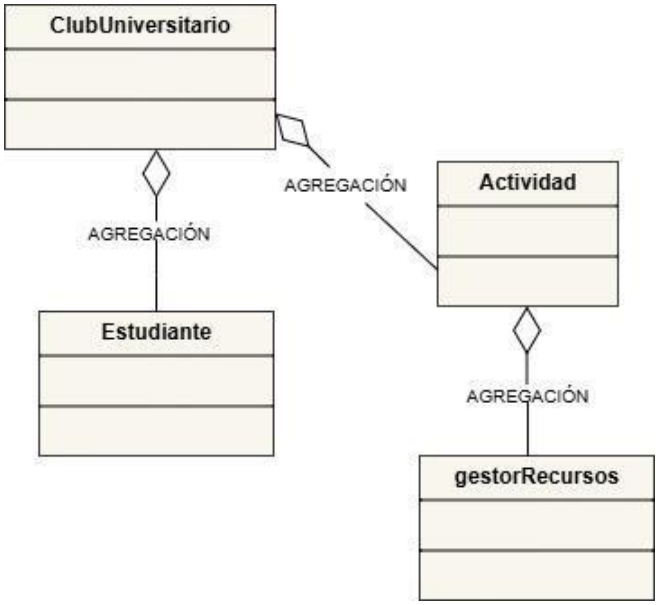
Relación "es – un" donde una clase hija extiende una clase padre.



Usuario es la clase base (abstracta) con atributos comunes (id, nombre, correo).
Estudiante y *Administrador* son subclases que heredan de *Usuario* y añaden atributos específicos (carrera, permisos).
Actividad es clase abstracta padre de *Científica* y *Cultural* (especializan **calcularDuracion()**).

4.2.2. Agregación (“tiene – un” relación débil)

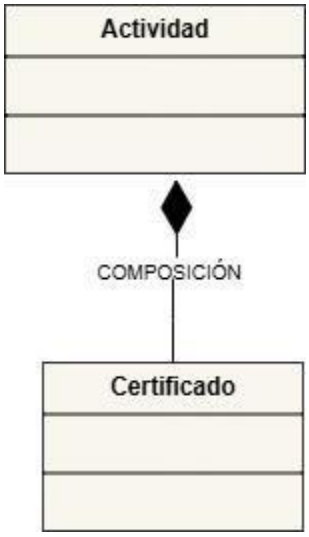
El objeto hijo puede existir independientemente del padre.



ClubUniversitario contiene *Estudiante*, pero los estudiantes pueden cambiar de club o existir sin uno.
Actividad referencia a *Estudiante* (participantes), pero no los gestiona.

4.2.3. Composición (“tiene – un” relación fuerte)

El objeto padre controla el ciclo de vida del objeto hijo. Si el padre se destruye, los hijos también.



ClubUniversitario crea y destruye su(s) Actividad(es) (multiplicidad 1..*).
Actividad genera *Certificado* al completarse (no existen sin actividad).

4.2.4. Interacción entre clases

Clase principal	Clase con la que interactúa	Tipo de relación
GestorRecursos<T>	Todas las clases que pueden ser recursos	Agregación
Certificado	Actividad	Composición
Certificado	Estudiante	Asociación
Actividad	Estudiante	Asociación
ClubUniversitario	Estudiante	Agregación
ClubUniversitario	Actividad	Agregación
Usuario	Administrador	Herencia
Usuario	Estudiante	Herencia
Cientifica	Actividad	Herencia
Cultural	Actividad	Herencia
Estudiante	Actividad	Asociación

V. DESARROLLO

5.1. Estructura general del código

5.1.1. Clases y subclases

1. Usuario (clase abstracta)
   - Atributos: idEst, nombre, edad.
   - Subclases:
     - Estudiante: Añade carrera y matIns.

- Administrador (no implementada en el código, pero sugerida por el UML).

2. Actividad (clase base)
   - Atributos: idAct, tema, duracion, certi (de tipo Certificado).
   - Subclases:
     - Cultural: Añade tipoArte.
     - Cientifica: Añade AreaDeInvestigacion.
3. ClubUniversitario
   - Atributos: nombre, id, listas de Actividad y Estudiante.
4. Certificado
   - Atributos: IdCert, fecha.
5. GestorRecurso<T> (clase genérica)
   - Maneja recursos de cualquier tipo (T).

5.1.2. Uso de genéricos

- Implementación:
  - Clase genérica para gestionar recursos asociados a un club (idClub).

5.1.3. Métodos sobreescritos (@Override)

- mostrar() y leer():
  - Sobrescritos en Actividad, Cultural, Cientifica, Certificado, y Estudiante para personalizar la visualización/lectura de datos.

5.1.4. Validaciones en clases

- archActividad.java y archEstudiantes.java:
  - Límite de capacidad: Verifican si hay espacio en los arreglos (count < actividad.length).
  - Búsquedas por ID o nombre:
    - Métodos como existeActividadPorNombre() o buscar(String id) verifican la existencia de datos antes de operar.
- ClubUniversitario.java:
  - Agregar actividades: No permite null en listas (control implícito al usar ArrayList).

Validaciones en persistencia

- Archivos binarios (.dat):
  - Verifican si el archivo existe antes de cargar (File.exists()).
  - Manejo de excepciones para FileNotFoundException y ClassNotFoundException.

5.1.5. Persistencia con archivos

- Serialización:
  - Todas las clases implementan Serializable para guardar/recuperar objetos en archivos binarios.

VI. PERSISTENCIA DE DATOS

El sistema utiliza archivos binarios (.dat) mediante serialización de objetos en Java (Serializable). Este formato permite guardar estructuras complejas (como listas de clubes o actividades) manteniendo las relaciones entre objetos. Las ventajas de la serialización

es que: 1) conserva la estructura de objetos completos (atributos + referencias), 2) es más eficiente que formatos de texto (.txt, .csv) para datos complejos y 3) cuenta con un soporte nativo en Java sin librerías externas.

Las clases encargadas de la lectura/escritura son:

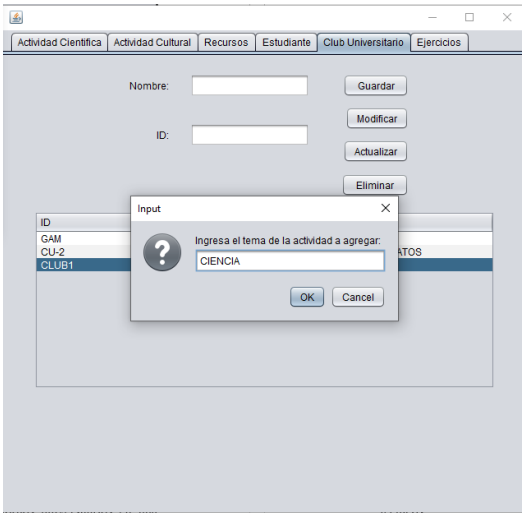
1. archClubUniversitario.java

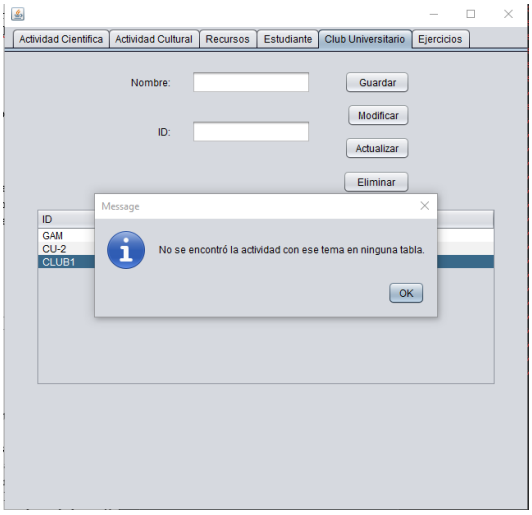
```java
public class archClubUniversitario implements Serializable{
    private final String rutaArchivo = "clubes.dat";
    private ClubUniversitario[] usuarios = new ClubUniversitario[999]; // Arreglo fijo
    private int count = 0;

    public void cargar() {
        try (ObjectInputStream ois = new ObjectInputStream(new FileInputStream(rutaArchivo))) {
            usuarios = (ClubUniversitario[]) ois.readObject();
            count = ois.readInt();
        } catch (FileNotFoundException e) {
            System.out.println("Archivo no encontrado, se creará al guardar.");
        } catch (IOException | ClassNotFoundException e) {
            e.printStackTrace();
        }
    }

    public void guardar() {
        try (ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(rutaArchivo))) {
            oos.writeObject(usuarios);
            oos.writeInt(count);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    public void guardar(ClubUniversitario[] clubes) {
        try (ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(rutaArchivo))) {
            oos.writeObject(clubes);
            oos.flush();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

2. AppendableObjectOutputStream.java (Extensión personalizada)
   Permite añadir objetos a un archivo existente sin corromperlo:

```java
public class AppendableObjectOutputStream extends ObjectOutputStream {

    public AppendableObjectOutputStream(OutputStream out) throws IOException {
        super(out);
    }

    @Override
    protected void writeStreamHeader() throws IOException {
        reset(); // No escribe la cabecera nuevamente
    }
}
```

VII. PRUEBAS Y VALIDACIÓN





VIII. CONCLUSIONES

El desarrollo del Sistema de Administración Universitaria de Actividades Culturales y Científicas representó una experiencia integral que permitió consolidar los conceptos fundamentales de la Programación Orientada a Objetos mediante la transformación de una problemática real de los entornos universitarios en una solución tecnológica estructurada y funcional.

La implementación demostró dominio efectivo de los principios fundamentales de POO a través de jerarquías bien estructuradas. La clase abstracta Usuario sirvió como base para Estudiante y Administrador, compartiendo atributos comunes mientras que cada subclase incorporó características específicas. De manera similar, la clase Actividad funcionó como clase padre y abstracta para las subclases Cultural y Científica, permitiendo que cada tipo implementara su propia lógica de cálculo de duración. El polimorfismo se dio mediante la sobrescritura del método mostrar() en todas las clases, facilitando interfaces uniformes con flexibilidad contextual. El encapsulamiento se aplicó protegiendo los atributos internos mediante modificadores de acceso apropiados, mientras la abstracción se logró a través de clases abstractas que definieron contratos comunes.

La implementación de la clase genérica GestorRecurso demostró la comprensión de la programación genérica en Java. Esta solución permitió manejar recursos de cualquier tipo de manera flexible y reutilizable, proporcionando type safety en tiempo de compilación y previniendo errores comunes relacionados con el manejo de tipos de datos incorrectos.

La decisión de implementar persistencia mediante serialización en archivos binarios (.dat) demostró comprensión práctica de lo necesario en el proyecto. Esta elección resultó acertada para un sistema académico de mediana escala, manteniendo la integridad de las relaciones entre objetos sin la complejidad adicional de una base de datos relacional. La implementación de la clase AppendableObjectOutputStream mostró creativamente la técnica al resolver el desafío específico de añadir objetos a archivos existentes sin corromper la estructura de datos.

El sistema de validaciones implementado, incluyendo límites como máximo siete actividades por estudiante y cien participantes por actividad, reflejó comprensión

profunda de las reglas de administración de dominios universitarios. Estas restricciones no solo previenen errores técnicos sino que abordan problemas reales como el desbalance en la participación estudiantil, demostrando que el desarrollo efectivo requiere competencia técnica y comprensión del contexto de los usuarios finales.

El proyecto se distingue por resolver problemáticas genuinas de entornos universitarios, transformando procesos manuales propensos a errores en sistemas automatizados confiables. La eliminación del manejo manual mediante hojas de cálculo, la automatización en la generación de certificados con identificadores únicos, y la centralización de información representan soluciones concretas a desafíos cotidianos. La implementación de un sistema de trazabilidad para certificados con validación automática abordó directamente las preocupaciones sobre falsificación identificadas en el análisis del problema.

Es importante en este proyecto su capacidad para ilustrar cómo los conceptos teóricos de la programación orientada a objetos se materializan en soluciones prácticas que impactan positivamente en comunidades reales. La experiencia obtenida en el diseño de jerarquías de clases, implementación de relaciones entre objetos, y creación de sistemas de persistencia proporcionó una base sólida para futuros desarrollos más complejos, representando un logro significativo en el proceso de formación como desarrolladores competentes preparados para enfrentar más posibles desafíos técnicos.

IX. DISTRIBUCIÓN DE ROLES DE EQUIPO

Rol/ Integrante	Responsabilidad principal
Jair Arias Flores	Código fuente de clases
Alejandra Palmenia Bravo Iriarte	Diagramas UML y documentación
Gabriel Zeus Cutile Lopez	Código fuente de clases
Alejandro Javier Farrachol Alcocer	Diagramas UML y documentación
Víctor Alberto Machaca Calle	Base de datos y Persistencia de datos

X. ANEXOS

https://github.com/bogas08/proyecto121.temporada.git