

Preyecto

Bimestral – POO

Integrantes

- JUAN TACURI
- ANTHONY VICENTE



EQUIPO

DE TRABAJO

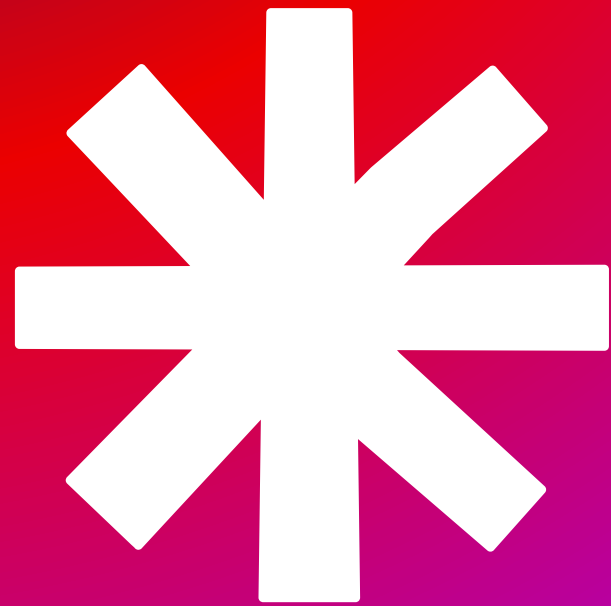
Sistema de gestión de admisiones en UTPL



Introduccion

La Universidad Técnica Particular de Loja (UTPL) ofrece 28 carreras presenciales para el período octubre 2024–febrero 2025, lo que implica un proceso complejo de inscripciones, exámenes y asignación de cupos. Para apoyar la toma de decisiones y optimizar la gestión, se desarrolla un prototipo de sistema de admisiones en Java que simula todo el flujo: registro de postulantes, aplicación de exámenes de admisión o diagnóstico, cálculo de puntos de mérito y asignación de plazas según puntajes y criterios adicionales. Este proyecto académico permite validar la lógica de negocio, garantizar la correcta priorización de aspirantes y ofrecer reportes estadísticos antes de implementar un sistema real.

Sistema de gestión de admisiones en UTPL



OBJETIVO

- Implementar un modelo de dominio (UML) que refleje postulantes, carreras, exámenes y resultados.
- Desarrollar la aplicación en Java, respetando MVC y usando serialización (.dat).
- Permitir registrar postulantes, asignar exámenes (admisión/diagnóstico), calcular méritos y asignar cupos.
- Generar reportes de estadísticas sobre ocupación de cupos y rechazados.

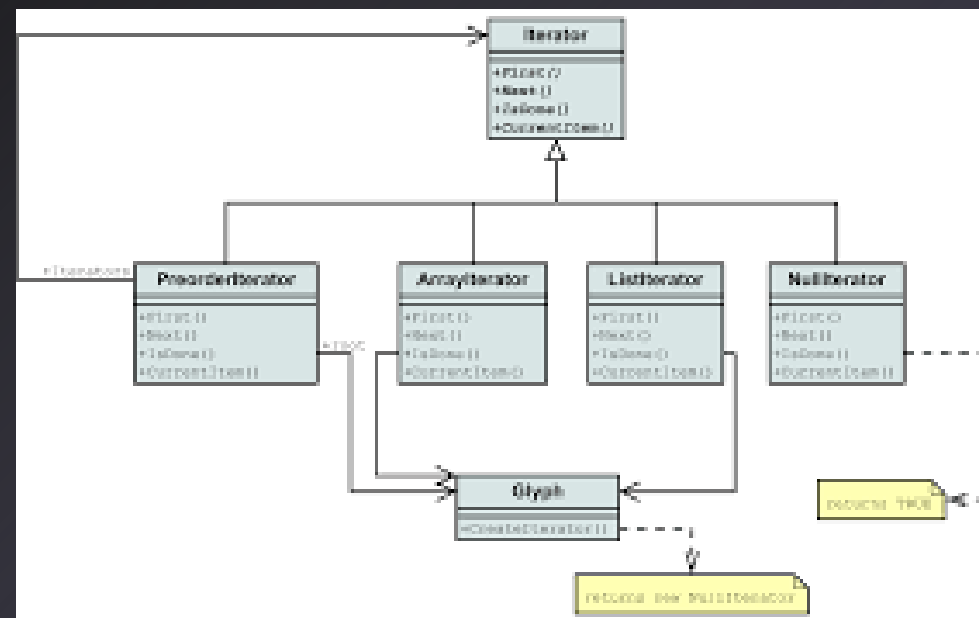


DIAGRAMA UML

Un diagrama UML es una forma de visualizar sistemas y software utilizando el Lenguaje Unificado de Modelado (UML). Los ingenieros de software crean diagramas UML online para comprender los diseños, la arquitectura del código y la implementación propuesta de sistemas de software complejos.

DIA-UML-MVC

Nombres de clases

M:

- Postulante
- Carrera
- Examen
- ExamenAdmision
- ExamenDiagnostico
- EstadoPostulacion
- Merito
- TipoExamen

C:

- AdmisionController

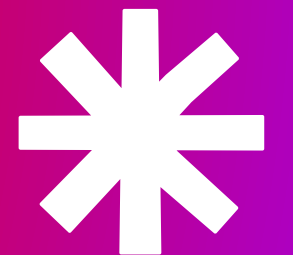
V:

- Main

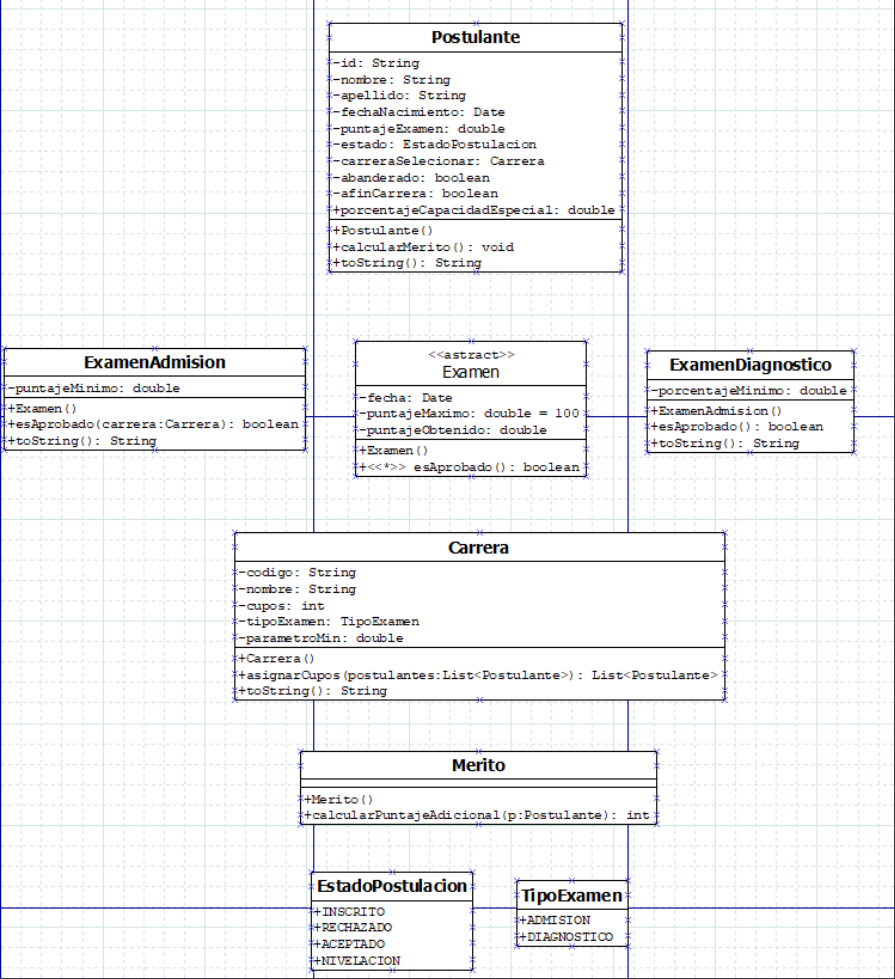
- M (Modelo): Entidades de dominio y lógica de negocio (Postulante, Carrera, Examen, etc.).
- C (Controlador): Orquesta la aplicación y llama al modelo (AdmisionController).
- V (Vista): Punto de entrada y manejo de consola (MainApp).

UML

MODELADO



DIAGRAMA



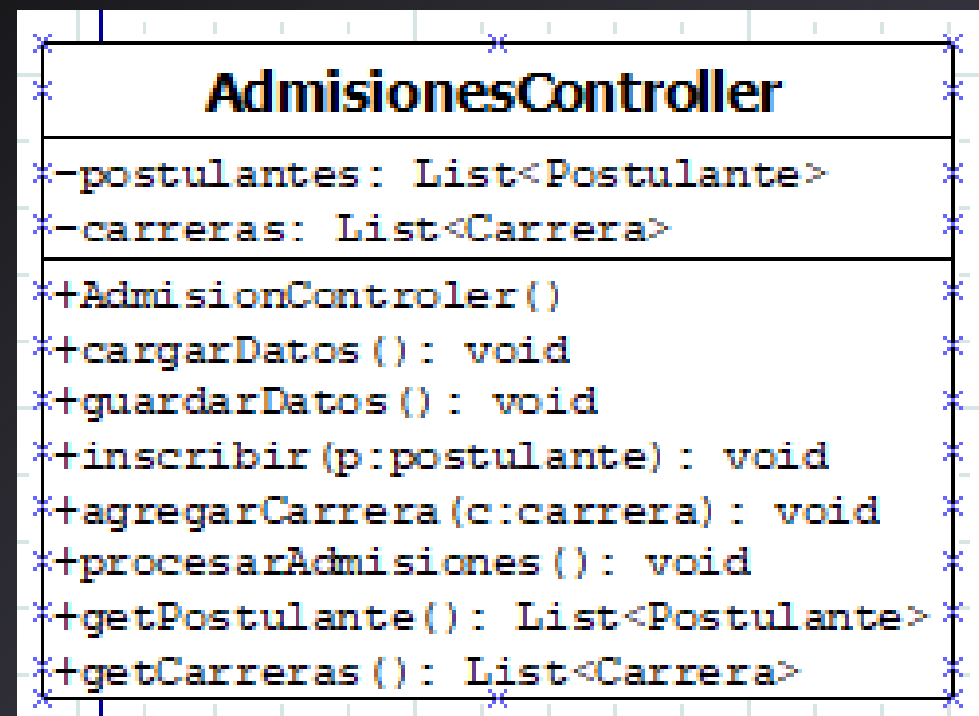
M:

- Postulante
- Carrera
- Examen
- ExamenAdmision
- ExamenDiagnostico
- EstadoPostulacion
- Merito
- TipoExamen

UML
MODELADO



DIAGRAMA

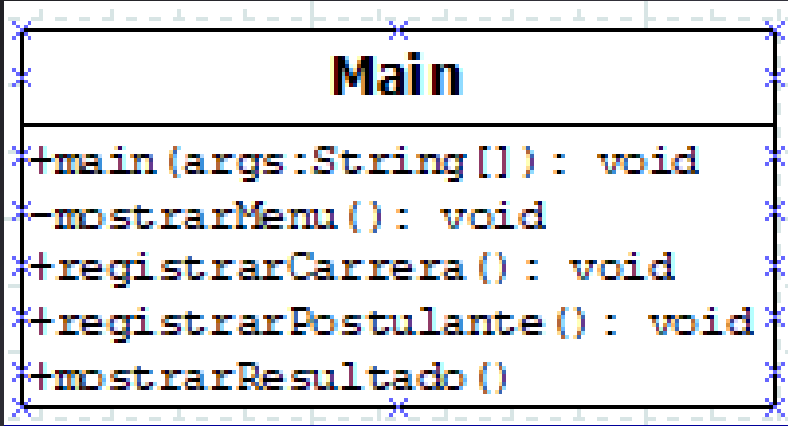


C:
• AdmisionController

UML
MODELADO



DIAGRAMA



V:
• Main

UML
MODELADO

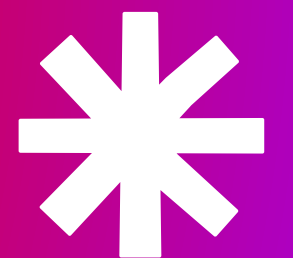


DIAGRAMA

1. Usuario (consola)
2. ↓
3. MainApp (V)
4. ↓ Llama a
5. AdmissionController (C)
6. ↓ Usa / modifica
7. Clases de M
8. ↓ Puede invocar
9. GestorDatos (M)
10. ↔ Archivos .dat

- El usuario ingresa datos desde consola → Ejecutor.
- Ejecutor invoca a AdmissionController para procesar la acción solicitada.
- AdmissionController lee/actualiza objetos de las clases MC.
- Si es necesario, llama a GestorDatos para guardar/cargar en discos (.dat).
- Luego, Ejecutor recibe la respuesta (por ejemplo, lista de estadísticas) y la muestra.

UML
MODELADO





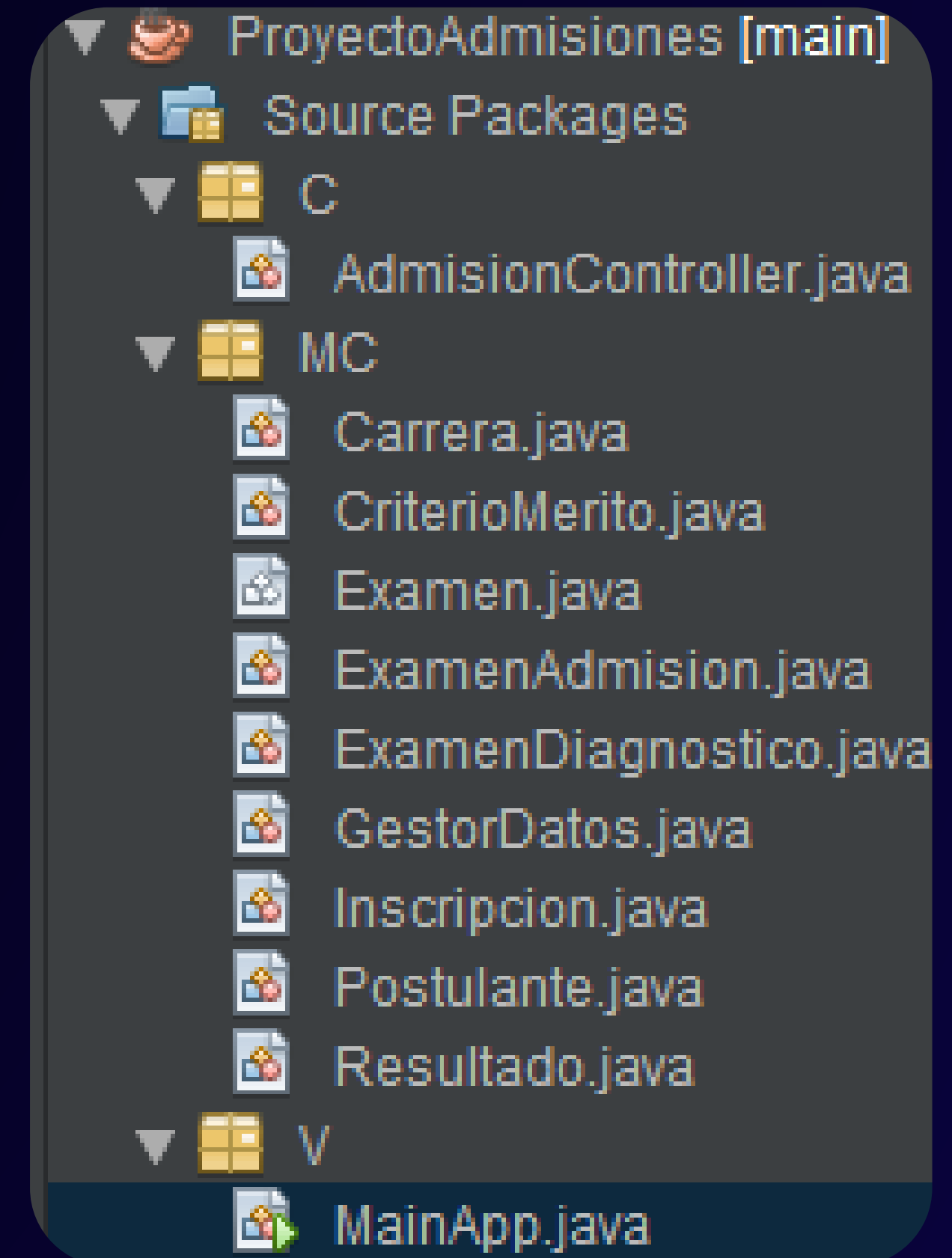
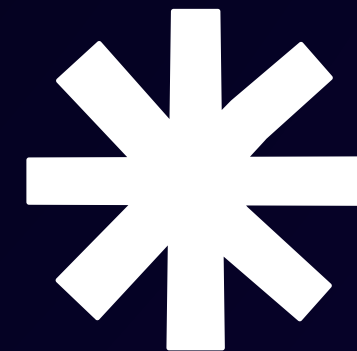
CODIGO JAVA

Java es un lenguaje de programación de alto nivel, versátil y ampliamente utilizado, especialmente en el desarrollo de aplicaciones empresariales y web, así como en la creación de software de escritorio y dispositivos móviles. Es conocido por su carácter multiplataforma, orientado a objetos y centrado en la red

.JAVA-Serializacion-CODIGO

- MC (Modelo)
 - Clases: Postulante, Carrera, Examen (+ subclases), CriterioMerito, Inscripcion, Resultado, GestorDatos.
- C (Controlador)
 - Clase: AdmissionController (gestiona lógica de admisiones).
- V (Vista/Ejecutor)
 - Clase: MainApp (maneja entrada/salida por consola y despliega menú).

Estructura del Proyecto Java



M – Clase Postulante

- Implementa Serializable para guardarse en postulantes.dat.
- Atributos clave:
 - cedula, nombre, apellido, fechaNacimiento → datos personales.
 - carreraPostulada → referencia a la carrera elegida.
 - puntajeExamen y puntajeAdicional → puntajes de admisión/diagnóstico y mérito.
 - Banderas de mérito: esAbanderado, afinidadBachillerato, porcentajeCapacidadEspecial.
 - tipoExamen y aprobado → para saber si rindió y pasó su examen.
- Constructores:
- Vacío (para deserialización y creación genérica).
- Completo (inicializa todos los atributos de un postulante).
- Método principal:
- calcularPuntajeAdicional(): suma 5 pts si es abanderado, 2 si hay afinidad, 3 si capac. esp. ≥35%.

```
1 public class Postulante implements Serializable {
2     // Atributos
3     private String cedula;
4     private String nombre;
5     private String apellido;
6     private LocalDate fechaNacimiento;
7     private Carrera carreraPostulada;
8     private double puntajeExamen;
9     private int puntajeAdicional;
10    private String tipoExamen; // "ADMISION" o "DIAGNOSTICO"
11    private boolean esAbanderado;
12    private boolean afinidadBachillerato;
13    private double porcentajeCapacidadEspecial;
14    private boolean aprobado;
15
16    // Constructor vacío
17    public Postulante() { ... }
18
19    // Constructor con todos los atributos
20    public Postulante(String cedula, String nombre, String apellido,
21                      LocalDate fechaNacimiento, Carrera carreraPostulada,
22                      double puntajeExamen, int puntajeAdicional,
23                      String tipoExamen, boolean esAbanderado,
24                      boolean afinidadBachillerato,
25                      double porcentajeCapacidadEspecial, boolean aprobado) { ... }
26
27    // Getters y setters (uno de ejemplo)
28    public String getCedula() { return cedula; }
29    public void setCedula(String cedula) { this.cedula = cedula; }
30
31    // Método clave: calcula puntos de mérito
32    public void calcularPuntajeAdicional() {
33        int total = 0;
34        if (esAbanderado) total += 5;
35        if (afinidadBachillerato) total += 2;
36        if (porcentajeCapacidadEspecial >= 35.0) total += 3;
37        this.puntajeAdicional = total;
38    }
39
40    @Override
41    public String toString() { /* muestra cedula, nombre, puntajes, etc. */ }
42 }
```

MC – Clase Carrera

```
1 public class Carrera implements Serializable {
2     // Atributos
3     private String idCarrera;
4     private String nombre;
5     private int cuposRegulares;
6     private double puntajeMinimoAdmi;
7     private double umbralDiagnostico;
8     private boolean requiereDiagnostico;
9     private ArrayList<Postulante> postulantes;
10
11     // Constructor vacío
12     public Carrera() {
13         this.idCarrera = "";
14         this.nombre = "";
15         this.cuposRegulares = 0;
16         this.puntajeMinimoAdmi = 0.0;
17         this.umbralDiagnostico = 0.0;
18         this.requiereDiagnostico = false;
19         this.postulantes = new ArrayList<>();
20     }
21
22     // Constructor con todos los atributos
23     public Carrera(String idCarrera, String nombre, int cuposRegulares,
24         double puntajeMinimoAdmi, double umbralDiagnostico,
25         boolean requiereDiagnostico, ArrayList<Postulante> postulantes) {
26         this.idCarrera = idCarrera;
27         this.nombre = nombre;
28         this.cuposRegulares = cuposRegulares;
29         this.puntajeMinimoAdmi = puntajeMinimoAdmi;
30         this.umbralDiagnostico = umbralDiagnostico;
31         this.requiereDiagnostico = requiereDiagnostico;
32         this.postulantes = postulantes != null ? postulantes : new ArrayList<>();
33     }
34
35     // Getters y setters (un ejemplo)
36     public String getIdCarrera() { return idCarrera; }
37     public void setIdCarrera(String idCarrera) { this.idCarrera = idCarrera; }
38     // ... otros getters/setters ...
39
40     // Método clave: agrega un postulante a la lista
41     public void agregarPostulante(Postulante p) {
42         if (p != null) {
43             postulantes.add(p);
44         }
45     }
46
47     // Devuelve cuántos postulantes hay inscritos
48     public int getCantidadPostulantes() {
49         return postulantes.size();
50     }
51
52     @Override
53     public String toString() {
54         return "Carrera{" +
55             "idCarrera='" + idCarrera + '\'' +
56             ", nombre='" + nombre + '\'' +
57             ", cuposRegulares=" + cuposRegulares +
58             ", puntajeMinimoAdmi=" + puntajeMinimoAdmi +
59             ", umbralDiagnostico=" + umbralDiagnostico +
60             ", requiereDiagnostico=" + requiereDiagnostico +
61             ", postulantesInscritos=" + postulantes.size() +
62             '}';
63     }
64 }
```

- Implementa Serializable para guardarse en carreras.dat.
- Atributos clave:
 - idCarrera, nombre → identificador y nombre legible de la carrera.
 - cuposRegulares → número de plazas disponibles.
 - puntajeMinimoAdmi → puntaje mínimo requerido en el examen de admisión.
 - umbralDiagnostico → porcentaje mínimo en el examen diagnóstico para evitar nivelación.
 - requiereDiagnostico → si esta carrera usa examen diagnóstico en lugar de puntaje mínimo.
 - postulantes → lista de Postulante inscrito a esta carrera.
- Constructores:
 - Vacío: inicializa valores por defecto y lista vacía de postulantes.
 - Completo: recibe todos los parámetros, incluida la lista inicial (puede partir vacía).
- Métodos principales:
 - agregarPostulante(Postulante p): añade un aspirante a la lista interna.
 - getCantidadPostulantes(): devuelve el tamaño de la lista de postulantes.
 - toString(): muestra, en una línea, todos los datos esenciales (ID, nombre, cupos, puntajes y cantidad de inscritos).

MC – Examen y Subclases

Examen (abstracta):

- Atributos: idExamen (String), fechaExamen (LocalDate), puntajeObtenido (double).
- Incluye constructor vacío y completo.
- Define el método abstracto calcularAprobado(double requisito) que las subclases deben implementar.
- Implementa toString() para mostrar datos en consola.

ExamenAdmision:

- Hereda de Examen.
- Atributo adicional: puntajeMinimo (double).
- En calcularAprobado(...), compara puntajeObtenido >= puntajeMinimo.
- El parámetro requisito del método se ignora porque usa su propio puntajeMinimo.

ExamenDiagnostico:

- Hereda de Examen.
- Atributo adicional: umbralNivelacion (double).
- En calcularAprobado(...), compara puntajeObtenido >= umbralNivelacion.
- Marca “aprobado” si alcanza el umbral; de lo contrario, el postulante requiere nivelación.

```
1 public abstract class Examen implements Serializable {
2     private String idExamen;
3     private LocalDate fechaExamen;
4     private double puntajeObtenido;
5
6     public Examen() { /*...*/ }
7     public Examen(String idExamen, LocalDate fechaExamen, double puntajeObtenido) {
8         this.idExamen = idExamen;
9         this.fechaExamen = fechaExamen;
10        this.puntajeObtenido = puntajeObtenido;
11    }
12
13    public String getIdExamen() { return idExamen; }
14    public void setIdExamen(String idExamen) { this.idExamen = idExamen; }
15    public LocalDate getFechaExamen() { return fechaExamen; }
16    public void setFechaExamen(LocalDate fechaExamen) { this.fechaExamen = fechaExamen; }
17    public double getPuntajeObtenido() { return puntajeObtenido; }
18    public void setPuntajeObtenido(double puntajeObtenido) { this.puntajeObtenido = puntajeObtenido; }
19
20    // Método abstracto que cada subclase implementa
21    public abstract boolean calcularAprobado(double requisito);
22
23    @Override
24    public String toString() { /*...*/ }
25 }
```

```
1 public class ExamenAdmision extends Examen {
2     private double puntajeMinimo;
3
4     public ExamenAdmision() { super(); this.puntajeMinimo = 0.0; }
5     public ExamenAdmision(String idExamen, LocalDate fechaExamen,
6                             double puntajeObtenido, double puntajeMinimo) {
7         super(idExamen, fechaExamen, puntajeObtenido);
8         this.puntajeMinimo = puntajeMinimo;
9     }
10
11    public double getPuntajeMinimo() { return puntajeMinimo; }
12    public void setPuntajeMinimo(double puntajeMinimo) { this.puntajeMinimo = puntajeMinimo; }
13
14    @Override
15    public boolean calcularAprobado(double requisito) {
16        // Para admisión se compara con puntajeMinimo interno
17        return getPuntajeObtenido() >= puntajeMinimo;
18    }
19
20    @Override
21    public String toString() { /*...*/ }
22 }
```

```
1 public class ExamenDiagnostico extends Examen {
2     private double umbralNivelacion;
3
4     public ExamenDiagnostico() { super(); this.umbralNivelacion = 0.0; }
5     public ExamenDiagnostico(String idExamen, LocalDate fechaExamen,
6                             double puntajeObtenido, double umbralNivelacion) {
7         super(idExamen, fechaExamen, puntajeObtenido);
8         this.umbralNivelacion = umbralNivelacion;
9     }
10
11    public double getUmbralNivelacion() { return umbralNivelacion; }
12    public void setUmbralNivelacion(double umbralNivelacion) { this.umbralNivelacion = umbralNivelacion; }
13
14    @Override
15    public boolean calcularAprobado(double requisito) {
16        // Para diagnóstico se compara con umbralNivelacion interno
17        return getPuntajeObtenido() >= umbralNivelacion;
18    }
19
20    @Override
21    public String toString() { /*...*/ }
22 }
```

MC – Clase Inscripcion

- Implementa Serializable para guardarse en inscripciones.dat.
- Atributos clave:
 - idInscripcion → código único de la inscripción.
 - postulante → objeto Postulante que se inscribe.
 - carrera → objeto Carrera a la que se inscribe.
 - fechaInscripcion → LocalDate en que se realizó la inscripción.
- Constructores:
 - Vacío: inicializa valores por defecto (cadenas vacías y null).
 - Completo: recibe id, referencia a postulante, referencia a carrera y fecha.
- Métodos:
 - Getters y setters para cada atributo.
 - toString(): imprime id de inscripción, cédula del postulante, id de carrera y fecha, útil al listar en consola.

```
1 public class Inscripcion implements Serializable {
2     // Atributos
3     private String idInscripcion;
4     private Postulante postulante;
5     private Carrera carrera;
6     private LocalDate fechaInscripcion;
7
8     // Constructor vacío
9     public Inscripcion() {
10         this.idInscripcion = "";
11         this.postulante = null;
12         this.carrera = null;
13         this.fechaInscripcion = null;
14     }
15
16     // Constructor con todos los atributos
17     public Inscripcion(String idInscripcion, Postulante postulante,
18         Carrera carrera, LocalDate fechaInscripcion) {
19         this.idInscripcion = idInscripcion;
20         this.postulante = postulante;
21         this.carrera = carrera;
22         this.fechaInscripcion = fechaInscripcion;
23     }
24
25     // Getters y setters
26     public String getIdInscripcion() { return idInscripcion; }
27     public void setIdInscripcion(String idInscripcion) {
28         this.idInscripcion = idInscripcion;
29     }
30     public Postulante getPostulante() { return postulante; }
31     public void setPostulante(Postulante postulante) {
32         this.postulante = postulante;
33     }
34     public Carrera getCarrera() { return carrera; }
35     public void setCarrera(Carrera carrera) {
36         this.carrera = carrera;
37     }
38     public LocalDate getFechaInscripcion() { return fechaInscripcion; }
39     public void setFechaInscripcion(LocalDate fechaInscripcion) {
40         this.fechaInscripcion = fechaInscripcion;
41     }
42
43     @Override
44     public String toString() {
45         return "Inscripcion{" +
46             "id='" + idInscripcion + '\'' +
47             ", postulante=" + (postulante != null ? postulante.getCedula() : "-") +
48             ", carrera=" + (carrera != null ? carrera.getIdCarrera() : "-") +
49             ", fecha=" + fechaInscripcion +
50             '}' ;
51     }
52 }
```



```

public class GestorDatos {
    private String rutaArchivoPostulantes;
    private String rutaArchivoCarreras;
    private String rutaArchivoInscripciones;
    private String rutaArchivoResultados;

    // Constructor vacío (rutas por defecto)
    public GestorDatos() {
        this.rutaArchivoPostulantes = "postulantes.dat";
        this.rutaArchivoCarreras = "carreras.dat";
        this.rutaArchivoInscripciones = "inscripciones.dat";
        this.rutaArchivoResultados = "resultados.dat";
    }

    // Constructor completo (rutas personalizadas)
    public GestorDatos(String rutaP, String rutaC, String rutaI, String rutaR) {
        this.rutaArchivoPostulantes = rutaP;
        this.rutaArchivoCarreras = rutaC;
        this.rutaArchivoInscripciones = rutaI;
        this.rutaArchivoResultados = rutaR;
    }

    // Inicializa archivos (si no existen, los crea)
    public void inicializarDatos() {
        try {
            File f1 = new File(rutaArchivoPostulantes);
            if (!f1.exists()) {
                f1.createNewFile();
                guardarPostulantes(new ArrayList<>());
            }
            File f2 = new File(rutaArchivoCarreras);
            if (!f2.exists()) {
                f2.createNewFile();
                guardarCarreras(new ArrayList<>());
            }
            File f3 = new File(rutaArchivoInscripciones);
            if (!f3.exists()) {
                f3.createNewFile();
                guardarInscripciones(new ArrayList<>());
            }
            File f4 = new File(rutaArchivoResultados);
            if (!f4.exists()) {
                f4.createNewFile();
                guardarResultados(new ArrayList<>());
            }
        } catch (IOException e) {
            System.err.println("Error inicializando archivos: " + e.getMessage());
        }
    }

    // Guarda lista de postulantes en postulantes.dat
    public void guardarPostulantes(ArrayList<Postulante> lista) {
        try (ObjectOutputStream oos = new ObjectOutputStream(
            new FileOutputStream(rutaArchivoPostulantes))) {
            oos.writeObject(lista);
        } catch (IOException e) {
            System.err.println("Error al guardar postulantes: " + e.getMessage());
        }
    }

    // Carga lista de postulantes desde postulantes.dat
    @SuppressWarnings("unchecked")
    public ArrayList<Postulante> cargarPostulantes() {
        ArrayList<Postulante> lista = new ArrayList<>();
        File archivo = new File(rutaArchivoPostulantes);
        if (!archivo.exists() || archivo.length() == 0) {
            return lista;
        }
        try (ObjectInputStream ois = new ObjectInputStream(
            new FileInputStream(rutaArchivoPostulantes))) {
            lista = (ArrayList<Postulante>) ois.readObject();
        } catch (IOException | ClassNotFoundException e) {
        }
    }
}

```

MC – Clase GestorDatos

- Su propósito: serializar y deserializar las listas de objetos en archivos .dat.
- Constructores:
 - Vacío: asigna nombres de archivos por defecto (postulantes.dat, carreras.dat, inscripciones.dat, resultados.dat).
 - Completo: permite personalizar rutas (e.g., "data/postulantes.dat").
- Método inicializarDatos():
 - Verifica si cada archivo existe.
 - Si no existe, lo crea y escribe una lista vacía (para evitar EOFException).
- Métodos guardarX y cargarX:
 - guardarX(ArrayList<Clase> lista): abre ObjectOutputStream para sobrescribir el archivo con la lista actual.
 - cargarX(): si el archivo existe y no está vacío, abre ObjectInputStream, lee el ArrayList<Clase> guardado y lo retorna; si está vacío o no existe, devuelve lista vacía.
 - Estos métodos existen para Postulante, Carrera, Inscripcion y Resultado.



```
public class AdmissionController {
    // Atributos
    private ArrayList<Carrera> listaCarreras;
    private ArrayList<Postulante> listaPostulantes;
    private ArrayList<Inscripcion> listaInscripciones;
    private ArrayList<Resultado> listaResultados;
    private GestorDatos gestor;

    // Constructor vacío
    public AdmissionController() {
        this.gestor = new GestorDatos();
        this.listaCarreras = new ArrayList<>();
        this.listaPostulantes = new ArrayList<>();
        this.listaInscripciones = new ArrayList<>();
        this.listaResultados = new ArrayList<>();
    }

    // Carga datos iniciales
    public void cargarDatosIniciales() {
        gestor.inicializarDatos();
        listaPostulantes = gestor.cargarPostulantes();
        listaCarreras = gestor.cargarCarreras();
        listaInscripciones = gestor.cargarInscripciones();
        listaResultados = gestor.cargarResultados();
    }

    // Guarda todas las colecciones
    public void guardarDatos() {
        gestor.guardarPostulantes(listaPostulantes);
        gestor.guardarCarreras(listaCarreras);
        gestor.guardarInscripciones(listaInscripciones);
        gestor.guardarResultados(listaResultados);
    }

    // Registrar un postulante
    public void registrarPostulante(Postulante p) {
        if (p != null) {
            listaPostulantes.add(p);
            guardarDatos();
        }
    }

    // Inscribir a carrera
    public void inscribirPostulanteACarrera(String cedulaP, String idCarrera, LocalDate fecha) {
        // Busca postulante y carrera, crea Inscripcion, añade a colecciones y guarda
        // ... (lógica resumida) ...
    }

    // Asignar examen a inscripción
    public void asignarExamenAInscripcion(String idInscripcion, Examen examen) {
        // Busca Inscripcion, asigna puntaje al Postulante, calcula "aprobado" y puntajeAdicional
        // ... (lógica resumida) ...
        guardarDatos();
    }

    // Procesar admisiones por carrera
    public void procesarAdmisionesPorCarrera(String idCarrera) {
        // Separa postulantes en ADMISION y DIAGNOSTICO,
        // ordena, asigna cupos, crea Resultados y guarda
        // ... (lógica resumida) ...
        guardarDatos();
    }

    // Generar estadísticas
    public void generarEstadisticas() {
        // Calcula ocupación < 50% y rechazos, imprime en consola
    }
}
```

C – Clase AdmissionController

- Responsabilidad principal: orquestar la lógica de negocio y coordinar modelo y persistencia.
- Atributos:
 - listaCarreras, listaPostulantes, listaInscripciones, listaResultados (todas ArrayList<>).
 - gestor (instancia de GestorDatos).
- Métodos clave:
 - cargarDatosIniciales():
 - guardarDatos():
 - registrarPostulante(Postulante p):
 - inscribirPostulanteACarrera(String cedulaP, String idCarrera, LocalDate fecha):
 - asignarExamenAInscripcion(String idInscripcion, Examen examen):
 - procesarAdmisionesPorCarrera(String idCarrera):
 - generarEstadisticas():


```

public class MainApp {
    private AdmissionController sistema;
    private Scanner sc;
    private DateTimeFormatter fmt;

    // Constructor vacío
    public MainApp() {
        this.sistema = new AdmissionController();
        this.sc = new Scanner(System.in);
        this.fmt = DateTimeFormatter.ofPattern("yyyy-MM-dd");
    }

    // main: punto de entrada
    public static void main(String[] args) {
        MainApp app = new MainApp();
        app.sistema.cargarDatosIniciales();
        app.mostrarMenuPrincipal();
    }

    // Muestra el menú principal y procesa opciones
    public void mostrarMenuPrincipal() {
        int opcion = -1;
        do {
            System.out.println("\n=== SISTEMA DE ADMISIONES UTPL ===");
            System.out.println("1) Registrar nuevo postulante");
            System.out.println("2) Listar carreras disponibles");
            System.out.println("3) Inscribir postulante a carrera");
            System.out.println("4) Registrar resultado de examen");
            System.out.println("5) Procesar admisiones para una carrera");
            System.out.println("6) Mostrar reportes/estadísticas");
            System.out.println("7) Salir");
            System.out.print("Seleccione [1-7]: ");
            opcion = leerEntero("");

            switch (opcion) {
                case 1: opcionRegistrarPostulante(); break;
                case 2: opcionListarCarreras(); break;
                case 3: opcionInscribirACarrera(); break;
                case 4: opcionRegistrarResultadoExamen(); break;
                case 5: opcionProcesarAdmisiones(); break;
                case 6: opcionMostrarEstadisticas(); break;
                case 7: System.out.println("Saliendo..."); break;
                default: System.out.println("Opción inválida.");
            }
        } while (opcion != 7);
    }

    // Métodos auxiliares para cada opción (resumido)
    private void opcionRegistrarPostulante() { /*...*/ }
    private void opcionListarCarreras() { /*...*/ }
    private void opcionInscribirACarrera() { /*...*/ }
    private void opcionRegistrarResultadoExamen() { /*...*/ }
    private void opcionProcesarAdmisiones() { /*...*/ }
    private void opcionMostrarEstadisticas() { /*...*/ }

    // Lectura desde consola
    private String leerTexto(String prompt) { /*...*/ }
    private int leerEntero(String prompt) { /*...*/ }
    private double leerDouble(String prompt) { /*...*/ }
}

```

V – Clase MainApp

- Propósito: único punto donde ocurre interacción con el usuario (consola).
- Atributos:
 - AdmissionController sistema → instancia del controlador.
 - Scanner sc → para leer datos del usuario.
 - DateTimeFormatter fmt → formatea LocalDate como "yyyy-MM-dd".
- main(String[] args):
 - a. Crea MainApp.
 - b. Llama a sistema.cargarDatosIniciales() para cargar/crear archivos .dat.
 - c. Invoca mostrarMenuPrincipal().
- mostrarMenuPrincipal():
- Opciones principales (resumen):
 1. Registrar nuevo postulante → opcionRegistrarPostulante().
 2. Listar carreras → opcionListarCarreras().
 3. Inscribir postulante → opcionInscribirACarrera().
 4. Registrar examen → opcionRegistrarResultadoExamen().
 5. Procesar admisiones → opcionProcesarAdmisiones().
 - Mostrar estadísticas → opcionMostrarEstadisticas().

CONSOLA *

with
NetBeans

JAVA CODIGO

run:

```
=== SISTEMA DE GESTION DE ADMISIONES UTPL ===  
=== BY JUAN TACURI / ANTHONY VICENTE ===  
1) Registrar nuevo postulante  
2) Listar carreras disponibles  
3) Inscribir postulante a una carrera  
4) Registrar resultado de examen  
5) Procesar admisiones para una carrera  
6) Mostrar reportes/estadísticas  
7) Salir  
Selecciona una opción [1-7]:
```


Thank You

***ING. NOS PONE 10 0
VALAMOS EN 10***

