

Documento de diseño: Proyecto 1

Daniel Vergara

Diego Mahecha

Santiago Quiroz Pintor

Samuel Molina

Universidad de Los Andes

Diseño y Programación Orientado a Objetos

23 de Octubre de 2024

1). Introducción: En el siguiente documento se hablará del desarrollo que se aplicó para la implementación del sistema, a través del diagrama UML y los distintos elementos del análisis para el caso específico de las actividades y los learnings paths para el profesor y el estudiante.

2). Decisiones de diseño:

En primer lugar se hablará del paquete users donde se mantuvo la clase abstracta de donde heredarán Professor y Student a su vez cada uno de ellos mantiene una implementación de la siguiente manera para lograr cumplir con los requerimientos funcionales establecidos en la primera entrega de Análisis.

* Paquete user

Clase User

```
package Users;

import java.util.HashMap;

public abstract class User {

    public String username;
    private String password;
    public static UsersVerifier crear;

    public User(String username, String password) {
        this.username = username;
        this.password = password;
        crear.usersDataBase.put(username, password);
    }

    public String getUsername() {
        return username;
    }
    public void setUsername(String username) {
        this.username = username;
    }
    public String getPassword() {
        return password;
    }
    public void setPassword(String password) {
        this.password = password;
    }
}
```

En donde se mantuvo el username y la password a su vez se intentó crear un users verifier pero al final se decidió cambiar por una clase controller que se encargará de la creación de User.

Clase Professor – Cambio de nombre de teacher a professor

```

import Programa.Activity;
import Programa.LearningPath;

class Teacher extends User {
    private HashMap<String, LearningPath> createdLearningPaths;
    private ArrayList<Activity> createdActivities;

    public Teacher(String username, String password) {
        super(username, password);
        this.createdLearningPaths = new HashMap<>();
        this.createdActivities = new ArrayList<>();
    }

    public LearningPath createLearningPath(String title, String description) {
        LearningPath path = new LearningPath(title, description);
        createdLearningPaths.put(title, path);
        return path;
    }

    public void deleteLearningPath(String title) {
        createdLearningPaths.remove(title);
    }

    public Activity createActivity(String title, String description) {
        Activity activity = new Activity(title, description);
        createdActivities.add(activity);
        return activity;
    }


    public void deleteActivity(Activity activity) {
        createdActivities.remove(activity);
    }

    public void addActivityToLearningPath(LearningPath path, Activity activity, int pos) {
        path.addActivity(activity, pos);
    }

    public void removeActivityFromLearningPath(LearningPath path, int pos) {
        path.removeActivity(pos);
    }
}

```

En esta clase se mantuvo con la implementación dada en un primer momento adicional se añadió un hashmap de learningPaths creados con el fin de que si el professor necesita acceder a un learning path viejo pueda acceder directamente. A su vez, tiene lo mismo respecto a actividades en el arrayList de createdActivities. Complementando, se añadieron los siguientes métodos para poder cumplir con los requerimientos, en un primer lugar se tiene createLearninPath el cual a través del constructor de learningPath se llama y se crea el nuevo learning path y luego se procede a ingresar en el HashMap<> de createdLearningPaths, a su vez tambien se usa el método de remove para eliminarlo del HashMap<> de learningPaths, con la lógica de lo anterior se repite el proceso para Activity solo que cambia la estructura de ingreso que en este caso es una List<> de tipo ArrayList, a su vez se implementó añadir actividades a un learningPath a través de posición y remover bajo el mismo principio.

|  Teacher |
|--|
| <ul style="list-style-type: none"> ❑ createdLearningPaths: HashMap<LearningPath> ❑ createdActivities: ArrayList<Activity> |
| <ul style="list-style-type: none"> ● createLearningPath(title: String, description: String): void ● deleteLearningPath(path: LearningPath): void ● createActivity(title: String, description: String): LearningPath ● deleteActivity(activity: Activity): void ● addActivityToLearningPath(path: LearningPath, activity: Activity): void ● addActivityToLearningPath(path: LearningPath, activity: Activity, int: pos): void ● removeActivityFromLearningPath(int: pos): void |

Clase Student

```
import java.util.ArrayList;

public class Student extends User {
    private List<String> interests;
    private List<ProgressTracker> progressTrackers;

    public Student(String username, String password) {
        super(username, password);
        this.interests = new ArrayList<>();
        this.progressTrackers = new ArrayList<>();
    }

    public void addInterest(String interest) {
        interests.add(interest);
    }

    public void removeInterest(int index) {
        if (index >= 0 && index < interests.size()) {
            interests.remove(index);
        }
    }

    public List<LearningPath> getLearningPathsByInterest(String interest) {
        List<LearningPath> arregloReturn = new ArrayList<>();
        int num=0; //recorrido :) 1 vez nada mas
        if(num == 0) {
            for(ProgressTracker elements : progressTrackers) {
                ArrayList<LearningPath> path = elements.getAllLearninPaths();
                for(LearningPath allPaths : path) {
                    ArrayList<String> interesesPath = allPaths.interest;
                    for(String elementosInterest: interesesPath) {
                        if (elementosInterest.equals(interest) || elementosInterest.contentEquals(interest)) {
                            arregloReturn.add(allPaths);
                        }
                    }
                }
            }
            num++;
        }
        return arregloReturn;
    }
}
```

En cuanto a la clase Student se decidió agregarle un arreglo de intereses para hacer la búsqueda de learninPahts por intereses para cumplir con aquel requerimiento, a su vez se decidió que cada estudiante ha de tener un arreglo de progressTracker donde cada uno de ellos se encargará de “monitorear” el progreso de alguna actividad en específico, con lo anterior se usa la lógica de agregación y eliminación de elementos que se implementó en el paso anterior, el gran cambio radica en cuanto al método de getLearninPaths debido al recorrido que se implementa pues debido a que se recorren los progressTracker(Arreglo) una única vez para obtener así el arreglo de todos los learningPath para así obtener el arreglo de intereses, si es positivo el interes del learningPath se agregara a la lista de retorno para así devolver la respuesta de los learningPaths recomendados por interés. Con lo anterior se logra cumplir con los siguientes requerimientos.

| C Student |
|--|
| <ul style="list-style-type: none"> ❑ interests: List<String> ❑ progressTrackers: List<progressTrackers> |
| <ul style="list-style-type: none"> ● addInterest(interest: String): void ● removeInterest(interestId: int): void ● getLearningPathsByInterest(): List<LearningPath> ● registerInLearningPath(path: LearningPath): void |

* Paquete tracker: Se decidió separar los paquetes de user de las clases ProgressTracker y ActivityTracker debido a que nos permitirá sectorizar las pruebas de mejor manera a la hora de la implementación también tener clases más definidas y separadas por paquetes consideramos que será una mejor implementación.

Clase ProgressTracker

```
package tracker;

import learningpath.LearningPath;

public class ProgressTracker implements Serializable{

    /**
     *
     */
    private static final long serialVersionUID = 1L;
    private String studentUsername;
    private LearningPath learningpath;
    private LinkedList<ActivityTracker> activityTrackers;
    private Date startDate;
    private Date completionDate;
    private float progress;
    private boolean completionStatus;

    public ProgressTracker(String studentUsername, LearningPath learningpath) {

        this.studentUsername = studentUsername;
        this.learningpath = learningpath;
        for (Activity activity: learningpath.getActivities()) {
            activityTrackers.add(new ActivityTracker(activity));
        }
        this.startDate = new Date();
        this.completionDate = null;
        this.progress = 0.0f;
        this.completionStatus = false;
    }
}
```

```

public void calculateProgress() {
    int totalMandatory = 0;
    int totalMandatoryCompleted = 0;
    boolean isCompleted;
    boolean isMandatory;

    for (ActivityTracker activitytracker: this.activityTrackers) {

        isCompleted = activitytracker.getCompletionStatus().equals("Completed");
        isMandatory = activitytracker.getActivity().isMandatory();

        if (isMandatory){

            totalMandatory++;

            if (isCompleted) {
                totalMandatoryCompleted++;
            }
        }
    }
    progress = (float) totalMandatoryCompleted /totalMandatory * 100;

    if (progress == 100.0f) {
        completionStatus = true;
        completionDate = new Date();
    }
}

public void recordActivityStart(ActivityTracker activityTracker) {
    activityTracker.recordActivityStart();
}

public void recordActivityCompletion(ActivityTracker activityTracker) {
    activityTracker.recordActivityCompletion();
    calculateProgress();
}

```

En esta clase se la idea es realizar el seguimiento del estudiante en un learning path asignado, es por esto que se tiene una relación a LearningPath para vincularlo con el progreso que lleve el estudiante en ese momento. Dentro del learning path a la vez se tiene el arreglo de activityTracker el cual almacena información detallada sobre el progreso de una actividad individual. A partir de esto se busca monitorear el progreso de cada estudiante mediante los métodos de calculateProgress el cual recorre el arreglo de actividades contando el número de estas para luego sacar el porcentaje actualizado del progreso. Así mismo se tiene la opción de registrar el inicio de una actividad y la finalización de la misma mediante los métodos recordActivityStart y recordActivityCompletion provenientes de la clase activityTracker los cuales serán explicados a continuación. Para este caso solo se hace un llamado a la función para declarar los cambios.

Clase ActivityTracker

```

package tracker;

import java.io.Serializable;

public class ActivityTracker implements Serializable{

    /**
     *
     */
    private static final long serialVersionUID = 1L;
    private Activity activity;
    private int dedicatedTime;
    private String completionStatus;
    protected Date startDate;
    protected Date completionDate;

    public ActivityTracker(Activity activity) {
        this.activity = activity;
        this.dedicatedTime = 0;
        this.completionStatus = "Not started";
        this.startDate = null;
        this.completionDate = null;
    }

```

```

    public void recordActivityStart() {
        this.completionStatus = "In Progress";
        this.startDate = new Date();
    }

    public void recordActivityCompletion() {
        this.completionStatus = "Completed";
        this.completionDate = new Date();
    }

```

Esta clase a diferencia del progressTracker hace un seguimiento del progreso de una actividad dentro del learning path, es decir cada instancia de la clase está asociada a una actividad específica y guarda información sobre su estado, tiempo y las fechas de inicio y finalización. Con lo anterior, para su función se tienen los métodos nuevamente para registrar el progreso mediante el estado de la actividad actual el cual se verifica si está completada o no. En el caso de recordActivityStart se entiende como “In progress” cuando se inicia la actividad con la fecha actual. A su vez se registra cuando una actividad ha sido completada cambiando su estado y actualizando la fecha correspondiente en completionDate. Es así que se habilita a otros componentes como el progressTracker para gestionar el progreso global del learning path.

* Paquete LearninPath: se decidió que el paquete que maneje Actividad y LearningPath ya que así el manejo de las distintas actividades será más comprensible desde el mismo paquete (Ya que un LearningPath debe contener las actividades).

Clase LearningPath

```

public class LearningPath implements Serializable {
    /**
     *
     */
    private static final long serialVersionUID = 1L;
    // Attributes of the LearningPath
    private final String id; // Unique identifier of the learning path
    private String title; // Title of the learning path
    private String description; // Description of the learning path
    private LinkedList<String> objectives; // Objectives of the learning path
    private int difficultyLevel; // Difficulty level of the learning path
    private int duration; // Duration of the learning path in hours
    private float rating; // Rating of the learning path
    private LinkedList<String> tags; // Tags associated with the learning path
    private final Professor professor; // Professor associated with the learning path
    private final Date creationDate; // Date when the learning path was created
    private Date modificationDate; // Date when the learning path was last updated
    private int version; // Version of the learning path
    private LinkedList<Activity> activities; // List of activities in the learning path
    private LinkedList<ProgressTracker> progressTrackers; // List of progress trackers for students enrolled in the
    private LinkedList<String> sessions; // learning path#

    /**
     * Constructor to initialize a LearningPath object with the given parameters.
     *
     * @param id The unique identifier of the learning path.
     * @param title Title of the learning path.
     * @param description Description of the learning path.
     * @param objectives Objectives of the learning path.
     * @param difficultyLevel Difficulty level of the learning path.
     * @param tags Tags associated with the learning path.
     * @param professor Professor associated with the learning path.
     */
    public LearningPath(String id, String title, String description, LinkedList<String> objectives, int difficultyLevel,
        LinkedList<String> tags, Professor professor) {
        Generator u = Generator.getInstance();
        this.id = u.generateId("LearningPath");
        this.title = title;
        this.description = description;
        this.objectives = objectives;
        this.difficultyLevel = difficultyLevel;
        this.duration = 0;
        this.rating = 0; // Initial rating is set to 0
        this.tags = tags;
        this.professor = professor;
        this.creationDate = new Date(); // Set creation date to current date
        this.modificationDate = new Date(); // Set update date to current date
        this.version = 1; // Initial version is set to 1
        this.activities = new LinkedList<>(); // Initialize the list of activities
        this.progressTrackers = new LinkedList<>(); // Initialize the list of progress trackers
    }
}

```



```

public void addActivityInPos(Activity activity, int index) {
    activities.add(index, activity);
}

/**
 * Moves an activity within the learning path from one index to another.
 *
 * @param currentIndex the current index of the activity.
 * @param newIndex     the new index of the activity.
 */
public void moveActivity(int currentIndex, int newIndex) {
    Activity activity = activities.remove(currentIndex);
    activities.add(newIndex, activity);
}

/**
 * Removes an activity from the learning path.
 *
 * @param index the index of the activity to be removed.
 */
public void removeActivityByIndex(int index) {
    activities.remove(index);
}

// Progress tracker management methods

/**
 * Gets the list of progress trackers associated with the learning path.
 *
 * @return the list of progress trackers.
 */
public LinkedList<ProgressTracker> getProgressTrackers() {
    return progressTrackers;
}

public ProgressTracker getProgressTrackerByIndex(int index) {
    return progressTrackers.get(index);
}

public void addProgressTracker(ProgressTracker progressTracker) {
    progressTrackers.add(progressTracker);
}

/**
 * Updates the version of the learning path.
 */
public void updateVersion() {
    this.version++;
}

/**
 * Updates the modification date of the learning path to the current date.
 */
public void updateModificationDate() {
    this.modificationDate = new Date();
}
}

```

De una forma general esta clase es esencial para gestionar los diferentes cursos propuestos haciendo énfasis en las funciones respecto a los roles creados, en este caso profesor y estudiante. Los métodos de esta clase permiten agregar, eliminar y mover actividades dentro del learning path, lo que brinda flexibilidad a los profesores para ajustar la secuencia de aprendizaje según las necesidades del curso. Además de esto, esta clase se relaciona directamente con ProgressTracker, la cual realiza un seguimiento del progreso de los estudiantes inscritos en el curso. Los métodos como addActivity y removeActivityByIndex permiten gestionar las actividades añadiéndolas a la lista o removiéndolas dado un índice inicial, mientras que addProgressTracker registra a nuevos estudiantes para llevar su progreso.

* Paquete Activity

Clase Activity:

```
public abstract class Activity {  
  
    protected String id;  
    protected String title;  
    protected String description;  
    protected String objective;  
    protected int expectedDuration;  
    protected boolean mandatory;  
    protected LinkedList<ActivityTracker> activityTrackers;  
    protected LinkedList<Activity> prerequisites;  
    protected LinkedList<Activity> followUpActivities;  
  
    public Activity(String title, String description, String objective, int expectedDuration, boolean mandatory) {  
        Generator u = Generator.getInstance();  
        this.id = u.generateId("Activity");  
        this.title = title;  
        this.description = description;  
        this.objective = objective;  
        this.expectedDuration = expectedDuration;  
        this.mandatory = mandatory;  
        this.activityTrackers = new LinkedList<>();  
        this.prerequisites = new LinkedList<>();  
        this.followUpActivities = new LinkedList<>();  
    }  
}
```

```
    public void addActivityTracker(ActivityTracker at) {  
        this.activityTrackers.add(at);  
    }  
  
    public void removeActivityTrackerByIndex(int index) {  
        this.activityTrackers.remove(index);  
    }  
  
    public void addPrerequisite(Activity activity) {  
        this.prerequisites.add(activity);  
    }  
  
    public void removePrerequisiteByIndex(int index) {  
        this.prerequisites.remove(index);  
    }  
  
    public void addFollowUp(Activity activity) {  
        this.followUpActivities.add(activity);  
    }  
  
    public void removeFollowUp(int index) {  
        this.followUpActivities.remove(index);  
    }  
}
```

Esta clase al ser abstracta, tiene el fin de dejar instanciadas las variables y métodos (getters y setters) que heredan las subclases de esta. Se almacena el título, la descripción, el objetivo, la duración esperada, si es obligatorio o no, y tres listas, una con los prerequisites, otra con las actividades de seguimiento, y la última con los seguimientos.

Clase ExamActivity

```
public class ExamActivity extends Activity {  
  
    private LinkedList<OpenQuestion> openQuestions;  
    private LinkedList<MultipleOptionQuestion> MOQuestions;  
  
    public ExamActivity(String title, String description, String objective, int expectedDuration, boolean mandatory,  
        LinkedList<OpenQuestion> openQuestions, LinkedList<MultipleOptionQuestion> MOQuestions) {  
  
        super(title, description, objective, expectedDuration, mandatory);  
        this.openQuestions = openQuestions != null ? openQuestions : new LinkedList<>();  
        this.MOQuestions = MOQuestions != null ? MOQuestions : new LinkedList<>();  
  
    }  
}
```

```
    public boolean addOpenQuestion(OpenQuestion question) {  
        if (this.containsOpenQuestion(question)) {  
            System.out.println("Question already added.");  
            return false;  
        }  
  
        if (question == null) {  
            System.err.println("Question can not be null.");  
        }  
        this.openQuestions.add(question);  
        return true;  
    }  
  
    public boolean addMOQuestion(MultipleOptionQuestion q) {  
        if (this.containsMOQuestion(q)) {  
            System.out.println("Question already added.");  
            return false;  
        }  
  
        if (q == null) {  
            System.err.println("Question can not be null.");  
        }  
        this.MOQuestions.add(q);  
        return true;  
    }  
}
```

```
    public boolean removeOpenQuestion(OpenQuestion question) {  
        if (question != null && this.containsOpenQuestion(question)) {  
            this.openQuestions.remove(question);  
            return true;  
        }  
        System.out.println("There's no question like that.");  
        return false;  
    }  
  
    public boolean removeMOQuestion(MultipleOptionQuestion q) {  
        if (q != null && this.containsMOQuestion(q)) {  
            this.MOQuestions.remove(q);  
            return true;  
        }  
        System.out.println("There's no question like that.");  
        return false;  
    }  
  
    public boolean containsOpenQuestion(OpenQuestion question) {  
        return this.openQuestions.contains(question);  
    }  
  
    public boolean containsMOQuestion(MultipleOptionQuestion q) {  
        return this.MOQuestions.contains(q);  
    }  
}
```

El objetivo de esta clase es dejar en claro que el tipo de actividad que se va a instanciar, es *Examen* por lo que contiene una lista de preguntas abiertas y preguntas de opción múltiple.

Clase *FormActivity*

```
public class FormActivity extends Activity {  
    private LinkedList<OpenQuestion> questions;  
    public FormActivity(String title, String description, String objective, int expectedDuration, boolean mandatory,  
        LinkedList<OpenQuestion> questions) {  
        super(title, description, objective, expectedDuration, mandatory);  
        this.questions = questions != null ? questions : new LinkedList<>();  
    }  
}
```

```
    public boolean addQuestion(OpenQuestion question) {  
        if (question == null) {  
            System.err.println("Question can not be null.");  
        }  
        if (this.containsQuestion(question)) {  
            System.out.println("Question already added.");  
            return false;  
        }  
        this.questions.add(question);  
        return true;  
    }  
    public boolean removeQuestion(OpenQuestion question) {  
        if (question == null) {  
            System.err.println("Question can not be null.");  
        }  
        if (!this.containsQuestion(question)) {  
            System.err.println("There's no question like that.");  
            return false;  
        }  
        this.questions.remove(question);  
        return true;  
    }  
    public boolean containsQuestion(OpenQuestion question) {  
        return this.questions.contains(question);  
    }  
}
```

El objetivo de esta clase es dejar en claro que el tipo de actividad que se va a instanciar, es *Formulario* por lo que contiene una lista de preguntas abiertas, sin calificación alguna.

Clase *QuizActivity*

```
public class QuizActivity extends Activity {  
    private double minScore;  
    private LinkedList<MultipleOptionQuestion> questions;  
    public QuizActivity(String title, String description, String objective, int expectedDuration, boolean mandatory,  
        LinkedList<MultipleOptionQuestion> questions, double minScore) {  
        super(title, description, objective, expectedDuration, mandatory);  
        this.questions = questions != null ? questions : new LinkedList<>();  
        this.minScore = minScore;  
    }  
}
```

```

• public boolean addQuestion(MultipleOptionQuestion question) {
    if (question == null) {
        System.err.println("Question can not be null.");
    }
    if (this.containsQuestion(question)) {
        System.out.println("Question already added.");
        return false;
    }
    this.questions.add(question);
    return true;
}

• public boolean removeQuestion(MultipleOptionQuestion question) {
    if (question != null && this.containsQuestion(question)) {
        this.questions.remove(question);
        return true;
    }
    System.out.println("There's no question like that.");
    return false;
}

• public boolean containsQuestion(MultipleOptionQuestion question) {
    return this.questions.contains(question);
}
}

```

El objetivo de esta clase es dejar en claro que el tipo de actividad que se va a instanciar, es *Quiz* por lo que contiene una lista de preguntas con opción múltiple, es calificable.

Clase ResourceActivity

```

public class ResourceActivity extends Activity {

    private String url;

    public ResourceActivity(String title, String description, String objective, int expectedDuration, boolean mandatory,
        String url) {
        super(title, description, objective, expectedDuration, mandatory);
        this.url = url;
    }

    public String getUrl() {
        return url;
    }

    public void setUrl(String url) {
        this.url = url;
    }

}

```

El objetivo de esta clase es dejar en claro que el tipo de actividad que se va a instanciar, es *Quiz* por lo que contiene una lista de preguntas con opción múltiple, es calificable.

Clase TaskActivity

```

public class TaskActivity extends Activity {
    private String toDo;

    public TaskActivity(String title, String description, String objective, int expectedDuration, boolean mandatory,
        String toDo) {
        super(title, description, objective, expectedDuration, mandatory);
        this.toDo = toDo;
    }

    public String getToDo() {
        return toDo;
    }

    public void setToDo(String toDo) {
        this.toDo = toDo;
    }
}

```

Tiene como objetivo definir la tarea (más no actividad), como pendiente.

* Paquete Question

Clase MultipleOptionQuestion

```

public class MultipleOptionQuestion {
    private String question;
    private LinkedList<Option> options;

    public MultipleOptionQuestion(String question, LinkedList<Option> options) {
        this.question = question;
        this.options = options != null ? options : new LinkedList<>();
    }

    public String getQuestion() {
        return question;
    }

    public void setQuestion(String question) {
        this.question = question;
    }

    public LinkedList<Option> getOptions() {
        return options;
    }

    public void setOptions(LinkedList<Option> options) {
        this.options = options;
    }
}

```

Tiene el objetivo de definir n cantidad de opciones a una pregunta y dentro de estas, se encuentra la opción respuesta.

Clase OpenQuestion

```

public class OpenQuestion {
    private String text;

    public OpenQuestion(String text) {
        this.text = text;
    }

    public String getText() {
        return this.text;
    }

    public void setText(String text) {
        this.text = text;
    }
}

```

Su objetivo es almacenar la pregunta para manejarla más fácil y hacerlo más explícito.

Clase Option

```

public class Option {
    private String text;
    private boolean correct;
    private String explanation;

    public Option(String text, boolean correct, String explanation) {
        this.text = text;
        this.correct = correct;
        this.explanation = explanation;
    }

    public String getText() {}

    public void setText(String text) {}

    public boolean isCorrect() {}

    public void setCorrect(boolean correct) {}

    public String getExplanation() {}

    public void setExplanation(String explanation) {}
}

```

Tiene como objetivo almacenar el texto de la opción, la explicación a porque es o no es esa opción, y si es correcta esa opción.

** Paquete Controller*

Se crea para imitar el MVC de forma en el que la implementación en el P2 sea mucho mas fácil a la hora de implementar los nuevos requerimientos y la conexión entre la lógica y lo visual

Clase Controller

```

public class Controller {

    protected HashMap<String, User> userHashMap;
    protected HashMap<String, LearningPath> learningPathHashMap;
    protected HashMap<String, Activity> activityHashMap;
    protected User currentUser;

    /**
     * Default constructor initializing the hash maps and setting the current user to null.
     */
    public Controller() {
        userHashMap = new HashMap<>();
        learningPathHashMap = new HashMap<>();
        activityHashMap = new HashMap<>();
        currentUser = null;
    }

    /**
     * Constructor initializing the hash maps and setting the current user.
     *
     * @param userHashMap A hash map of users.
     * @param learningPathHashMap A hash map of learning paths.
     * @param activityHashMap A hash map of activities.
     * @param currentUser The current user.
     */
    public Controller(HashMap<String, User> userHashMap, HashMap<String, LearningPath> learningPathHashMap,
        HashMap<String, Activity> activityHashMap, User currentUser) {
        this.userHashMap = userHashMap;
        this.learningPathHashMap = learningPathHashMap;
        this.activityHashMap = activityHashMap;
        this.currentUser = currentUser;
    }
}

```

Clase StudentController: Sigue la misma lógica explicada anteriormente

```

package controller;

import learningpath.*;

public class StudentController extends Controller {

    private Student student;
    private LearningPath currentLearningPath;
    private ProgressTracker currentProgressTracker;
    private ActivityTracker currentActivityTracker;
    private LinkedList<LearningPath> learningPathsByInterest;

    public StudentController(HashMap<String, User> userHashMap, HashMap<String, LearningPath> learningPathHashMap,
        HashMap<String, Activity> activityHashMap, User currentUser) {
        super(userHashMap, learningPathHashMap, activityHashMap, currentUser);
        student = (Student) currentUser;
    }

    // Query methods

    public LinkedList<LearningPath> getLearningPathsByInterest(String interest) {

        LinkedList<LearningPath> learningPaths = new LinkedList<>();

        for(LearningPath learningPath : learningPathHashMap.values()) {

            LinkedList<String> tags = learningPath.getTags();
            if(tags.contains(interest)) {
                learningPaths.add(learningPath);
            }
        }
        learningPathsByInterest = learningPaths;
        return learningPaths;
    }
}

```


Clase ProfessorController:

```
package controller;

import java.util.Collection;

public class ProfessorController extends Controller {

    private Professor professor;
    private LearningPath currentLearningPath;
    private Activity currentActivity;

    /**
     * Constructs a new ProfessorController with the specified hash maps and
     * current user.
     *
     * @param userHashMap A hash map of users.
     * @param learningPathHashMap A hash map of learning paths.
     * @param activityHashMap A hash map of activities.
     * @param currentUser The current user.
     */
    public ProfessorController(HashMap<String, User> userHashMap, HashMap<String, LearningPath> learningPathHashMap,
                               HashMap<String, Activity> activityHashMap, User currentUser) {
        super(userHashMap, learningPathHashMap, activityHashMap, currentUser);
        professor = (Professor) currentUser;
        currentLearningPath = null;
        currentActivity = null;
    }

    // Query methods
    /**
     * Retrieves the learning paths associated with a professor.
     *
     * @param professor The professor whose learning paths are to be retrieved.
     * @return A linked list of learning paths associated with the professor.
     */
    public LinkedList<LearningPath> getProfessorLearningPaths(Professor professor) {
        return professor.getCreatedLearningPaths();
    }
}
```

* Paquete Utils

Clase Generator

```
public class Generator {

    private static Generator instance = null;
    private final HashMap<String, LinkedList<String>> db = new HashMap<>();

    private Generator() {
        db.put("Activity", new LinkedList<>());
        db.put("LearningPath", new LinkedList<>());
    }

    public static Generator getInstance() {
        if (instance == null) {
            instance = new Generator();
        }
        return instance;
    }

    public static void deleteInstance() {
        instance = null;
    }
}
```

```

private String nanoid(String input) {
    int length = input.length();
    String san = input.replace("\\W", "");

    int randomLength = Math.max(0, length - san.length());
    StringBuilder randomP = new StringBuilder();
    Random rand = new Random();

    for (int i = 0; i < randomLength; i++) {
        randomP.append((char) (rand.nextInt(36) + 'a'));
    }

    return (san + randomP.toString()).substring(0, length);
}

private String interleave(String s1, String s2) {
    StringBuilder result = new StringBuilder();
    int i = 0;
    int j = 0;

    while (i < s1.length() && j < s2.length()) {
        result.append(s1.charAt(i++));
        result.append(s2.charAt(j++));
    }

    result.append(s1.substring(i));
    result.append(s2.substring(j));

    return result.toString();
}

```

```

private boolean existsInDatabase(String type, String id) {
    if (type == null) {
        System.err.println("Type can not be null");
        return false;
    }

    if (id == null) {
        System.err.println("Id can not be null");
        return false;
    }
    LinkedList<String> typeList = this.db.get(type);
    if (typeList == null) {
        System.err.println("Invalid type \"" + type + "\"");
        return false;
    }

    return typeList.contains(id);
}

private String checkId(String type, String id) {
    boolean exists = this.existsInDatabase(type, id);

    if (!exists) {
        return id;
    }

    int l = type.length() + (int) Math.floor(type.length() / 2.0);
    String newId = this.interleave(type, this.nanoid(Integer.toString(l)));
    return checkId(type, newId);
}

```

```

public String generateId(String type) {
    int l = type.length() + (int) Math.floor(type.length() / 2.0);
    String id = interleave(type, nanoid(Integer.toString(l)));
    String finalId = checkId(type, id);

    if (!existsInDatabase(type, finalId)) {
        db.get(type).add(finalId);
    }

    return finalId;
}

```

Tiene la función de generar un identificador único por **Actividad** y **LearningPath**, por lo que tiene que verificar en la variable denotada como *db*, que este identificador no se encuentre repetido, y así asignar otro al azar.

* Paquete Persistencia

Clase CentralPersistencia

Principalmente se creó la clase de persistencia serializable debido a la facilidad del trabajo de los datos adicionalmente se implementaron 3 métodos para llamar a la persistencia y serializar 3 estructuras de datos que harán de bases de datos, serializando un arreglo donde se encuentran todas las actividades, un arreglo donde se encuentran todos los estudiantes, uno de profesores, a su vez uno de learningPaths y los profesores.

```
public class CentralPersistencia {  
    private static final long serialVersionUID = 1L;  
    public void guardar(Serializable object) {  
        String path = "./Data/database.ser";  
        File directory = new File("./Data/");  
        if (!directory.exists()) {  
            directory.mkdirs();  
        }  
  
        try (ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(path))) {  
            oos.writeObject(object);  
        } catch (IOException e) {  
            System.err.println("Error al guardar " + path + ": " + e.getMessage());  
            e.printStackTrace();  
        }  
    }  
  
    public Object cargar() {  
        String path = "./Data/database.ser";  
        File file = new File(path);  
        if (!file.exists()) {  
            return null;  
        }  
  
        try (ObjectInputStream ois = new ObjectInputStream(new FileInputStream(file))) {  
            return ois.readObject();  
        } catch (IOException | ClassNotFoundException e) {  
            System.err.println("Error loading object from " + path + ": " + e.getMessage());  
            e.printStackTrace();  
            return null;  
        }  
    }  
}
```

Método en professor:

```
public void guardarInfo() {  
  
    centralPersistencia.guardar(createdLearningPaths);  
    centralPersistencia.guardar(createdActivities);  
  
}
```

Guarda los learningPaths y las actividades

Método en generatos:

```
public void guardarInfo() {  
  
    centralPersistencia.guardar(db);  
  
}
```

```
private final HashMap<String, LinkedList<String>> db = new HashMap<>();
```

donde se guardan a partir de las id las contraseñas de los usuarios y a su vez y por último se tiene en users verifíer donde guarda todos los usuarios (profesores, estudiantes)

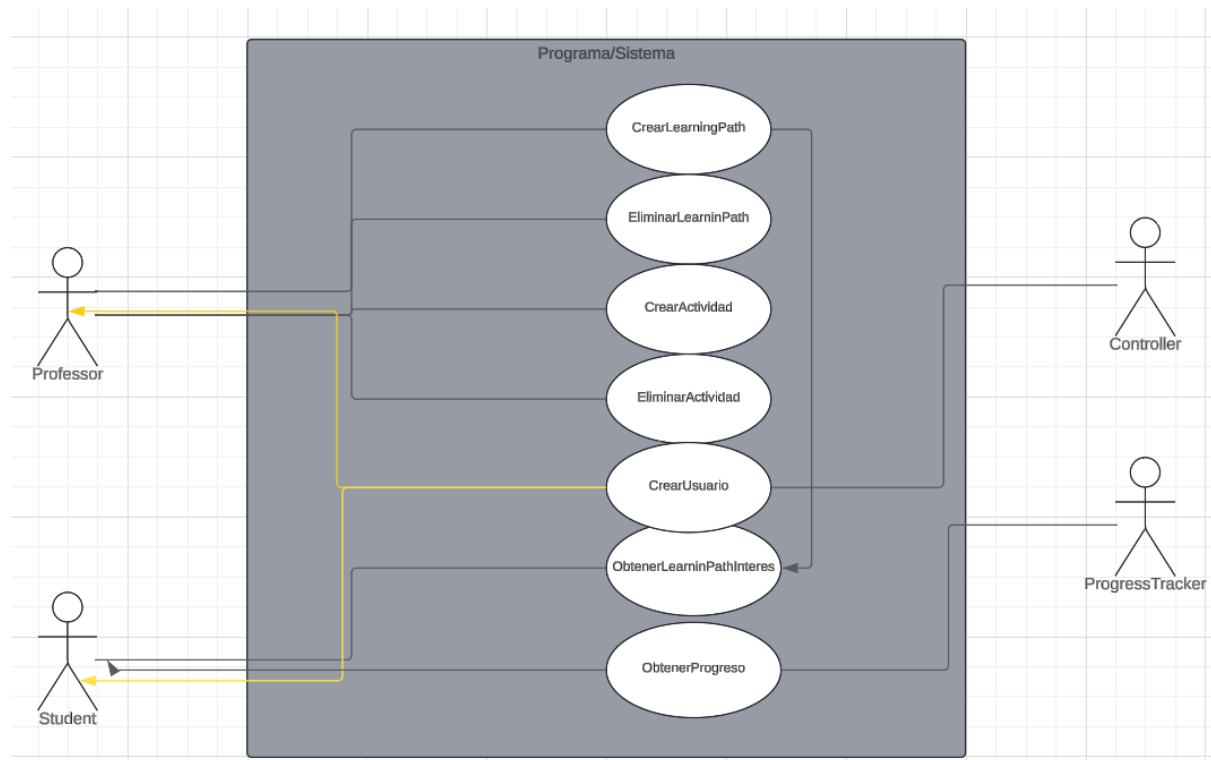
```
public static HashMap<String, String> usersDataBase = new HashMap<>();
```

Y en el main se agrega la siguiente línea para persistir todo

```
centralPersistencia.guardar(usersDataBase);
```

3). Historia de Usuarios:

Entendemos que progressTracker ni Controller son personas pero son la entidad que devuelve los datos específicos así que se añaden para mejorar la visualización y permitir un entendimiento más profundo del cómo funciona.



4). Requerimientos del Sistema

* Requerimientos Funcionales:

1. En primer lugar, se tiene el registro de los usuarios a través de un administrador que hereda de usuario así que el registra a todos los usuarios ya sean estudiantes o profesores, adicional posee toda la información
2. Los estudiantes pueden acceder y seguir Learning Paths (Por ende, acceder a experiencias de aprendizaje personalizadas).
3. Los profesores pueden acceder, crear, editar y gestionar learning paths, además pueden adjuntar contenidos como enlaces, videos y documentos en las actividades.
4. El sistema puede rastrear actividades y el progreso de los estudiantes (Administrador tiene acceso al sistema).
5. Los profesores pueden tomar Learning Paths ajenos y editarlos para crear uno único o copiarlo.
6. Los estudiantes pueden inscribir nuevos cursos con sus actividades. (Restricción de prerrequisitos).

7. Se puede agregar reseñas (los profesores y los estudiantes). Adicional, se puede agregar un rating.
8. Por lo anterior debe ser posible sacar un rating de actividades mejor calificadas.
9. Los docentes pueden mandar notificaciones por correo, LMS adicional calificar las respuestas.
10. Los estudiantes pueden agregar reseñas

* Requerimientos No Funcionales:

- La persistencia 😊

5). Diseño Detallado

→ insertar diagrama de clases

