Índice

[Información base del proyecto – TraLaLa: plataforma de música Streaming 3](#_Toc197339508)

[Introducción 3](#_Toc197339509)

[Objetivos 3](#_Toc197339510)

[Datos del proyecto 3](#_Toc197339511)

[Información básica: 3](#_Toc197339512)

[Herramientas recomendadas 3](#_Toc197339513)

[Entregables 3](#_Toc197339514)

[Penalizaciones 4](#_Toc197339515)

[Enunciado del proyecto TraLaLa 5](#_Toc197339516)

[Información detallada para modelado UML – TraLaLa 5](#_Toc197339517)

[Gestión de canciones 5](#_Toc197339518)

[Gestión de usuarios 5](#_Toc197339519)

[Amigos entre usuarios 6](#_Toc197339520)

[Recomendaciones 6](#_Toc197339521)

[Sistema de puntos 6](#_Toc197339522)

[Reproducciones 6](#_Toc197339523)

[Rol del Administrador 7](#_Toc197339524)

[Requisitos Funcionales – TraLaLa 7](#_Toc197339525)

[Actividades a realizar en el proyecto TraLaLa 9](#_Toc197339526)

[**Análisis del sistema y Modelado UML (3 puntos)** 9](#_Toc197339527)

[**1.** **Diagrama de casos de uso (1,5 puntos)** 9](#_Toc197339528)

[**2.** **Diagrama de clases (1,5 puntos)** 9](#_Toc197339529)

[**Desarrollo en Eclipse (1 punto)** 10](#_Toc197339530)

[**1.** **Generación de código (1 punto)** 10](#_Toc197339531)

[**Análisis de calidad (4 puntos)** 12](#_Toc197339532)

[**1.** **Complejidad ciclomática (2 puntos)** 12](#_Toc197339533)

[1.1. Definición de nodos y Grafo de flujo (0,5 puntos) 12](#_Toc197339534)

[1.2. Cálculo de la complejidad Ciclomática (0,25 puntos) 12](#_Toc197339535)

[1.3. Caminos básicos (0,25 puntos) 13](#_Toc197339536)

[1.4. Tabla Combinada de Caminos y Casos de Prueba (1 punto) 13](#_Toc197339537)

[**2.** **Automatización pruebas unitarias - JUnit (2 puntos)** 13](#_Toc197339538)

[2.1. Pruebas Unitarias (1,5 puntos) 13](#_Toc197339541)

[2.2. Cobertura (0,5 puntos) 13](#_Toc197339542)

[**Control de versiones del proyecto (2 puntos)** 14](#_Toc197339543)

[1. Crea un repositorio compartido en GitHub (0,25 puntos): 14](#_Toc197339544)

[2. Manejo y control del contenido de los ejercicios tanto de código como no código a Git (0,75 puntos): 14](#_Toc197339545)

[3. Fusión de ramas (Merge) y resolución de conflictos (1 punto): 15](#_Toc197339546)

[Criterios de evaluación del proyecto 16](#_Toc197339547)

# Información base del proyecto – TraLaLa: plataforma de música Streaming

## Introducción

En este proyecto vamos a trabajar sobre el sistema TraLaLa, una plataforma de música en streaming. A partir del enunciado funcional proporcionado, los estudiantes deberán analizar y definir el sistema mediante el uso de modelos UML, para después implementar una solución en código y aplicar técnicas de calidad del software.

## Objetivos

Puntualización de los objetivos del proyecto en base a las distintas fases del desarrollo del software:

* Fase de análisis y diseño:
  + Modelar diagramas de casos de uso y de clases en base al enunciado y los requisitos definidos.
* Fase de desarrollo:
  + Generar el código base del proyecto, tomando como base el diagrama de clases diseñado y añadirlo a un proyecto de código Java en Eclipse.
* Fase de pruebas:
  + Calcular la complejidad ciclomática de un método definido.
    - Obtener el diagrama de grafo de flujo
    - Calcular las V(G)
    - Obtener los caminos para realizar las pruebas
  + Crear las pruebas unitarias para el método en concreto en Junit
* Controlar versiones de un sistema de software completo a partir de un enunciado funcional con git/GitHub.

## Datos del proyecto

### ****Información básica:****

* **Nombre del Proyecto:** TraLaLa
* **Asignatura:** Entornos de Desarrollo
* **Duración:** 9 horas
* **Fecha de Inicio:** 06/05/2025
* **Fecha de fin:** 23/05/2025
* **Grupo:** 2

### Herramientas recomendadas

* Modelado UML: Modelio
* Lenguaje de programación: Java
* Creación del proyecto Java: Eclipse
* Sistema de control de versiones: Git + GitHub
* Testing: JUnit (Java)

### Entregables

* **Por grupo de trabajo:**
  + Diagramas UML (casos de uso y clases) :
    - Añadir las imágenes de los diagramas al documento del enunciado
    - Proyecto Modelio en un fichero .zip
  + Proyecto Java creado en Eclipse (.zip)
    - Código fuente del sistema generado a partir del diagrama de clases
    - Método para pruebas añadido
    - Unit Test creados para ese método
  + Documentación: documento del enunciado del proyecto cumplimentado con los requisitos específicos, que incluyen:
    - Diagrama de casos de uso
    - Diagrama de clases
    - Complejidad ciclomática
    - Caminos de casos de pruebas
    - Detalle Registro completo en Git: historial de commits, ramas, tags y aplicación del flujo Git Flow.

### Penalizaciones

* **No se permitirá la utilización de ChatGPT o herramientas de inteligencia artificial:** en caso de detección de su uso elproyecto **será suspendido**.

# Enunciado del proyecto TraLaLa

**TraLaLa** es una plataforma digital de música en streaming donde los usuarios pueden escuchar canciones online a un precio económico y ver la letra mientras suena la música. Para usar el servicio, los usuarios deben registrarse y añadir métodos de pago.

Cada usuario puede guardar canciones en dos listas:

* **Favoritas**, por las que paga cada vez que las escucha.
* **Colección permanente**, que puede escuchar gratis e ilimitadamente.

Los usuarios pueden tener **amigos** dentro del sistema y **recomendar canciones** entre ellos. Cada vez que alguien envía una recomendación, **gana 1 punto**, independientemente de si su amigo escucha la canción.  
Cuando acumulan al menos **100 puntos**, pueden **canjearlos** para mover canciones favoritas a su colección permanente (1 canción por cada 100 puntos).

El sistema registra cada **reproducción**, guardando la canción, el usuario y la hora en que se escuchó.

Además, hay un **Administrador** encargado de gestionar el catálogo musical (añadir, editar y eliminar canciones) y de supervisar el correcto funcionamiento de la plataforma, incluyendo la gestión de cuentas de usuario en caso de actividad sospechosa.

## Información detallada para modelado UML – TraLaLa

### Gestión de canciones

El sistema TraLaLa almacena un catálogo musical compuesto por miles de canciones. Cada canción se identifica por su **título** y tiene asociado el **nombre del artista o grupo**, así como su **duración** (en minutos y segundos), la **fecha de publicación**, el **precio por reproducción** (importe que paga el usuario por escucharla), y el **texto completo de la letra de la canción**. Además, cada canción está vinculada a un archivo de música en formato MPEG, por lo que también se guarda la **URL** donde se encuentra alojado dicho archivo.

El sistema permite acceder a esta información, mostrarla y reproducir tanto la música como la letra asociada.

### Gestión de usuarios

Para acceder a los servicios de TraLaLa, es necesario que los usuarios se registren. Para cada usuario, el sistema almacena un **nickname único**, una **contraseña**, su **correo electrónico**, y una **auto-descripción o biografía breve** que puede servir como perfil visible.

Los usuarios pueden asociar uno o varios **métodos de pago**, entre los que se incluyen **tarjeta de crédito (en la que se almacenan el número de tarjeta, nombre del titular y apellido del titular)**, **PayPal** **(en la que se almacena el email)** y **Bizum (en la que se almacena el número de teléfono).** Estos métodos se utilizan para abonar el coste por reproducción de canciones.

Cada usuario puede gestionar dos listas personales de canciones:

* Una **lista de canciones favoritas**, donde almacena aquellas canciones que escucha frecuentemente y por las que está dispuesto a pagar.
* Una **colección permanente**, que contiene canciones que el usuario puede escuchar gratuitamente siempre que quiera, sin límite de veces ni coste adicional.

Los usuarios pueden escuchar canciones, añadir o quitar canciones de sus listas, ver sus puntos acumulados, reproducir canciones recomendadas, y canjear puntos por canciones de su lista de favoritas para moverlas a la colección permanente.

### Amigos entre usuarios

Los usuarios pueden **añadir a otros usuarios como amigos**. Esta relación permite habilitar las funcionalidades de recomendación. Cada usuario mantiene una lista de sus amigos. El sistema debe validar que ambos usuarios existan antes de establecer una relación de amistad. Esta relación permite enviar recomendaciones de canciones exclusivamente entre amigos.

### Recomendaciones

Una funcionalidad clave del sistema es la posibilidad de enviar **recomendaciones de canciones** entre amigos. Para ello, el usuario selecciona una canción de su catálogo y uno de sus amigos, y el sistema genera una **recomendación** que incluye:

* El **usuario emisor** (quien envía la recomendación)
* El **usuario receptor** (quien la recibe)
* La **canción recomendada**
* La **fecha de envío**
* Un **identificador único** para la recomendación

Al enviarse una recomendación, el sistema otorga automáticamente **1 punto al emisor**, sin importar si el receptor escucha la canción o no. Cuando el receptor quiera acceder a la canción recomendada, podrá hacerlo introduciendo el identificador recibido, lo que iniciará la reproducción correspondiente.

### Sistema de puntos

Los puntos obtenidos mediante recomendaciones sirven para ampliar la colección permanente del usuario. Por cada recomendación enviada, el usuario gana 1 punto. Estos puntos se van acumulando y pueden ser **canjeados**:  
cuando un usuario tiene **al menos 100 puntos**, puede elegir **una canción de su lista de canciones favoritas** y moverla a su colección permanente.

Por cada 100 puntos acumulados se puede realizar esta operación una vez. Por ejemplo, con 300 puntos se pueden canjear hasta 3 canciones favoritas.

El proceso de canjeo incluye:

* Verificar cuántos puntos tiene el usuario
* Mostrar las canciones de la lista de canciones favoritas
* Permitir seleccionar una o varias, según la cantidad de puntos
* Mover las canciones seleccionadas a la colección permanente
* Eliminar dichas canciones de la lista de canciones favoritas
* Restar los puntos correspondientes (100 por canción)

### Reproducciones

Cada vez que un usuario escucha una canción, el sistema registra una **reproducción**. Esta contiene la siguiente información:

* El **usuario** que escucha la canción
* La **canción reproducida**
* La **fecha y la hora de inicio de la reproducción**

Esta información se guarda automáticamente al iniciar la reproducción y sirve para llevar un registro de uso, contabilizar pagos y analizar comportamiento del sistema.

### Rol del Administrador

El sistema cuenta con un perfil especial de **Administrador**, cuya función es gestionar y supervisar el correcto funcionamiento de la plataforma.

Entre sus responsabilidades se incluyen:

* **Añadir nuevas canciones** al catálogo musical, introduciendo todos los datos necesarios (título, artista, duración, fecha, letra, precio, URL).
* **Editar información existente** de una canción, como por ejemplo corregir letras o actualizar metadatos (nombre, precio, etc.).
* **Eliminar canciones** que ya no deban formar parte del catálogo, bien por estar desactualizadas o por contener contenido inapropiado.
* **Gestionar cuentas de usuario**: si se detecta comportamiento anómalo o uso indebido, el administrador puede revisar, bloquear o modificar el estado de un usuario.
* **Supervisar el sistema** en general, detectando errores, inconsistencias o mal uso del servicio.

El administrador no necesita reproducir canciones ni tener amigos o puntos, ya que su papel es únicamente operativo y de control.

## Requisitos Funcionales – TraLaLa

* **PU#1 – Registro de usuario:** el sistema debe permitir a nuevos usuarios registrarse, proporcionando nickname, contraseña, email, descripción y métodos de pago.
* **PU#2 – Iniciar sesión:** el usuario debe poder iniciar sesión en el sistema para acceder a sus funcionalidades.
* **PU#3 – Escuchar una canción:** un usuario autenticado puede seleccionar una canción del catálogo y reproducirla, mostrándose también la letra.
* **PU#4 – Marcar canción como favorita:** el usuario puede añadir canciones a su lista de canciones favoritas.
* **PU#5 – Visualizar colección permanente:** el usuario puede consultar su lista de canciones disponibles para reproducción gratuita e ilimitada.
* **PU#6 – Añadir amigo:** un usuario puede añadir a otro como amigo, si ambos están registrados en el sistema.
* **PU#7 – Recomendar canción a un amigo:** un usuario puede enviar una recomendación de canción a cualquiera de sus amigos, ganando 1 punto automáticamente.
* **PU#8 – Escuchar una recomendación:** un usuario puede introducir el identificador de recomendación recibido y reproducir la canción asociada.
* **PU#9 – Canjear puntos por canciones favoritas:** si el usuario tiene al menos 100 puntos, puede seleccionar una o más canciones favoritas y moverlas a su colección permanente (1 canción por cada 100 puntos). El sistema debe actualizar las listas y descontar los puntos.
* **PU#10 – Registrar reproducción:** cada vez que un usuario escucha una canción, el sistema debe registrar la reproducción con fecha y hora de inicio.
* **PU#11 – Añadir nueva canción (Administrador):** el administrador puede añadir nuevas canciones al catálogo.
* **PU#12 – Editar información de canción (Administrador):** el administrador puede modificar los datos de una canción (como letra o título).
* **PU#13 – Eliminar canción (Administrador):** el administrador puede eliminar canciones que sean obsoletas o inapropiadas.
* **PU#14 – Gestionar cuentas de usuario (Administrador):** el administrador puede revisar y modificar cuentas de usuario en caso de uso indebido o comportamiento sospechoso.
* **PU#15 – Ver puntos acumulados:** el usuario puede consultar la cantidad de puntos que ha acumulado por recomendaciones.

# Actividades a realizar en el proyecto TraLaLa

**Análisis del sistema y Modelado UML (3 puntos)**

A partir del enunciado de TraLaLa, el alumnado deberá:

* Analizar los requisitos del sistema.
* Identificar los actores y funcionalidades del sistema y sus relaciones para crear un diagrama de casos de uso.
* Identificar las clases principales, atributos, métodos y relaciones entre ellas para crear un diagrama de clases.

**Requerimientos mínimos para aprobar:**

* Crear en Modelio los diagramas de casos de uso y de clases. Asignar al proyecto Modelio a crear el siguiente nombre: **6\_1\_MODELIO\_APELLIDO1\_APELLIDO2\_2425**, donde APELLIDO1 es el primer apellido de la persona del grupo y APELLIDO2 es el apellido de la segunda persona. No debe haber espacios en blanco entre las diferentes letras y palabras del apellido. En caso de que el número de miembros del equipo no sea 2, se acordará con el docente el nombre a dar al proyecto Modelio.
* Al finalizar ambos diagramas se deberá de crear un único fichero zip a entregar con el proyecto de Modelio.

1. **Diagrama de casos de uso (1,5 puntos)**

Realizar en Modelio el diagrama de casos de uso en base al análisis de requisitos funcionales definidos en el enunciado:

**Requerimientos mínimos para aprobar:**

* Crear en el proyecto Modelio el diagrama de casos de uso.
* Incluir todos los casos de uso, actores, contexto (ámbito o frontera del sistema que se está modelando, es decir lo que está dentro y lo que está fuera del sistema) y relaciones del diagrama.
* Subir a la plataforma un único archivo comprimido con el contenido de la carpeta del Workspace de Modelio en la que se alojan los diferentes archivos del proyecto creado en Modelio.
* Pegar un pantallazo del diagrama en el documento.

**Diagrama de casos de uso:**

1. **Diagrama de clases (1,5 puntos)**

Realizar en Modelio el diagrama de clases en base al análisis de requisitos funcionales definidos en el enunciado:

**Requerimientos mínimos para aprobar:**

* Crear en el proyecto Modelio el diagrama de clases.
* Incluir todas las clases, atributos, métodos, relaciones y multiplicidades.
* Subir a la plataforma un único archivo comprimido con el contenido de la carpeta del Workspace de Modelio en la que se alojan los diferentes archivos del proyecto creado en Modelio.
* Todos los atributos se han definido con visibilidad privada.
* Pegar un pantallazo del diagrama en el documento.

**Diagrama de clases:**

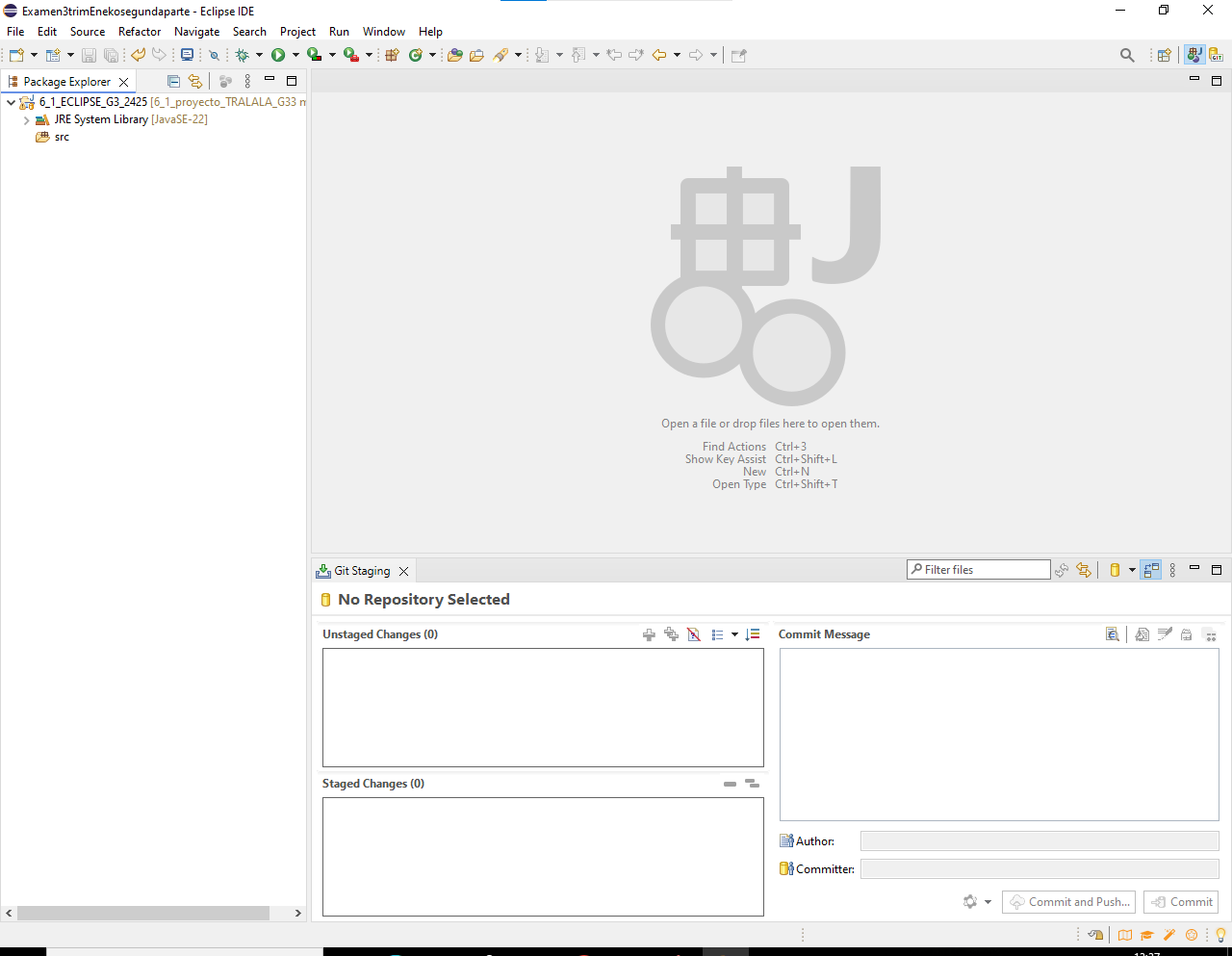
**Desarrollo en Eclipse (1 punto)**

1. **Generación de código (1 punto)**

Realizar en Modelio la generación de código y trasladar el resultado a un proyecto Java en Eclipse.

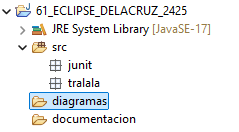
**Requerimientos mínimos para aprobar:**

* Generar un proyecto nuevo vacío de Java en Eclipse. El proyecto en Eclipse debe tener el siguiente nombre: **6\_1\_ECLIPSE\_APELLIDO1\_APELLIDO2\_2425**, donde APELLIDO1 es el primer apellido de la persona del grupo y APELLIDO2 es el apellido de la segunda persona. No debe haber espacios en blanco entre las diferentes letras y palabras del apellido. En caso de que el número de miembros del equipo no sea 2, se acordará con el docente el nombre a dar al proyecto Modelio.

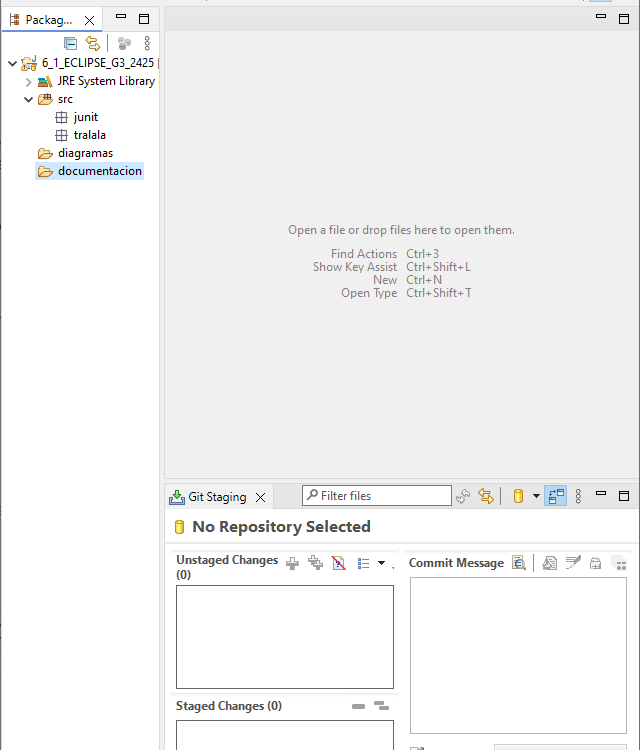


* Cuando se finalice el proyecto su creará un fichero .zip y se entregará en Moodle.

**Tareas a realizar:**

* Estructura de carpetas (0,25 puntos): El repositorio guardará todo el contenido utilizado en el proyecto en la siguiente estructura de carpetas:
* **src**: xonde la carpeta contendrá el código Java.
  + Paquete Tralala: contenido de clases Java generadas en Modelio desde el diagrama de clases.
  + Paquete JUnit: clases de los test realizados en JUnit.
* **diagramas**: carpeta que contendrá los ficheros de Modelio con ambos diagramas y un fichero -zip con el contenido completo.
* **documentación**: carpeta donde se guardarán las distintas versiones del documento del enunciado que se irá completando en las distintas fases del proyecto. Contendrá:
  + Imagen del diagrama de casos de uso
  + Imagen del diagrama de clases
  + Complejidad ciclomática – grafo de flujo
  + Cálculos de complejidad ciclomatica

Con el diagrama de clases creado previamente generar el código base del sistema (Java). El código debe:



(0,25 puntos):

* + Seguir la estructura del diagrama de clases.
  + Representar de forma fiel las asociaciones y responsabilidades de cada clase.

Incluir la nueva estructura de código generada en Modelio en el nuevo proyecto Java en Eclipse dentro de la carpeta **src y adaptarlo a la nueva estructura y eliminar los errores dejando el código compilable**. (0,25 puntos)

El proyecto tiene definido el método **canjearCanciones** en el apartado siguiente relacionado con los controles de calidad. Este código, adjunto a continuación, se deberá añadir a la clase **Canción** dentro del proyecto de Eclipse. Este método será el objeto de análisis de calidad y pruebas en la siguiente fase. (0,25 puntos)

**public** **static** **void** canjearCanciones(Usuario usuario) {

// Comprobamos si el usuario tiene suficientes puntos

**if** (usuario.getPuntos() < 100) {

System.***out***.println("No tienes suficientes puntos para canjear canciones.");

**return**;

}

// Obtenemos las canciones favoritas del usuario

List<Cancion> favoritas = **new** ArrayList<>(usuario.getCancionesFavoritas());

// Inicializamos las variables

**int** maxCanciones = usuario.getPuntos() / 100;

List<Cancion> cancionesACanjear = **new** ArrayList<>();

// Iniciamos el proceso de canje de canciones

**while** (cancionesACanjear.size() < maxCanciones && !favoritas.isEmpty()) {

System.***out***.println("Selecciona una canción para canjear:");

Cancion cancion = *seleccionarCancion*(favoritas);

**if** (!cancionesACanjear.contains(cancion)) {

cancionesACanjear.add(cancion);

usuario.getColeccionPermanente().add(cancion);

favoritas.remove(cancion);

usuario.getCancionesFavoritas().remove(cancion);

}

}

// Restamos los puntos usados

usuario.setPuntos(usuario.getPuntos() - 100 \* cancionesACanjear.size());

System.***out***.println(cancionesACanjear.size() + " canciones añadidas a tu colección permanente.");

}

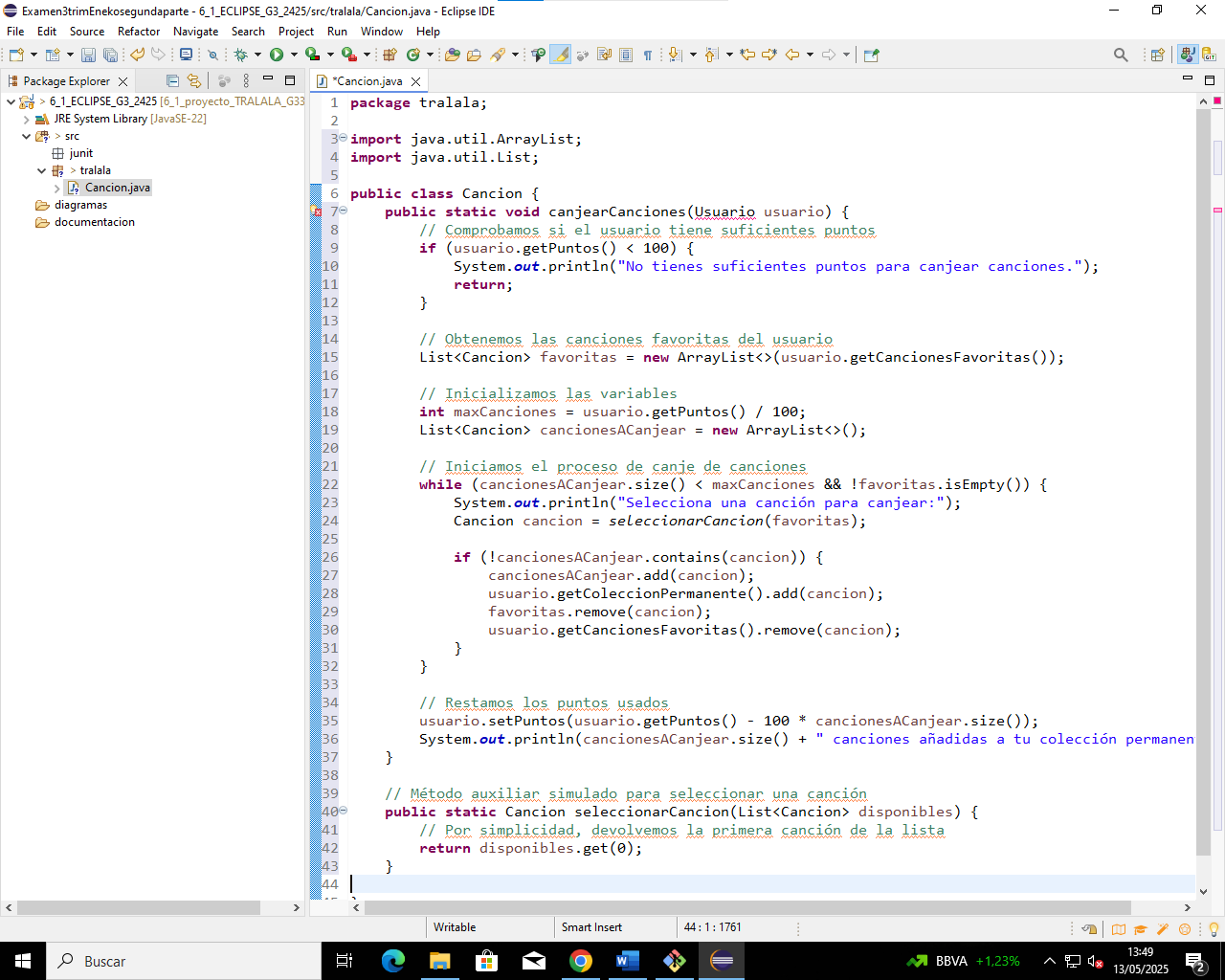
// Método auxiliar simulado para seleccionar una canción

**public** **static** Cancion seleccionarCancion(List<Cancion> disponibles) {

// Por simplicidad, devolvemos la primera canción de la lista

**return** disponibles.get(0);

}

****

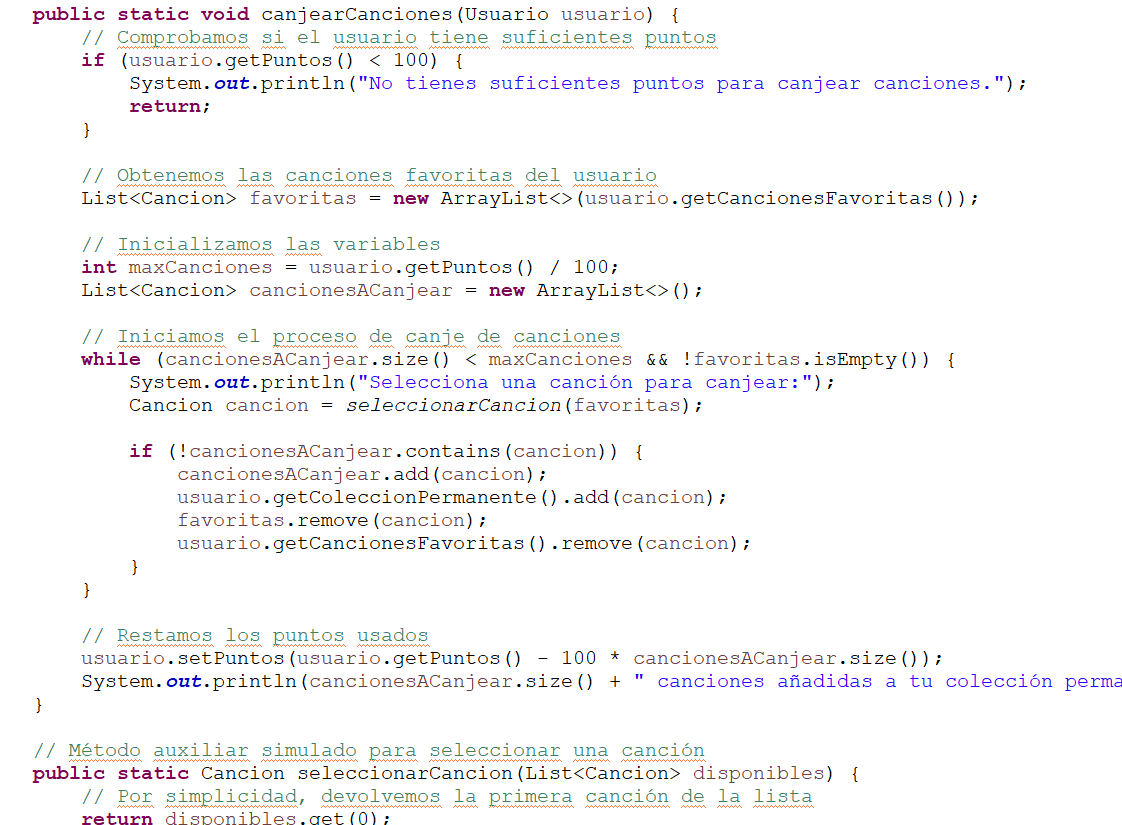
**Análisis de calidad (4 puntos)**

En este apartado se va a realizar un análisis de calidad del método canjearCanciones(Usuario usuario), que se habrá incorporado previamente al proyecto Eclipse en la clase Usuario en el apartado anterior. Este método formará parte del sistema TraLaLa. Este análisis tiene como objetivo evaluar la calidad estructural del código y su capacidad para ser probado correctamente mediante la automatización de pruebas unitarias.

**Objetivos del análisis**

* Calcular la **complejidad ciclomatica** del método.
* Identificar los **caminos independientes de ejecución**.
* Diseñar y justificar **casos de prueba unitarios** que cubran esos caminos.
* Implementar los **tests unitarios con JUnit**, asegurando una cobertura completa del código (100%).

**Código a analizar**

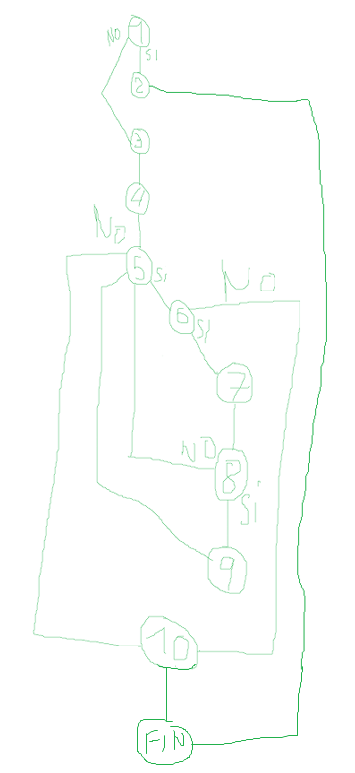


1. **Complejidad ciclomática (2 puntos)**

Cada estructura de control (como if, while) representa una decisión. Esta métrica permite conocer cuántos caminos lógicos deben cubrirse con pruebas unitarias para garantizar un testeo completo.

### Definición de nodos y Grafo de flujo (0,5 puntos)





### Cálculo de la complejidad Ciclomática (0,25 puntos)

* V(G) = 5 regiones
* V(G) = 14-11+2=5
* V(G) = 4+1 = 5

### Caminos básicos (0,25 puntos)

* Camino 1: 1-2-fin
* Camino 2: 1-3-4-5-10-fin
* Camino 3: 1-3-4-5-6-10-fin
* Camino 4: 1-3-4-5-6-7-8-5-10-fin
* Camino 5: 1-3-4-5-6-7-8-9-5-10-fin

### Tabla Combinada de Caminos y Casos de Prueba (1 punto)

| **Camino** | **Caso de Prueba** | **Datos de Entrada** | **Resultado Esperado** |
| --- | --- | --- | --- |
| 1 | 1-2-fin | Puntos usuario= 50 | Retorna “no tiene suficientes puntos para canjear canciones” |
| 2 | 1-3-4-5-10-fin | Puntos usuario= 150  Canciones a canjear = 0 | La primera vez el sistema si o si tendrá 0 para canjear canciones asique no podrá canjear nada, pero si mostrara en pantalla el mensaje “canciones añadidas a tu colección perma” |
| 3 | 1-3-4-5-6-10-fin | Puntos usuario= 150  Canciones a canjear = 1  Canciones favoritas = 0 | En la primera parte del while del programa entrará, pero en la segunda parte en caso de que este vacio, pero al final seguirá mostrando en pantalla “canciones añadidas a tu colección perma” |
| 4 | 1-3-4-5-6-7-8-5-10-fin | Puntos usuario= 150  Canciones a canjear = 1  Canciones favoritas = 1  Canciones a canjear (c1) = c1 | Entra dentro del while en el programa, pero en este caso la canción a canjear será la misma que ya este canjeada asique no hará nada saldrá del while y al final mostrara en pantalla “canciones añadidas a tu colección perma” |
| 5 | 1-3-4-5-6-7-8-9-5-10-fin | Puntos usuario= 150  Canciones a canjear = 1  Canciones favoritas = 1  Canciones a canjear (c1) = c2 | El programa actúa bien entrando en el while y añadiendo una canción distinta a las que ya hay dentro mostrando bien al final el siguiente mensaje: “canciones añadidas a tu colección perma” |

1. **Automatización pruebas unitarias - JUnit (2 puntos)**

Los tests de pruebas unitarias se implementarán utilizando **JUnit** dentro del entorno de desarrollo **Eclipse**. **Cada prueba cubrirá un camino básico del método**. Se validarán tanto salidas como efectos secundarios (por ejemplo, cambios en las listas de canciones o en los puntos del usuario). Si al realizar estas pruebas unitarias se detectase algún error en el código proporcionado, se deberá solucionar y mostrar evidencias de su solución.



### Pruebas Unitarias (1,5 puntos)

Crea pruebas en un fichero de pruebas(@Test) que contenga todas las pruebas unitarias para el código proporcionado.

* Usa JUnit (JUnit 5 preferiblemente).
* Usa aserciones (assertEquals, assertTrue, assertFalse, assertNull etc.).
* Utiliza anotaciones para condiciones previas a los test si fueran necesarias (@beforeEach, @BeforeAll, @AfterEach, @AfterAll etc.).
* Es necesario tener en cuenta que se deben cubrir todos los caminos básicos definidos durante la complejidad ciclomática.

### Cobertura (0,5 puntos)

Asegura la cobertura del 100% del código proporcionado y adjunta una evidencia (captura de pantalla)

* Asegúrate de que todos los caminos posibles de ejecución estén cubiertos por tus pruebas. Se puede utilizar para ello la herramienta **EclEmma Java Code Coverage que se puede descargar del Eclipse Marketplace.**
* Si usas estructuras como if, switch o bucles (while, for), asegúrate de probar todos los casos.

**Control de versiones del proyecto (2 puntos)**

Toda la gestión del proyecto, incluido código fuente y los contenidos del proyecto deberá realizarse utilizando Git como sistema de control de versiones y GitHub como repositorio remoto compartido.

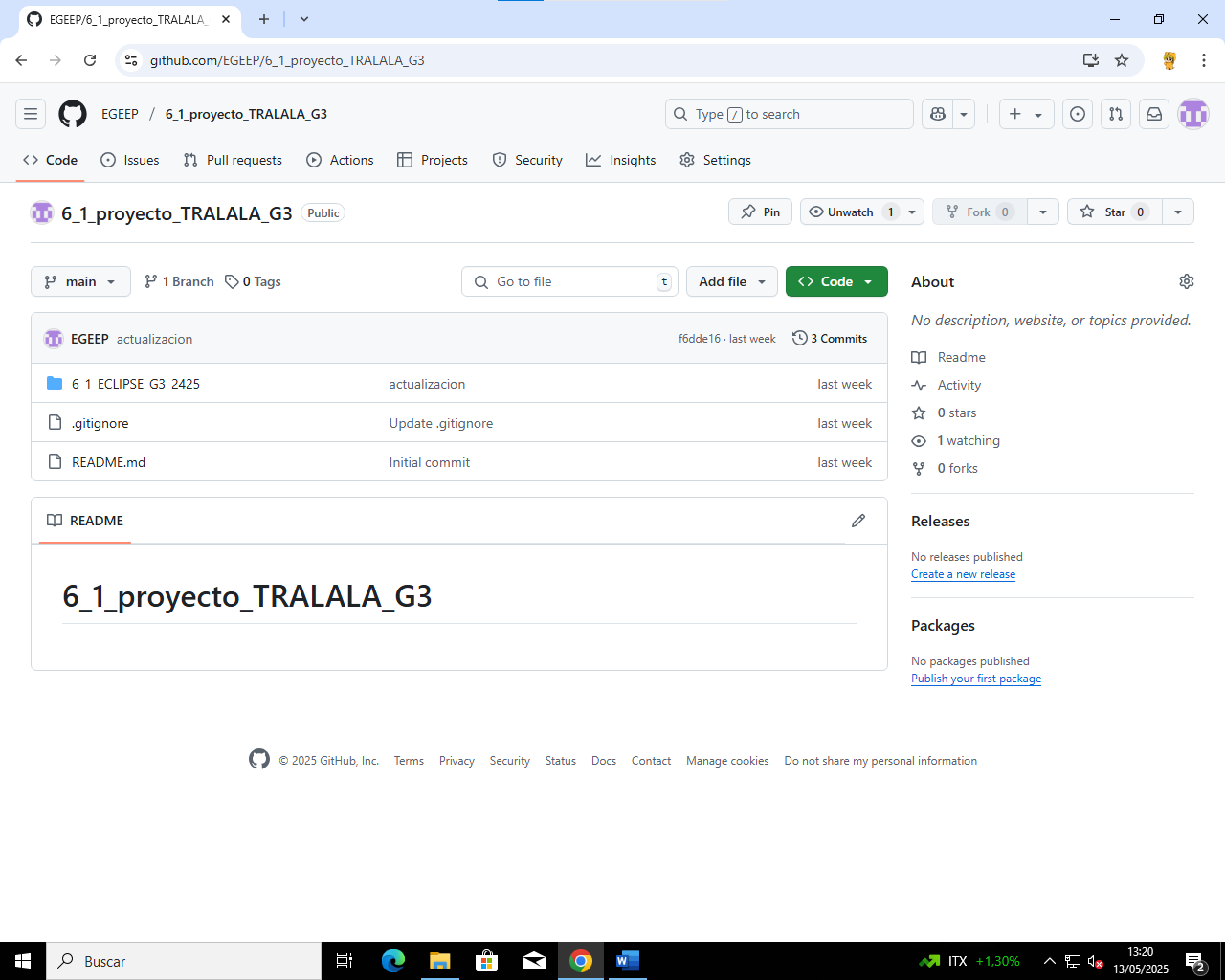
El trabajo se desarrollará en grupos de dos personas, siguiendo buenas prácticas de desarrollo colaborativo, como el uso de ramas personales, commits significativos, fusión de ramas (merge), resolución de conflictos y etiquetado de versiones (tags).

Además, se deberá aplicar el flujo de trabajo Git Flow para organizar correctamente el desarrollo del proyecto. Todos los cambios y entregas deberán quedar correctamente registrados en el historial del repositorio.

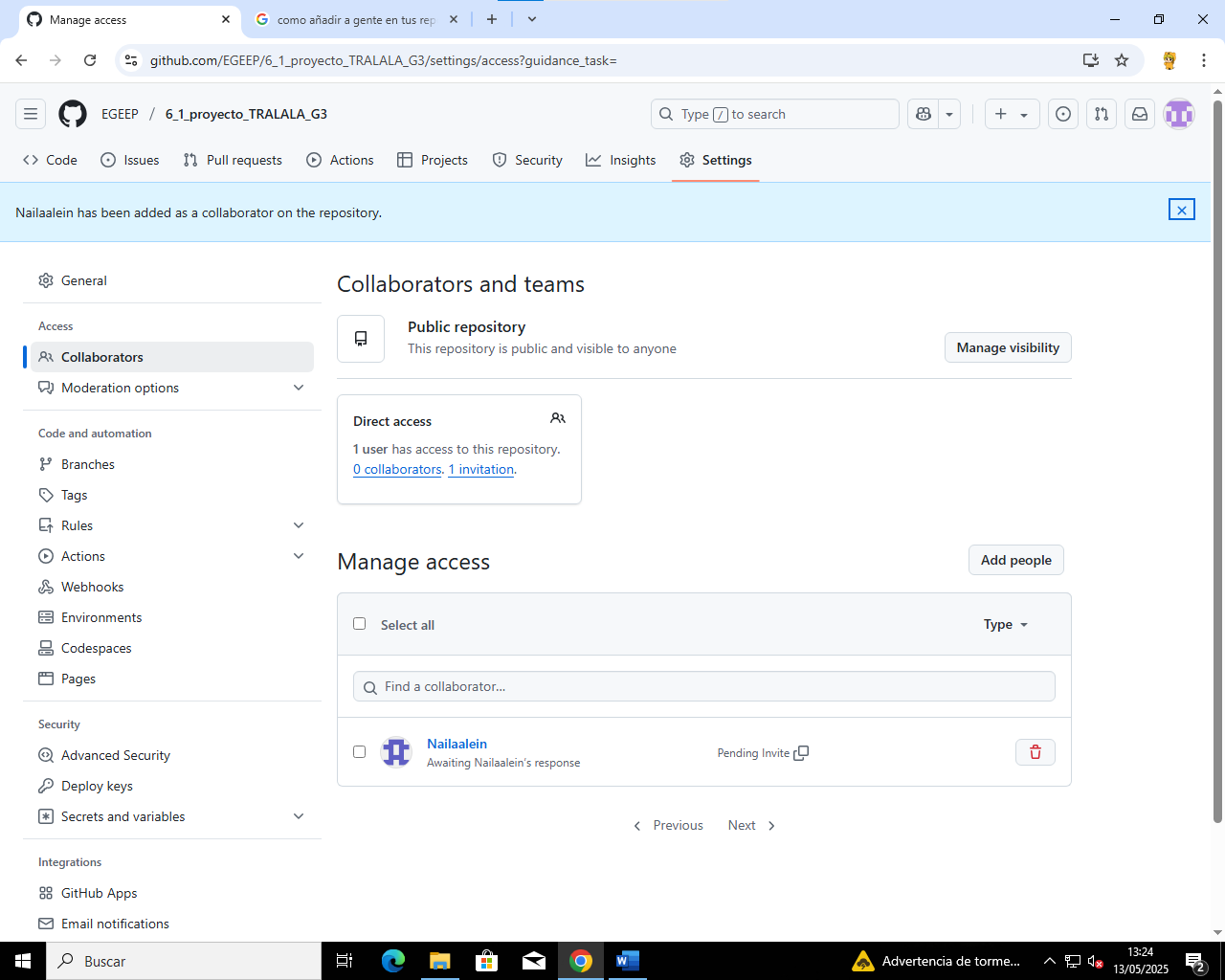
**GitHub (2 puntos)**

### Crea un repositorio compartido en GitHub (0,25 puntos):

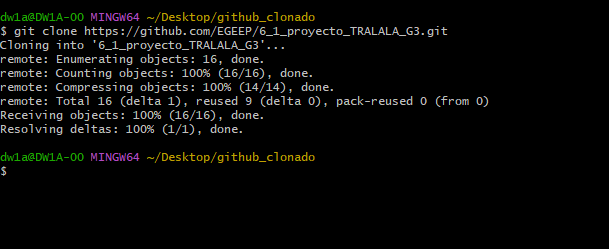
* Uno de los dos debe crear un repositorio en GitHub (puede ser privado o público).



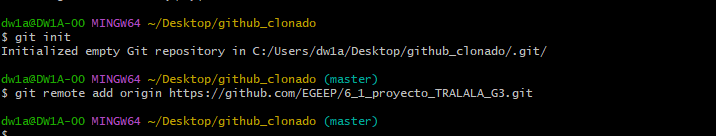
* Añade al compañero/a como colaborador y al profesor/a.



* Clonadlo los dos en vuestros respectivos ordenadores.

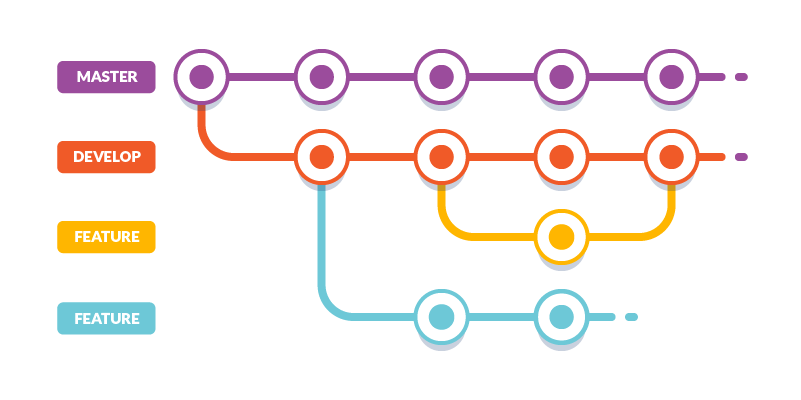


* Se utilizará el control de versiones desde el principio del proyecto



### Manejo y control del contenido de los ejercicios tanto de código como no código a Git (0,75 puntos):

* Una vez finalizados los diagramas se deberán de subir las imágenes y el código generado para los **diagramas UML** (casos de uso, clases, etc.) en la ubicación correspondiente.
* Es importante que los commits reflejen **quién ha subido qué**. Usad mensajes claros.
* Git debe seguir la definición del **Git Flow**, creando una rama main sobre la que se hará la entrega, una rama develop sobre la que se unirán los cambios de ambos integrantes y ramas independientes para el desarrollo de ambos.



* Debéis crear una rama personalizada desde la rama **develop**. La rama debe seguir el formato feature\_ alumnoX, donde X es vuestro número o nombre.
* En vuestra rama personal, implementad el código necesario para crear los JUnit que permitan hacer el test para validar los cambios definidos en la complejidad ciclomática.
* Aseguraos de que todos los tests cubren las funcionalidades del código proporcionado y que el Coverage es del %100. Insertar una evidencia (captura de pantalla).
* Una vez que hayáis completado la implementación y los tests, realizad un commit de todos los cambios a vuestra rama y luego pasad ese mismo código a la rama develop para juntar con el resto del contenido.

### Fusión de ramas (Merge) y resolución de conflictos (1 punto):

* Después de que ambos alumnos hayan subido su código a sus ramas correspondientes, se procederá a fusionar las ramas feature\_alumno1 y feature\_alumno2 con la rama develop.
* **Importante:** debe surgir un **conflicto de fusión** que debéis resolver. Si no se consigue a la primera, tendréis que cambiar algo en vuestros locales para conseguir un conflicto y forzaros a la resolución. Mostrar las evidencias.
* Para hacer el merge y resolver conflictos:
  + Si el conflicto es detectado, debéis descargar el repositorio y resolverlo localmente en el editor de código.
  + Después de resolver el conflicto, realizad un commit para registrar la resolución.
* **Fusión final de develop en main:**
  + Una vez resueltos todos los conflictos y que las ramas estén fusionadas, se realizará una fusión de la rama develop en main. Para ello será necesario abrir el repositorio en GitHub y abrir un **Pull Request** desde vuestra rama (develop) hacia la rama main.
  + La Pull Request deberá ser validada por el otro miembro del grupo.
  + Una vez aceptada, se realizará automáticamente el merge de develop en main.
* **Etiqueta de versión:**
  + Al iniciar el proyecto será obligatoria hacer un tag inicial del proyecto en la rama main. Nombre v0.0
  + Una vez completada la fusión y subida la versión final a main, añadid una etiqueta (tag) para marcar la versión del código entregado. Nombre v1.0

# Criterios de evaluación del proyecto

| **Bloque** | **Elemento evaluado** | **Puntos** | **Criterios de evaluación** |
| --- | --- | --- | --- |
| **Análisis del sistema y Modelado UML** | Diagrama de casos de uso | 1,5 | - Todos los actores identificados. - Casos de uso correctos según los requisitos funcionales. - Relaciones adecuadas (inclusión, extensión, etc). - Representación gráfica clara y sin errores. |
|  | Diagrama de clases | 1,5 | - Clases principales bien identificadas. - Atributos y métodos correctos según el enunciado. - Relaciones correctas (asociaciones, composiciones, agregaciones, etc).  - Multiplicidades bien definidas. - Visibilidades bien definidas. |
| **Desarrollo en Eclipse** | Generación de código | 1,0 | - Código generado correctamente desde Modelio. - Importado sin errores a Eclipse. - Estructura del proyecto organizada (paquetes, carpetas). - Código compilable y adaptado. |
| **Análisis de calidad** | Complejidad ciclomática | 1,0 | - Diagrama de flujo de control correcto. - Cálculo V(G) bien hecho. - Caminos básicos bien definidos. |
|  | Tabla combinada de caminos y pruebas | 1,0 | - Correspondencia clara entre caminos y casos de prueba. - Datos de entrada y resultados esperados razonables. - Pruebas alineadas con caminos. |
|  | Automatización pruebas unitarias con JUnit | 2,0 | - Uso de JUnit correcto (JUnit5 preferido). - Cobertura del 100% del método objetivo (canjearCanciones). - Uso de anotaciones (@Test, @BeforeEach, etc.). - Pruebas bien justificadas. Evidencia incluida (captura). |
| **Control de versiones del proyecto** | Repositorio compartido en GitHub | 0,25 | - Repositorio creado y compartido correctamente. - Acceso del profesor garantizado. |
|  | Gestión de contenido y estructura | 0,75 | - Carpeta organizada según especificación (src, diagramas, documentación, etc.). - Commits significativos y con autoría clara. - Subida correcta de todo el contenido. |
|  | Fusión de ramas y resolución de conflictos | 0,5 | - Uso correcto de ramas feature\_alumnoX. - Uso de develop como rama de integración. - Conflicto generado y resuelto correctamente. - Evidencia incluida. |
|  | Fusión final a main y etiquetado de versiones | 0,5 | - Fusión develop → main mediante Pull Request. - Validación cruzada del compañero. - Tag inicial (v0.0) y final (v1.0) creados y subidos correctamente. |