Práctica Final “Examinator 3000”

11/12/2025

—

Álvaro Rico Refoyo
DNI: 70976790D
Curso: 2º Ingeniería Informática 25/26
Asignatura: Programación III - Universidad De Salamanca

ÍNDICE

1.Introducción

Este proyecto implementa un sistema completo de gestión de preguntas tipo test desarrollado en Java siguiendo el patrón de diseño Modelo-Vista-Controlador (MVC). El objetivo es proporcionar una aplicación modular, extensible y fácil de mantener, capaz de gestionar un banco de preguntas, realizar operaciones CRUD (crear, listar, modificar y eliminar), importar y exportar datos en formato JSON, generar preguntas automáticamente y ejecutar exámenes simulados, todo además de guardar la información entre ejecuciones.

La aplicación se ejecuta en modo consola. El sistema permite crear preguntas con varios temas, autor, enunciado y cuatro opciones de respuesta, garantizando que solo una de ellas sea correcta. Además, incluye herramientas para consultar el banco de preguntas, filtrar por tema, visualizar el detalle de cada pregunta y modificar cualquiera de sus atributos.

La práctica integra un mecanismo de persistencia basado en dos métodos complementarios:

- **Exportación e importación en JSON**, utilizada para realizar copias de seguridad o restaurar conjuntos de preguntas, evitando duplicados mediante identificadores únicos (UUID).

- **Repositorio binario**, encargado de almacenar automáticamente o al salir el estado del banco de preguntas.

Como funcionalidad adicional, el sistema incorpora un generador automático de preguntas mediante gemini, así como un modo examen que permite seleccionar un número de preguntas, filtrar por tema y obtener un resultado final con aciertos, errores y preguntas no respondidas.

Este informe presenta las distintas funcionalidades implementadas, acompañadas de capturas y explicaciones técnicas que demuestran el correcto funcionamiento del software y el cumplimiento del enunciado. No se incluye código fuente más que el que se va ha explicar de manera clara y estructurada de manera que actúa como manual técnico para cualquier desarrollador que desee entender, ejecutar o ampliar el sistema.

2.Arquitectura del Programa

La aplicación ha sido desarrollada siguiendo el patrón arquitectónico **Modelo–Vista–Controlador (MVC)**, lo que permite separar de manera clara la lógica de negocio, la interacción con el usuario y la gestión interna de los datos. Esta división favorece la mantenibilidad, extensibilidad y robustez del software, además de facilitar que cada componente pueda evolucionar de manera independiente.

2.1. Patrón MVC aplicado al proyecto

Modelo (Model)

El modelo es el responsable de gestionar el banco de preguntas y de aplicar toda la lógica interna de la aplicación.
Incluye:

- La representación de las entidades (*Question*, *Option*, *ExamResult..*).
- Los mecanismos de validación y modificación de preguntas.
- La persistencia mediante repositorio binario.
- La importación y exportación en formato JSON.
- La generación automática de preguntas a través de *QuestionCreator*.

Es la parte del sistema que contiene la lógica del negocio y que trabaja directamente con los datos.

Vista (View)

La vista corresponde a la interfaz de usuario basada en consola.
Su función es:

- Mostrar menús.
- Solicitar información.
- Presentar resultados.
- Mostrar errores y mensajes informativos.

No realiza cálculos ni gestiona datos: simplemente comunica al usuario con el controlador, asegurando que la experiencia de interacción sea clara y estructurada.

Controlador (Controller)

El controlador actúa como intermediario entre la vista y el modelo.
Sus funciones principales son:

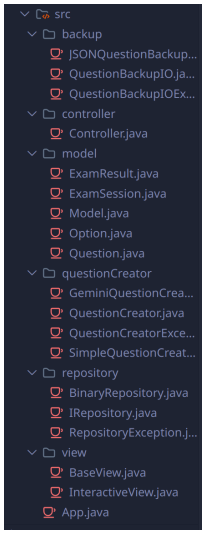
- Recibir solicitudes de la vista.
- Invocar al modelo para ejecutar la lógica correspondiente.
- Devolver los resultados para que la vista los muestre.
- Coordinar el ciclo de ejecución completo de la aplicación.

Gracias al controlador, la vista permanece independiente del modelo.

2.2. Estructura de paquetes

El proyecto se organiza en un conjunto de paquetes coherentes que facilitan la comprensión del sistema y un desarrollo más ordenado:

- **backup/** → gestiona exportación e importación en JSON.
- **controller/** → contiene la lógica de coordinación de la aplicación.
- **model/** → incluye las clases del dominio y los componentes de negocio.
- **questionCreator/** → contiene los generadores automáticos de preguntas.
- **repository/** → gestiona la persistencia binaria de datos.
- **view/** → contiene la interfaz interactiva del usuario



Esta organización permite que cada módulo se mantenga independiente y modular.

2.3. Ciclo de vida de la aplicación

El ciclo de ejecución se desarrolla de la siguiente manera:

1. **Inicio del programa**
    - Se instancia el modelo, el controlador y la vista.
    - Se carga el banco de preguntas desde el repositorio binario (si existe).
2. **Ejecución interactiva**
    - El usuario navega por los menús.
    - La vista solicita datos y envía las peticiones al controlador.
    - El controlador coordina las operaciones con el modelo.
    - El modelo valida, procesa y devuelve resultados.
    - La vista los muestra en pantalla.
3. **Finalización**
    - El modelo guarda automáticamente o al cerrar el estado actualizado del banco de preguntas.
    - La aplicación muestra un mensaje de cierre correctamente finalizado.

Este ciclo asegura una interacción estable, segura y coherente con los datos almacenados.

2.4. ¿Por qué en Inglés?

La aplicación ha sido desarrollada utilizando el inglés como idioma principal en la interfaz, los nombres de clases, métodos y variables (El contenido del banco de preguntas y las preguntas generadas por gemini no tienen porque). Esta decisión se basa en varios motivos:

1. **El inglés es el idioma universal de la programación.**
   La mayoría de lenguajes, librerías, documentación técnica y comunidades profesionales emplean el inglés como estándar. Utilizarlo en el código facilita la lectura, comprensión y mantenibilidad del proyecto en un entorno técnico global.

2. **Mejora la interoperabilidad con código de terceros.**
   Al estar sometidos continuamente a bibliotecas, frameworks y repositorios escritos en inglés, mantener la coherencia de idioma en el código propio reduce la dificultad y permite una mejor integración y revisión por parte de otros programadores.

3. **Favorece la profesionalización y la práctica realista.**
   Acostumbrarse a programar en inglés es muy útil para el futuro profesional, tanto en la universidad como en cualquier trabajo relacionado con tecnología. Por eso, este proyecto también sirve como práctica para adoptar un estilo de programación más cercano al que se usa en la industria.

3.Funcionalidades del Sistema

En esta sección se describen de forma detallada todas las funcionalidades implementadas en la aplicación. Para cada una de ellas se presentan dos perspectivas complementarias:

1. **Experiencia de usuario:**
   Se explica cómo interactúa el usuario con la función desde la interfaz de consola, qué información recibe en pantalla y cuál es el flujo habitual de uso. Esta parte está acompañada de capturas reales de la ejecución del programa para ilustrar el comportamiento del sistema, es lo más parecido a una guía de usuario.

2. **Funcionamiento técnico:**
   Una vez mostrada la interacción práctica, se expone el mecanismo interno que hace posible cada funcionalidad.
   Aquí se incluyen fragmentos de código relevantes y una explicación clara de cómo intervienen el Modelo, la Vista y el Controlador en cada proceso. Esta presentación permite comprender la lógica implementada sin repetir demasiado contenido ya disponible en los archivos entregados.

Siguiendo este enfoque, cada subapartado presenta primero la visión operativa del usuario (qué ve y qué sucede), y posteriormente la explicación técnica del funcionamiento interno (cómo se implementa y qué responsabilidades asume cada componente del MVC).

3.1. Funcionalidad CRUD de preguntas

La aplicación implementa un conjunto completo de operaciones CRUD (Create, Read, Update, Delete) que permiten al usuario gestionar el banco de preguntas directamente desde la consola. Estas funcionalidades garantizan que el usuario pueda mantener actualizado y organizado su conjunto de preguntas.

A continuación se describen cada una de las opciones del menú CRUD acompañadas de capturas de la ejecución.

3.1.1 Crear una nueva pregunta

Esta funcionalidad permite introducir una pregunta completa en el sistema. El usuario debe introducir:

- Autor
- Enunciado
- Lista de temas (separados por comas)
- Cuatro opciones de respuesta
- Índice de la opción correcta

El sistema valida que solo una opción sea correcta y almacena la pregunta tanto en el modelo interno como en el repositorio binario.

3.1.1.1 Ejemplo de uso:

```
============================================================
★  MAIN MENU
============================================================
Navigate with numbers and press ENTER
📝  1. CRUD (Questions)
📦  2. Import / Export
🤖  3. Automatic Question Creation
🧠  4. Exam Mode
------------------------------------------------------------
🚪  0. Exit
Select an option:  (0 <= numero <= 4) 1
```

```
============================================================
★  CRUD MENU
============================================================
Manage your bank of questions
➕  1. Create new question
📋  2. List questions
------------------------------------------------------------
🔙 0. Back to main menu
Select an option:  (0 <= numero <= 2) 1
```

```
✓ === Create New Question ===
Enter author: [Autor]
Enter question statement: [Enunciado]
Enter topics (comma separated): [Tema1,Tema2...]

--- Option 1 ---
Enter option text: [Opcion 1]
Enter option rationale: [Explicacion]

--- Option 2 ---
Enter option text: [Opcion2]
Enter option rationale: [Explicacion]

--- Option 3 ---
Enter option text: [Opcion 3]
Enter option rationale: [Explicacion]

--- Option 4 ---
Enter option text: [Opcion 4]
Enter option rationale: [Explicacion]

Which option is correct? (1-4):  (1 <= numero <= 4) [Opcion correcta (1-4)]
```

Parámetros de la pregunta:

1.Enter author: [Autor]

Corresponde al **nombre del autor** o creador de la pregunta.

Ejemplo:
[Autor] = "Álvaro Rico"

2.Enter question statement: [Enunciado]

Es el **texto principal de la pregunta**, es decir, lo que se le plantea al usuario durante un examen o visualización.

Ejemplo:
[Enunciado] = "¿Cuál es la capital de Francia?"

3.Enter topics (comma separated): [Tema1,Tema2...]

Representa los **temas asociados a la pregunta**, separados por comas.
Cada tema se almacena en mayúsculas para garantizar uniformidad al filtrar.

Ejemplo:
[Tema1,Tema2...] = "GEOGRAFÍA, EUROPA"

4.Enter option text: [Opcion X]

Para cada una de las cuatro opciones, el usuario introduce el **texto de la posible respuesta**.

Ejemplos:

- [Opción 1] = "París"
- [Opción 2] = "Roma"
- [Opción 3] = "Berlín"
- [Opción 4] = "Madrid"

5.Enter option rationale: [Explicación]

Cada opción debe incluir una **breve explicación o justificación** que permita entender por qué esa respuesta es correcta o incorrecta.
Este campo es útil en modo estudio o revisión.

Ejemplo:
[Explicación] = "París es la capital oficial de Francia desde 508 d.C."

6. Which option is correct? (1–4): [Opción correcta (1–4)]

Finalmente, el usuario indica **cuál de las cuatro opciones es la respuesta correcta**.
Debe ser un número entero entre 1 y 4.

Ejemplo:
[Opción correcta] = 1

3.1.1.2 Código que implementa esta función

Vista:

El menú CRUD permite gestionar el banco de preguntas del sistema. Al acceder, la interfaz limpia la pantalla, muestra un encabezado claro y presenta tres opciones: crear una nueva pregunta, listar las existentes o volver al menú principal.

El usuario selecciona una opción y la vista recoge esa elección para enviarla al controlador, que se encarga de ejecutar la acción correspondiente. Si la entrada no es válida, se muestra un mensaje de error y se solicita una nueva selección.

```
private void showCRUDMenu() {
    clearScreen();
    printHeader("CRUD MENU");
    pulseMessage("Manage your bank of questions", GREEN, 2);
    printMenuItem(1, "Create new question", "➕", CYAN);
    printMenuItem(2, "List questions", "📋", CYAN);
    printDivider();
    System.out.println(colorize("🔙 0. Back to main menu", RED));
}


private void optionCRUD() {
    boolean back = false;
    while (!back) {
        showCRUDMenu();
        int option = Esdia.readInt("Select an option: ", 0, 2);
        switch (option) {
            case 1 -> createQuestion();
            case 2 -> listQuestions();
            case 0 -> back = true;
            default -> showErrorMessage("Invalid option.");
        }
    }
}
```

Modelo:

El modelo es el encargado de llevar a cabo todo el proceso real de creación de la pregunta. Para ello, ejecuta varias etapas internas:

1. Validación de campos básicos (fnc: `validateBaseFields()`)

Antes de crear la pregunta, el modelo verifica:

- Que el nombre del autor no esté vacío.
- Que el enunciado de la pregunta tenga contenido.
- Que exista al menos un tema asociado.
- Que ninguno de los temas sea una cadena vacía.

Si alguno de los datos no es válido, se genera una excepción indicando el error exacto. Esta validación garantiza que todas las preguntas creadas tengan una estructura mínima coherente.

2.Validación de las opciones de respuesta (fnc: `validateOptions()`)

El modelo comprueba que:

- Se han introducido exactamente cuatro opciones.
- Todas las opciones contienen texto.
- Todas las explicaciones (rationales) están presentes.
- El índice de la opción correcta esté entre 1 y 4.

Si alguna de estas condiciones no se cumple, se lanza una excepción para evitar preguntas incompletas o inconsistentes.

3.Construcción interna de las opciones (fnc: `createQuestion()`)

Una vez validados los datos, el modelo transforma las cuatro opciones introducidas por el usuario en objetos internos que incluyen:

- El texto de la respuesta.
- La explicación.
- Un indicador que identifica cuál es la respuesta correcta.

Este paso convierte la entrada textual del usuario en una estructura estándar que será usada posteriormente en los exámenes, listados y exportaciones.

4.Creación del objeto Pregunta (fnc: `createQuestion()`)

Con las opciones ya procesadas, el modelo construye la pregunta final, asociando:

- Autor
- Enunciado
- Lista de temas
- Lista de opciones
- Identificador único (UUID)

Este objeto queda listo para ser gestionado en cualquier funcionalidad del sistema.

5.Persistencia en el repositorio (fnc: `persistIfNeeded()`) **(Explicada más adelante)**

La nueva pregunta se añade al repositorio configurado (ya sea en formato binario o JSON). Posteriormente, el modelo ejecuta acciones internas como:

- Actualizar la caché en memoria.
- Guardar automáticamente los datos si la configuración lo requiere.

Gracias a este mecanismo, la información está siempre sincronizada y lista para ser recuperada o exportada.

```
public Question createQuestion(
        String author,
        String statement,
        Set<String> topics,
        List<String> optionTexts,
        List<String> optionRationales,
        int correctIndex
 ) throws RepositoryException {
    validateBaseFields(author, statement, topics);
    validateOptions(optionTexts, optionRationales, correctIndex);

    List<Option> options = new ArrayList<>();
    for (int i = 0; i < optionTexts.size(); i++) {
        boolean isCorrect = (i + 1 == correctIndex);
        options.add(new Option(optionTexts.get(i), optionRationales.get(i), isCorrect));
    }

    Question q = new Question(author, statement, topics, options);
    repository.addQuestion(q);
    refreshCache();
    persistIfNeeded();
    return q;
 }
```

```
   private void validateOptions(List<String> texts, List<String> rationales, int
correctIndex)
     throws RepositoryException {
       if (texts == null || rationales == null || texts.size() != 4 || rationales.size() !=
4){
           throw new RepositoryException("Questions must have exactly 4 options.");
       }
       if (correctIndex < 1 || correctIndex > 4) {
           throw new RepositoryException("Correct option must be between 1 and 4.");
       }
       for (String t : texts) {
           if (t == null || t.trim().isEmpty()) {
               throw new RepositoryException("Option text cannot be empty.");
           }
       }
       for (String r : rationales) {
           if (r == null || r.trim().isEmpty()) {
               throw new RepositoryException("Option rationale cannot be empty.");
           }
       }
   }

   private void validateBaseFields(String author, String statement, Set<String> topics)
throws
  RepositoryException {
       if (author == null || author.trim().isEmpty()) {
           throw new RepositoryException("Author cannot be empty.");
       }
       if (statement == null || statement.trim().isEmpty()) {
           throw new RepositoryException("Statement cannot be empty.");
       }
       if (topics == null || topics.isEmpty()) {
           throw new RepositoryException("Topics cannot be empty.");
       }
       for (String t : topics) {
           if (t == null || t.trim().isEmpty()) {
               throw new RepositoryException("Topics cannot contain empty values.");
           }
       }
   }
```

Controlador:

El controlador no realiza ninguna lógica propia, sino que simplemente recibe los datos provenientes de la vista y los envía directamente al modelo.

```
   public void createQuestion(
       String author,
       String statement,
       Set<String> topics,
       List<String> optionTexts,
       List<String> optionRationales,
       int correctIndex
   ) throws RepositoryException {
       model.createQuestion(author, statement, topics, optionTexts, optionRationales,
correctIndex);
   }
```

Clases distintas de mvc involucradas :

- Option: Cada posible respuesta de la pregunta, con su texto, justificación y marca de si es la correcta.

- Question: Objeto que agrupa enunciado, autor, temas, fecha y las 4 opciones de respuesta (con 1 correcta) para una pregunta tipo test

3.1.2 Listar preguntas

La funcionalidad List Questions permite al usuario consultar de forma organizada todas las preguntas almacenadas en el banco del sistema. Desde este módulo, el usuario puede elegir entre visualizar el conjunto completo de preguntas o filtrar el listado por un tema específico. Esta herramienta facilita localizar rápidamente una pregunta concreta, revisar su contenido o acceder a sus detalles para modificarla o eliminarla.

3.1.2.1 Ejemplo de uso:

```
============================================================
★  MAIN MENU
============================================================
Navigate with numbers and press ENTER
📝  1. CRUD (Questions)
📦  2. Import / Export
🤖  3. Automatic Question Creation
🧠  4. Exam Mode
------------------------------------------------------------
🚪 0. Exit
Select an option:  (0 <= numero <= 4) 1
```

```
============================================================
★  CRUD MENU
============================================================
Manage your bank of questions
➕  1. Create new question
📋  2. List questions
------------------------------------------------------------
🔙 0. Back to main menu
Select an option:  (0 <= numero <= 2) 2
```

```
============================================================
★  LIST QUESTIONS
============================================================
Choose how you want to see the questions
🔶 Questions loaded: 7                                    🔶
📚  1. List all questions
🎯  2. List questions by topic
------------------------------------------------------------
🔙 0. Back to CRUD menu
Select an option:  (0 <= numero <= 2) [1/2]
```

SI el usuario introduce 1 se listan todas las preguntas existentes en el banco de preguntas

```
--- Questions ---
1. [Enunciado de la pregunta] [2025-12-06T11:39:49.102035837]

Select a question to view details (0 to cancel): 1
```

Si por el contrario el usuario hubiera introducido 2 se pregunta qué temas quiere listar y se listan todas las preguntas que contengan ese tema.

```
Available topics:
1. TEMA2..]
2. [TEMA1
Select topic:  (1 <= numero <= 2) 1
```

```
--- Questions ---
1. [Enunciado de la pregunta] [2025-12-06T11:39:49.102035837]

Select a question to view details (0 to cancel):
```

3.1.2.2 Código que implementa esta función

Vista:

1. Menú CRUD (fnc: `showCRUDMenu()`)

La vista muestra un menú claro y estructurado donde el usuario puede seleccionar entre crear una nueva pregunta, listar las existentes o volver al menú principal.
Este menú se refresca en cada iteración para mantener una experiencia de uso ordenada e intuitiva.

2. Navegación del usuario (fnc: `optionCRUD()`)

La función que gestiona la opción CRUD mantiene un bucle que recoge la selección del usuario y ejecuta la acción correspondiente.
El objetivo es permitir una navegación fluida: cada selección redirige a la función adecuada y cualquier entrada incorrecta genera un mensaje de error informativo.

3. Crear una pregunta (fnc: `createQuestion()`)

Cuando el usuario elige crear una pregunta, la vista solicita los datos necesarios de forma guiada:

- Autor
- Enunciado
- Temas separados por comas
- Cuatro opciones con su explicación
- Identificación de la opción correcta

Una vez completado el formulario, la vista envía los datos al controlador para que el modelo cree la pregunta.

4. Listar preguntas (fnc `listQuestions()`)

La vista permite consultar el banco de preguntas mediante dos opciones: listado completo o filtrado por tema.
Una vez generado el listado, el usuario puede seleccionar una pregunta para ver sus detalles completos o volver atrás.
Esta funcionalidad facilita revisar el contenido almacenado antes de modificarlo o eliminarlo.

```
private void showCRUDMenu() {

    clearScreen();
    printHeader("CRUD MENU");
    pulseMessage("Manage your bank of questions", GREEN, 2);
    printMenuItem(1, "Create new question", "➕", CYAN);
    printMenuItem(2, "List questions", "📋", CYAN);
    printDivider();
    System.out.println(colorize("🔙 0. Back to main menu", RED));
}


private void optionCRUD() {

    boolean back = false;

    while (!back) {
        showCRUDMenu();

        int option = Esdia.readInt("Select an option: ", 0, 2);
```

```
            switch (option) {
                case 1 -> createQuestion();
                case 2 -> listQuestions();
                case 0 -> back = true;
                default -> showErrorMessage("Invalid option.");
            }
        }
    }
```

```
    private void createQuestion() {

        clearScreen();
        showMessage("=== Create New Question ===");

        String author = readNonEmptyString("Enter author: ");

        String statement = readNonEmptyString("Enter question statement: ");


        String topicsInput = readNonEmptyString("Enter topics (comma separated): ");


        HashSet<String> topics = new HashSet<>();
        for (String t : topicsInput.split(",")) {
            topics.add(t.trim().toUpperCase());
        }

        List<String> optionTexts = new ArrayList<>();
        List<String> optionRationales = new ArrayList<>();

        for (int i = 1; i <= 4; i++) {
            System.out.println("\n--- Option " + i + " ---");


            String optText = readNonEmptyString("Enter option text: ");
            optionTexts.add(optText);

            String optRat = readNonEmptyString("Enter option rationale: ");
            optionRationales.add(optRat);
        }


        int correctIndex = Esdia.readInt("\nWhich option is correct? (1-4): ", 1, 4);

        try {
            controller.createQuestion(author, statement, topics, optionTexts,
optionRationales, correctIndex);
            showMessage("Question created successfully.");
```

```
        } catch (Exception e) {
            showErrorMessage("Could not create question: " + e.getMessage());
        }
    }
```

```
private void listQuestions() {

    clearScreen();

    printHeader("LIST QUESTIONS");
    pulseMessage("Choose how you want to see the questions", MAGENTA, 2);
    renderStatusBar("Questions loaded: " + controller.getAllQuestions().size(), "");
    printMenuItem(1, "List all questions", "📚", CYAN);
    printMenuItem(2, "List questions by topic", "🎯", CYAN);
    printDivider();
    System.out.println(colorize("🔙 0. Back to CRUD menu", RED));

    int choice = Esdia.readInt(    "Select an option: ",0,2);

    List<Question> questions = new ArrayList<>();

    try {
        if (choice == 1) {


            questions = controller.getAllQuestions();

        } else if (choice == 2) {


            String topic = chooseTopic(false);
            if (topic == null) return;

            questions = controller.getQuestionsByTopic(topic);

        } else if (choice == 0) {
            return;
        }else{

            showErrorMessage("Invalid option.");
            return;
        }


        if (questions.isEmpty()) {
            showMessage("No questions found.");
            return;
        }


        System.out.println("\n--- Questions ---");
```

```
        int index = 1;
        for (Question q : questions) {
            System.out.println(index + ". " + q.getStatement() + " [" +
q.getCreationDate() + "]");
            index++;
        }


        int selected = Esdia.readInt("\nSelect a question to view details (0 to cancel):
");

        if (selected == 0) return;

        if (selected < 1 || selected > questions.size()) {
            showErrorMessage("Invalid selection.");
            return;
        }


        viewQuestionDetail(questions.get(selected - 1));

    } catch (Exception e) {
        showErrorMessage("Could not list questions: " + e.getMessage());
    }
    }
```

Controlador:

El controlador proporciona a la vista una forma sencilla de acceder al banco de preguntas sin que esta tenga que comunicarse directamente con el modelo. Su función es simplemente recibir la petición de la vista y delegarla en el modelo, manteniendo así la separación del patrón MVC.

- **Obtener todas las preguntas (fnc `getAllQuestions()`):**
  El controlador solicita al modelo el listado completo ya ordenado y lo devuelve a la vista para su presentación.

- **Obtener preguntas por tema (fnc `getQuestionsByTopic()`):**
  Cuando la vista pide solo las preguntas de un tema concreto, el controlador pasa ese tema al modelo, que realiza el filtrado y devuelve únicamente las que corresponden.

```java
public List<Question> getAllQuestions() {
    return model.getAllQuestionsSorted();
}


public List<Question> getQuestionsByTopic(String topic) {
    return model.getQuestionsByTopic(topic);
}
```

Modelo:

El modelo es el encargado de gestionar y organizar los datos del sistema. En el contexto de la consulta de preguntas, ofrece funciones que permiten recuperar el banco de preguntas de forma ordenada y filtrada según lo que necesite la vista. Estas funciones no interactúan con el usuario directamente, sino que proporcionan al controlador los datos ya preparados.

1. Obtener todas las preguntas ordenadas (fnc: `getAllQuestionsSorted()`)

El modelo devuelve la lista completa de preguntas ordenada por fecha de creación.
Esto garantiza que los listados mostrados al usuario mantengan un criterio consistente y que las preguntas más nuevas o más antiguas aparecen siempre en el mismo orden.

2. Obtener preguntas filtradas por tema (fnc: `getQuestionsByTopic()`)

Cuando se solicita un tema concreto, el modelo filtra internamente la lista de preguntas y devuelve solo aquellas que pertenecen al tema indicado.
Además, igual que en el caso anterior, el resultado se devuelve ordenado por fecha, facilitando su lectura en la vista.

3. Obtener todos los temas disponibles (fnc: `getAvailableTopics()`)

El modelo proporciona un conjunto con todos los temas existentes en el banco de preguntas.

```java
public List<Question> getAllQuestionsSorted() {
    return questions.stream()
            .sorted(Comparator.comparing(Question::getCreationDate))
            .collect(Collectors.toList());
}

public List<Question> getQuestionsByTopic(String topic) {
```

```
        return questions.stream()
                .filter(q -> q.getTopics().contains(topic.toUpperCase()))
                .sorted(Comparator.comparing(Question::getCreationDate))
                .collect(Collectors.toList());
    }

    public Set<String> getAvailableTopics() {
        Set<String> topics = new HashSet<>();
        for (Question q : questions) {
            topics.addAll(q.getTopics());
        }
        return topics;
    }
```

Clases distintas de mvc involucradas :

- <u>Question</u>: Objeto que agrupa enunciado, autor, temas, fecha y las 4 opciones de respuesta (con 1 correcta) para una pregunta tipo test, se listan objetos de este tipo

3.1.3 Modificar una pregunta

La aplicación permite modificar cualquier atributo de una pregunta existente, excepto su identificador único. Esta funcionalidad se inicia desde la vista de detalle, donde el usuario selecciona la opción de editar. A partir de ese momento, la interfaz ofrece un menú específico que permite actualizar el autor, los temas, el enunciado o las opciones de respuesta.

Cada modificación se realiza de manera guiada: la vista solicita los nuevos valores, valida que no estén vacíos y envía la información al controlador. Este, a su vez, delega la actualización en el modelo, que modifica directamente los atributos de la pregunta y mantiene la consistencia del banco.

3.1.3.1 Ejemplo de uso:

```
--- Questions ---
1. [Enunciado de la pregunta] [2025-12-06T11:39:49.102035837]

Select a question to view details (0 to cancel):
```

Partimos del menú Questions al que se llega como lo vimos en **Listar Preguntas**

```
============================================================
★  QUESTION DETAIL 🔍=
============================================================
ID: 28124640-534e-4c95-83fb-cc4fed8a5920
Author: Alvaro
Created: 2025-12-06T11:39:49.102035837
Topics: [[TEMA1, TEMA2..]]
Statement: [Enunciado de la pregunta]

Options:
1. [Opcion 1]
  Rationale: [Justificacion 1]
  Correct: true
2. [Opcion 2]
  Rationale: [Justificacion 2]
  Correct: false
3. [Opcion 3]
  Rationale: [justificacion 3]
  Correct: false
4. [Opcion 4]
```

```
  Rationale: [Justificacion 4]
  Correct: false
------------------------------------------------------------
AVAILABLE ACTIONS
1. Modify this question
2. Delete this question
0. Back
Select an option:  (0 <= numero <= 2)
```

```
============================================================
★  MODIFY QUESTION ✏
============================================================
1. Modify author
2. Modify topics
3. Modify statement
4. Modify options
0. Back
Select an option:  (0 <= numero <= 4)
```

```
Select an option:  (0 <= numero <= 4)1
Enter new author: [Nuevo Autor]
```

```
Select an option:  (0 <= numero <= 4)2
Enter new topics (comma separated): [Nuevo Tema 1],[Nuevo Tema 2]
```

```
Select an option:  (0 <= numero <= 4)3
Enter new statement: [Nuevo Enunciado]
```

```
Select an option:  (0 <= numero <= 4)4

--- Option 1 ---
Enter option text: [Nueva opcion 1]
Enter option rationale: [Nueva justificacion 1]

--- Option 2 ---
Enter option text: [Nueva Opcion 2]
Enter option rationale: [Nueva justificacion 2]

--- Option 3 ---
```

```
Enter option text: [Nueva opcion 3]
Enter option rationale: [Nueva justificacion 3]
--- Option 4 ---
Enter option text: [Nueva opcion 4]
Enter option rationale: [Nueva justificacion 4]
Which option is correct? (1-4): 1
```

1. Enter new author: [Nuevo Autor]

Es el **nuevo autor de la pregunta**, es decir, la persona que lo modifica.

Ejemplo:
[Nuevo Autor] = "Álvaro"

2. Enter new topics (comma separated): [Nuevo Tema 1],[Nuevo Tema 2]

Son los **nuevos temas de la pregunta**, si hay más de uno separados por comas

Ejemplo:
[Nuevo Tema 1] = "Ciencia"

[Nuevo Tema 2] = "Biología"

3. Enter new statement: [Nuevo Enunciado]

Es el **nuevo enunciado de la pregunta**,.

Ejemplo:
[Nuevo Enunciado] = "¿Cuál es el planeta más grande del sistema solar?"

4. Enter option text: [Nueva Opción x]

Es una **nueva opción de la pregunta**.

Ejemplo:
[Nueva Opción x] = "Tierra"

5.Enter option rationale: [Nueva justificación x]

Es la nueva explicación **de la pregunta**.

Ejemplo:
[Nuevo Justificación x] = "No, es la 3º más grande"

3.1.3.2 Código que implementa esta función

Vista:

La vista ofrece un conjunto de herramientas que permiten al usuario consultar en profundidad el contenido de una pregunta y modificar cualquiera de sus elementos de forma interactiva.

1. Visualización del detalle de una pregunta (fnc: `viewQuestionDetail()`)

Cuando el usuario selecciona una pregunta desde un listado, la vista presenta una pantalla de detalle que muestra toda la información relevante:

- Identificador único (UUID)
- Autor
- Fecha de creación
- Temas asociados
- Enunciado
- Las cuatro opciones de respuesta, junto con su explicación y marcadas como correctas o incorrecta.

Esta presentación completa facilita revisar la estructura de la pregunta antes de editarla o eliminarla.

Además, desde esta misma pantalla se ofrece un pequeño menú de acciones que permite modificar la pregunta o eliminarla del banco de datos.

2. Modificación de una pregunta (fnc: `modifyQuestion()`)

La vista integra un editor interactivo que permite actualizar cualquier atributo de la pregunta, excepto su ID. El usuario puede escoger entre varias opciones:

- Modificar el autor
- Actualizar los temas, introducidos nuevamente como lista separada por comas
- Cambiar el enunciado
- Redefinir por completo las cuatro opciones, incluyendo textos y justificaciones

En cada uno de estos apartados, la vista solicita los nuevos datos mediante formularios simples y valida que ningún campo esté vacío. Una vez recogida la información, la vista la envía al controlador, que se encarga de actualizar el modelo.

3. Edición de las opciones de respuesta (fncs: `modifyAuthor()`, `modifyTopics()`, `modifyStatement()`, `modifyOptions()`)

Durante la modificación de opciones, el usuario vuelve a introducir los textos y justificaciones de las cuatro respuestas. Finalmente, selecciona cuál de ellas debe marcarse como correcta.
Este proceso asegura que la pregunta mantenga siempre el formato exigido: cuatro opciones y una única respuesta válida.

```
private void modifyAuthor(Question question) {

    String newAuthor = readNonEmptyString("Enter new author: ");

    try {
        controller.modifyAuthor(question, newAuthor);
        showMessage("Author updated successfully.");
    } catch (Exception e) {
        showErrorMessage("Error updating author: " + e.getMessage());
    }
}

private void modifyTopics(Question question) {

    String input = readNonEmptyString("Enter new topics (comma separated): ");

    HashSet<String> newTopics = new HashSet<>();

    for (String t : input.split(",")) {
        newTopics.add(t.trim().toUpperCase());
    }

    try {
        controller.modifyTopics(question, newTopics);
        showMessage("Topics updated successfully.");
    } catch (Exception e) {
        showErrorMessage("Error updating topics: " + e.getMessage());
    }
}

private void modifyStatement(Question question) {

    String newStatement = readNonEmptyString("Enter new statement: ");

    try {
```

```
            controller.modifyStatement(question, newStatement);
            showMessage("Statement updated successfully.");
        } catch (Exception e) {
            showErrorMessage("Error updating statement: " + e.getMessage());
        }
    }


    private void modifyOptions(Question question) {

        List<String> newTexts = new ArrayList<>();
        List<String> newRationales = new ArrayList<>();

        for (int i = 1; i <= 4; i++) {
            System.out.println("\n--- Option " + i + " ---");

            newTexts.add(readNonEmptyString("Enter option text: "));

            newRationales.add(readNonEmptyString("Enter option rationale: "));
        }

        int correctIndex = -1;

        while (correctIndex < 1 || correctIndex > 4) {
                correctIndex = Esdia.readInt("Which option is correct? (1-4): ");
        }

        try {
            controller.modifyOptions(question, newTexts, newRationales, correctIndex);
            showMessage("Options updated successfully.");
        } catch (Exception e) {
            showErrorMessage("Error updating options: " + e.getMessage());
        }
    }


}
```

Controlador:

El controlador actúa como intermediario entre la vista y el modelo, garantizando que cualquier modificación solicitada por el usuario se procese de manera correcta y coherente. La vista nunca altera directamente los datos; en su lugar, delega en el controlador, que valida, centraliza y coordina la actualización de la información.

A continuación, se describen las funciones relacionadas con la modificación de preguntas.

1. Modificar el autor de una pregunta (fnc: `modifyAuthor()`)

Cuando el usuario introduce un nuevo autor desde la vista, el controlador recibe este valor y lo transmite al modelo. De esta manera, asegura que la actualización se realice siguiendo las reglas del sistema y gestionando cualquier excepción derivada de datos inválidos.

2. Modificar los temas asociados (fnc: `modifyTopics()`)

El controlador también se encarga de gestionar la actualización de los temas de una pregunta. La vista proporciona una lista normalizada de temas, y el controlador la envía al modelo para sustituir los valores anteriores. El modelo valida que no haya temas vacíos y mantiene la consistencia interna del banco de preguntas.

3. Modificar el enunciado (fnc: `modifyStatement()`)

Cuando el usuario cambia el enunciado, el controlador transmite el nuevo texto al modelo, que se encarga de comprobar que no esté vacío y actualizarlo en la pregunta correspondiente. Esto mantiene la separación de responsabilidades entre interacción (vista) y lógica de negocio (modelo).

4. Modificar las opciones de respuesta (fnc: `modifyOptions()`)

La modificación de opciones es la operación más completa:
el usuario introduce nuevos textos, justificaciones y la opción correcta.
El controlador envía estos elementos al modelo, que reconstruye la lista de opciones, valida que haya exactamente cuatro y que una sola sea correcta.

De este modo, el controlador garantiza que la estructura obligatoria de las preguntas se mantenga y que cualquier error sea notificado adecuadamente.

```
  public void modifyAuthor(Question q, String newAuthor) throws RepositoryException {
       model.modifyAuthor(q, newAuthor);
   }

   public void modifyTopics(Question q, Set<String> newTopics) throws
RepositoryException {
       model.modifyTopics(q, newTopics);
   }

   public void modifyStatement(Question q, String newStatement) throws
RepositoryException {
       model.modifyStatement(q, newStatement);
   }
```

```
public void modifyOptions(
        Question q,
        List<String> texts,
        List<String> rationales,
        int correctIndex
) throws RepositoryException {
    model.modifyOptions(q, texts, rationales, correctIndex);
}
```

Modelo:

El modelo es la capa encargada de gestionar la lógica de negocio y garantizar la coherencia interna del banco de preguntas. A diferencia del controlador, que se limita a coordinar operaciones, el modelo valida, actualiza y persiste los datos, asegurando que cualquier modificación cumpla las reglas establecidas por el sistema.

Esta sección describe cómo el modelo gestiona cada operación de modificación.

1. Modificar el autor (fnc: `modifyAuthor()`)

El modelo verifica que el nuevo autor no sea una cadena vacía ni nula.
Si los datos son válidos, actualiza el campo correspondiente en la pregunta y delega la persistencia en el repositorio. Después, refresca la caché interna para que los cambios se reflejen inmediatamente.

Esta validación evita inconsistencias que podrían dañar la integridad del banco de preguntas.

2. Modificar los temas (fnc: `modifyTopics()`)

Al actualizar los temas, el modelo reutiliza su propio sistema de validación de campos base. Comprueba que el conjunto de temas no esté vacío y que ninguno de ellos sea nulo o contenga solo espacios.

Una vez validados, los nuevos temas se asignan a la pregunta y el modelo persiste los cambios.
Este proceso garantiza que una pregunta nunca quede asociada a un conjunto de temas inválido.

3. Modificar el enunciado (fnc: `modifyStatement()`)

Para cambiar el enunciado, el modelo verifica que el texto no esté vacío.
Si la validación se supera, se actualiza el enunciado y se registra la modificación en el repositorio.

Al igual que en los casos anteriores, el modelo asegura que nunca haya enunciados nulos, cumpliendo los requisitos mínimos de coherencia definidos en la práctica.

4. Modificar las opciones de respuesta (fnc: `modifyOptions()`)

Esta es la operación más compleja.
El modelo valida que:

- existan exactamente cuatro opciones,
- todas tengan texto y justificación,
- la opción correcta esté dentro del rango permitido.

Tras superar esta validación estricta, el modelo reconstruye completamente la lista de opciones de la pregunta, marcando una única opción como correcta.
Después persiste los cambios y actualiza la caché interna.

Este enfoque asegura que la estructura de las preguntas se mantenga siempre uniforme y que cualquier modificación preserve la integridad del formato exigido.

5.Persistencia en el repositorio (fnc: `persistIfNeeded()`) **(Explicada más adelante)**

La nueva pregunta modificada se sustituye a la anterior en el repositorio configurado (ya sea en formato binario o JSON).
Posteriormente, el modelo ejecuta acciones internas como:

- Actualizar la caché en memoria.
- Guardar automáticamente los datos si la configuración lo requiere.

Gracias a este mecanismo, la información está siempre sincronizada y lista para ser recuperada o exportada.

```
  public void modifyAuthor(Question q, String newAuthor) throws RepositoryException {
       if (newAuthor == null || newAuthor.trim().isEmpty()) {
           throw new RepositoryException("Author cannot be empty.");
       }
       q.setAuthor(newAuthor);
       repository.modifyQuestion(q);
       refreshCache();
       persistIfNeeded();
   }

   public void modifyTopics(Question q, Set<String> newTopics) throws
RepositoryException {
       validateBaseFields(q.getAuthor(), q.getStatement(), newTopics);
       q.setTopics(newTopics);
```

```
        repository.modifyQuestion(q);
        refreshCache();
        persistIfNeeded();
    }

    public void modifyStatement(Question q, String newStatement) throws
RepositoryException {
        if (newStatement == null || newStatement.trim().isEmpty()) {
            throw new RepositoryException("Statement cannot be empty.");
        }
        q.setStatement(newStatement);
        repository.modifyQuestion(q);
        refreshCache();
        persistIfNeeded();
    }

    public void modifyOptions(
            Question q,
            List<String> texts,
            List<String> rationales,
            int correctIndex
    ) throws RepositoryException {
        validateOptions(texts, rationales, correctIndex);

        List<Option> newOptions = new ArrayList<>();
        for (int i = 0; i < texts.size(); i++) {
            boolean isCorrect = (i + 1 == correctIndex);
            newOptions.add(new Option(texts.get(i), rationales.get(i), isCorrect));
        }
        q.setOptions(newOptions);
        repository.modifyQuestion(q);
        refreshCache();
        persistIfNeeded();
    }
```

Clases distintas de mvc involucradas :

- Option: Cada posible respuesta de la pregunta, con su texto, justificación y marca de si es la correcta.

- Question: Objeto que agrupa enunciado, autor, temas, fecha y las 4 opciones de respuesta (con 1 correcta) para una pregunta tipo test

3.1.4 Eliminar una pregunta

La aplicación permite eliminar cualquier pregunta del banco de datos de forma directa desde la vista de detalle. Esta funcionalidad es esencial para mantener el banco limpio y actualizado, especialmente cuando se detectan preguntas obsoletas, incorrectas o duplicadas.

El proceso de eliminación sigue el flujo estándar del patrón MVC:
la vista muestra la pregunta seleccionada, el usuario confirma su decisión y el controlador delega la operación en el modelo, que finalmente la elimina del repositorio y actualiza el estado interno del sistema.

La operación se ejecuta de manera segura, mostrando mensajes informativos y gestionando posibles errores (por ejemplo, intentar borrar una pregunta inexistente).

3.1.4.1 Ejemplo de uso:

Partimos del menú Questions al que se llega como lo vimos en **Listar Preguntas**

```
--- Questions ---
1. [Enunciado de la pregunta] [2025-12-06T11:39:49.102035837]

Select a question to view details (0 to cancel):1
```

```
============================================================
★  QUESTION DETAIL 🔍=
============================================================
ID: 28124640-534e-4c95-83fb-cc4fed8a5920
Author: [Nuevo Autor]
Created: 2025-12-06T11:39:49.102035837
Topics: [[NUEVO TEMA 1], [NUEVO TEMA 2]]
Statement: [Nuevo Enunciado]

Options:
1. [Opcion 1]
  Rationale: [Justificacion 1]
  Correct: true
2. [Opcion 2]
  Rationale: [Justificacion 2]
  Correct: false
3. [Opcion 3]
  Rationale: [Justificacion 3]
  Correct: false
4. [Opcion 4]
```

```
  Rationale: [Justificacion 4]
  Correct: false
------------------------------------------------------------
AVAILABLE ACTIONS
1. Modify this question
2. Delete this question
0. Back
Select an option:  (0 <= numero <= 2) 2
```

Al introducir la opción 2 la pregunta seleccionada se borrará

3.1.4.2 Código que implementa esta función

Vista:

Dentro de la vista, la eliminación de una pregunta se gestiona desde la pantalla de detalle. Cuando el usuario selecciona una pregunta desde la lista, la interfaz muestra toda su información y ofrece un menú de acciones disponibles.

Si el usuario elige la opción de eliminar, la vista llama al controlador para ejecutar la acción correspondiente. La vista solo muestra datos y recoge la interacción del usuario, mientras que la lógica de modificación del sistema queda completamente delegada en otras capas.

Tras enviar la solicitud al controlador, la vista informa al usuario del resultado, ya sea confirmando que la pregunta se ha eliminado correctamente o mostrando un mensaje de error en caso de que algo falle.

```java
   private void viewQuestionDetail(Question question) {

       clearScreen();

       printHeader("QUESTION DETAIL 🔍");

       System.out.println("ID: " + question.getId());
       System.out.println("Author: " + question.getAuthor());
       System.out.println("Created: " + question.getCreationDate());
       System.out.println("Topics: " + question.getTopics());
       System.out.println("Statement: " + question.getStatement());

       System.out.println("\nOptions:");
       int i = 1;
       for (Option op : question.getOptions()) {
           System.out.println(i + ". " + op.getText());
           System.out.println("   Rationale: " + op.getRationale());
           System.out.println("   Correct: " + op.isCorrect());
           i++;
       }
```

```
        printDivider();
        System.out.println(colorize("AVAILABLE ACTIONS", BOLD + YELLOW));
        System.out.println("1. Modify this question");
        System.out.println("2. Delete this question");
        System.out.println("0. Back");

        int choice = Esdia.readInt("Select an option: ",0,2);

        switch (choice) {

            case 1 -> {
                modifyQuestion(question);
            }

            case 2 -> {
                try {
                    controller.deleteQuestion(question);
                    showMessage("Question deleted successfully.");
                } catch (Exception e) {
                    showErrorMessage("Error deleting question: " + e.getMessage());
                }
            }

            case 0 -> {
                return;
            }

            default -> showErrorMessage("Invalid option.");
        }
    }
```

Controlador:

El controlador actúa como intermediario entre la vista y el modelo durante el proceso de eliminación. Cuando la vista solicita borrar una pregunta, el controlador recibe el objeto correspondiente y delega la operación directamente en el modelo, que es el responsable final de realizar la eliminación en el repositorio.

Para garantizar la estabilidad del sistema, esta función está envuelta en un bloque de gestión de excepciones. Si el modelo detecta algún problema como inconsistencias o intentos de borrar una pregunta inexistente lanza una excepción que el controlador captura y reenvía a la vista como un mensaje de error claro para el usuario.

Con este diseño, el controlador mantiene una ejecución segura, evita fallos inesperados y asegura que la vista siempre reciba una respuesta informativa, tanto si la operación se completa con éxito como si se produce algún error.

```
    public void deleteQuestion(Question q) {
        try {
            model.deleteQuestion(q);
        } catch (RepositoryException e) {
            view.showErrorMessage(e.getMessage());
        }
    }
```

Modelo:

En el modelo reside la lógica central encargada de eliminar una pregunta del sistema. Cuando el controlador solicita esta operación, el modelo delega la eliminación en el repositorio correspondiente, el cual gestiona el almacenamiento persistente de los datos.

Una vez eliminada la pregunta del repositorio, el modelo actualiza su caché interna para asegurar que el estado en memoria refleje correctamente los cambios aplicados. Finalmente, ejecuta un guardado automático si la configuración lo requiere, garantizando que la modificación quede registrada de forma permanente.

Si durante este proceso se produce algún problema, por ejemplo, un fallo de acceso al repositorio o una inconsistencia en los dato, el método lanza una `RepositoryException`. Esta excepción se propaga al controlador para que la vista pueda informar al usuario de manera clara y segura.

Este diseño asegura que la eliminación sea coherente, fiable y totalmente alineada con el flujo de persistencia del sistema.

```
public void deleteQuestion(Question q) throws RepositoryException {
        repository.removeQuestion(q);
        refreshCache();
        persistIfNeeded();
    }
```

Clases distintas de mvc involucradas :

- Question: Objeto que agrupa enunciado, autor, temas, fecha y las 4 opciones de respuesta (con 1 correcta) para una pregunta tipo test

3.2 Importar/Exportar en JSON

El sistema incorpora un mecanismo completo de importación y exportación en formato JSON que permite gestionar copias de seguridad y facilitar el intercambio de datos entre diferentes ejecuciones o equipos. Esta funcionalidad es esencial para preservar el banco de preguntas, restaurarlo en cualquier momento y trabajar con conjuntos de datos consistentes y fácilmente transportables.

La exportación genera un archivo JSON con todas las preguntas almacenadas, incluyendo sus opciones, autor, temas, fecha de creación y metadatos relevantes. Por su parte, la importación permite cargar un fichero previamente generado, añadiendo únicamente aquellas preguntas que no estén ya presentes en el sistema, gracias al uso de identificadores únicos (UUID).

3.2.1 Exportación

La funcionalidad de exportación permite generar un archivo JSON que contiene todo el banco de preguntas almacenado en el sistema. Este archivo actúa como una copia de seguridad completa y portable, ideal para preservar los datos, compartirlos o transferirlos a otra ejecución del programa.

Durante el proceso de exportación, el modelo recopila todas las preguntas, las convierte en una estructura JSON mediante la librería Gson y las guarda en el directorio del usuario siguiendo el formato especificado en el enunciado. El resultado es un fichero legible, estructurado y compatible con futuras importaciones.

Esta operación garantiza que cualquier modificación realizada durante la sesión quede reflejada en un archivo persistente, manteniendo la integridad del banco de preguntas y facilitando su mantenimiento o reutilización

3.2.1.1 Ejemplo de uso:

```
============================================================
★  MAIN MENU
============================================================
Navigate with numbers and press ENTER
📝  1. CRUD (Questions)
📦  2. Import / Export
🤖  3. Automatic Question Creation
🧠  4. Exam Mode
------------------------------------------------------------
🚪 0. Exit
Select an option:  (0 <= numero <= 4) 1
```

```
============================================================
★  IMPORT / EXPORT
============================================================
Backup your work or restore it
🟠 Backup: JSON                                          🟠
------------------------------------------------------------
✦ Choose backup action ✦
------------------------------------------------------------
⬆  1. Export questions to JSON
⬇  2. Import questions from JSON
------------------------------------------------------------
🔙 0. Back to main menu
Select an option:  (0 <= numero <= 2) 1
```

```
Enter filename to import (from your home): [Nombre del archivo]
```

[Nombre del archivo] es el nombre del archivo json sin la extensión (.json), si no se pone nada se le asigna **backup.json**

3.2.1.2 Código que implementa esta función:

Vista:

La función **`optionImportExport()`** gestiona el menú dedicado a las operaciones de copia de seguridad y restauración de preguntas en formato JSON. Esta sección de la vista permite al usuario exportar su banco de preguntas a un archivo externo o importar preguntas previamente guardadas, actuando como puente entre el usuario y las funciones de persistencia del modelo.

Al acceder a esta opción, la interfaz limpia la pantalla y presenta un menú visual mejorado que incluye encabezados, animaciones y mensajes de ayuda, facilitando una navegación clara incluso en modo consola. El usuario dispone de dos acciones principales:

1. **Exportar preguntas a JSON**
   Guarda todo el banco de preguntas en un archivo JSON legible, útil para realizar copias de seguridad o transportar datos a otra máquina.

2. **Importar preguntas desde JSON**
   Explicado en el siguiente apartado: **Importación con control de duplicados por UUID**

La función mantiene al usuario dentro de este menú hasta que seleccione explícitamente la opción Back, gestionando errores básicos (como opciones inválidas) desde la propia vista, mientras delega la lógica real de importación/exportación al controlador.

El método **`exportQuestions()`** gestiona toda la interacción del usuario durante el proceso de exportación. Primero solicita el nombre del archivo donde se guardará la copia de seguridad y muestra una animación de progreso para indicar que la operación está en curso. Una vez obtenido el nombre, delega la tarea real en el Controlador, que a su vez coordina la exportación con el Modelo.

Si la operación finaliza correctamente, la Vista informa al usuario con un mensaje de éxito. En caso de error (por ejemplo, problemas de escritura o nombres inválidos), se muestra el mensaje correspondiente y se espera a que el usuario pulse ENTER antes de regresar al menú.

Este método mantiene la experiencia de usuario fluida mientras el resto del proceso se ejecuta en capas inferiores.

```
private void optionImportExport() {

    boolean back = false;

    while (!back) {
        clearScreen();
        // -- Menú Import/Export --
        printHeader("IMPORT / EXPORT");
        pulseMessage("Backup your work or restore it", BLUE, 2);
        renderStatusBar("Backup: JSON", "");
        animateSectionTransition("Choose backup action");
        printMenuItem(1, "Export questions to JSON", "⬆️", CYAN);
        printMenuItem(2, "Import questions from JSON", "⬇️", CYAN);
        printDivider();
        System.out.println(colorize("🔙 0. Back to main menu", RED));

        int option = Esdia.readInt("Select an option: ", 0, 2);

        switch (option) {

            case 1 -> exportQuestions();

            case 2 -> importQuestions();

              case 0 -> back = true;

              default -> showErrorMessage("Invalid option.");
         }}}
```

```
    private void exportQuestions() {
        try {
            String filename = Esdia.readString("Enter filename (stored in your
home): ").trim();
            animateProgress("Exporting");
            controller.exportQuestions(filename);
            showMessage("Questions exported successfully.");
        } catch (Exception e) {
            showErrorMessage("Export failed: " + e.getMessage());
            Esdia.readString("Press ENTER to continue.");
        }
    }
```

Controlador:

El método **`exportQuestions()`** simplemente recibe el nombre del fichero indicado por el usuario y delega la exportación en el Modelo. El Controlador no realiza comprobaciones adicionales, manteniendo su función como coordinador entre Vista y Modelo. Si ocurre un error durante la escritura del archivo JSON, la excepción se envía de vuelta para que la Vista pueda gestionarla y mostrar el mensaje correspondiente.

```
    public void exportQuestions(String fileName) throws
QuestionBackupIOException {
        model.exportQuestions(fileName);
    }
```

Modelo:

El método **`exportQuestions()`** se encarga de generar un archivo JSON con todas las preguntas almacenadas en el sistema. Para ello, delega la escritura en `backupHandler`, el componente responsable de transformar la lista de preguntas en un fichero válido.

Antes de exportar, el nombre del archivo se normaliza mediante **normalizeFileName()** para garantizar que se utilice una ruta correcta o un nombre por defecto si el usuario no especifica ninguno.

El método **`normalizeFileName()`** garantiza que el proceso de exportación o importación siempre utilice un nombre de archivo válido. Si el usuario no introduce ningún nombre o deja el campo vacío, el sistema asigna automáticamente `"backup.json"` como nombre por defecto. Este mecanismo evita errores comunes y asegura que las operaciones de lectura y escritura se realicen de forma consistente y segura.

```
    public void exportQuestions(String fileName) throws
QuestionBackupIOException {
        backupHandler.exportQuestions(questions, normalizeFileName(fileName));
    }
```

```
    private String normalizeFileName(String fileName) {
        if (fileName == null || fileName.isBlank()) {
            return "backup.json";
        }
        return fileName;
    }
```

3.2.2 Importación con control de duplicados por UUID

El sistema incluye un proceso de importación diseñado para garantizar la integridad del banco de preguntas. Durante esta operación, el fichero JSON seleccionado se analiza y se transforman sus datos en objetos de tipo Question. Sin embargo, antes de añadir cada pregunta al repositorio, el sistema aplica un control estricto de duplicados basado en el identificador único (UUID) asociado a cada una de ellas.

Este UUID actúa como huella inequívoca de cada pregunta. Cuando el modelo detecta que el JSON contiene una pregunta cuyo UUID coincide con una ya existente en el repositorio, dicha entrada simplemente se ignora. De este modo, se evita la creación de copias redundantes, se preserva la coherencia de los datos y se respeta el principio de unicidad definido en el enunciado.

Gracias a este mecanismo, la importación es segura y predecible: permite recuperar preguntas previamente exportadas, incorporar nuevas sin alterar las existentes y evitar conflictos derivados de modificaciones o duplicaciones accidentales. Este flujo asegura un banco de preguntas siempre consistente, incluso tras múltiples operaciones de importación.

3.2.2.1 Ejemplo de uso:

```
============================================================
★  MAIN MENU
============================================================
Navigate with numbers and press ENTER
📝  1. CRUD (Questions)
📦  2. Import / Export
🤖  3. Automatic Question Creation
🧠  4. Exam Mode
------------------------------------------------------------
🚪  0. Exit
Select an option:  (0 <= numero <= 4) 1
```

```
============================================================
★  IMPORT / EXPORT
============================================================
Backup your work or restore it
🟧 Backup: JSON                                          🟧
------------------------------------------------------------
✦ Choose backup action ✦
------------------------------------------------------------
⬆  1. Export questions to JSON
⬇  2. Import questions from JSON
------------------------------------------------------------
🔙 0. Back to main menu
Select an option:  (0 <= numero <= 2) 2
```

```
Enter filename to import (from your home): [Nombre del archivo]
```

[Nombre del archivo] es el nombre del archivo json sin la extensión (.json), si no se pone nada se le asigna **backup.json**

3.2.2.2 Código que implementa esta función

Vista:

La función `optionImportExport()` gestiona el menú dedicado a las operaciones de copia de seguridad y restauración de preguntas en formato JSON. Esta sección de la vista permite al usuario exportar su banco de preguntas a un archivo externo o importar preguntas previamente guardadas, actuando como puente entre el usuario y las funciones de persistencia del modelo.

Al acceder a esta opción, la interfaz limpia la pantalla y presenta un menú visual mejorado que incluye encabezados, animaciones y mensajes de ayuda, facilitando una navegación clara incluso en modo consola. El usuario dispone de dos acciones principales:

3. **Exportar preguntas a JSON**
   Explicado en el apartado anterior: **Exportación**

4. **Importar preguntas desde JSON**
   Permite cargar preguntas desde un archivo externo. La aplicación incorpora control automático de duplicados mediante UUID, garantizando que no se repitan preguntas en el sistema.

El método `importQuestions()` gestiona la interacción con el usuario durante la importación de datos. Primero solicita el nombre del archivo JSON a cargar y muestra una animación de progreso para indicar que el proceso está en marcha. A continuación, delega la operación al Controlador, que coordina la validación y carga real del contenido a través del Modelo.

Si la importación se realiza con éxito, la Vista informa al usuario mediante un mensaje de confirmación. En caso de error —ya sea por un archivo inexistente, información inválida o duplicados detectados— se muestra un mensaje de fallo y se espera a que el usuario pulse ENTER antes de volver al menú.

```
   private void optionImportExport() {

       boolean back = false;

       while (!back) {
           clearScreen();
           printHeader("IMPORT / EXPORT");
           pulseMessage("Backup your work or restore it", BLUE, 2);
           renderStatusBar("Backup: JSON", "");
           animateSectionTransition("Choose backup action");
           printMenuItem(1, "Export questions to JSON", "⬆️", CYAN);
           printMenuItem(2, "Import questions from JSON", "⬇️", CYAN);
           printDivider();
           System.out.println(colorize("🔙 0. Back to main menu", RED));

           int option = Esdia.readInt("Select an option: ", 0, 2);

           switch (option) {

               case 1 -> exportQuestions();

               case 2 -> importQuestions();

               case 0 -> back = true;

               default -> showErrorMessage("Invalid option.");
           }
       }
   }
```

```
   private void importQuestions() {
             try {
           String filename = Esdia.readString("Enter filename to import (from your
home): ").trim();
           animateProgress("Importing");
           controller.importQuestions(filename);
           showMessage("Questions imported successfully.");
       } catch (Exception e) {
           showErrorMessage("Import failed: " + e.getMessage());
           Esdia.readString("Press ENTER to continue.");
       }
   }
```

Controlador:

El método **importQuestions()** actúa como intermediario entre la Vista y el Modelo durante la importación de preguntas desde un archivo JSON. Su responsabilidad es mínima: recibe el nombre del archivo introducido por el usuario y delega directamente la operación en el Modelo, que es quien realiza la lectura, validación y almacenamiento de los datos.

```
   public void importQuestions(String fileName) throws QuestionBackupIOException,
RepositoryException {
        model.importQuestions(fileName);
   }
```

Modelo:

La función **importQuestions()** se encarga de leer un archivo JSON, validar su contenido y actualizar el banco de preguntas sin comprometer la integridad del sistema.

Primero, el archivo indicado por el usuario es procesado por el componente encargado de la copia de seguridad, obteniendo una lista de preguntas externas. Antes de incorporarlas al repositorio, el Modelo aplica una política estricta: si una sola pregunta importada contiene datos inválidos, la importación completa se cancela, garantizando que nunca se mezcle información corrupta con datos válidos.

Una vez validadas, cada pregunta se compara con las ya existentes mediante su identificador único (UUID). Esto permite detectar duplicados de forma fiable. Solo aquellas preguntas que no estén registradas previamente se añaden al repositorio.

Después de completar las inserciones, el Modelo actualiza su caché interna y ejecuta el mecanismo de persistencia binaria automática para asegurar que los cambios queden guardados en disco. Este enfoque garantiza que la importación sea segura, coherente y totalmente transparente para el usuario.

El método **normalizeFileName()** garantiza que el proceso de exportación o importación siempre utilice un nombre de archivo válido. Si el usuario no introduce ningún nombre o deja el campo vacío, el sistema asigna automáticamente `"backup.json"` como nombre por defecto.

Este mecanismo evita errores comunes y asegura que las operaciones de lectura y escritura se realicen de forma consistente y segura.

```
public void importQuestions(String fileName) throws QuestionBackupIOException,
RepositoryException {
       List<Question> imported =
backupHandler.importQuestions(normalizeFileName(fileName));
       for (Question q : imported) {
           validateImportedQuestion(q);
       }
       for (Question q : imported) {
           boolean exists = questions.stream()
                   .anyMatch(x -> x.getId().equals(q.getId()));
           if (!exists) {
               repository.addQuestion(q);
           }
       }
       refreshCache();
       persistIfNeeded();
   }
```

```
   private String normalizeFileName(String fileName) {
       if (fileName == null || fileName.isBlank()) {
           return "backup.json";
       }
       return fileName;
   }
```

Clases distintas de mvc involucradas :

- Option: Cada posible respuesta de la pregunta, con su texto, justificación y marca de si es la correcta.

- Question: Objeto que agrupa enunciado, autor, temas, fecha y las 4 opciones de respuesta (con 1 correcta) para una pregunta tipo test

- QuestionBackupIO: Define la estructura general para cualquier sistema encargado de importar y exportar preguntas.

- JSONQuestionBackupIO: Implementa la importación y exportación de preguntas en formato JSON utilizando la librería Gson.

3.3. Persistencia binaria automática

El sistema incorpora un mecanismo de persistencia binaria que garantiza que cualquier cambio en el banco de preguntas quede almacenado de forma automática sin intervención del usuario. Este proceso se ejecuta cada vez que se modifica el repositorio interno (creación, edición, eliminación o importación de preguntas).

La persistencia se gestiona a través del repositorio binario del Modelo, que escribe los datos estructurados en un archivo binario privado. De esta forma, el estado del sistema se conserva entre ejecuciones, evitando pérdidas de información y asegurando una carga rápida al iniciar la aplicación.

3.3.1 Carga inicial

Al iniciar la aplicación, el Modelo intenta restaurar automáticamente el estado previo del banco de preguntas mediante la lectura del archivo binario de persistencia. Este proceso carga en memoria todas las preguntas previamente almacenadas, permitiendo que el usuario continúe trabajando exactamente donde lo dejó en la sesión anterior.

Si el archivo binario no existe (por ejemplo, en la primera ejecución), el sistema simplemente arranca con un banco vacío, creando el archivo más adelante cuando se produzcan las primeras modificaciones.

3.3.1.1 Ejemplo de uso

```
✓ Loaded 0 questions from storage. Backup handler: JSON backup in user
home directory
============================================================
★  WELCOME
============================================================
Navigate with numbers and press ENTER
Do you want to autosave after each operation? (y/n):  (y,n) ?
```

Informa de las preguntas cargadas desde la persistencia al arranque del programa

3.3.1.2 Código que implementa esta función

Modelo:

Para garantizar que el banco de preguntas se conserva entre ejecuciones, el sistema incorpora un mecanismo de carga automática durante la inicialización del modelo. El método `createDefault()` cosas configura el Modelo con un repositorio binario (`BinaryRepository`) responsable de recuperar desde disco todas las preguntas almacenadas previamente en el archivo ***questions.bin***.

De este modo, al arrancar la aplicación:

1. **Se crea el repositorio binario**, que inmediatamente intenta leer el archivo persistido.

2. **Las preguntas almacenadas en ejecuciones anteriores se reconstruyen en memoria**, conservando su ID, autor, temas, opciones y fecha de creación.

3. **El Modelo queda listo para operar** con los datos cargados, evitando que el usuario tenga que importar o recrear manualmente sus preguntas.

El constructor principal del Modelo es el encargado de recuperar automáticamente todas las preguntas almacenadas en el sistema. Al recibir un repositorio binario y un gestor de copias JSON, el Modelo se inicializa cargando en memoria todas las preguntas persistidas previamente.

Durante la construcción:

1. **Se asignan los componentes de persistencia**:

    - El repositorio binario, responsable de guardar y recuperar datos.
    - El gestor JSON, utilizado para importaciones y exportaciones.

2. **Se registran los generadores automáticos de preguntas**, si existen.

3. **Se cargan las preguntas guardadas**, copiándolas directamente desde el repositorio mediante `repository.getAllQuestions()`.
   Esto reconstruye en memoria el estado exacto del banco de preguntas antes de cerrar la aplicación.

```
   public static Model createDefault(List<QuestionCreator> questionCreators)
throws RepositoryException {
       IRepository repository = new BinaryRepository("questions.bin");
       QuestionBackupIO backupHandler = new JSONQuestionBackupIO();
       return new Model(repository, backupHandler, questionCreators);
}
```

```
   public Model(IRepository repository, QuestionBackupIO backupHandler,
List<QuestionCreator> questionCreators) throws RepositoryException {
       this.repository = repository;
       this.backupHandler = backupHandler;
       this.questionCreators = questionCreators != null ? questionCreators :
new ArrayList<>();
       this.questions = new ArrayList<>(repository.getAllQuestions());
   }
```

Controlador:

La función `start()` Muestra un mensaje inicial con información del sistema, incluyendo:

- La cantidad total de preguntas cargadas desde la persistencia.
- El tipo de mecanismo de copia de seguridad configurado (por ejemplo, JSON).

Este mensaje funciona como una verificación de arranque que confirma al usuario que los datos se han recuperado correctamente.

Lanza el método `init()` de la vista, que abre el menú principal y comienza la interacción.

```
   public void start() {
       if (view != null) {
           view.showMessage("Loaded " + model.getQuestionCount() + " questions
from storage. Backup handler: " + model.getBackupDescription());
           view.init();
       }
   }
```

3.3.2 Guardado tras modificaciones

Cada vez que el usuario realiza una operación que modifica el banco de preguntas —como crear, editar, eliminar o importar nuevas preguntas— el sistema activa automáticamente el mecanismo de persistencia binaria.

Tras aplicar el cambio en el repositorio, el Modelo ejecuta `persistIfNeeded()`, que guarda el estado actualizado en el archivo binario correspondiente. Este proceso es transparente para el usuario y garantiza que todas las modificaciones queden almacenadas de forma segura sin necesidad de acciones manuales.

3.3.2.1 Ejemplo de uso

```
✓ Loaded 0 questions from storage. Backup handler: JSON backup in user
home directory
============================================================
★  WELCOME
============================================================
Navigate with numbers and press ENTER
Do you want to autosave after each operation? (y/n):  (y,n) ?
```

3.3.2.2 Código que implementa esta función

Al comenzar la ejecución, la vista presenta una pantalla de bienvenida donde el usuario recibe instrucciones básicas de navegación. En este punto, se ofrece una decisión importante: **activar o desactivar el guardado automático**.

El sistema pregunta al usuario si desea que todas las operaciones —crear, modificar o eliminar preguntas— se guarden automáticamente en el repositorio binario.

- Si el usuario responde *Sí*, el controlador activa el modo *autosave*, garantizando que cada cambio se persista de inmediato.

- Si responde *No*, la aplicación trabajará en modo manual, informando claramente que **el guardado solo se realizará al salir de la aplicación**, momento en el que se invocará la persistencia global del estado.

El método `persistState()` del controlador se encarga de asegurar que todos los cambios realizados en el banco de preguntas queden guardados de forma permanente. Su función es actuar como intermediario entre la vista y el modelo, delegando en este último la lógica de persistencia real.

Cuando es llamado:

1. **Solicita al modelo que guarde el estado actual**, utilizando el mecanismo de persistencia configurado (habitualmente el repositorio binario).
   Esto incluye nuevas preguntas, modificaciones o eliminaciones.

2. **Controla posibles errores de escritura**, como archivos inaccesibles o datos corruptos.
   Si la operación falla, informa a la vista para que muestre un mensaje de error claro al usuario.

Cuando el usuario decide salir del sistema, la vista ejecuta el método `end()`, encargado de cerrar la aplicación de forma controlada. Antes de finalizar, se solicita al controlador que realice la operación de persistencia del estado. Esto garantiza que cualquier cambio pendiente —especialmente si el modo *autosave* no estaba activado— quede almacenado de manera definitiva en el repositorio binario.

La vista encapsula esta llamada dentro de un bloque `try/catch` para asegurar que, incluso si ocurre un error en el guardado, el cierre de la aplicación no se interrumpe. Tras ello, se muestran los mensajes de despedida que confirman al usuario que el programa se ha cerrado correctamente.

Cuando el usuario decide salir del sistema, la vista ejecuta el método `end()`, encargado de cerrar la aplicación de forma controlada. Antes de finalizar, se solicita al controlador que realice la operación de persistencia del estado. Esto garantiza que cualquier cambio pendiente —especialmente si el modo *autosave* no estaba activado— quede almacenado de manera definitiva en el repositorio binario.

La vista encapsula esta llamada dentro de un bloque `try/catch` para asegurar que, incluso si ocurre un error en el guardado, el cierre de la aplicación no se interrumpe. Tras ello, se muestran los mensajes de despedida que confirman al usuario que el programa se ha cerrado correctamente.

a responsabilidad de guardar permanentemente el conjunto de preguntas recae en el método `persistState()` del modelo. Este método delega la operación en el repositorio binario mediante `saveAll()`, asegurando que la colección actual de preguntas se escriba en el archivo de almacenamiento.

Al lanzar **`RepositoryException`** en caso de error, la clase respeta la separación de responsabilidades:

- **El modelo** realiza la operación técnica de persistencia.
- **El controlador** decide cómo reaccionar ante un fallo (por ejemplo, mostrando un mensaje al usuario).

```
        printHeader("WELCOME");
        pulseMessage("Navigate with numbers and press ENTER", CYAN, 3);
        boolean autoSave = Esdia.yesOrNo("Do you want to autosave after each operation?
(y/n): ");
        controller.setAutoSave(autoSave);
        if (!autoSave) {
            showMessage("El guardado se realizará al salir de la aplicación.");
        }
```

```
    public void persistState() {
        try {
            model.persistState();
        } catch (RepositoryException e) {
            if (view != null) {
                view.showErrorMessage("Error saving data: " + e.getMessage());
            }
        }
    }
```

```
    @Override
    public void end(){

    try {
        controller.persistState();
    } catch (Exception ignored) {
        // handled in controller
    }
    showMessage("Closing application...");
    showMessage("Goodbye!");
    return;
    }
```

```
    public void persistState() throws RepositoryException {
        repository.saveAll(questions);
    }
```

```java
public void setAutoSave(boolean autoSave) {
    this.autoSave = autoSave;
}
```

```java
private void persistIfNeeded() throws RepositoryException {
    if (autoSave) {
        persistState();
    }
}
```

Clases distintas de mvc involucradas :

- Option: Cada posible respuesta de la pregunta, con su texto, justificación y marca de si es la correcta.

- Question: Objeto que agrupa enunciado, autor, temas, fecha y las 4 opciones de respuesta (con 1 correcta) para una pregunta tipo test

- **BinaryRepository:** Gestiona la persistencia automática de preguntas almacenándolas y recuperándolas en formato binario.

- JSONQuestionBackupIO: Proporciona soporte adicional para copias de seguridad en JSON, complementando al repositorio binario.

3.4 Generación automática de preguntas

El sistema incluye un mecanismo de generación automática de preguntas basado en objetos `QuestionCreator`. Estos componentes actúan como "fabricantes" de preguntas, capaces de producir nuevos ítems a partir de un tema seleccionado por el usuario.

Cuando el usuario accede a esta función, la Vista solicita un tema y delega la creación en el Modelo, que selecciona uno de los generadores disponibles. El `QuestionCreator` construye una pregunta completa —autor, enunciado, temas, opciones y justificaciones— aplicando sus propias reglas internas. La pregunta generada se muestra al usuario, quien puede revisarla y decidir si desea añadirla al banco de preguntas.

Este sistema permite ampliar automáticamente el repositorio sin necesidad de redactar cada pregunta manualmente, ofreciendo una funcionalidad útil para pruebas rápidas, prácticas educativas o escenarios donde se requiera un conjunto extenso de preguntas de forma ágil.

3.4.1 Ejemplo de uso

```
============================================================
★  MAIN MENU
============================================================
Navigate with numbers and press ENTER
📝  1. CRUD (Questions)
📦  2. Import / Export
🤖  3. Automatic Question Creation
🧠  4. Exam Mode
------------------------------------------------------------
🚪 0. Exit
Select an option:  (0 <= numero <= 4) 3
```

```
============================================================
★  GEMINI QUESTION CREATION 🤖=
============================================================
Let the AI propose a new question
Available generators:
1. Gemini (gemini-1.5-pro)
2. Gemini (gemini-2.5-flash)
Choose generator:  (1 <= numero <= 2) 1
```

```
Enter topic for the automatic question: [Tema]
```

1.Tema de la pregunta [Tema]: Tema que usará gemini para generar la pregunta

```
============================================================
★  AUTOMATIC QUESTION PREVIEW 🤖=
============================================================
------------------------------------------------------------
Statement:
¿Cuál es el planeta más grande de nuestro Sistema Solar?
------------------------------------------------------------
Topics: [PLANETAS, ASTRONOMÍA, CIENCIA]
Author: Gemini (gemini-2.5-flash)
------------------------------------------------------------
Options:
✔ 1. Júpiter
  Rationale: Júpiter es, con diferencia, el planeta más grande del Sistema
Solar. Su masa es dos veces y media la de todos los demás planetas juntos,
y su volumen podría contener más de 1.300 Tierras.
✖ 2. Saturno
  Rationale: Saturno es el segundo planeta más grande del Sistema Solar,
conocido por sus impresionantes anillos, pero es considerablemente más
pequeño que Júpiter.
✖ 3. Marte
  Rationale: Marte es un planeta rocoso y es significativamente más pequeño
que la Tierra y los gigantes gaseosos. No es el planeta más grande del
Sistema Solar.
✖ 4. Tierra
  Rationale: La Tierra es el quinto planeta más grande del Sistema Solar,
pero es un planeta rocoso y mucho más pequeño que Júpiter, que es un
gigante gaseoso.
------------------------------------------------------------
Do you want to add this question to the database? (Y/N): [Y/N]
```

(Para este ejemplo se ha usado el tema [Tema]=Ciencia)

2.Do you want to add this question to the database? (Y/N) [Y/N] : Confirmación para añadir la pregunta al pulsar Y se añade y N se elimina.

3.4.2 Código que implementa esta función:

Vista:

La función **`optionAutomaticQuestion()`** gestiona toda la interacción con el usuario para crear preguntas automáticas mediante un generador basado en IA.

Primero comprueba si hay generadores disponibles; si no, informa al usuario de que debe iniciar la aplicación con el parámetro adecuado.

Después, muestra la lista de generadores instalados y permite elegir uno. A continuación, solicita un tema y lanza una animación de carga mientras el sistema solicita la pregunta al motor seleccionado.

Cuando la generación finaliza, se muestra una vista previa completa con enunciado y opciones. El usuario puede confirmar si desea guardarla o cancelar la operación.

Si se confirma, la pregunta se envía al controlador para ser añadida al banco de preguntas.

```
   private void optionAutomaticQuestion() {

       clearScreen();

       if (!controller.hasQuestionCreators()) {
           showErrorMessage("There are no automatic question generators available.
Please start the app with -question-creator.");
           Esdia.readString("Press ENTER to continue.");
           return;
       }

       printHeader("GEMINI QUESTION CREATION 🤖");
       pulseMessage("Let the AI propose a new question", CYAN, 2);

       List<String> descriptions = controller.getQuestionCreatorDescriptions();
       System.out.println(colorize("Available generators:", BOLD + CYAN));
       for (int i = 0; i < descriptions.size(); i++) {
           System.out.println(colorize((i + 1) + ". " + descriptions.get(i), CYAN));
       }
       int selectedGenerator = Esdia.readInt("Choose generator: ", 1,
descriptions.size()) - 1;


       String topic = Esdia.readString("Enter topic for the automatic question:
").trim().toUpperCase();

       try {

           AtomicBoolean running = new AtomicBoolean(true);
           Thread spinner = startSpinner("Contacting Gemini", running);
           Question generated;
```

```
            try {
                generated = controller.generateAutomaticQuestion(selectedGenerator,
topic);
            } finally {
                running.set(false);
                try {
                    spinner.join();
                } catch (InterruptedException ignored) {}
            }

            if (generated == null) {
                showErrorMessage("No question could be generated for this topic.");
                return;
            }


            showGeneratedQuestionPreview(generated);


            String confirm;
            while (true) {
                confirm = Esdia.readString("Do you want to add this question to the
database? (Y/N): ").trim().toUpperCase();
                if (confirm.equals("Y") || confirm.equals("N") || confirm.equals("S")) {
                    break;
                }
                showErrorMessage("Please answer Y or N.");
            }

            if (confirm.equals("Y") || confirm.equals("S")) {
                try {
                    controller.addGeneratedQuestion(generated);
                    showMessage("Question added successfully.");
                } catch (Exception ex) {
                    showErrorMessage("Could not save question: " + ex.getMessage());
                }
            } else {
               showMessage("Operation cancelled.");
            }

        } catch (Exception e) {
         showErrorMessage("Could not generate automatic question: " + e.getMessage());
        }
    }
```

Controlador:

El metodo `hasQuestionCreators()` implemente consulta al modelo para verificar si existen generadores automáticos de preguntas configurados. Su función es permitir que la vista determine si la opción de creación automática debe estar disponible o mostrar un mensaje de aviso al usuario en caso contrario.

El metodo `generateAutomaticQuestion()` delega en el modelo la generación automática de una nueva pregunta, indicando qué generador utilizar (según su índice) y el tema solicitado por el usuario. Su único propósito es actuar como intermediario entre la vista y la lógica interna del modelo, trasladando posibles excepciones para que puedan ser gestionadas adecuadamente.

El metodo `addGeneratedQuestion()` envía al modelo la pregunta generada automáticamente para que sea almacenada de forma permanente. Su función es actuar como intermediario entre la vista y la lógica del modelo, garantizando que la pregunta se valide, se persista y quede integrada en el banco de preguntas del sistema.

```
    public boolean hasQuestionCreators() {
        return model.hasQuestionCreators();
    }
```

```
    public void addGeneratedQuestion(Question q) throws RepositoryException {
        model.addGeneratedQuestion(q);
    }
```

```
    public Question generateAutomaticQuestion(int creatorIndex, String topic) throws
QuestionCreatorException {
        return model.generateAutomaticQuestion(creatorIndex, topic);
    }
```

Modelo:

El metodo **`hasQuestionCreators()`** comprueba si existen generadores automáticos de preguntas disponibles en el sistema. Devuelve true cuando la lista de questionCreators no está vacía, permitiendo a la vista habilitar o deshabilitar la funcionalidad de creación automática según la configuración real del modelo.

El metodo **`getQuestionCreatorDescriptions()`** devuelve una lista de descripciones textuales de todos los generadores automáticos disponibles. Cada QuestionCreator proporciona su propia descripción, y el modelo las recopila para que la vista pueda mostrarlas al usuario cuando este debe elegir qué generador utilizar en la creación automática de preguntas.

El metodo **`generateAutomaticQuestion()`** solicita a uno de los generadores automáticos (QuestionCreator) la creación de una nueva pregunta basada en un tema indicado por el usuario. Antes de generarla, valida que el índice seleccionado sea correcto; si no, lanza una excepción. Si es válido, delega la creación en el generador correspondiente, devolviendo una pregunta completamente construida.

El metodo **`addGeneratedQuestion()`** incorpora al sistema una pregunta generada automáticamente, asegurando antes que cumple todas las reglas de validación. Primero verifica que la pregunta no sea nula y comprueba que los campos básicos (autor, enunciado y temas) son correctos. Después valida las opciones, garantizando que existan cuatro, todas con texto y justificación, y una sola marcada como correcta.
Una vez validada, la pregunta se añade al repositorio, se actualiza la caché interna y se ejecuta la persistencia si el autosave está activado. Finalmente, el método devuelve la pregunta almacenada.

```
    public boolean hasQuestionCreators() {
        return !questionCreators.isEmpty();
    }
```

```
    public List<String> getQuestionCreatorDescriptions() {
        return questionCreators.stream()
                .map(QuestionCreator::getQuestionCreatorDescription)
                .collect(Collectors.toList());
}
```

```
    public Question generateAutomaticQuestion(int creatorIndex, String topic) throws
QuestionCreatorException {
        if (creatorIndex < 0 || creatorIndex >= questionCreators.size()) {
            throw new QuestionCreatorException("Invalid generator selected");
        }
        return questionCreators.get(creatorIndex).createQuestion(topic);
    }
```

```
    public Question addGeneratedQuestion(Question q) throws RepositoryException {
        if (q == null) {
            throw new RepositoryException("Generated question is empty.");
        }
        validateBaseFields(q.getAuthor(), q.getStatement(), q.getTopics());
        List<String> texts =
q.getOptions().stream().map(Option::getText).collect(Collectors.toList());
```

```
        List<String> rationales =
q.getOptions().stream().map(Option::getRationale).collect(Collectors.toList());
        int correctIndex = extractSingleCorrectIndex(q.getOptions());
        validateOptions(texts, rationales, correctIndex);

        repository.addQuestion(q);
        refreshCache();
        persistIfNeeded();
        return q;
    }
```

Clases distintas de mvc involucradas :

- Option: Cada posible respuesta de la pregunta, con su texto, justificación y marca de si es la correcta.

- Question: Objeto que agrupa enunciado, autor, temas, fecha y las 4 opciones de respuesta (con 1 correcta) para una pregunta tipo test

- BinaryRepository: Almacena de forma persistente las preguntas generadas automáticamente en un fichero binario.

- JSONQuestionBackupIO: Permite exportar e importar preguntas generadas automáticamente mediante archivos JSON.

- IRepository: Define las operaciones necesarias para guardar, recuperar y actualizar preguntas, incluidas las generadas por IA.

- RepositoryException: Representa errores producidos al intentar guardar o actualizar preguntas generadas automáticamente.

- QuestionCreator: Interfaz que define cómo un generador automático debe construir una pregunta a partir de un tema.

- GeminiQuestionCreator: Implementación específica de `QuestionCreator` que utiliza la API de Gemini para generar preguntas tipo test de forma automática.

3.5. Modo Examen

El modo Examen permite al usuario realizar una prueba simulada utilizando las preguntas almacenadas en el sistema. Al iniciar esta opción, el programa solicita dos datos: el número de preguntas que compondrán el examen y el tema sobre el que deben seleccionarse (o **ALL** para incluir todas).

El sistema valida que existan suficientes preguntas y genera una lista de forma controlada. Cada pregunta se muestra individualmente junto a sus cuatro opciones de respuesta, permitiendo al usuario elegir una opción o dejarla en blanco.

Una vez finalizado el cuestionario, el modelo calcula automáticamente el resultado mediante un objeto **ExamResult**, indicando:

- Aciertos
- Fallos
- Preguntas no respondidas
- Nota final sobre 10

La vista presenta este resumen de forma clara para que el usuario pueda evaluar su rendimiento.

3.4.1 Ejemplo de uso

```
============================================================
★  MAIN MENU
============================================================
Navigate with numbers and press ENTER
📝  1. CRUD (Questions)
📦  2. Import / Export
🤖  3. Automatic Question Creation
🧠  4. Exam Mode
------------------------------------------------------------
🚪  0. Exit
Select an option:  (0 <= numero <= 4)
```

```
============================================================
★  EXAM MODE 🧠=
============================================================
🔶 Backup: JSON backup in user home directory                🔶

Available topics:
1. ASTRONOMÍA
2. CIENCIA
3. PLANETAS
4. ALL
Select topic:  (1 <= numero <= 4)
```

```
Enter the number of questions for the exam (1-1):  (1 <= numero <= 1)
```

```
Question 1 of 1:
¿Cuál es el planeta más grande de nuestro Sistema Solar?
1. Júpiter
2. Saturno
3. Marte
4. Tierra
Select an option (1-4). Press ENTER to leave unanswered:
```

```
★  EXAM SUMMARY
============================================================
Correct answers: 1
Wrong answers:   0
Unanswered:      0
Score: ████████████████████  10,00/10
Time (s):        13
✓ Exam finished. Press ENTER to continue.
```

3.4.2 Código que implementa esta función:

```
private void optionExamMode() {
    clearScreen();
    printHeader("EXAM MODE 🧠");
    renderStatusBar("Backup: " + controller.getBackupDescription(), "");

    if (controller.getQuestionCount() == 0) {
        showErrorMessage("No questions available. Create or import before starting an
exam.");
        Esdia.readString("Press ENTER to continue.");
        return;
    }

    String topic = chooseTopic(true);
    if (topic == null) return;

    int maxQuestions = controller.getMaxQuestionsForTopic(topic);
    if (maxQuestions == 0) {
        showErrorMessage("No questions available for that topic.");
        return;
    }

    int numQuestions = Esdia.readInt("Enter the number of questions for the exam (1-" +
maxQuestions + "): ", 1, maxQuestions);

    ExamSession session;
    try {
        session = controller.configureExam(topic, numQuestions);
    } catch (Exception e) {
        showErrorMessage("Could not prepare exam: " + e.getMessage());
        return;
    }

    printDivider();
    System.out.println(colorize("Exam starting... Good luck!", GREEN));

    List<Question> examQuestions = session.getQuestions();

    for (int i = 0; i < examQuestions.size(); i++) {
        Question q = examQuestions.get(i);

        System.out.println("\nQuestion " + (i + 1) + " of " + examQuestions.size() +
":");
        System.out.println(q.getStatement());

        List<Option> options = q.getOptions();
        for (int j = 0; j < options.size(); j++) {
```

```
            System.out.println((j + 1) + ". " + options.get(j).getText());
        }

        String answer = Esdia.readString_ne("Select an option (1-4). Press ENTER to
leave unanswered: ");
        int selected = 0;
        if (!answer.isBlank()) {
            try {
                selected = Integer.parseInt(answer);
                if (selected < 1 || selected > 4) {
                    selected = 0;
                }
            } catch (NumberFormatException e) {
                selected = 0;
            }
        }

        String feedback = controller.answerQuestion(session, i, selected);
        showMessage(feedback);
    }

    ExamResult result = controller.finishExam(session);
    showExamSummary(result);
}

private void showExamSummary(ExamResult result) {

    clearScreen();

    printHeader("EXAM SUMMARY");

    System.out.println("Correct answers: " + result.getCorrect());
    System.out.println("Wrong answers:   " + result.getWrong());
    System.out.println("Unanswered:      " + result.getUnanswered());
    renderScoreBar(result.getGrade());
    System.out.println("Time (s):        " + result.getDurationSeconds());

    showMessage("Exam finished. Press ENTER to continue.");
    Esdia.readString(" ");
}

private String chooseTopic(boolean includeAll) {
    Set<String> topics = controller.getAvailableTopics();
    if (topics.isEmpty()) {
        showErrorMessage("No topics available.");
        return null;
    }
    List<String> topicList = new ArrayList<>(topics);
    topicList.sort(String::compareTo);
    if (includeAll) {
        topicList.add("ALL");
    }
```

```
    System.out.println("\nAvailable topics:");
    for (int i = 0; i < topicList.size(); i++) {
        System.out.println((i + 1) + ". " + topicList.get(i));
    }
    int choice = Esdia.readInt("Select topic: ", 1, topicList.size());
    return topicList.get(choice - 1);
}
```

4. Clases distintas de MVC y sus funciones:

4.1 Clases abstractas e interfaces

4.1.1 BaseView

BaseView es una clase abstracta que define la estructura fundamental de cualquier vista dentro del patrón Modelo–Vista–Controlador (MVC). Su función es establecer un contrato común para todas las interfaces de usuario del sistema, garantizando coherencia y separación clara entre presentación y lógica de negocio.

Esta clase proporciona los atributos y métodos necesarios para interactuar con el controlador, a la vez que obliga a las vistas concretas a implementar ciertas operaciones clave: inicializar la aplicación (**init()**), finalizarla de forma ordenada **(end()**), mostrar mensajes informativos **(showMessage()**) y gestionar errores (**showErrorMessage()**).

Al centralizar estas responsabilidades, **BaseView** asegura que cualquier vista derivada—como la consola interactiva utilizada en este proyecto—mantenga un comportamiento consistente, permitiendo evolucionar o sustituir la interfaz sin afectar al resto del sistema.

```
package view;
import controller.Controller;

public abstract class BaseView {

   protected Controller controller;

   protected BaseView(Controller controller){
       this.controller = controller;
   }

   protected void setController(Controller controller){
       this.controller = controller;
   }

   protected Controller getController(){
       return this.controller;
   }

   public abstract void init();

   public abstract void end();

   public abstract void showMessage(String message);
```

```
    public abstract void showErrorMessage(String errorMessage);
}
```

4.1.2 QuestionBackupIO

QuestionBackupIO es una clase abstracta que define la interfaz de entrada y salida necesaria para realizar copias de seguridad del banco de preguntas. Su propósito es establecer un contrato común para cualquier mecanismo de importación o exportación, permitiendo que el sistema trabaje de forma independiente al formato utilizado (JSON, XML, CSV u otros). Esta clase obliga a implementar dos operaciones fundamentales: **exportar una colección de preguntas a un archivo** y **cargar preguntas desde un archivo externo**.

Gracias a esta abstracción, el modelo puede delegar completamente el proceso de backup en objetos especializados sin acoplarse a un formato concreto. Esto hace posible extender fácilmente el sistema—por ejemplo, añadiendo nuevos formatos de almacenamiento—sin modificar el resto de la arquitectura. En este proyecto, la implementación principal es **JSONQuestionBackupIO**, que utiliza la biblioteca Gson para leer y escribir preguntas en archivos JSON.

```
package backup;

import java.util.List;

import model.Question;

public interface QuestionBackupIO {

    void exportQuestions(List<Question> questions, String fileName) throws
QuestionBackupIOException;

    List<Question> importQuestions(String fileName) throws QuestionBackupIOException;

    String getBackupIODescription();
}
```

4.1.3 IRepository

`IRepository` es una interfaz que define las operaciones fundamentales para la persistencia interna del banco de preguntas en la aplicación. Actúa como una capa de abstracción entre el modelo y el sistema de almacenamiento, permitiendo que el código del modelo trabaje con un repositorio sin conocer su implementación concreta.

La interfaz especifica métodos esenciales como **añadir**, **modificar**, **eliminar** y **recuperar** preguntas, además de operaciones para **guardar** y **cargar** el conjunto completo desde un medio persistente. Al utilizar una interfaz en lugar de una clase concreta, el sistema permite intercambiar fácilmente distintas estrategias de persistencia—por ejemplo, un repositorio binario, uno basado en texto o incluso una base de datos—sin alterar la lógica de negocio del modelo. En este proyecto, la implementación utilizada es **`BinaryRepository`**, que almacena los datos en un archivo binario serializado y se integra automáticamente con el sistema de guardado y carga del programa.

```
package repository;

import java.util.List;

import model.Question;

public interface IRepository {

   Question addQuestion(Question q) throws RepositoryException;

   void removeQuestion(Question q) throws RepositoryException;

   Question modifyQuestion(Question q) throws RepositoryException;

   List<Question> getAllQuestions() throws RepositoryException;

   void saveAll(List<Question> questions) throws RepositoryException;
}
```

4.1.4 QuestionCreator

`QuestionCreator` es una clase abstracta que define el mecanismo general para generar preguntas de forma automática a partir de un tema suministrado por el usuario. Su función principal es establecer un contrato que deben seguir todas las implementaciones de generación automática de preguntas, de modo que el modelo pueda solicitar nuevas preguntas sin depender de cómo se producen internamente.

La clase expone un método central, generalmente `createQuestion(String topic)`, que las subclases deben implementar para construir una pregunta completa: autor, enunciado, temas, opciones y solución correcta. Además, `QuestionCreator` proporciona un método para describir el generador, lo que permite a la vista mostrar al usuario distintos motores disponibles (por ejemplo, generadores basados en IA o generadores locales).

Este diseño facilita la extensibilidad del proyecto, ya que permite incorporar nuevos generadores automáticos sin modificar el controlador ni el modelo. En esta práctica, uno de los generadores concretos es `GeminiQuestionCreator`, que conecta con un modelo de IA para producir preguntas coherentes y estructuradas a partir de un tema elegido por el usuario.

```
package questionCreator;

import model.Question;

public interface QuestionCreator {

   Question createQuestion(String topic) throws QuestionCreatorException;

   String getQuestionCreatorDescription();
}
```

4.2 Clases POJO

4.2.1 Option

Option es el POJO que modela cada respuesta posible de una pregunta tipo test. Se usa en creación, edición, examen y exportación/persistencia.

Campos y comportamiento:

Text (String): texto visible de la respuesta.

Rationale (String): explicación de por qué esta opción es correcta o incorrecta; se muestra en feedback/examen.

Correct (boolean): marca si esta opción es la única correcta de la pregunta.

Constructores/getters/setters serializables para que pueda guardarse en binario/JSON.

Se valida junto con la pregunta: no admite texto ni rationale vacíos y debe haber exactamente una opción con correct=true por pregunta.

```java
package model;

import java.io.Serializable;

public class Option implements Serializable {

   private static final long serialVersionUID = 1L;

   private String text;
   private String rationale;
   private boolean correct;

   /** Constructor completo */
   public Option(String text, String rationale, boolean correct) {
       this.text = text;
       this.rationale = rationale;
       this.correct = correct;
   }

   // --- GETTERS & SETTERS --- //

   public String getText() {
       return text;
   }

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

```
    public String getRationale() {
        return rationale;
    }

    public void setRationale(String rationale) {
        this.rationale = rationale;
    }

    public boolean isCorrect() {
        return correct;
    }

    public void setCorrect(boolean correct) {
        this.correct = correct;
    }

    @Override
    public String toString() {
        return "Option{text='" + text + "', correct=" + correct + "}";
    }
}
```

4.2.2 Question

Question es el POJO que representa una pregunta tipo test completa.

Agrupa:

Id (UUID): identificador único.

Author (String): autor o modelo generador.

Statement (String): enunciado.

Topics (Set<String>): temas normalizados en mayúsculas.

Options (List<Option>): las 4 respuestas, con exactamente una marcada como correcta.

CreationDate (LocalDateTime): fecha/hora de creación.

Incluye un método **normalizeTopics** que limpia y homogeneiza los temas: recorre el conjunto recibido, descarta nulos o cadenas en blanco, aplica trim() y convierte cada tema a mayúsculas antes de guardarlos en el HashSet<String>. Así se garantiza que los temas se comparen y filtran de forma consistente (p. ej., “math”, “ MATH ” → “MATH”).

Se usa en todas las operaciones del CRUD, en modo examen y en exportación/persistencia. El modelo válida que no haya campos vacíos, que haya 4 opciones y solo una correcta antes de aceptar o importar una pregunta.

```java
package model;

import java.io.Serializable;
import java.time.LocalDateTime;
import java.util.HashSet;
import java.util.List;
import java.util.Set;
import java.util.UUID;

/**
* Representa una pregunta del banco de preguntas.
* Cada pregunta tiene un UUID único, autor, enunciado, temas y 4 opciones.
*/
public class Question implements Serializable {

   private static final long serialVersionUID = 1L;

   private UUID id;
   private String author;
   private String statement;
   private Set<String> topics;
   private List<Option> options;
   private LocalDateTime creationDate;

   /** Constructor usado cuando se crea una nueva pregunta */
   public Question(
           String author,
           String statement,
           Set<String> topics,
           List<Option> options
   ) {
       this.id = UUID.randomUUID();
       this.author = author;
       this.statement = statement;
       this.topics = normalizeTopics(topics);
       this.options = options;
       this.creationDate = LocalDateTime.now();
   }

   /** Constructor usado cuando se importa desde JSON */
   public Question(
           UUID id,
           String author,
           String statement,
           Set<String> topics,
           List<Option> options,
           LocalDateTime creationDate
   ) {
       this.id = id;
       this.author = author;
       this.statement = statement;
       this.topics = normalizeTopics(topics);
```

```
        this.options = options;
        this.creationDate = creationDate;
    }

    // ---- NORMALIZADOR DE TEMAS ---- //
    private Set<String> normalizeTopics(Set<String> raw) {
        Set<String> result = new HashSet<>();
        for (String t : raw) {
            if (t != null && !t.isBlank()) {
                result.add(t.trim().toUpperCase());
            }
        }
        return result;
    }

    // ----------------- GETTERS ----------------- //

    public UUID getId() {
        return id;
    }

    public String getAuthor() {
        return author;
    }

    public String getStatement() {
        return statement;
    }

    public Set<String> getTopics() {
        return topics;
    }

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

    public LocalDateTime getCreationDate() {
        return creationDate;
    }

    // ---------------- MODIFICADORES ---------------- //

    public void setAuthor(String newAuthor) {
        this.author = newAuthor;
    }

    public void setStatement(String newStatement) {
        this.statement = newStatement;
    }

    public void setTopics(Set<String> newTopics) {
```

```
        this.topics = normalizeTopics(newTopics);
    }

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

    @Override
    public String toString() {
        return "Question{id=" + id + ", statement='" + statement + "'}";
    }
}
```

4.2.3 ExamResult

`ExamResult` es una clase POJO diseñada para almacenar y representar los resultados obtenidos por el usuario al finalizar un examen simulado dentro de la aplicación. Su finalidad es agrupar, de forma estructurada, toda la información relevante del rendimiento del usuario, evitando cálculos dispersos y facilitando tanto la presentación de los resultados en la vista como su posible reutilización por otras partes del sistema.

La clase suele contener atributos como:

- **Número de aciertos**, es decir, cuántas preguntas fueron respondidas correctamente.
- **Número de fallos**, aquellas respuestas incorrectas seleccionadas por el usuario.
- **Número de preguntas no respondidas**, en caso de que el usuario omita alguna.
- **Nota final sobre 10**, calculada independientemente del número de preguntas del examen.
- **Listado de respuestas del usuario**, si se requiere un análisis más detallado.

Además, **`ExamResult`** encapsula la lógica necesaria para calcular la nota final a partir de estos datos, evitando duplicar cálculos en el controlador o la vista. Su uso mejora la claridad del flujo del Modo Examen, pues permite que el controlador solo se encargue de recopilar las respuestas y delegue en esta clase el análisis cuantitativo del desempeño.

Gracias a esta separación de responsabilidades, la clase facilita futuras ampliaciones, como añadir penalización por fallos, estadísticas más detalladas o exportación de resultados.

```
package model;

public class ExamResult {
```

```
    private final int correct;
    private final int wrong;
    private final int unanswered;
    private final double grade;
    private final long durationSeconds;

    public ExamResult(int correct, int wrong, int unanswered, double grade, long
durationSeconds) {
        this.correct = correct;
        this.wrong = wrong;
        this.unanswered = unanswered;
        this.grade = grade;
        this.durationSeconds = durationSeconds;
    }

    public int getCorrect() {
        return correct;
    }

    public int getWrong() {
        return wrong;
    }

    public int getUnanswered() {
        return unanswered;
    }

    public double getGrade() {
        return grade;
    }

    public long getDurationSeconds() {
        return durationSeconds;
    }

    @Override
    public String toString() {
        return "ExamResult{correct=" + correct +
                ", wrong=" + wrong +
                ", unanswered=" + unanswered +
                ", grade=" + grade +
                ", durationSeconds=" + durationSeconds + "}";
    }
}
```

4.3 Clases de persistencia

4.3.1 JSONQuestionBackupIO

`JSONQuestionBackupIO` es la clase encargada de gestionar la **importación y exportación de preguntas en formato JSON**, actuando como puente entre el modelo interno del programa y un archivo externo legible y portable. Su función principal es serializar y deserializar listas de objetos `Question`, permitiendo realizar copias de seguridad, restaurar estados previos o intercambiar bancos de preguntas entre diferentes usuarios o máquinas.

Esta clase implementa la interfaz `QuestionBackupIO`, lo que garantiza un conjunto coherente de operaciones (`exportQuestions()` e `importQuestions()`) independientes del resto del sistema. Para realizar la conversión entre objetos Java y JSON, utiliza la biblioteca **Gson**, facilitada por el enunciado de la práctica, asegurando un manejo robusto y estándar del formato.

En la exportación, `JSONQuestionBackupIO` transforma todas las preguntas del sistema en un JSON estructurado, generando un archivo —habitualmente `backup.json`— dentro del directorio home del usuario. Durante la importación, lee el archivo indicado, reconstruye los objetos `Question` con todas sus opciones y metadatos, y devuelve la lista resultante para que el modelo la valide antes de integrarla.

Su diseño modular permite sustituirla fácilmente por otros formatos en el futuro (XML, YAML) sin necesidad de modificar el modelo o el controlador, y convierte al JSON en un método simple, compatible y transparente para preservar la información del banco de preguntas.

```
package backup;

import com.google.gson.Gson;
import com.google.gson.GsonBuilder;
import com.google.gson.JsonDeserializer;
import com.google.gson.JsonSerializer;
import com.google.gson.reflect.TypeToken;

import java.io.FileReader;
import java.io.FileWriter;
import java.lang.reflect.Type;
import java.nio.file.Files;
import java.nio.file.Path;
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;
import java.util.ArrayList;
import java.util.List;
```

```
import model.Question;

public class JSONQuestionBackupIO implements QuestionBackupIO {

   private final Gson gson;
   private final Path dataDir;

   public JSONQuestionBackupIO() {
       this.dataDir = Path.of(System.getProperty("user.home"));
       try {
           Files.createDirectories(dataDir);
       } catch (Exception ignored) {
       }
       this.gson = new GsonBuilder()
               .registerTypeAdapter(LocalDateTime.class,
(JsonSerializer<LocalDateTime>) (src, typeOfSrc, context) ->

context.serialize(src.format(DateTimeFormatter.ISO_LOCAL_DATE_TIME)))
               .registerTypeAdapter(LocalDateTime.class,
(JsonDeserializer<LocalDateTime>) (json, type, context) ->
                       LocalDateTime.parse(json.getAsString(),
DateTimeFormatter.ISO_LOCAL_DATE_TIME))
               .create();
   }

   @Override
   public void exportQuestions(List<Question> questions, String fileName) throws
QuestionBackupIOException {
       Path target = resolvePath(fileName);
       try (FileWriter writer = new FileWriter(target.toFile())) {
           gson.toJson(questions, writer);
       } catch (Exception e) {
           throw new QuestionBackupIOException("Error exporting JSON to " + target, e);
       }
   }

   @Override
   public List<Question> importQuestions(String fileName) throws
QuestionBackupIOException {
       Path target = resolvePath(fileName);
       if (!Files.exists(target) || !Files.isRegularFile(target)) {
           throw new QuestionBackupIOException("File not found: " + target);
       }
       try (FileReader reader = new FileReader(target.toFile())) {
           Type listType = new TypeToken<List<Question>>() {}.getType();
           List<Question> list = gson.fromJson(reader, listType);
           if (list == null) {
               return new ArrayList<>();
           }
           return list;
       } catch (Exception e) {
```

```
            throw new QuestionBackupIOException("Error importing JSON from " + target,
e);
        }
    }

    @Override
    public String getBackupIODescription() {
        return "JSON backup in user home directory";
    }

    private Path resolvePath(String fileName) {
        String normalized = (fileName == null || fileName.isBlank()) ? "backup.json" :
fileName;
        return dataDir.resolve(normalized);
    }
}
```

4.3.2 BinaryRepository

`BinaryRepository` es la clase responsable de gestionar la **persistencia automática del banco de preguntas** utilizando un archivo binario. Su función es almacenar y recuperar de forma eficiente todas las preguntas del sistema, permitiendo que la aplicación conserve su estado entre ejecuciones sin necesidad de que el usuario realice ninguna acción manual.

Este repositorio implementa la interfaz **`IRepository`**, lo que le obliga a proporcionar operaciones fundamentales como **`addQuestion()`**, **`removeQuestion()`**, **`modifyQuestion()`** y **`saveAll()`**. Gracias a esta abstracción, el modelo puede trabajar con cualquier tipo de repositorio sin conocer su implementación interna, cumpliendo estrictamente con el principio de inversión de dependencias.

El archivo binario gestionado por **`BinaryRepository`** (habitualmente `questions.bin`) contiene una representación compacta del conjunto completo de preguntas, incluida su información estructural, opciones, temas y metadatos. Durante el arranque de la aplicación, la clase se encarga de **cargar automáticamente** su contenido —si el archivo existe— para reconstruir el banco de preguntas. Durante la ejecución, cuando el usuario modifica el contenido (crea, elimina o edita preguntas), el repositorio puede actualizar el archivo, según el modo de guardado seleccionado (automático o al salir).

Su enfoque binario ofrece ventajas clave: lectura rápida, tamaño de archivo reducido y serialización directa, lo que convierte a **`BinaryRepository`** en un mecanismo de persistencia ideal para asegurar el correcto mantenimiento y continuidad del banco de preguntas dentro de la aplicación.

```java
package repository;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.ArrayList;
import java.util.List;

import model.Question;

public class BinaryRepository implements IRepository {

   private final Path filePath;
   private List<Question> cache;

   public BinaryRepository(String fileName) {
       if (fileName == null || fileName.isBlank()) {
           fileName = "questions.bin";
       }
       Path homeDir = Path.of(System.getProperty("user.home"));
       try {
           Files.createDirectories(homeDir);
       } catch (Exception e) {

       }
       this.filePath = homeDir.resolve(fileName);
       this.cache = loadFromDisk();
   }

   @Override
   public Question addQuestion(Question q) throws RepositoryException {
       cache.add(q);
       return q;
   }

   @Override
   public void removeQuestion(Question q) throws RepositoryException {
       cache.removeIf(existing -> existing.getId().equals(q.getId()));
   }

   @Override
   public Question modifyQuestion(Question q) throws RepositoryException {
       boolean replaced = false;
       for (int i = 0; i < cache.size(); i++) {
           if (cache.get(i).getId().equals(q.getId())) {
               cache.set(i, q);
               replaced = true;
               break;
           }
```

```
        }
        if (!replaced) {
            throw new RepositoryException("Question not found: " + q.getId());
        }
        return q;
    }

    @Override
    public List<Question> getAllQuestions() throws RepositoryException {
        return new ArrayList<>(cache);
    }

    @Override
    public void saveAll(List<Question> questions) throws RepositoryException {
        try (ObjectOutputStream out = new ObjectOutputStream(new
FileOutputStream(filePath.toFile()))) {
            cache = new ArrayList<>(questions);
            out.writeObject(cache);
        } catch (Exception e) {
            throw new RepositoryException("Error saving binary data", e);
        }
    }

    @SuppressWarnings("unchecked")
    private List<Question> loadFromDisk() {
        if (!Files.exists(filePath)) {
            return new ArrayList<>();
        }
        try (ObjectInputStream in = new ObjectInputStream(new
FileInputStream(filePath.toFile()))) {
            Object obj = in.readObject();
            if (obj instanceof List) {
                return (List<Question>) obj;
            }
            return new ArrayList<>();
        } catch (Exception e) {
            return new ArrayList<>();
        }
    }
}
```

4.4 Clases manejadoras de excepciones

4.4.1 RepositoryException

`RepositoryException` es una clase de excepción personalizada diseñada para representar errores relacionados con la persistencia de datos dentro del sistema. Su propósito principal es **distinguir claramente los fallos derivados de operaciones de almacenamiento** (como guardar, eliminar, modificar o cargar preguntas) de otros errores de la aplicación, permitiendo una gestión más precisa y controlada de situaciones inesperadas.

Esta excepción se utiliza en todas las clases que interactúan con el repositorio —como **`BinaryRepository`**, **`Model`** o los mecanismos de importación/exportación— para señalar problemas tales como:

- Fallos al leer o escribir el archivo binario.
- Inconsistencias en los datos almacenados.
- Intentos de modificar o insertar preguntas inválidas.
- Problemas de acceso al sistema de archivos.

Gracias a **`RepositoryException`**, el controlador puede interceptar estos errores y notificar a la vista de forma clara y comprensible, evitando que detalles técnicos influyan negativamente en la experiencia del usuario.
De este modo, el sistema mantiene una separación adecuada entre la lógica de negocio y la gestión de fallos, asegurando robustez, claridad y un manejo elegante de situaciones anómalas en la persistencia de datos.

```
package repository;

public class RepositoryException extends Exception {

   public RepositoryException(String message) {
       super(message);
   }

   public RepositoryException(String message, Throwable cause) {
       super(message, cause);
   }
}
```

4.4.2 QuestionBackupIOException

`QuestionBackupIOException` es una excepción personalizada especializada en representar errores producidos durante los procesos de **importación y exportación de preguntas en formato JSON**. A diferencia de **`RepositoryException`**, que abarca la persistencia general del sistema, esta excepción se centra exclusivamente en los problemas derivados del manejo de archivos de respaldo externos.

Esta clase se utiliza principalmente en la implementación **`JSONQuestionBackupIO`**, donde pueden ocurrir errores como:

- Archivos JSON inexistentes o inaccesibles.
- Formatos incorrectos o dañados que impiden reconstruir las preguntas.
- Problemas de permisos al intentar leer o escribir en el directorio del usuario.
- JSON mal formado o con una estructura incompatible con lo esperado.

Al encapsular estos fallos en una excepción propia, el sistema puede diferenciarlos claramente de otros tipos de errores, permitiendo al controlador informar al usuario de manera específica sobre qué ha fallado y cómo resolverlo (por ejemplo, verificando la ruta del archivo o el formato del backup).

De este modo, **`QuestionBackupIOException`** contribuye a la robustez del módulo de importación/exportación, mejorando la trazabilidad de errores y manteniendo el principio de responsabilidad única dentro del diseño del software.

```
package backup;

public class QuestionBackupIOException extends Exception {

   public QuestionBackupIOException(String message) {
       super(message);
   }

   public QuestionBackupIOException(String message, Throwable cause) {
       super(message, cause);
   }
}
```

4.4.3 QuestionCreatorException

`QuestionCreatorException` es una excepción personalizada diseñada específicamente para gestionar los errores derivados del proceso de **generación automática de preguntas**. Su función es encapsular cualquier problema que surja en las implementaciones de `QuestionCreator`, especialmente en aquellas que dependen de servicios externos o lógica compleja, como `GeminiQuestionCreator`.

Entre las situaciones que pueden provocar esta excepción se incluyen:

- Fallos en la comunicación con el generador automático (por ejemplo, un servicio de IA).
- Temas inválidos, vacíos o no soportados por el generador.
- Respuestas incompletas o incoherentes devueltas por el sistema de generación.
- Problemas internos en la construcción de la pregunta generada (opciones faltantes, datos inconsistentes, etc.).
- Selección de un generador inexistente o índice fuera de rango.

Al centralizar este tipo de errores en una excepción especializada, el modelo y el controlador pueden reaccionar de forma uniforme, informando claramente a la vista sobre la causa del fallo y evitando interrupciones inesperadas en el flujo de la aplicación.

Gracias a `QuestionCreatorException`, el sistema mantiene un diseño robusto y extensible, permitiendo integrar nuevos motores de generación automática sin comprometer la estabilidad del programa ni mezclar errores de distinta naturaleza.

```
package questionCreator;

public class QuestionCreatorException extends Exception {

   public QuestionCreatorException(String message) {
       super(message);
   }

   public QuestionCreatorException(String message, Throwable cause) {
       super(message, cause);
   }
}
```

4.5 Clases auxiliares

4.5.1 GeminiQuestionCreator

`GeminiQuestionCreator` es una implementación concreta de la interfaz **`QuestionCreator`** que utiliza un servicio externo basado en IA (Gemini) para generar preguntas tipo test de forma automática.
Su función es construir una pregunta completa —enunciado, autor, temas y cuatro opciones con su justificación— a partir de un **tema proporcionado por el usuario**.

Este generador encapsula toda la lógica necesaria para comunicarse con la IA, formatear el prompt, interpretar la respuesta recibida y transformarla en un objeto `Question` válido.
En caso de errores del servicio externo (respuestas mal formadas, falta de opciones, contenido insuficiente, etc.), lanza una **`QuestionCreatorException`**.

Es la pieza central que permite que el sistema ofrezca **creación automática asistida por IA**, haciendo posible extender la aplicación más allá del CRUD tradicional.

```
package questionCreator;

import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Set;

import model.Option;
import model.Question;

public class GeminiQuestionCreator implements QuestionCreator {

   private final String modelId;
   private final String apiKey;
   private final String description;

   public GeminiQuestionCreator(String modelId, String apiKey) {
       this.modelId = modelId;
       this.apiKey = apiKey;
       this.description = "Gemini (" + modelId + ")";
   }

   @Override
   public Question createQuestion(String topic) throws QuestionCreatorException {
       if (topic == null || topic.isBlank()) {
           throw new QuestionCreatorException("Topic required");
       }
```

```
        try {

            Class<?> cfgClass = Class.forName("es.usal.genai.GenAiConfig");
            Class<?> facadeClass = Class.forName("es.usal.genai.GenAiFacade");
            Class<?> schemasClass = Class.forName("es.usal.genai.SimpleSchemas");
            Class<?> schemaClass = Class.forName("com.google.genai.types.Schema");

            Object config = cfgClass
                    .getMethod("forGemini", String.class, String.class)
                    .invoke(null, modelId, apiKey);

            Object schema = schemasClass.getMethod("from", Class.class).invoke(null,
QuestionDTO.class);

            String prompt = buildPrompt(topic.trim());

            QuestionDTO dto;

            Object facade = facadeClass.getConstructor(cfgClass).newInstance(config);
            try {
                dto = (QuestionDTO) facadeClass
                        .getMethod("generateJson", String.class, schemaClass,
Class.class)
                        .invoke(facade, prompt, schema, QuestionDTO.class);
            } finally {
                try {
                    facadeClass.getMethod("close").invoke(facade);
                } catch (Exception ignored) {}
            }

            if (dto == null || dto.options == null || dto.options.size() != 4) {
                throw new QuestionCreatorException("Invalid response from Gemini");
            }

            List<Option> options = new ArrayList<>();
            for (OptionDTO opt : dto.options) {
                options.add(new Option(opt.text, opt.rationale, opt.correct));
            }

            Set<String> topics = new HashSet<>();
            topics.addAll(normalizeTopics(dto.topics));
            topics.add(topic.trim().toUpperCase());

            return new Question(description, dto.statement, topics, options);
        } catch (Exception e) {

            return fallbackQuestion(topic, e.getMessage());
        }
    }

    private String buildPrompt(String topic) {
```

```
        return "Return ONLY valid JSON with schema
{\"author\":string,\"statement\":string,"
                +
"\"topics\":string[],\"options\":[{\"text\":string,\"rationale\":string,\"correct\":boo
lean}]}. "
                + "Write everything in Spanish. "
                + "Generate a multiple-choice question about \"" + topic + "\" with
exactly 4 options and only one correct. "
                + "Include at least one TOPIC in uppercase related to \"" + topic + "\".
"
                + "The rationale must explain why each option is correct or incorrect.";
    }

    private Set<String> normalizeTopics(List<String> raw) {
        Set<String> result = new HashSet<>();
        if (raw == null) return result;
        for (String t : raw) {
            if (t != null && !t.isBlank()) {
                result.add(t.trim().toUpperCase());
            }
        }
        return result;
    }

    @Override
    public String getQuestionCreatorDescription() {
        return description;
    }

    private Question fallbackQuestion(String topic, String reason) {
        String normalizedTopic = topic == null ? "GENERAL" : topic.trim().toUpperCase();
        String statement = "Fallback question for topic: " + normalizedTopic + " (Gemini
unavailable)";
        List<Option> options = new ArrayList<>();
        options.add(new Option("Correct answer about " + normalizedTopic,
                "Generated locally because Gemini failed: " + reason, true));
        options.add(new Option("Incorrect option 1", "Placeholder distractor", false));
        options.add(new Option("Incorrect option 2", "Placeholder distractor", false));
        options.add(new Option("Incorrect option 3", "Placeholder distractor", false));

        Set<String> topics = new HashSet<>();
        topics.add(normalizedTopic);

        return new Question(description, statement, topics, options);
    }

    public static class QuestionDTO {
        public String author;
        public String statement;
        public List<String> topics;
        public List<OptionDTO> options;
    }
```

```
    public static class OptionDTO {
        public String text;
        public String rationale;
        public boolean correct;
    }
}
```

4.5.2 ExamSession

`ExamSession` es la clase encargada de gestionar todo el proceso interno de un examen dentro del sistema.
Su responsabilidad es mantener el estado asociado a una sesión de examen, incluyendo:

- El conjunto de preguntas seleccionadas para ese examen.
- Las respuestas proporcionadas por el usuario.
- La determinación de aciertos, fallos y preguntas no contestadas.

Además, es la encargada de producir el resultado final mediante un objeto `ExamResult`, que resume el rendimiento del usuario (nota, número de aciertos, errores y omisiones).

`ExamSession` actúa como una capa de lógica especializada que permite separar claramente el **flujo del examen** de la vista y del modelo general, proporcionando un diseño más modular y fácil de mantener.

```
package model;

import java.util.ArrayList;
import java.util.List;

public class ExamSession {

    private final List<Question> questions;
    private final List<Integer> answers;
    private final long startMillis;
    private long endMillis;

    public ExamSession(List<Question> questions) {
        this.questions = new ArrayList<>(questions);
        this.answers = new ArrayList<>();
        for (int i = 0; i < questions.size(); i++) {
            this.answers.add(0);
        }
        this.startMillis = System.currentTimeMillis();
```

```
        this.endMillis = -1;
    }

    public List<Question> getQuestions() {
        return questions;
    }

    public List<Integer> getAnswers() {
        return answers;
    }

    public void recordAnswer(int index, int answer) {
        if (index < 0 || index >= answers.size()) return;
        answers.set(index, answer);
    }

    public void finish() {
        this.endMillis = System.currentTimeMillis();
    }

    public long getDurationSeconds() {
        long end = endMillis == -1 ? System.currentTimeMillis() : endMillis;
        return Math.max(0, (end - startMillis) / 1000);
    }
}
```

4.5.3 SimpleQuestionCreator

SimpleQuestionCreator es un generador básico de preguntas utilizado como implementación mínima de **QuestionCreator**.
A diferencia de **GeminiQuestionCreator**, no depende de servicios externos ni IA: su comportamiento es totalmente local y deterministicamente genera preguntas sencillas a partir del tema indicado.

Su propósito principal es:

- Demostrar el funcionamiento del sistema de generación automática.
- Servir como alternativa cuando no se dispone de conexión o recursos externos.
- Facilitar pruebas, validación y depuración del módulo sin depender de la IA.

Aunque limitado, garantiza que la característica de generación automática siempre esté disponible y operativa.

```
package questionCreator;

import java.util.*;
```

```java
import model.Option;
import model.Question;

public class SimpleQuestionCreator implements QuestionCreator {

   @Override
   public Question createQuestion(String topic) {

       String statement = "Automatically generated question for topic: " + topic;

       List<Option> options = new ArrayList<>();

       options.add(new Option("Correct answer", "Generated automatically", true));
       options.add(new Option("Wrong answer 1", "Generated automatically", false));
       options.add(new Option("Wrong answer 2", "Generated automatically", false));
       options.add(new Option("Wrong answer 3", "Generated automatically", false));

       Set<String> topics = new HashSet<>();
       topics.add(topic.toUpperCase());

       return new Question("AUTO", statement, topics, options);
   }

   @Override
   public String getQuestionCreatorDescription() {
       return "Simple local generator";
   }
}
```

5. Problemáticas encontradas y soluciones aplicadas

Durante el desarrollo del sistema de gestión de preguntas se identificaron diversas problemáticas técnicas relacionadas con la arquitectura MVC, la persistencia de datos, la validación del modelo y la experiencia de usuario en consola. A continuación se describen las más relevantes y las soluciones implementadas.

5.1. Separación estricta de responsabilidades (MVC)

Problema:
Inicialmente, parte de la lógica de negocio (por ejemplo, validaciones, ordenación de preguntas o control de duplicados) se encontraba distribuida entre la vista y el controlador, violando el principio MVC.

Solución aplicada:
Reestructuración completa para trasladar toda la lógica al modelo, dejando la vista únicamente para interacción con el usuario y el controlador como intermediario.
Esto mejoró la mantenibilidad y permitió documentar cada capa de manera independiente.

5.2. Validación de preguntas incompletas o inconsistentes

Problema:
Durante la creación y modificación de preguntas era posible introducir valores vacíos, opciones sin justificación o varias respuestas correctas, lo que generaba datos inválidos en el sistema.

Solución aplicada:
Implementación de validadores dedicados en el modelo (validateBaseFields y validateOptions).
El sistema ahora garantiza que toda pregunta tenga exactamente cuatro opciones, una única opción correcta y campos no vacíos. Si una validación falla, se lanza una RepositoryException.

5.3. Persistencia automática y manual

Problema:
La práctica pedía gestionar dos modos de guardado:

- Guardado automático tras cada operación.
- Guardado final al salir.

Esto podía ocasionar inconsistencias si el usuario realizaba muchas operaciones sin guardar.

Solución aplicada:
Implementación del método setAutoSave(boolean) en el controlador y llamadas a persistState() tras las operaciones críticas.
El usuario elige el modo al inicio, y el controlador gestiona uniformemente la persistencia siguiendo esa política.

5.4. Integración de repositorios binarios y JSON

Problema:
Coordinar dos mecanismos de entrada/salida (binario automático y JSON manual) generó conflictos iniciales en el flujo de datos.

Solución aplicada:
Unificación de ambos sistemas bajo una capa de repositorio y un backupHandler, con reglas claras:

- **Binario** → almacenamiento principal del sistema.
- **JSON** → import/export manual para copias de seguridad.

Además, se añadió normalización del nombre de archivo (normalizeFileName) para evitar errores comunes del usuario.

5.5. Gestión del examen y respuestas incompletas

Problema:
El usuario podía dejar preguntas sin responder o introducir valores inválidos, generando errores en el cálculo de la nota.

Solución aplicada:
El sistema detecta respuestas vacías, fuera de rango o no numéricas, convirtiéndolas en "no respondidas".
El modelo calcula la nota final mediante un objeto ExamResult, lo que estandariza la lógica y permite mostrar resúmenes claros.

6. Mejoras o extras aplicados

Además de cumplir con todos los requisitos del enunciado, el proyecto incorpora una serie de mejoras que optimizan la experiencia de uso, la mantenibilidad del código y la robustez del sistema. Estas extensiones no eran obligatorias, pero aportan un valor añadido significativo.

6.1. Interfaz de consola mejorada con diseño visual

Se ha implementado una interfaz enriquecida con:

- Colores ANSI configurables
- Encabezados dinámicos (printHeader)
- Animaciones sutiles (pulseMessage, animateProgress)
- Barras de estado y divisores estilizados
- Menús más legibles y jerarquizados

Esto convierte la línea de comandos en una experiencia mucho más clara e intuitiva.

6.2. Funciones auxiliares para evitar código repetitivo

La vista incorpora diversas funciones utilitarias que simplifican y estandarizan la presentación:

- printMenuItem
- printDivider
- renderStatusBar
- chooseTopic
- readNonEmptyString, etc.

Esto reduce drásticamente la duplicación de código y mejora la mantenibilidad a largo plazo.

6.3. Normalización inteligente del nombre de archivo

En operaciones de importación/exportación se implementó:

- normalizeFileName(fileName)
  que garantiza un nombre válido (“backup.json”) cuando el usuario lo deja en blanco o introduce cadenas no válidas.

Evita errores comunes y hace el sistema más robusto frente a entradas inesperadas.

6.4. Gestión avanzada de importación con abortado seguro

El proceso de importación implementa mejoras significativas sobre el estándar:

- Validación exhaustiva de cada pregunta importada
- Abortado completo si alguna es inválida
- Prevención de duplicados por UUID
- Mensajes claros de error

Esto garantiza la integridad del banco de preguntas.

6.10. Animaciones y mensajes guiados para mejorar la UX

Incluye pequeñas animaciones:

- Barras de progreso

- Destellos en títulos

- Transiciones entre secciones

Logran una experiencia más profesional y pulida en modo consola.

6.11. Carga flexible de múltiples QuestionCreator desde argumentos

Una mejora adicional implementada consiste en permitir que la aplicación reciba desde la línea de comandos uno o varios generadores automáticos de preguntas (QuestionCreator), separados por comas.

En lugar de depender únicamente de un creador por defecto, ahora el sistema puede procesar cadenas como:

--creators=SimpleQuestionCreator,AdvancedCreator,AIQuestionCreator

La vista o el punto de entrada del programa se encarga de:

1. Leer el argumento.
2. Separarlo por comas.
3. Instanciar dinámicamente cada QuestionCreator.
4. Pasar la lista resultante al modelo.

Gracias a esta ampliación:

- El sistema se vuelve más modular y extensible.
- Se pueden combinar distintos estilos de generación automática sin modificar el código base.
- Es posible añadir nuevos generadores en el futuro simplemente incorporando el archivo correspondiente y pasándolo en los argumentos.

Esta funcionalidad no estaba prevista en el enunciado original, pero aporta una gran flexibilidad para escenarios más avanzados o evolutivos del proyecto.

7. Posibles mejoras futuras

Aunque la aplicación cumple con todos los requisitos del enunciado y ofrece funcionalidades ampliadas (como persistencia automática, validación avanzada, interfaz mejorada y soporte para varios QuestionCreators), existen múltiples líneas de evolución que podrían enriquecer el sistema:

7.1. Sistema de permisos y perfiles de usuario

La aplicación podría ampliarse para distinguir entre:

- Administradores (permiten CRUD completo)
- Estudiantes (solo modo examen)
- Profesores (importación/exportación, generación automática, estadísticas)

Esto permitiría un uso real en entornos docentes y facilitaría el control de acceso.

7.2. Soporte para más formatos de importación/exportación

Actualmente, todo se gestiona en JSON.
Podrían añadirse:

- CSV (compatible con Excel)
- XML
- Base de datos SQL (MySQL, SQLite...)

Esto mejoraría la interoperabilidad con otros sistemas educativos.

7.3. Estadísticas avanzadas de uso

El sistema podría registrar:

- Número de veces que se usa cada pregunta
- Resultados promedio de exámenes por tema
- Dificultad estimada según tasa de acierto

Esto permitiría construir un módulo de learning analytics.

7.4. Banco de preguntas categorizado por dificultad

Además del tema, cada pregunta podría etiquetarse con:

- Nivel (Fácil, Medio, Difícil)
- Relevancia
- Peso en el examen

Esto abriría la puerta a exámenes adaptables.

7.5. Integración con APIs externas de IA

Actualmente se usan QuestionCreator básicos, pero podrían integrarse servicios externos como:

- OpenAI / ChatGPT
- Hugging Face
- Google Gemini

para generar preguntas completas con justificativa y distractores inteligentes.

7.6. Interfaz gráfica (GUI) o versión web

La vista en consola funciona bien, pero una evolución natural sería:

- Interfaz JavaFX
- Aplicación web con Spring Boot
- Frontend en React conectado a un backend Java

Esto haría el sistema más accesible para usuarios no técnicos.

7.7. Tests automáticos

Incorporar:

- Unit tests para el modelo (JUnit)
- Integration tests para el repositorio
- Validación de flujo MVC completo

Aumentaría la fiabilidad del sistema ante futuras ampliaciones.

7.8. Mecanismos de copia de seguridad programada

Además de exportación manual:

- Backups automáticos diarios/semanales
- Gestión de versiones (snapshot histórico)
- Restauración de estados anteriores

Esto sería útil en entornos educativos donde el banco de preguntas crece continuamente.

7.9. Carga dinámica de QuestionCreators desde ficheros o plugins

Actualmente se cargan desde argumentos.
Una mejora sería implementar un sistema de plugins, donde cada creador externo pueda añadirse simplemente colocándolo en una carpeta:

/creators/MyNewCreator.class

El sistema lo detecta automáticamente mediante reflection.

7.10. Modo examen avanzado

Nuevas capacidades:

- Tiempo límite
- Penalización por fallos configurable
- Preguntas aleatorias con pesos probabilísticos
- Revisión de examen con explicación detallada

Convertiría la herramienta en un simulador profesional.

7.11. Exportación de exámenes en PDF

Permitir generar el examen y su plantilla de corrección en PDF con formato profesional, útil para impresión.

7.12. Sincronización en la nube

Guardar automáticamente en:

- Google Drive
- OneDrive
- Dropbox

permitiendo el uso del banco de preguntas desde varios equipos.

8. Conclusión

El desarrollo de este proyecto ha permitido construir un sistema completo de gestión de preguntas tipo test siguiendo una arquitectura MVC bien definida y fácilmente ampliable. A

lo largo del trabajo se han implementado todas las funcionalidades requeridas en el enunciado, garantizando una separación clara entre responsabilidades y una estructura modular que facilita el mantenimiento.

El uso de repositorios externos para la persistencia, junto con mecanismos de validación estricta, asegura la integridad de los datos y evita inconsistencias durante el ciclo de vida del programa. Además, la implementación de utilidades visuales en la vista en consola mejora la experiencia de usuario y demuestra atención al diseño de interfaces incluso en entornos no gráficos.

Las problemáticas encontradas durante el desarrollo han permitido mejorar la robustez del sistema, y los extras añadidos, como la persistencia binaria automática o el soporte para múltiples *QuestionCreators*, muestran un esfuerzo por ir más allá de los requisitos mínimos, acercando la aplicación a un escenario real de uso.

Finalmente, el proyecto queda preparado para evolucionar en múltiples direcciones, ya sea mediante una interfaz gráfica, ampliación de formatos de almacenamiento o la integración con servicios de IA avanzados. Con ello, esta práctica no solo cumple los objetivos académicos planteados, sino que establece una base sólida y extensible para futuros desarrollos relacionados con la gestión y generación de contenido educativo.