PRÁCTICA FINAL
PROGRAMACIÓN III

CURSO 2025/2026

Fecha límite de entrega: 11/12/2025

Facultad de Ciencias - Universidad de Salamanca

Introducción

Una empresa nos encarga el desarrollo de *Examinator 3000*, una aplicación en Java para cualquier plataforma de escritorio (MacOS, Linux, Windows, etc.) cuyo objetivo es permitir a estudiantes y docentes gestionar bancos de preguntas y realizar simulacros de examen.

La aplicación inicialmente será de consola, pero la empresa prevé incorporar en el futuro otros tipos de vistas (como interfaces gráficas o incluso un modo asistente que permita navegar por voz y leer las preguntas con síntesis de voz). Por ello, es fundamental que el diseño siga buenas prácticas de programación orientada a objetos, establezca capas de abstracción claras y adopte una arquitectura adecuada que permita hacer evolucionar la aplicación y automatizar pruebas en el futuro.

Requisitos

Los requisitos funcionales de la aplicación se resumen en los siguientes puntos:

- **Capacidad de realizar operaciones CRUD en un banco de preguntas**: La aplicación debe permitir gestionar un banco de preguntas tipo test. Cada pregunta tendrá una serie de atributos obligatorios (detallados en el apartado de diseño del modelo). La interfaz por consola deberá permitir:
    - **Crear** nuevas preguntas.
    - **Listar preguntas**, incluyendo:
        - Listado completo ordenado por fecha de creación de la pregunta.
        - Listado filtrado por tema introducido por el usuario.
    - **Ver detalle** de una pregunta seleccionada desde los listados.
        - Desde el detalle se podrá:
            a) Modificar cualquier dato salvo su identificador interno.
            b) Eliminar la pregunta.

  El CRUD también deberá estar preparado para funcionar en futuros modos de visualización (GUI, TTS, etc.), por lo que se exige una separación clara entre el modelo y la vista. La vista se limitará a realizar lógica relacionada con E/S por consola.

- **Modo Examen:** la aplicación debe contar con una opción denominada “Modo examen” que permitirá al usuario realizar un examen sobre un tema **T** con **N** preguntas. Esta opción le presentará los (**T**) temas disponibles al usuario en una lista para que el usuario elija el de su preferencia, además al listado se añadirá una opción denominada “todos” que abarcará todos los temas. Los temas se obtienen dinámicamente a partir de los disponibles en las preguntas. Posteriormente se le solicitará el número de preguntas **N** del examen que deberá estar entre 1 y el número máximo de preguntas disponibles para ese tema. El programa configurará un objeto Examen en el modelo con preguntas que la vista irá mostrando al usuario para que vaya respondiendo. En cada pregunta, el usuario podrá responder o no a la pregunta. En caso de que el usuario responda, se deberá dar *feedback* al usuario con la información pertinente (mostrando los motivos por los que una respuesta es correcta o incorrecta en cada caso). Al finalizar, se mostrará el resultado del examen señalando la nota que habría sacado el estudiante (sobre 10 puntos independientemente del número de preguntas), el número de preguntas acertadas, falladas y no contestadas y el tiempo que ha tardado. El cálculo de la nota se realizará teniendo en cuenta que las preguntas falladas penalizan 1/3 del valor de una pregunta. Se debe separar claramente la lógica de negocio del Examen y de presentación de preguntas al usuario.

- **Creador automático de preguntas**: la aplicación contará con una opción que permitirá crear una pregunta de forma automática de un tema proporcionado por el usuario. Para ello el modelo tendrá un ArrayList de objetos de tipo QuestionCreator. Este QuestionCreator será una interfaz que permite crear una pregunta a partir de un tema proporcionado como String. La vista ofrecerá tantas opciones como questions creators existan en el modelo y permitirá al usuario seleccionar uno de ellos e introducir un tema para crear y añadir dicha pregunta al banco de preguntas. La lógica para añadir la pregunta deberá residir en el modelo. La aplicación deberá contar únicamente con un QuestionCreator de tipo GeminiQuestionCreator que se creará si se indica por consola el parámetro -question-creator seguido del id del modelo y de la API_KEY necesaria. En caso contrario la aplicación no tendrá question creators y esta opción estará deshabilitada para el usuario. Ver el apartado *One More Thing* para la implementación de esta parte.

- **Exportación/Importación como JSON:** La aplicación debe contar con una opción que permita importar y exportar las preguntas disponibles en el modelo el formato JSON. La política de **importación** debe asegurar que no existen dos preguntas con un mismo identificador único. Tanto la exportación como la importación se realizarán a/de un fichero con el nombre introducido por el usuario que se encuentra en el **home** del usuario. El modelo deberá contar con un objeto de tipo QuestionBackupIO que permita abstraer este proceso para ampliarlo a otros formatos el día de mañana.

- **Persistencia**: El programa **deberá ser capaz de recuperar y guardar automáticamente el estado completo de la aplicación (banco de preguntas)**. Por defecto se usará la serialización binaria de Java. El estado se cargará al iniciar la aplicación y se podrá elegir si guardar al finalizar o en cada operación. Esta persistencia se abstraerá mediante una interfaz (IRepository) para que pueda ser sustituido por otro tipo de persistencia distinta en el futuro (hoy se guarda en un binario, mañana podría ser una BBDD).

- **Interfaces de usuario**: El prototipo deberá contar con una interfaz por consola con un menú interactivo, pero también deberá estar preparado para extenderse en el futuro con nuevas vistas (GUI, Command-CLI, TTS, etc.). Un diseño adecuado es esencial para permitir esta evolución. Se deberá implementar al menos la siguiente interfaz de usuario:
    - Vista Interactiva por consola: incluirá un menú y submenús para acceder a todas las opciones de la aplicación. Será la vista por defecto.

  [OPCIONAL-NO EVALUABLE] Vista TTS una vista que narra el texto de las opciones y preguntas utilizando una biblioteca de *Text-To-Speech* como esta. Es posible implementarla de forma muy sencilla a partir de la Vista Interactiva por consola, ya que simplemente es añadir la narración de cada uno de los textos (revisad el uso de super).

- **Aplicación robusta y resistente a fallos**: la funcionalidad de la aplicación debe de haber sido probada manualmente en modo interactivo, revisando toda la funcionalidad implementada. **Si una característica no funciona la empresa no la valorará.** Se deberán manejar excepciones e informar al usuario de posibles errores debidamente, sin la parada abrupta de la aplicación.

- **Arquitectura MVC**: un punto clave del desarrollo de la aplicación es que siga las guías de diseño del estilo arquitectónico Modelo Vista Controlador tratadas a lo largo del curso en la asignatura. No solo debe funcionar, sino que se debe seguir la arquitectura especificada. Se debe tener especial cuidado a la hora de implementar cada parte, separando responsabilidades.

GUÍA DE DISEÑO E IMPLEMENTACIÓN

A continuación, se proporciona una guía de diseño que se deberá tener en cuenta para el desarrollo de la aplicación. No se proporciona una guía exhaustiva pero sí lo suficientemente detallada para poder abordar todos los puntos de la práctica.

Si no comprende algo o necesita alguna aclaración, escríbalo por el foro de dudas de Studium habilitado para tal fin, resolverá su duda y la de sus compañeros.

Orden recomendado para la realización de la práctica:

0. Crear un nuevo repositorio Git privado en GitHub para la práctica.
1. Andamiaje (*scaffolding* del MVC). Armar las clases principales del MVC.
2. CRUD preguntas. Realizar pruebas manuales para comprobar que funciona cada parte. Empezad por la creación de preguntas y los listados simples de preguntas. Probar paulatinamente que todo funciona.
3. Serialización: comprobar que se restaura el estado previo del banco de preguntas y que se guarda en cada modificación.
4. Exportación/Importación JSON: comprobar que se importan exportan correctamente las preguntas y se usan los objetos adecuados.
5. Modo Examen: vigilad donde implementáis la lógica de este proceso.
6. QuestionCreator: probad primero el proyecto proporcionado de muestra y posteriormente integradlo en vuestro proyecto.

A continuación, se proporciona una guía de diseño que se deberá tener en cuenta para el desarrollo de la aplicación. No se proporciona una guía exhaustiva pero sí lo suficientemente detallada para poder abordar todos los puntos de la práctica.

Arquitectura MVC:

La aplicación deberá presentar una arquitectura MVC. Esta arquitectura se construirá en el inicio de la aplicación, creando y ensamblando los objetos que intervendrán en el funcionamiento del sistema. La aplicación siempre trabajará con las clases **BaseView**, **Model** y **Controller**, mientras que las instancias concretas de estas (p. ej., diferentes vistas, repositorios, etc.) puede que dependan de los parámetros recibidos en String[] args. La configuración completa del MVC podrá realizarse en el método *main* de la clase App.java como se ha realizado habitualmente en clase.

Revise previamente los ejemplos de MVC con herencia e interfaces proporcionados durante las clases prácticas.

A continuación, se muestra un diagrama UML (**incompleto**) que representa un posible enfoque para el diseño del proyecto. **En él NO se muestran todas las clases, métodos y atributos que puede llegar a contener la solución, pero sí una guía de las principales piezas del sistema. Puede (y debe) añadir más métodos y atributos siempre que lo considere necesario.** Es importante observar que el Controlador tiene una referencia a BaseView (clase abstracta) y que la clase Model contiene atributos cuyo tipo son interfaces (IRepository, QuestionBackupIO). Esto implica que:

- El controlador siempre trabajará únicamente con los métodos definidos en la superclase abstracta BaseView, sin conocer la vista concreta seleccionada.

- El modelo solo podrá utilizar los métodos expuestos por las interfaces IRepository y QuestionBackupIO, sin conocer las implementaciones concretas. Será el encargado de implementar la lógica de negocio solicitada empleando objetos de dichos tipos.

Este diseño permite abstraer y desacoplar dichas partes de la aplicación, de modo que puedan sustituirse fácilmente por otras (por ejemplo, una vista TTS en lugar de la vista interactiva, o un exportador XML en lugar de uno JSON) sin afectar a la lógica de negocio (cómo se realiza la importación teniendo en cuenta que no debe haber objetos con UUID repetidos, por ejemplo). **Las instancias concretas se decidirán durante la inicialización del programa en el main, asegurando que el comportamiento del sistema se mantenga estable, aunque se cambien las implementaciones utilizadas**.

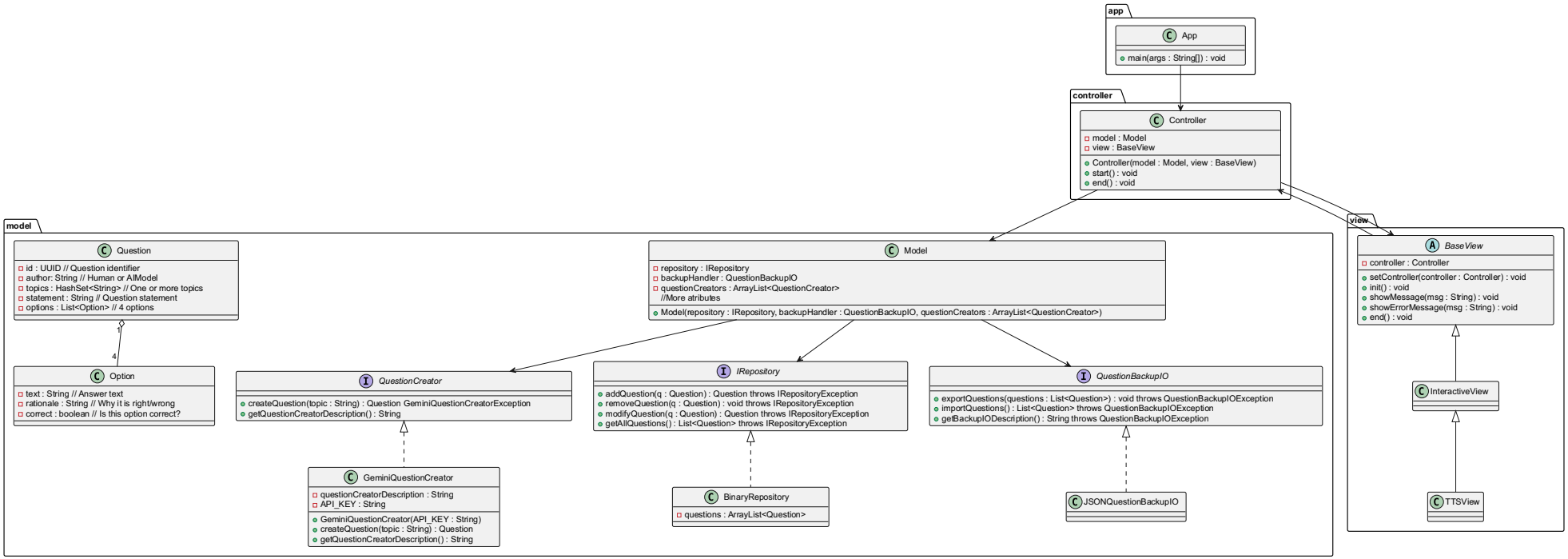


Ilustración 1 Diagrama de clases (incompleto) de la práctica

Paquete view:

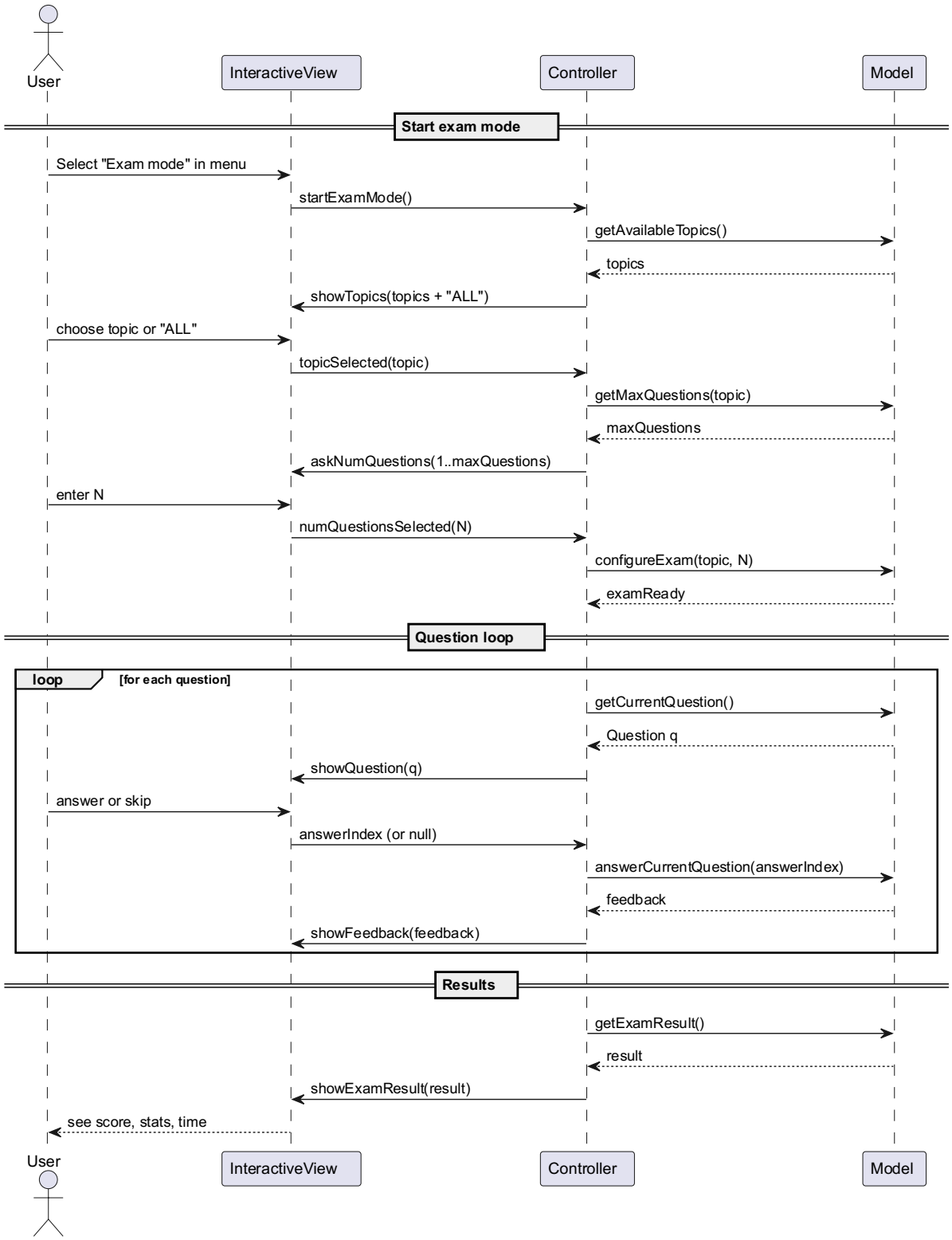
- **BaseView**: es una clase abstracta que tiene, al menos, los siguientes métodos abstractos y atributos:
  Atributos:
  - **Controller** controller.

  Métodos: tendrá como mínimo los siguientes **métodos abstractos**:
  - void **init**(): inicia la vista y desencadena la lógica de la vista.
  - void **showMessage**(): permite notificar de un mensaje al usuario.
  - void **showErrorMessage**(): permite notificar de un mensaje de error al usuario.
  - void **end**(): finaliza la vista ordenadamente.

La funcionalidad de los métodos abstractos la implementaran las clases que heredan de **BaseView**, en este caso como mínimo se debe implementar la siguiente:

- **InteractiveView (Interfaz de consola simple interactiva):** Será una clase que hereda de BaseView. Será la encargada de mostrar los mensajes de inicio de la aplicación, iniciar el menú con las opciones del programa y gestionar toda la interacción con el usuario. Las opciones mínimas que deberá ofrecer son:
  - **Opción CRUD que permite:**
    - **Crear nuevas preguntas**. Solicitando la información necesaria al usuario y asegurándose de que solo hay una opción verdadera.
    - **Listar preguntas**, incluyendo:
      - Listado completo ordenado por fecha de creación de la pregunta.
      - Listado filtrado por tema introducido por el usuario.
    - **Ver detalle** de una pregunta seleccionada desde los listados. Desde el detalle se podrá:
      - **Modificar cualquier atributo de la pregunta** salvo su identificador interno.
      - **Eliminar la pregunta.**
  - **Opción exportación/importación:** Exportar todas las preguntas y temas a ficheros en el directorio home del usuario en formato **JSON** (por ejemplo: backup.json).
    - Importar a partir de esos mismos ficheros.
    - La política de importación será **no importar elementos que tengan el mismo identificador único UUID** a los existentes en el banco de preguntas.
  - **Opción creación de pregunta automática:** solo estará disponible si hay algún QuestionCreator cargado en el modelo. Solicitará un tema al usuario por consola y creará una pregunta empleando el QuestionCreator seleccionado. Esta pregunta se le presentará al usuario y confirmará si desea agregarla al banco de preguntas. Más información sobre esta opción en la sección "*One More Thing*".
  - **Menú Modo Examen:** Permitir al usuario realizar un examen simulado:
    - Solicitar el **número de preguntas** del simulacro.
    - Permitir seleccionar **un tema** de entre los disponibles o la opción todos.
    - Formular las preguntas de forma secuencial y registrar las respuestas del usuario.
    - Al finalizar, mostrar un **resumen de resultados** (aciertos, fallos, no respondidas y nota que hubiera sacado sobre 10, independientemente del número de preguntas).

Para el menú de simulacro de examen os dejamos un pequeño diagrama de secuencia simplificado que os puede ayudar a implementar la lógica respetando las responsabilidades de MVC:



Paquete controller:

El **controlador** será un intermediario y orquestador de la aplicación. Sus responsabilidades principales serán:

- **Inicialización de la aplicación**: Indicará al modelo que cargue la información previa (si existiera) y ordenará a la vista mostrar un mensaje con el resultado de dicha carga.
- **Arranque de la vista:** Pedirá a la vista que se inicie mediante init(), desencadenando el menú principal (CRUD de preguntas, exportación/importación y simulacro de examen).
- **Gestión de órdenes desde la vista:** El controlador recibirá solicitudes como:
  - crear, modificar o eliminar preguntas,
  - listar preguntas,
  - exportar o importar preguntas,
  - ejecutar un simulacro de examen especificando un tema o todos.
- Tras recibir la orden, el controlador trasladará la petición al modelo y devolverá el resultado o propagará las excepciones necesarias.
- Las **excepciones** que se produzcan en el modelo podrán propagarse hasta la vista; será la vista la que realice el try/catch y muestre el mensaje adecuado al usuario.
- **Finalización ordenada de la aplicación**: El controlador indicará al modelo que guarde el estado actual de la aplicación y, después, pedirá a la vista mostrar un mensaje indicando si el guardado se realizó correctamente o si ocurrió algún error.

Paquete model:

Existirá una clase que se corresponde con el **modelo (Model)** y que debe tener, al menos, los siguientes atributos:

- **IRepository:** Se trata de un objeto que implementa la interfaz IRepository y que permitirá hacer las operaciones CRUD sobre la colección interna de **preguntas** que posea el modelo. Esta colección se cargará desde un origen de datos dependiendo del IRepository concreto proporcionado al modelo por constructor:
  - En el caso de ser un **BinaryRepository**, se deberá crear una colección de objetos a partir de un fichero binario en el home del usuario llamado “questions.bin”, si este existiera. Las preguntas se mantendrán en memoria durante la ejecución de la aplicación y, cuando termine, se guardarán de nuevo en el mismo fichero binario.

  Los métodos que implementará esta interfaz deberán ser (al menos):

  - `Question addQuestion(Question q) throws RepositoryException`
  - `void removeQuestion(Question q) throws RepositoryException`
  - `void modifyQuestion(Question q) throws RepositoryException`
  - `ArrayList<Question> getAllQuestions() throws RepositoryException`

  Todas lanzan una excepción personalizada denominada RepositoryException, que envuelve las posibles excepciones que se puedan producir al realizar las acciones concretas de cada implementación, por ejemplo, en el caso de BinaryRepository se podrían producir excepciones relacionas con ficheros. El IRepository concreto lo obtendrá el modelo por **constructor** en el inicio de la aplicación.

- **QuestionBackupIO:** Se trata de un objeto que implementa la interfaz QuestionBackupIO y que permite **exportar o importar** las preguntas. Contará con dos métodos, exportQuestions e importQuestions, para exportar/importar las preguntas. Si se produce un error, se deberá lanzar una QuestionBackupIOException personalizada que envuelva las posibles excepciones que se den.
    - Se deberá crear un JSONQuestionBackupIO que implemente QuestionBackupIO y será el que se proporcione por constructor al modelo. Realizará las operaciones empleando formato JSON.

- **QuestionCreator:** el modelo podrá contar con un atributo de tipo ArrayList<QuestionCreator> que contendrá los question creator disponibles y se proporcionará por constructor al modelo. El modelo utilizará este objeto para crear nuevas preguntas de forma automática sin depender de la implementación concreta.
    - En la inicialización de la aplicación en App.java, si se proporciona el parámetro -question-creator seguido del identificador del modelo y de la API_KEY, se deberá crear un GeminiQuestionCreator que implementa QuestionCreator. Consultar la sección "*One More Thing*".

- **Clases del modelo de dominio:** Habrá una clase POJO que represente una pregunta (**Question**) y una para las diferentes opciones de respuesta (**Option**) que por defecto serán 4.

  Question:
    - **id:** será de tipo UUID que podéis generar con UUID.randomUUID() de java.util.UUID.
    - **author:** autor de la pregunta o identificador del question creator en caso de que la pregunta sea generada. Será de tipo String.
    - **topics:** temas en forma de string, se deberán normalizar para que sean siempre en mayúsculas. Será de tipo HashSet<String>
    - **statement:** enunciado de la pregunta. De tipo String.
    - **options:** lista de posibles respuestas (List<Option>).

  Option:
    - **text:** texto de la opción en tipo String.
    - **rationale:** texto que justifica por qué la pregunta es cierta o falsa.
    - **correct:** booleano que indica si es correcta o no.

  Se deberán utilizar obligatoriamente estos nombres para los atributos de cara a poder realizar las pruebas de una forma más unificada por parte de los docentes que evalúan la práctica.

"One more thing"

La empresa piensa ofrecer como característica especial de *Examinator 3000* la posibilidad de crear preguntas de forma automática mediante diferentes modelos de IA (dejando la aplicación preparada para añadir más en el futuro). Para ello se propone incluir un primer QuestionCreator denominado GeminiQuestionCreator, aprovechando el boom de los últimos modelos de Gemini. Este GeminiQuestionCreator será una clase que implementará QuestionCreator y que utilizará la SDK de Gemini para Java. Concretamente se utilizará la función de "salida estructurada" para generar un objeto de una clase concreta a partir de un ***prompt*** de entrada y la clase objetivo. Se deberá especificar cuidadosamente el **prompt** para incluir el tema proporcionado por el usuario y cumplir los requisitos de la aplicación (que sean 4 opciones y una sola correcta con sus correspondientes justificaciones). Se proporciona un proyecto de ejemplo y los JAR necesarios en el siguiente enlace:

- Repositorio de ejemplo

- Código de ejemplo

La clase GeminiQuestionCreator deberá obtener el tema proporcionado, incluirlo en el prompt y devolver una Question válida. Deberás utilizar una clase o varias clases POJO extra con todos los miembros públicos para generar la pregunta y las opciones, puedes llamarlas QuestionDTO y OptionDTO y a partir de estas crear un Question del modelo propuesto. Para poder utilizar la SDK de Gemini será necesario que obtengáis una API_KEY. Las posibles excepciones que produzcan deberán ser envueltas con QuestionCreatorException.

NOTAS IMPORTANTES SOBRE EL CICLO DE VIDA DE LA APLICACIÓN

Al ejecutar la aplicación por consola, el usuario podrá pasarle parámetros que definirán cómo se ejecutará la aplicación. Como sabe, los parámetros pasados se encontrarán en String[] args del método main. La aplicación podrá recibir el siguiente parámetro:

-question-creator: seguido de un identificador de modelo y una API_KEY (en este caso de Gemini)

```
java -jar app.jar -question-creator gemini-2.5-flash API_KEY
```

Este parámetro define si se creará un GeminiQuestionCreator y se añadirá a la lista de QuestionCreator que tendrá disponible el modelo. De otro modo el modelo recibirá un ArrayList vacío y no ofrecerá la opción al usuario de crear preguntas de forma automática.

REQUISITOS TÉCNICOS Y RECOMENDACIONES

- La aplicación deberá poder ejecutarse en **Linux, Windows y macOS**.
- La aplicación es **multiplataforma,** por tanto, no se debe hacer referencia a ningún aspecto de un sistema operativo concreto.
- La aplicación debe presentar una **E/S por consola robusta**.
- Se deben **controlar las posibles excepciones** y gestionarlas correctamente.
- Se deben seguir los **principios de diseño del MVC**, respetando las responsabilidades de cada una de las partes de la aplicación. Los errores restarán puntos en función de su gravedad.
- Se debe utilizar una **sintaxis camelCase**, propia de Java.
- **Revisad los proyectos proporcionados** a lo largo del curso y las indicaciones proporcionadas en este enunciado.
- Si se presentan dudas, dificultades o necesitáis aclaraciones sobre el enunciado, lo mejor es consultarlas con los profesores de la asignatura y en el foro de Studium.
- Si se trata de *bugs*, tras haber invertido un mínimo de tiempo y esfuerzo en resolverlos, los podéis consultar también al profesorado. Saber cómo depurar el código es una competencia que hay que adquirir en esta asignatura.

REQUISITOS DE LA ENTREGA EN STUDIUM **(MUY IMPORTANTE)**

Estos requisitos son obligatorios, la ausencia de alguno de ellos supondrá un 0 suspenso en este trabajo.

- Fecha límite de entrega: 11 diciembre de 2025 23:59.
- **Proyecto completo en VSCode**, JDK mínima 21 con las bibliotecas JAR incorporadas. Se deberá entregar en un zip con la nomenclatura **Apellido1Apellido2Nombre.zip. Se deberá incluir la carpeta .git y el JAR de la solución final del proyecto. Recomendamos crear un repositorio a parte privado para la práctica.**
- **Informe del trabajo en PDF con resultados de la ejecución del programa** y unas mínimas explicaciones de cada sección que demuestren la realización del trabajo. El fichero tiene que tener el nombre Apellido1Apellido2Nombre_Informe.pdf. Se debe enfocar este informe como un manual técnico que acompaña el software con explicaciones de la funcionalidad y capturas de pantalla del resultado de la ejecución de las funcionalidades del proyecto. Algo que le proporcionaría a otro desarrollador. Utilice una plantilla de Google Docs o Word y procure que tenga un buen formato. Repetimos, el informe es un requisito, sin él no se evalúa la práctica. No se debe incluir un "*copy paste*" del código en el informe (ya lo estáis subiendo con el proyecto). No debe haber capturas de pantalla sin explicación y no se debe incluir de nuevo el enunciado completo. No se evalúa al peso, esto busca demostrar que habéis probado el software y que domináis lo que habéis entregado. Posible esquema de contenidos del informe:
    - Portada. Nombre, DNI etc.
    - Introducción: breve explicación
    - Funcionalidad 1: demostración/captura y explicación
    - Funcionalidad 2: demostración/captura y explicación
    - …
    - Problemáticas, aspectos a mejorar, posibles líneas futuras.

RÚBRICA PARA LA PRÁCTICA

Característica de la aplicación	%	Excelente (9-10)	Notable(7-8)	Suficiente (5-6)	Deficiente (0-4)
CRUD PREGUNTAS Listados ordenados Alta Modificación Borrado	25%	El proyecto cumple con los conceptos de diseño MVC y no tiene errores. Pasa todos los test realizados.	El proyecto cumple con los conceptos de diseño del MVC, con algún error puntual.	El proyecto cumple con los conceptos de diseño del MVC, pero presenta algunos errores leves.	El proyecto no cumple con los conceptos de diseño del MVC, o presenta errores graves en la utilización de este patrón.
Modo Examen Siguiendo MVC	25%	Se presenta esta característica completamente funcional y sin fallos y cumpliendo MVC.	El proyecto presenta completamente funcional esta característica con algún fallo puntual.	El proyecto presenta esta característica, pero con errores o deficiencias.	No funciona esta característica o presenta errores graves de diseño.
Exportación Importación JSON	10%	Se presenta esta característica completamente funcional y sin fallos y cumpliendo MVC.	El proyecto presenta completamente funcional esta característica con algún fallo puntual.	El proyecto presenta esta característica, pero con errores o deficiencias.	No funciona esta característica o presenta errores graves de diseño.
Serialización	5%	Se presenta esta característica completamente funcional y sin fallos y cumpliendo MVC.	El proyecto presenta completamente funcional esta característica con algún fallo puntual.	El proyecto presenta esta característica, pero con errores o deficiencias.	No funciona esta característica o presenta errores graves de diseño.
Parámetros entrada y configuración MVC inicial	5%	Se presenta esta característica completamente funcional y sin fallos. Con parámetros y configuración del MVC correcta.	Se utilizan parámetros, pero existe algún error puntual.	No se utilizan parámetros, pero si se construye el MVC en el main.	No funciona esta característica o presenta errores graves de diseño.
"One more thing" Creador de preguntas IA (Uso de bibliotecas externas e integración de código de terceros)	20%	Se presenta esta característica completamente funcional y sin fallos.	El proyecto presenta completamente funcional esta característica con algún fallo puntual.	El proyecto presenta esta característica, pero con errores o deficiencias.	No funciona esta característica o presenta errores graves de diseño.
Calidad del informe[1]	10%	Se presenta un informe detallado, siguiendo las indicaciones de la práctica, mostrando la funcionalidad con capturas y con unas mínimas explicaciones. Además, se cuida la presentación, la ortografía, etc.			El informe presentado no tiene capturas o no explica las diferentes funcionalidades de la aplicación. Presenta faltas de ortografía, deficiencias de formato, etc.

[1] **El informe es un requisito**. Si no se entrega no se evaluará la práctica y se considerará como no presentado. Además, el informe tiene su peso en la nota de la práctica ya que demuestra los tests realizados y cómo se ha abordado la práctica.