AIM

The aim of this coursework is to apply and reinforce the design principles discussed throughout the lectures, showcasing an understanding of key concepts in object-oriented programming and software design. This project involves the development of interfaces and classes that adhere to best practices, such as the appropriate overriding of methods from the Object class, including the toString method, and implementing a static valueOf method where applicable. The coursework emphasizes the design of interface-based hierarchies, allowing for programming through interfaces and late binding, which promotes flexibility and scalability in software development. Furthermore, it explores the use of factory patterns to manage object instantiation, ensuring the creation of unique instances and enhancing code maintainability. Defensive programming principles, including immutability, are employed to safeguard the integrity of data and prevent unintended modifications. The coursework also incorporates appropriate classes and interfaces from the Collections framework to manage data effectively, alongside the use of Javadoc comments for thorough documentation of interfaces and classes, facilitating better understanding and usability of the code. Finally, unit testing is conducted using JUnit to validate the functionality and reliability of the implemented classes and interfaces, ensuring that the system meets its design specifications and performs as expected.

COURSEWORK

The **Quiz Management System** is designed to efficiently manage the construction, conducting, and assessment of quizzes containing both fixed-response and multiple-choice questions in a user-friendly environment. It will be developed in Java to enhance interactivity in learning by making it easier for instructors to maintain quizzes while assessing learners in an engaging manner. This system adheres to object-oriented programming principles, ensuring maintainability, scalability, and clear separation of concerns.

The system is structured into key components such as:

- **Database** for storing questions,
- A set of **POJOs** representing the question attributes,
- A **quiz engine** that manages quiz flow and evaluates answers,
- A set of **tests** for system robustness verification.

The database component comprises two primary classes: `**FixedResponseQuestionSet**` and `**MultipleChoiceQuestionSet**`, which manage question sets. These classes generate predefined questions and provide access for the quiz system.

The POJO layer includes classes that define the characteristics and behavior of questions. Specifically, `**FixedResponseQuestion**` and `**MultipleChoiceQuestion**` implement a common interface, `**Question**`, unifying the structure of all question types. This design simplifies quiz generation and grading by allowing the system to handle different question types uniformly. Each question class provides methods to access the question text and evaluate user responses, promoting encapsulation and data integrity.

The core functionality of the quiz system is implemented in the `**Quiz**` class, which facilitates quiz-taking for students. This class randomly selects questions from the question sets and records student answers to provide immediate feedback on their performance. It tracks the number of attempts and modifications made by each student to generate comprehensive statistics. Finally, the system compares student performances against a predefined passing threshold to make a final decision based on aggregated data.

Additionally, the project includes a testing apparatus to ensure the system functions correctly without producing errors. Tests are designed to check question sets, the quiz system, and individual question classes for proper functionality. They cover typical operations, edge cases, and error conditions, validating the system's predictable behavior across various scenarios. By adopting rigorous testing methodologies, the project aims to build a resilient and reliable quiz management system capable of handling real-world demands.

Potential enhancements for the Quiz Management System include:

- **User authentication** for a personalized experience,
- **Multimedia-based questions**,
- The development of **adaptive learning capabilities** that adjust quiz difficulty based on student performance.

Incorporating mocking or stubbing during testing could further improve system reliability by isolating individual components and simulating different situations. With continued improvements and new features, this project could evolve into a comprehensive learning tool catering to the diverse needs of students and educators.

In summary, the **Quiz Management System** represents a significant step forward in educational technology. It provides an effective platform for creating and assessing quizzes with a user-friendly interface and a robust backend. The system is both maintainable and scalable, laying the groundwork for further innovations that could make learning more effective for students and quiz administration easier for educators.

TEST CASES EXPLANATION

1. LIBRARYSYSTEMFIXEDRESPONSETEST

This class contains a series of test cases designed to validate the functionality of a quiz system that uses fixed-response questions. Each test case evaluates different scenarios that a student may encounter while taking a quiz.

TEST CASES:

i. **test_AttemptOneFail():**
   Objective: Verify that a student fails the quiz if they answer only one out of three questions correctly.
   Process:
   Create a student and a quiz instance.
   Add three fixed-response questions related to programming languages, cloud services, and HTTP ports.
   Provide incorrect answers for two questions and one correct answer.
   Assert that the final verdict is null (indicating failure) and that the score reflects the correct answer ratio (1 out of 3).

ii. **testAttemptOneFail_AttemptTwoFail():**
   Objective: Ensure the student fails after two consecutive attempts with the same incorrect answers.
   Process:
   Similar to the first test, but the student takes the quiz twice with the same incorrect answers.
   Assert that the final verdict is FAIL and both attempts have the same score.

iii. **test_AttemptOnePass():**
   Objective: Confirm that a student passes the quiz on their first attempt when answering correctly.
   Process:
   The student answers all three questions correctly.
   Assert that the final verdict is PASS and the score reflects the correct answer ratio (2 out of 3).

iv. **test_AttemptOnePass_AttemptTwoInvalid():**
   Objective: Validate that a second attempt is ignored if the student passes the first attempt.
   Process:
   The student answers correctly on the first attempt.
   The second attempt is made with different answers, but the final verdict should not change.
   Assert that the student's final verdict remains PASS and that the second attempt score is 0.

v. **test_ExtraSpacesAndIgnoreCase():**
   Objective: Check if the system handles extra spaces and case differences in answers correctly.
   Process:
   Provide answers with extra spaces and different casing.
   The first quiz attempt should reflect the student's score accurately.
   A second attempt with normalized answers should allow the student to pass.
   Assert that the final verdict reflects the change.

2. LIBRARYSYSTEMMULTIPLECHOICETEST

This class tests the multiple-choice functionality of the quiz system. It ensures that the system can handle multiple-choice questions and verify answers correctly.

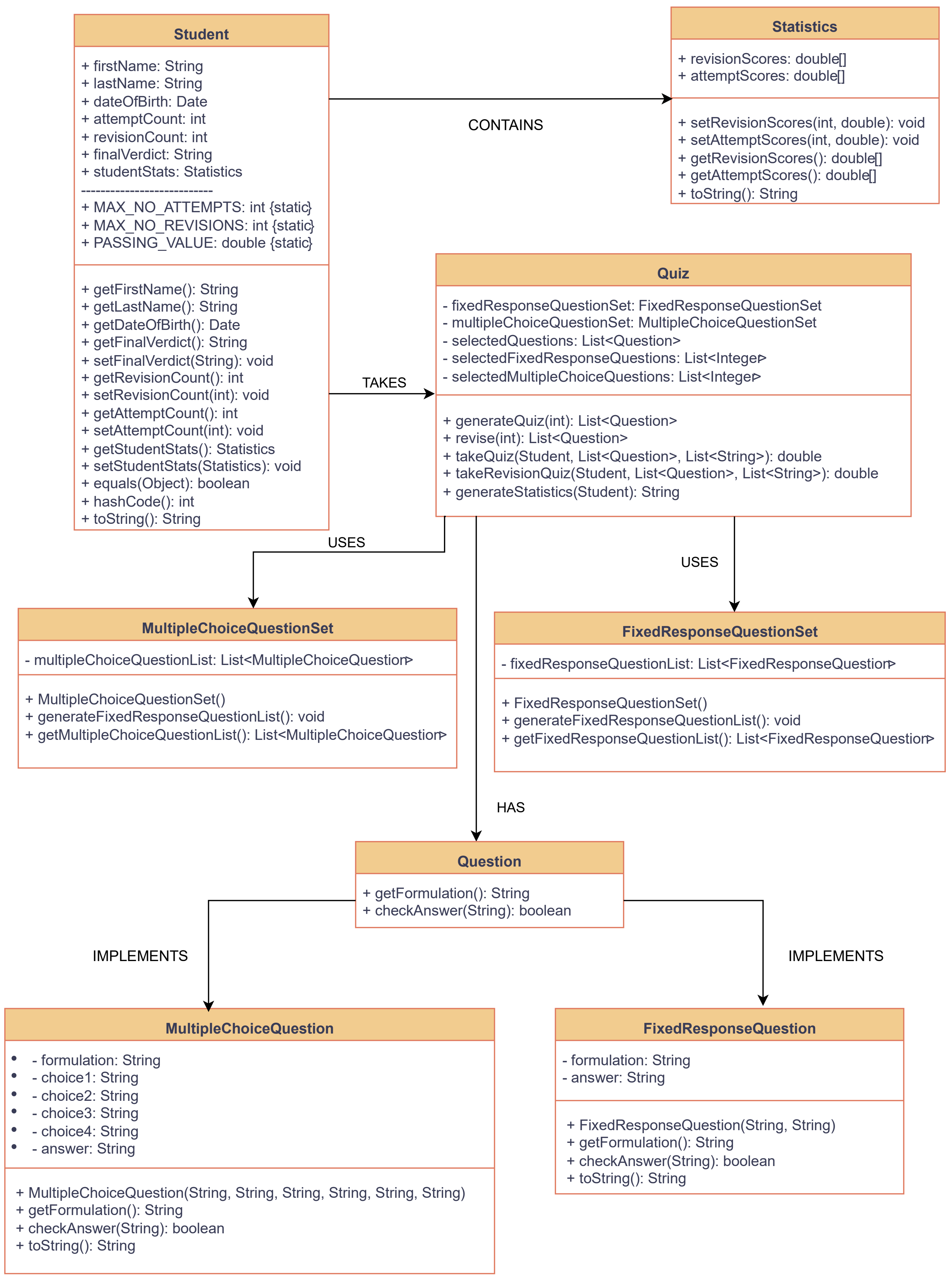
TEST CASES:

vi. **test_AttempOnePass():**
   Objective: Verify that a student passes the multiple-choice quiz on their first attempt when answering correctly.
   Process:
   Create a student and a quiz instance.
   Add three multiple-choice questions covering programming languages, databases, and cloud platforms.
   Provide the correct set of answers as a comma-separated list.
   Assert that the final verdict is PASS and that the score reflects 100% correct answers (1.0).

REFERENCES

a. **draw.io:** a tool for creating UML diagrams.
b. **Stack overflow:** a community-driven platform for Java programming questions and answers

UML DIAGRAM:


Student
+ firstName: String
+ lastName: String
+ dateOfBirth: Date
+ attemptCount: int
+ revisionCount: int
+ finalVerdict: String
+ studentStats: Statistics

+ MAX_NO_ATTEMPTS: int {static}
+ MAX_NO_REVISIONS: int {static}
+ PASSING_VALUE: double {static}
+ getFirstName(): String
+ getLastName(): String
+ getDateOfBirth(): Date
+ getFinalVerdict(): String
+ setFinalVerdict(String): void
+ getRevisionCount(): int
+ setRevisionCount(int): void
+ getAttemptCount(): int
+ setAttemptCount(int): void
+ getStudentStats(): Statistics
+ setStudentStats(Statistics): void
+ equals(Object): boolean
+ hashCode(): int
+ toString(): String
CONTAINS
Statistics
+ revisionScores: double[]
+ attemptScores: double[]
+ setRevisionScores(int, double): void
+ setAttemptScores(int, double): void
+ getRevisionScores(): double[]
+ getAttemptScores(): double[]
+ toString(): String
TAKES
Quiz
- fixedResponseQuestionSet: FixedResponseQuestionSet
- multipleChoiceQuestionSet: MultipleChoiceQuestionSet
- selectedQuestions: List<Question>
- selectedFixedResponseQuestions: List<Integer>
- selectedMultipleChoiceQuestions: List<Integer>
+ generateQuiz(int): List<Question>
+ revise(int): List<Question>
+ takeQuiz(Student, List<Question>, List<String>): double
+ takeRevisionQuiz(Student, List<Question>, List<String>): double
+ generateStatistics(Student): String
USES
USES
MultipleChoiceQuestionSet
- multipleChoiceQuestionList: List<MultipleChoiceQuestion>
+ MultipleChoiceQuestionSet()
+ generateFixedResponseQuestionList(): void
+ getMultipleChoiceQuestionList(): List<MultipleChoiceQuestion>
FixedResponseQuestionSet
- fixedResponseQuestionList: List<FixedResponseQuestion>
+ FixedResponseQuestionSet()
+ generateFixedResponseQuestionList(): void
+ getFixedResponseQuestionList(): List<FixedResponseQuestion>
HAS
Question
+ getFormulation(): String
+ checkAnswer(String): boolean
IMPLEMENTS
IMPLEMENTS
MultipleChoiceQuestion
- formulation: String
- choice1: String
- choice2: String
- choice3: String
- choice4: String
- answer: String
+ MultipleChoiceQuestion(String, String, String, String, String, String)
+ getFormulation(): String
+ checkAnswer(String): boolean
+ toString(): String
FixedResponseQuestion
- formulation: String
- answer: String
+ FixedResponseQuestion(String, String)
+ getFormulation(): String
+ checkAnswer(String): boolean
+ toString(): String