Parson’s Problems Application for Learning Management Systems
Design Document

Authors:

Devin Allen
Stephen Cox
Benjamin Maymir
Zhenyu Yang

Group: 17

Document Revision History

Date	Version	Description	Author
11/9/2022	1.0	Initial draft.	Devin Allen Stephen Cox Benjamin Maymir Zhenyu Yang

Table of Contents

1 Introduction

1.1 Purpose

This document describes the design, specifications, uses, and constraints of the Parson's Problems Appliance for Learning Management Systems (PPALMS) application. The document begins with an overview of the PPALMS system – describing its intended purpose, base functionality, and intended audience – followed by the system's design and architecture (Section 2), subsystems and constraints, interface and data stores (Section 3), structure and classes (Section 4), use diagrams (Section 5), fulfillment of non-functional requirements (Section 6), and traceability of requirements (Section 7).

1.2 System Overview

PPALMS is a stand-alone application that allows users to import source code from a file and generate quiz or exam questions from lines of programming code. Question types include multiple-choice, line reordering, 2D problems (indentation), fill-in-the-blank, and bug selection. The system saves these questions to a file of type (.qti) which can be imported into Learning Management Systems (LMS): Canvas, Moodle, or Blackboard. PPALMS is intended for use by instructors, institutions, and researchers using Learning Management Systems (LMS) at all education levels. PPALMS shall operate on current versions of Windows, Linux, and Mac computers, in an environment in which the user can create, modify, and execute files.

1.3 Design Objectives

PPALMS allows for easy, simple generation of computer-science-related exam questions by automating the action of modifying or rearranging lines of source code. This relieves users (instructors and organizations) of the manual process and reduces the time and resources necessary to create quizzes for students.

1.4 References

This document may refer to the Software Requirements Specification for PPALMS (version 1.1) as the 'Requirements Document' or 'Requirements'.

1.5 Definitions, Acronyms, and Abbreviations

For simplicity, the Parson's Problems Appliance for Learning Management Systems will be denoted as **PPALMS** in most cases. Learning Management System (those such as Canvas, Moodle, or Blackboard) will be denoted as **LMS**. Any mention of an instructor is assumed to be a computer science educator using the system to generate questions. The words 'questions' or 'problems' carry identical meaning and are used interchangeably. A 'mutation' denotes a version of a generated problem for selected lines of code. Other definitions can be found in Appendix A: Glossary.

2 Design Overview

2.1 Introduction

PPALMS shall be built using an object-oriented approach to allow communication and data processing between system components. The system shall feature a type of Generic Layered Architecture containing interconnected parts: Interface, Controller, Question Generator, and Questions. Question Generators shall be built using the strategy design pattern so the Controller can use proper question-generating algorithms based on the user's chosen settings.

2.2 Environment Overview

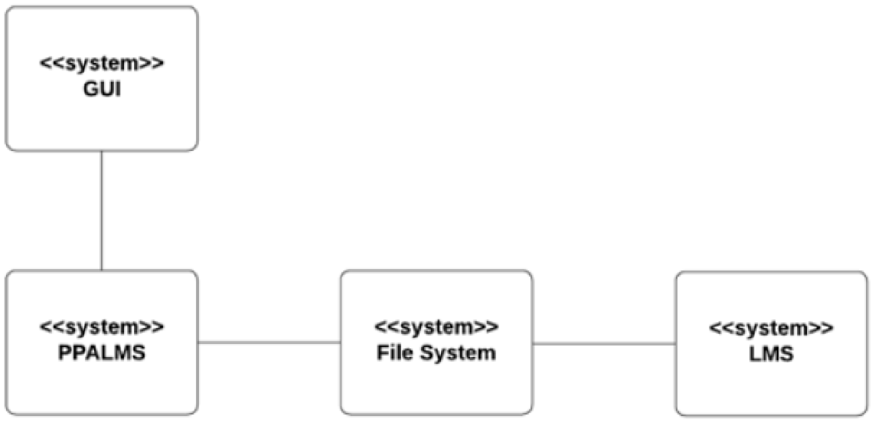

Figure 1: PPALMS Context Model

PPALMS is a stand-alone application that will be executable within a Windows, Linux, or other operating system. The system shall be able to output .qti files to the directory from which it receives imported source code.

2.3 System Architecture

2.3.1 Top-level system structure of PPALMS

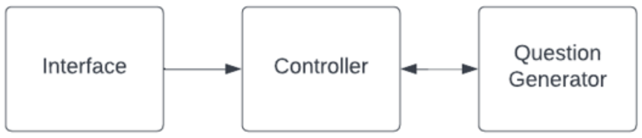

Figure 2: Top-level PPALMS Structure

The system consists of three major components: a graphical Interface, a system Controller, and a Question Generator. The user interacts with the Interface to send information to the Controller – the source code file path and the settings which affect question generation. The Controller also contains a type of Question Generator based on the chosen question type and saves all of the questions generated by the Generator.

2.3.2 Interface

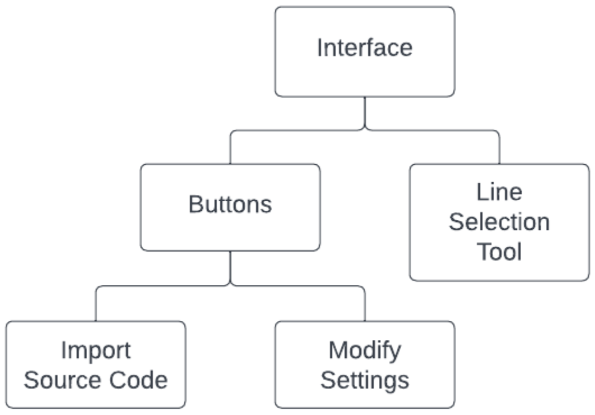

Figure 3: Interface Context Diagram

The Interface consists of buttons for modifying the intended LMS target, question type, and number of generated questions; it also contains a button which allows the user to select a file to import from their system or a text box to enter a direct file path. The Interface allows the user to select which lines of code to use in question generation after importing a valid file. The lines, source code, and settings are passed into the Controller.

2.3.3 Controller

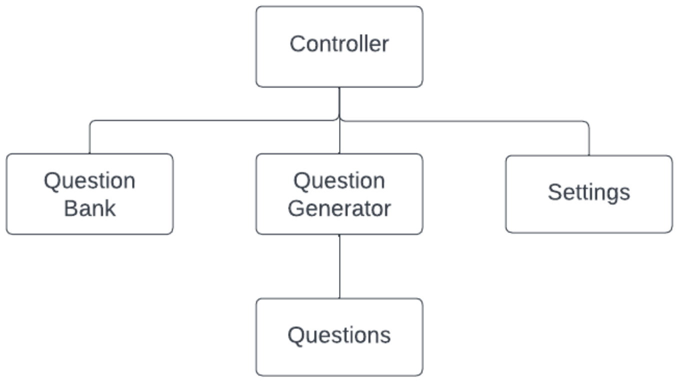

Figure 4: Controller Context Diagram

The Controller contains all system settings (mutation limit, intended LMS, question type, and source code file path) used in Question Generation. These settings are passed to the Generator whose type is decided by the selected question type by the user. Each type of Question Generator uses its own algorithm (along with the initial settings) to create mutations of problems of the intended type. These Questions are passed back to the Controller and saved in a Question Bank for later export into a .qti file.

2.3.4 Question Generator

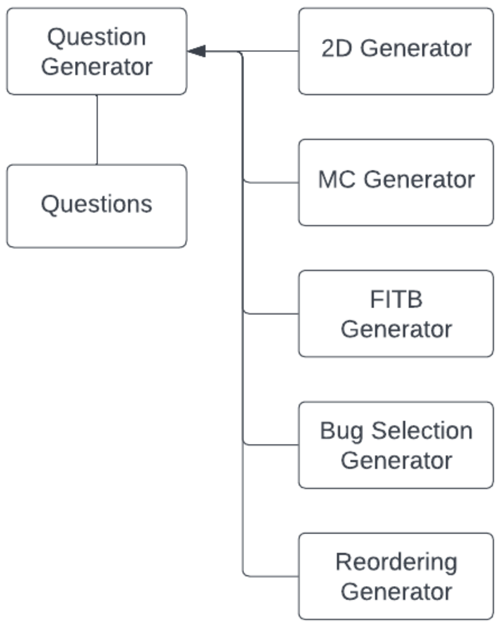

Figure 5: Question Generator Context Diagram

A Question Generator is selected based on the chosen question type and then uses its own algorithms to generate Questions. These Questions contain pertinent correct and incorrect answers (depending on the question type) along with text pertaining to that question and the type of question. Generated questions are then passed back to the Controller for saving and exporting.

2.4 Constraints and Assumptions

1. Question generation should not stagger system performance.

   This is addressed by limiting the number of questions generated to a maximum of 25 with a default amount set to 5. The system creates mutations based on selected lines of code and must have a limit to…

2. PPALMS must allow users to select lines of code from an imported source file.

   The system shall be developed using the object-oriented programming language, C++, which allows data reading and writing from files using file paths or file descriptors. The system can be coupled with the GTK development tool kit which allows for a graphical user interface for simple line-selection and buttons for changing the system's settings.

3. Large source code files may inhibit problem generation or output.

   PPALMS shall require the computer to contain enough memory and storage to allow for problem generation and creation of a .qti file (the common file type for use with an LMS) containing the sets of problems.

3 Interfaces and Data Stores

3.1 System Interfaces

3.1.1 Graphical User Interface

This interface is used to record the lines the user selected to generate related problems. The lines that the user specified will determine whether it exceeds the line limit or not, if it does, a warning message will be displayed on the screen.

3.1.2a Source Code Interface

This interface is used to import the source code. The code shall be stored in the local file system and the data can be accessed by the PPALMS using I/O stream.

3.1.2b Settings Configuration Interface

This interface is used by the instructor to set the system's configuration settings. Upon completion the settings shall be stored on the file system. The settings detail mutation amount, intended LMS, and question type. The data can later be accessed by the PPALMS system at any given point during runtime.

3.1.2c QTI File Interface

This interface is used to export the generated questions to a .qti file, after the system generates problems, the problems will be saved into a question bank and then exported to the instructor's file system using this interface.

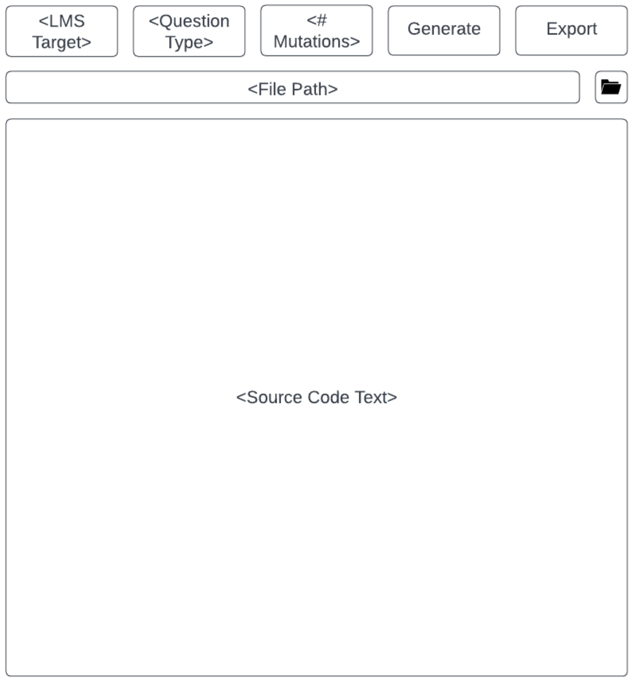

Figure 6: Sketch of PPALMS' Graphical User Interface

3.2 Data Stores

The PPALMS system shall create a data store of the instructor's configured settings upon initial configuration interaction with the system's GUI. Additionally the multitude of questions produced from question generation shall be stored in a question bank object which can then be exported to the instructor's hard disk drive as a .qti file.

4 Structural Design

4.1 Class Diagrams

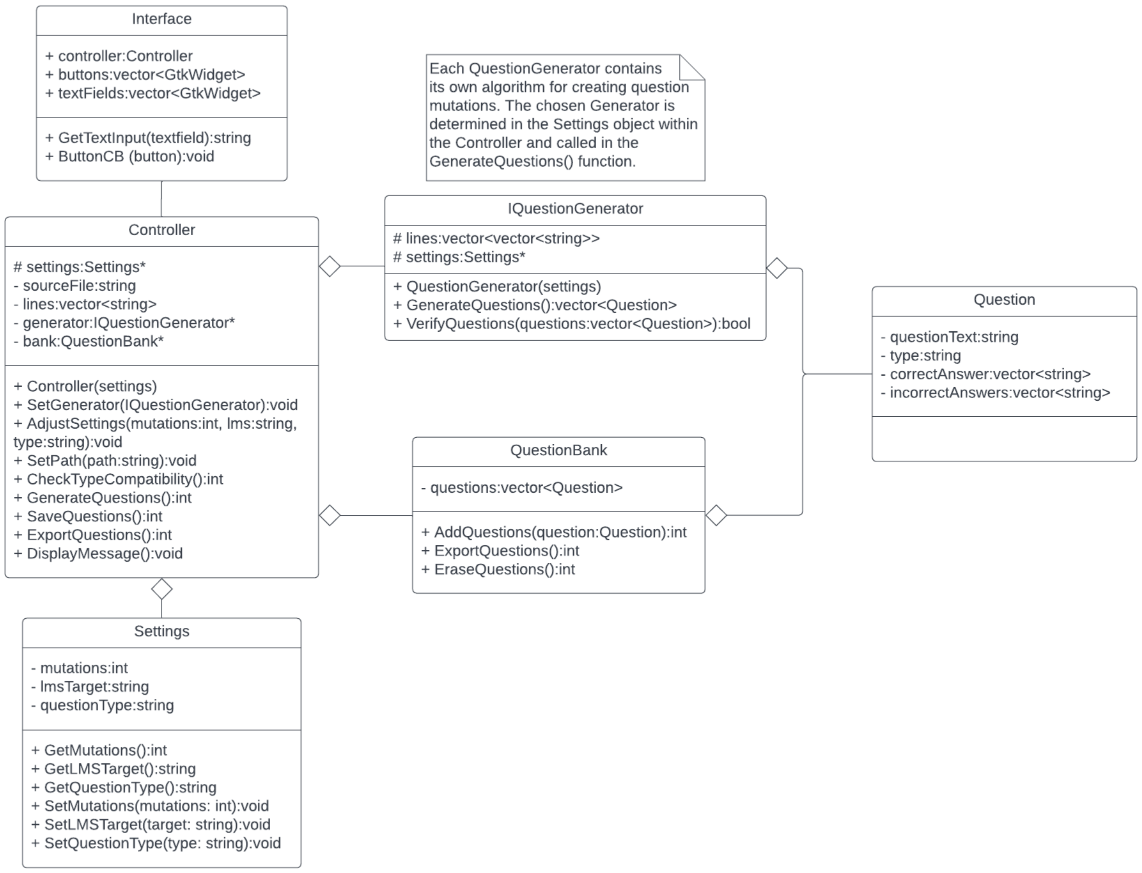
Interface
+ controller:Controller
+ buttons:vector<GtkWidget>
+ textFields:vector<GtkWidget>
+ GetTextInput(textfield):string
+ ButtonCB (button):void
Each QuestionGenerator contains its own algorithm for creating question mutations. The chosen Generator is determined in the Settings object within the Controller and called in the GenerateQuestions() function.
IQuestionGenerator
# lines:vector<vector<string>>
# settings:Settings*
+ QuestionGenerator(settings)
+ GenerateQuestions():vector<Question>
+ VerifyQuestions(questions:vector<Question>):bool
Controller
# settings:Settings*
- sourceFile:string
- lines:vector<string>
- generator:IQuestionGenerator*
- bank:QuestionBank*
+ Controller(settings)
+ SetGenerator(IQuestionGenerator):void
+ AdjustSettings(mutations:int, lms:string, type:string):void
+ SetPath(path:string):void
+ CheckTypeCompatibility():int
+ GenerateQuestions():int
+ SaveQuestions():int
+ ExportQuestions():int
+ DisplayMessage():void
Question
- questionText:string
- type:string
- correctAnswer:vector<string>
- incorrectAnswers:vector<string>
QuestionBank
- questions:vector<Question>
+ AddQuestions(question:Question):int
+ ExportQuestions():int
+ EraseQuestions():int
Settings
- mutations:int
- lmsTarget:string
- questionType:string
+ GetMutations():int
+ GetLMSTarget():string
+ GetQuestionType():string
+ SetMutations(mutations: int):void
+ SetLMSTarget(target: string):void
+ SetQuestionType(type: string):void

Figure 7: Class Diagram 1 - PPALMS Overview

The user interacts with the system's Interface, which allows them to change the system's settings through the system's main Controller class. The Controller class contains all methods for event handling, which prompt changes to the Settings object class, save questions to the Question Bank class, or generate questions through a type of Question Generator class (determined by the question type from Settings). Question Generators create questions containing a question type, correct and incorrect answers, and a string of text to explain the question. These questions are passed back through the Controller and saved into the Question Bank.

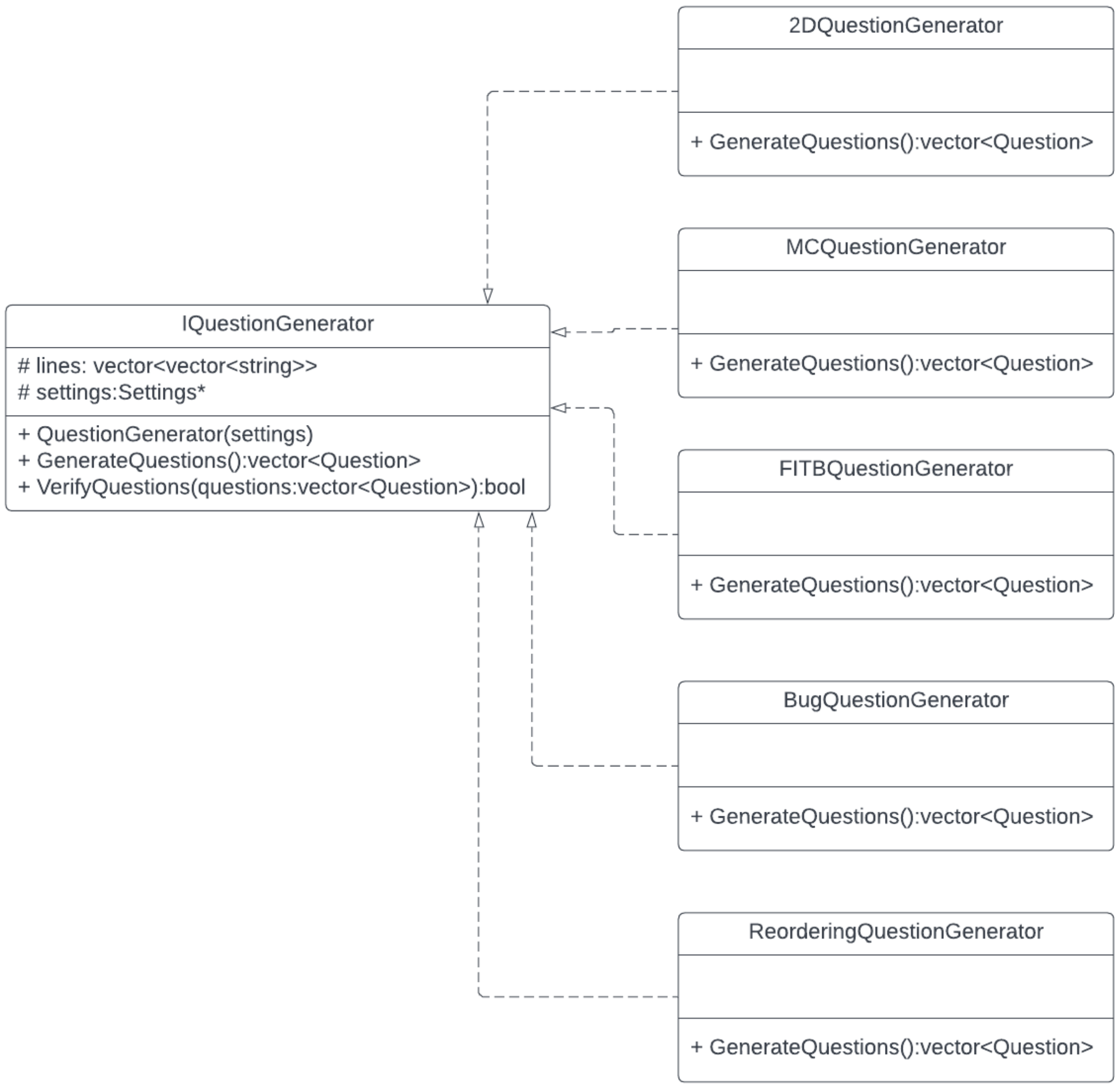
2DQuestionGenerator
+ GenerateQuestions():vector<Question>
MCQuestionGenerator
+ GenerateQuestions():vector<Question>
IQuestionGenerator
# lines: vector<vector<string>>
# settings:Settings*
+ QuestionGenerator(settings)
+ GenerateQuestions():vector<Question>
+ VerifyQuestions(questions:vector<Question>):bool
FITBQuestionGenerator
+ GenerateQuestions():vector<Question>
BugQuestionGenerator
+ GenerateQuestions():vector<Question>
ReorderingQuestionGenerator
+ GenerateQuestions():vector<Question>

Figure 8: Class Diagram 2 - Question Generators

Each Question Generator subclass derives from IQuestionGenerator, each having access to a '*lines*' variable which contains the lines of code to generate questions from. '*lines*' has been noted as a vector of vectors, as Reordering Questions require the lines of code to be rearranged by sets or tuples. Each Question Generator uses these lines and the passed Settings to generate questions using their own unique algorithms. A number of questions equal to the mutation amount from the Settings are created and passed back to the system's Controller as a set.

4.2 Class Descriptions

4.2.1 Class: Interface

- Purpose: Allow users to interact with the system.
- Constraints: Limited to changing problem-generation settings, importing source code, selecting lines of code, and exporting questions.
- Persistent: No (created at system initialization from other available data)

4.2.1.1 Attribute Descriptions

1. Attribute: Controller
   Type: class (object)
   Description: The system's main controller that handles all events.
   Constraints: N/A

2. Attribute: buttons
   Type: vector<GtkWidget>
   Description: The buttons on the interface that allow the user to change settings and prompt the Controller to generate or export questions.
   Constraints: Only function if the user has imported source code.

3. Attribute: textFields
   Type: vector<GtkWidget>
   Description: A text box that the user can enter a file path.
   Constraints: N/A

4.2.1.2 Method Descriptions

1. Method: GetTextInput(textfield)
   Return Type: string
   Parameters: textField – The text box that the user enters a file path.
   Return value: A string containing the file path.
   Pre-condition: There exists a source code file in the file system.
   Post-condition: The Controller's sourceFile attribute is set to the string.
   Attributes read/used: textFields, sourceFile
   Methods called: N/A

   Processing logic: The file path is entered into the textField by the user. The Controller takes the string from the textField and saves it into its sourceFile variable.

2. Method: ButtonCB(button)
   Return Type: void
   Parameters: a button to invoke a method.
   Return value: N/A
   Pre-condition: the button exists.
   Post-condition: the button calls a Controller's method.
   Attributes read/used: Controller
   Methods called: a pertinent method on the Controller.

   Processing logic: The user clicks on one of the GUI's buttons. Each button corresponds to different methods on the Controller that are called when that button is clicked.

4.2.2 Class: Controller

- Purpose: Control the flow of execution and link all subsystems together.
- Constraints: Interface required to call methods of the Controller.
- Persistent: No (exists only when system is running)

4.2.2.1 Attribute Descriptions

1. Attribute: settings
   Type: Settings* (pointer)
   Description: an object containing all system settings.
   Constraints: N/A

2. Attribute: sourceFile
   Type: string
   Description: the file path for the desired source code file.
   Constraints: file path must exist.

3. Attribute: lines
   Type: vector<string>
   Description: selected lines of source code by the user.
   Constraints: source code must be imported before selecting. Otherwise, null.

4. Attribute: generator
   Type: IQuestionGenerator
   Description: One of five different QuestionGenerator objects each with their own algorithms for generating questions.
   Constraints: a question type must be selected by the user. Otherwise, null.

5. Attribute: bank
   Type: QuestionBank* (pointer)
   Description: an object that holds onto all saved and generated questions.
   Constraints: will not contain questions unless they are generated.

4.2.2.2 Method Descriptions

1. Method: Controller(settings), constructor
   Return Type: N/A
   Parameters: settings – a Settings object for which to save configured settings.
   Return value: N/A
   Pre-condition: N/A
   Post-condition: the Controller will contain a new Settings object.
   Attributes read/used: settings
   Methods called: N/A

   Processing logic: The Controller needs to hold onto any settings modified by the user in order to generate the proper number and type of questions.

2. Method: SetGenerator(IQuestionGenerator)
   Return Type: void
   Parameters: one of five different Question Generators based on the configured Settings.
   Return value: N/A
   Pre-condition: a question type has been selected by the user.
   Post-condition: the Controller will have the correct Question Generator object.
   Attributes read/used: questionType (from Settings)
   Methods called: N/A
   Processing logic: The Controller creates a new Question Generator and holds onto it.

3. Method: AdjustSettings(mutations, lms, type)
   Return Type: void
   Parameters: mutations (int), lms (string), type (string)
   Return value: N/A
   Pre-condition: the user adjusts a setting through the Interface.
   Post-condition: the Settings object on the Controller holds the new desired settings.
   Attributes read/used: the Settings object on the Controller.
   Methods called: SetMutations(mutations), SetLMSTarget(target), SetQuestionType(type)

   Processing logic: The user modifies one of the three settings through the Interface. The Controller then modifies its Settings object to hold the desired settings configuration.

4. Method: SetPath(path)
   Return Type: void
   Parameters: a file path entered by the user.
   Return value: N/A
   Pre-condition: the user has entered a file path or imported one through the GUI.
   Post-condition: the Controller object holds onto the file path in its sourceFile variable.
   Attributes read/used: sourceFile, textFields
   Methods called: N/A

   Processing logic: The user enters a file path in the text field or clicks a button on the interface to find a file path. The Controller takes that file path and saves it into a variable.

5. Method: CheckTypeCompatibility()
   Return Type: int
   Parameters: N/A
   Return value: on success returns 1, on failure returns 0.
   Pre-condition: a question type has been selected by the user.
   Post-condition: a warning is shown to the user if their lines of code cannot be used to generate questions of that type.
   Attributes read/used: questionType (string)
   Methods called: GetQuestionType() from Settings

   Processing logic: The user enters a question type through the interface. When the user selects the Generate button through the interface, the system determines if the lines of code can be used to generate questions of their selected type. If yes, no further events occur. If not, a warning message is shown to the user.

6. Method: GenerateQuestion()
   Return Type: int
   Parameters: N/A
   Return value: on success returns 1, on failure returns 0.
   Pre-condition: settings have been configured and the user has selected lines of code.
   Post-condition: questions are generated and saved based on the given settings and lines.
   Attributes read/used: Settings (the values in the object), IQuestionGenerator, QuestionBank
   Methods called: GenerateQuestions(), VerifyQuestions(questions), SaveQuestions()

   Processing logic: The user modifies system settings then prompts the system to generate questions. The system's Controller then invokes the GenerateQuestions method on the QuestionGenerator it holds and then saves those questions into the QuestionBank.

7. Method: SaveQuestions()
   Return Type: int
   Parameters: N/A
   Return value: on success returns 1, on failure returns 0.
   Pre-condition: the Controller's QuestionGenerator object has generated questions.
   Post-condition: the generated questions are saved into the QuestionBank object.
   Attributes read/used: QuestionGenerator, questions (in QuestionBank)
   Methods called: AddQuestions(question)

   Processing logic: After questions are generated by the Controller's QuestionGenerator, they are saved into a QuestionBank object. If saving fails, the method returns a 0 and shows an error message to the user. On success, the method returns a 1 and shows a success message.

8. Method: ExportQuestions()
   Return Type: int
   Parameters: N/A
   Return value: on success returns 1, on failure returns 0.
   Pre-condition: questions have been saved into the QuestionBank object.
   Post-condition: questions have been exported into a .qti file in the file system.
   Attributes read/used: QuestionBank
   Methods called: ExportQuestions()

   Processing logic: After questions are generated and saved into the QuestionBank, the user may export the sets of questions to a .qti file, which can be imported into their desired LMS.

9. Method: DisplayMessage()
   Return Type: void
   Parameters: N/A
   Return value: N/A (outputs error and success messages to the Interface)
   Pre-condition: an event prompting a message to the user has occurred in the system.
   Post-condition: a message is displayed to the user.
   Attributes read/used: dependent on the Controller event.
   Methods called: dependent on the Controller event.

   Processing logic: The Controller encounters an event in which a system message should be displayed to the user (usually successes or failures of certain function calls or events). A pertinent message is then displayed to the user.

4.2.3 Class: Settings

- Purpose: Collects and holds settings related to question generation
- Constraints: Limited to holding values specified by the user
- Persistent: No (only exists while system is running)

4.2.3.1 Attribute Descriptions

1. Attribute: mutations
   Type: int
   Description: The number of mutations that will be created per selected line
   Constraints: Must hold a value that is greater than or equal to 0.

2. Attribute: lmsTarget
   Type: string
   Description: The LMS target for which the .qti file will be generated
   Constraints: Must be one of “Canvas”, “Moodle” or “Blackboard”.
3. Attribute: questionType
   Type: string
   Description: The type of question that will be generated.
   Constraints: Must be one of “2D”, “Multiple Choice”, “Fill in the Blank”, “Bug Selection”, or “Reordering”.

4.2.3.2 Method Descriptions

1. Method: GetMutations()
   Return Type: int
   Parameters: none
   Return value: the value of mutations
   Pre-condition: mutations holds a value
   Post-condition: mutations remains unchanged
   Attributes read/used: mutations
   Methods called: N/A

   Processing logic: The value of mutations is returned.

2. Method: GetLMSTarget()
   Return Type: string
   Parameters: none
   Return value: the value of LMSTarget
   Pre-condition: LMSTarget holds a value
   Post-condition: LMSTarget remains unchanged
   Attributes read/used: LMSTarget
   Methods called: N/A

   Processing logic: The value of LMSTarget is returned.

3. Method: GetQuestionTypet()
   Return Type: string
   Parameters: none
   Return value: the value of questionType
   Pre-condition: questionType holds a value
   Post-condition: questionType remains unchanged
   Attributes read/used: questionType
   Methods called: N/A

   Processing logic: The value of questionType is returned.

4. Method: SetMutations(mutations)
   Return Type: void
   Parameters: the desired number of mutations
   Return value: N/A
   Pre-condition: N/A
   Post-condition: the value of mutations is equal to the the mutations parameter
   Attributes read/used: mutations
   Methods called: none

   Processing logic: mutations is set to the value of the mutations parameter

5. Method: SetLMSTarget(LMSTarget)
   Return Type: void
   Parameters: the desired LMS target
   Return value: N/A
   Pre-condition: N/A
   Post-condition: the value of LMSTarget is equal to the LMSTarget parameter
   Attributes read/used: LMSTarget
   Methods called: none

   Processing logic: LMSTarget is set to the value of the LMSTarget parameter

6. Method: SetQuestionType(questionType)
   Return Type: void
   Parameters: the desired question type
   Return value: N/A
   Pre-condition: N/A
   Post-condition: the value of question type is equal to the questionType parameter
   Attributes read/used: questionTYpe
   Methods called: none

   Processing logic: questionType is set to the value of the questionType parameter.

4.2.4 Class: IQuestionGenerator

- Purpose: Generates questions after the user has selected lines of code and settings. Exists as a base class for the five types of Question Generators.
- Constraints: question types are limited as the class diagram 2 shows.
- Persistent: No (only exists while system is running)

4.2.4.1 Attribute Descriptions

1. Attribute: lines
   Type: vector<vector<string>>
   Description: the lines used to generate problems.
   Constraints: the number of lines should be within the limit we set, a warning message displays otherwise.

2. Attribute: settings
   Type: Settings* (pointer)
   Description: Configuration settings that user set.
   Constraints: must always be true and no errors in the configuration.

4.2.4.2 Method Descriptions

1. Method: QuestionGenerator(settings)
   Return Type: int
   Parameters: settings
   Return value: on success returns 1, on failure returns 0.
   Pre-condition: settings are True.
   Post-condition: questions generated if 1 return, 0 otherwise.
   Attributes read/used: settings,
   Methods called: GenerateQuestions(), VerifyQuestions(questions:vector<Questions>)
   Processing logic: The method calls the GenerateQuestions() function to generate questions based on the desired settings, then verifies that the questions are valid. 1 will be returned if the question generated successfully, 0 otherwise.

2. Method: GenerateQuestions()
   Return Type: int
   Parameters: None
   Return value: on success returns 1, on failure returns 0.
   Pre-condition: settings have been configured and the user has selected lines of code.
   Post-condition: questions generated if 1 return, 0 otherwise.
   Attributes read/used: settings.
   Methods called: None
   Processing logic: The method generates questions based on the desired question type from Settings. If the questions are generated successfully, 1 will be returned, 0 otherwise.

3. Method: VerifyQuestions(questions:vector<Question>)
   Return Type: int
   Parameters: questions
   Return value: on success returns 1, on failure returns 0.
   Pre-condition: question generated
   Post-condition: return 1 if question verified with no problem, 0 otherwise
   Attributes read/used: questions
   Methods called: None
   Processing logic: Questions are generated and must be verified to contain solutions and be mutations of the provided lines of code. If solutions exist and the code has been modified, 1 will be returned, 0 otherwise.

4.2.5 Class: QuestionBank

- Purpose: To store the generated questions
- Constraints: None
- Persistent: No (object is initialized at runtime and deleted upon system termination)

4.2.5.1 Attribute Descriptions

1. Attribute: questions
   Type: vector<Question>
   Description: A vector of the questions generated (questions stored as question objects)
   Constraints: None

4.2.5.2 Method Descriptions

1. Method: AddQuestions(Question)
   Return Type: int
   Parameters: None
   Return value: on success returns 1, on failure returns 0.
   Pre-condition: None
   Post-condition: Expanded questions vector
   Attributes read/used: questions
   Methods called: None

   Processing logic: The method takes in a Question object as its sole parameter and attempts to append it to the questions attribute (vector<Question>). It returns 1 upon successfully appending the question, otherwise 0 is returned.

2. Method: ExportQuestions()
   Return Type: int
   Parameters: None
   Return value: on success returns 1, on failure returns 0.
   Pre-condition: None
   Post-condition: Question objects written to .qti file
   Attributes read/used: questions.
   Methods called: None

   Processing logic: The method loops through the questions vector and attempts to write each Question object to a given .qit file. If an error occurs during any of the write operations return 0, else return 1.

3. Method: EraseQuestions()
   Return Type: int
   Parameters: None
   Return value: on success returns 1, on failure returns 0.
   Pre-condition: None
   Post-condition: the questions erased if return 1, 0 otherwise.
   Attributes read/used: questions.
   Methods called: None

   Processing logic: The method attempts to erase all the Question objects in the questions vector. If an error occurs during any of the remove operations it returns 0, else returns 1 on completion.

4.2.6 Class: Question

- Purpose: Contains a question to be saved to a QuestionBank.
- Constraints: Only generated by QuestionGenerator objects based on Settings.
- Persistent: No (only exists upon generation by a QuesitonGenerator object)

4.2.6.1 Attribute Descriptions

1. Attribute: questionText
   Type: string
   Description: text pertaining to the generated question.
   Constraints: determined by the type of question from Settings.

2. Attribute: type
   Type: string
   Description: the type of question generated (determined by the user).
   Constraints: determined by the type of question from Settings.

3. Attribute: correctAnswer
   Type: vector<string>
   Description: contains any correct answers for the generated question.
   Constraints: contains only one element if it is a Multiple Choice, Fill-in-the-Blank, or Bug Selection question type.

4. Attribute: incorrectAnswers
   Type: vector<string>
   Description: contains any incorrect answers for the question if they need to be recorded.
   Constraints: only contains incorrect answers for Multiple Choice questions.

5 Dynamic Model

5.1 Scenarios

5.1.1 Importing Source Code

Instructors and other users shall select the source code file they wish to import by using the system's interface. The system's controller receives the source code file and saves its path until it is ready to generate questions. The interface shall display a message to the user on success or an error if their selected file is invalid or does not exist.

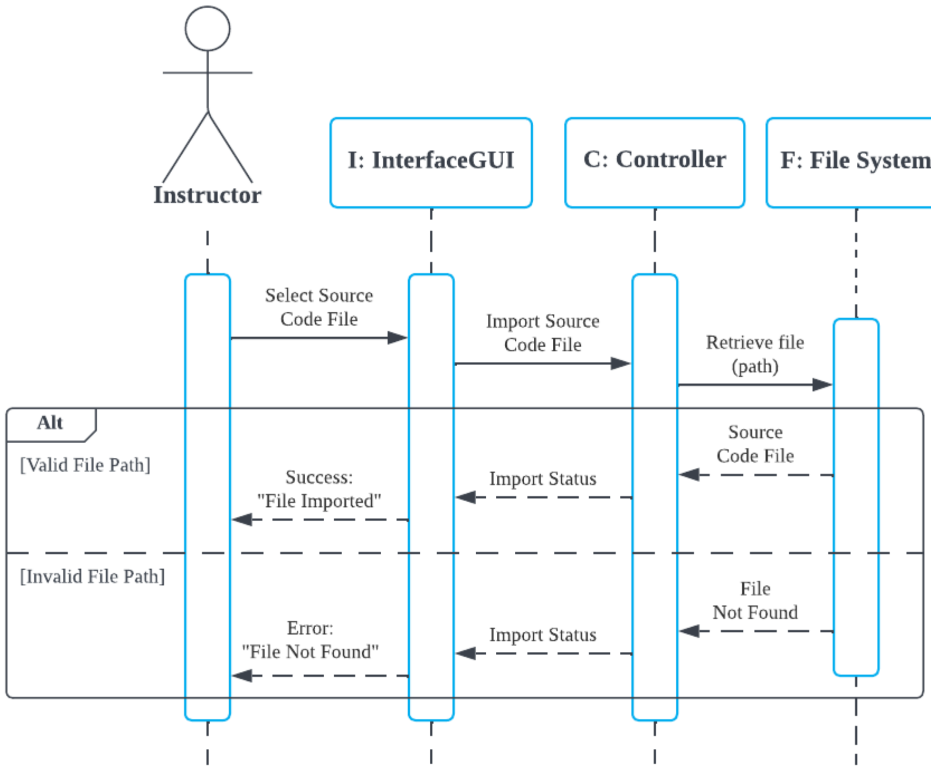

Figure 9: Sequence diagram detailing use case 1

5.1.2 Changing System Settings

Instructors and other users shall be able to change the system settings used in question generation: question mutation amount, question type, and intended LMS. The system's controller receives and saves these settings in a Settings object (Config). The interface shall display an error message to the user if the desired settings cannot be set.

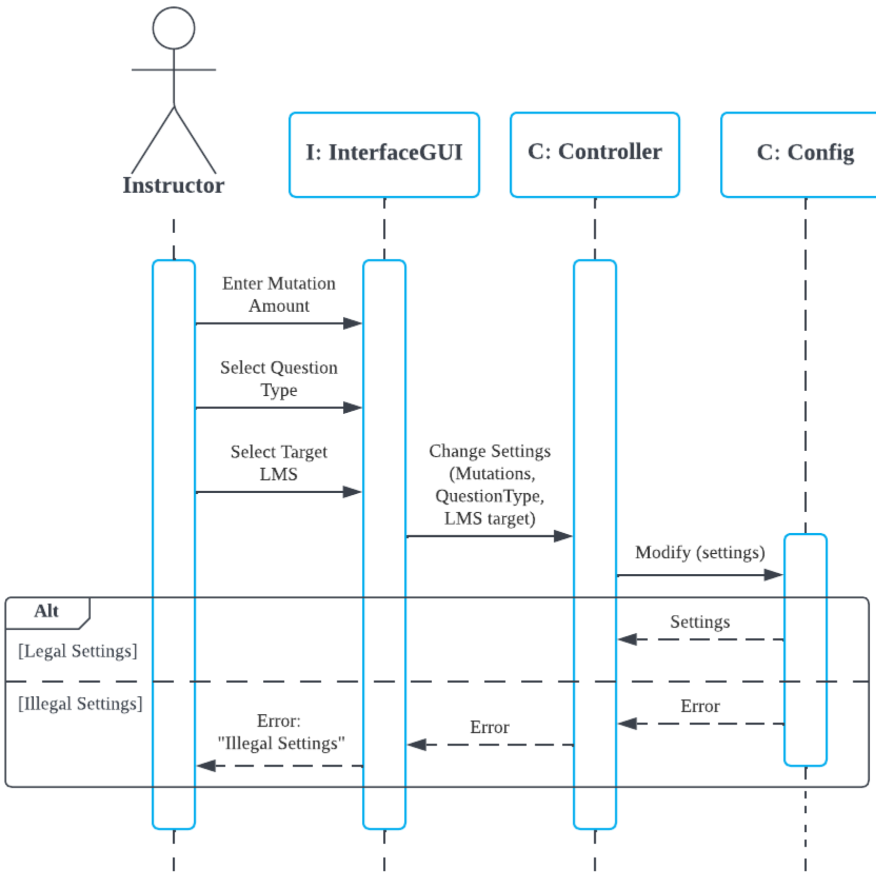

Figure 10: Sequence diagram detailing use case 2

5.1.3 Selecting Lines of Code

All users shall be able to select the lines from which questions will be generated. The system's controller shall inform the user of the current selected lines via the graphical interface. If the user selects more than 200 lines, the system shall display a warning informing the user that they may experience a delay in completion due to the size of their selection. If the user selected no lines, the system shall display an error informing the user that they must make a selection.

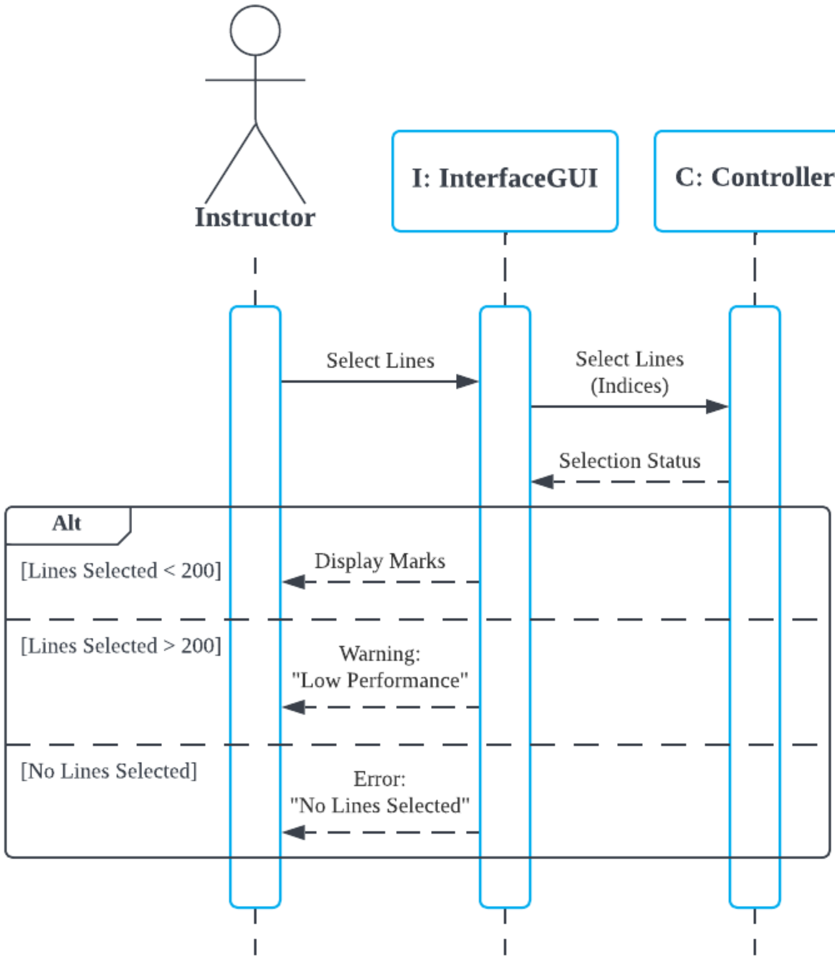

Figure 11: Sequence diagram detailing use case 3

5.1.4 Generate Questions

The system shall generate questions from the settings and lines selected by the user. The user shall request questions be generated by clicking on the Generate button. The controller shall check whether the question type is compatible with the lines selected, and display an error if it is not. Then the controller shall provide the question generator with the user's settings and selected lines, with which the generator shall create a bank of questions. Then the generator shall then verify that each question has a solution. The question generator shall finally return the questions generated to the controller. If no errors were encountered, the system shall display a message indicating that the questions were successfully created, and send the questions to the question bank. Otherwise the system shall display a message indicating that an error occurred.

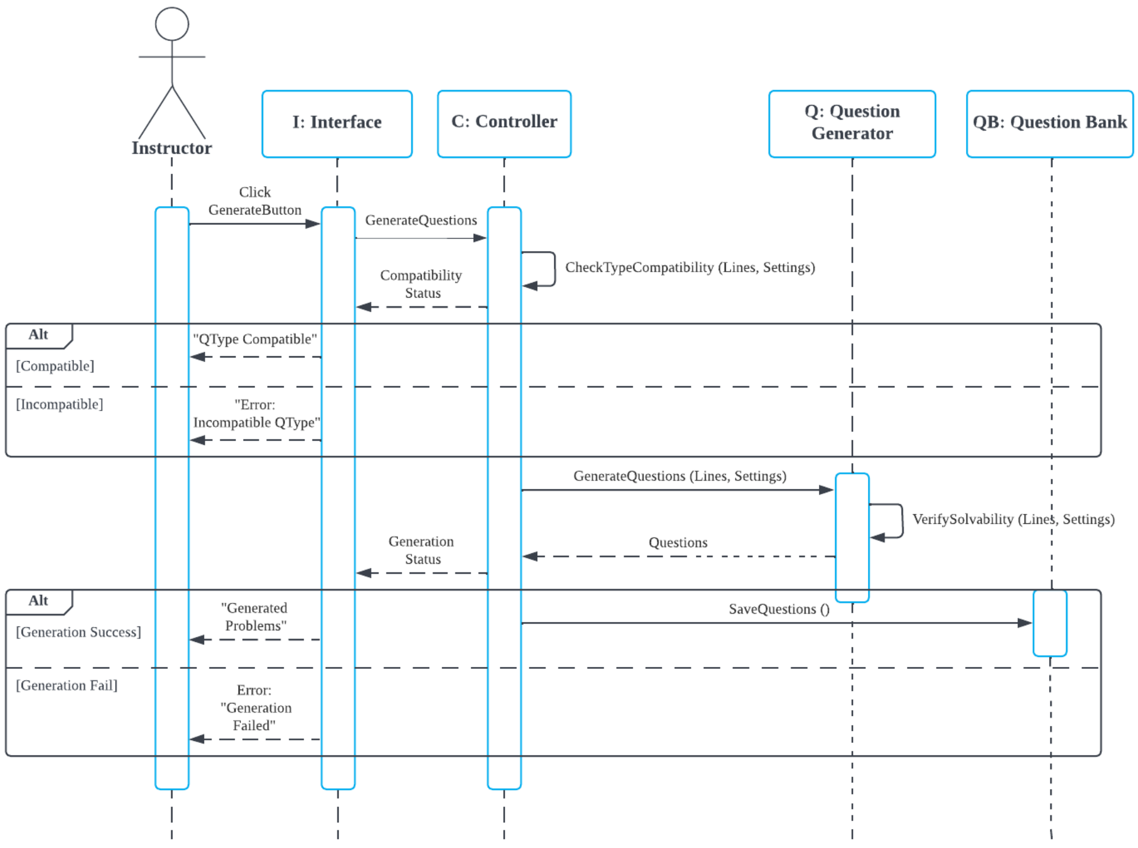

Figure 12: Sequence diagram detailing use case 4

5.1.5 Export Questions to File

The system shall export a question bank as a .qti file.. The user shall request questions be exported by clicking the Export button.. The controller shall call the question bank to create a .qti file from all questions previously generated.The question bank shall then save the file to the file system, and return a status to the controller. If the file was successfully created and saved, the system shall inform the user that the export was successful. Otherwise it will inform the user that an error occurred.

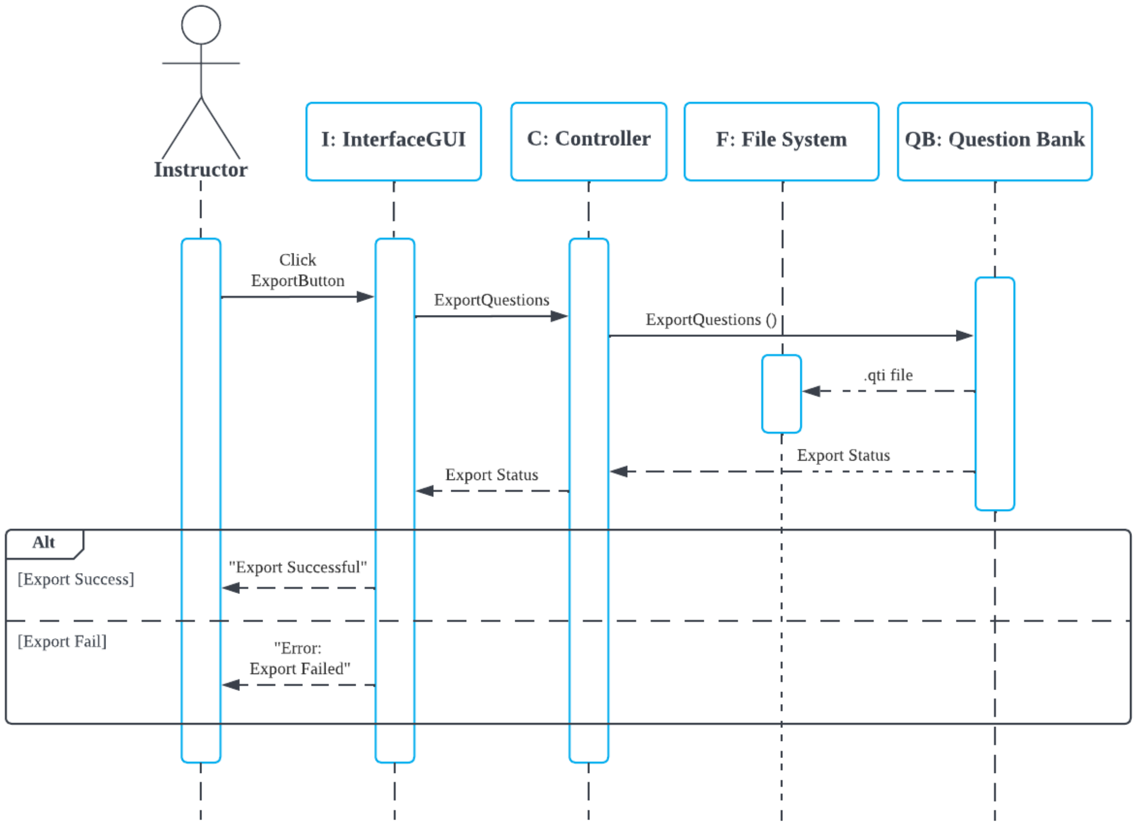

Figure 13: Sequence diagram detailing use case 5

6 Non-functional Requirements

6.1 Performance Requirements

PPALMS allows source code (as a text file) that contains correct, compilable programming language to be imported and used to generate problems. The number of mutations shall be set to 5 by default and be configurable by the user within a range from 1 to 25. The system shall strictly limit this number to maintain higher performance. If the user selects a large number of lines for problem generation, a warning shall be displayed indicating that system performance will be impacted and that the system will take more time to complete question generation.

6.2 Safety Requirements

The source code shall not be affected by the system; the system shall not add-to, delete, or modify imported source code in any way. The system shall not have capability to execute, correct, or compile the source code. Users shall only be able to generate problems from imported source code to be exported to an LMS – the system contains no other functionality. The source code will not be verified for compiling or runtime errors and will be assumed to be executable and functional to its full extent. Users should compile their code and make sure it is free from errors before importing to PPALMS.

6.3 Security Requirements

PPALMS will always be packaged along with a list of hashes for each file in the product. The user has the ability to verify the authenticity of the program by hashing the files they received and comparing the hashes to the hash list. The system should always be concerned with potential vulnerabilities like stack overflow. The system shall contain ASLR (Address Space Layout Randomization) to avoid vulnerabilities.

6.4 Software Quality Attributes

The PPALMS system shall expand to support source code containing other programming languages which are not currently supported. The system shall implement an I/O stream to import the source code and let the user select lines by GUI to generate the corresponding problems. Generated problems are tested against the source code for equivalency – to prevent problems from having no correct or incorrect answers.

6.5 Business Rules

The permissions of the source code should be set to read-only, the system will not modify or execute the source code. The source code will not be verified for compiling or runtime errors and will be assumed to be executable and functional to its full extent. Users should compile their code and make sure it is free from errors before importing to PPALMS.

7 Requirements Traceability Matrix

	1 - Importing Source Code	2 - Warning on Line Selection	3 - Original Source Code File Will Not Be Modified by the System	4 - Selecting Lines from Source Code	5 - Selecting the Intended LMS	6 - Problem Generation	7 - Reordering Problems	8 - 2D Problems	9 - Multiple-Choice Problems	10 - Bug and Error Problems	11 - Fill-in-the-blank Problems	12 - Maximum Number of Generated Questions	13 - Generated Problems Stored in System	14 - Generated Problems Output to a File	15 - Incorrect Answers do not Function the Same as Correct Answers	16 - Starting Positions for Lines in Reordering Problems	17 - System Hash Included in Releases	18 - Usable Languages for 2D Problems
Interface	■	■		■	■	■						■		■				
Controller			■			■							■	■				
Settings					■							■						
IQuestion Generator						■	■	■	■	■	■	■			■	■		■
2D Question Generator						■		■							■			■
MC Question Generator						■			■						■			
FITB Question Generator						■					■				■			
Bug Question Generator						■				■					■			
Reordering Question Generator						■	■								■	■		
Question Bank													■	■				
Question						■	■	■	■	■	■	■	■	■	■	■		■
Performance		■	■			■								■			■	
Scenario 5.1.1	■		■															
Scenario 5.1.2					■							■						
Scenario 5.1.3		■	■	■														
Scenario 5.1.4															■			
Scenario 5.1.5													■	■				

Figure 14: Traceability Matrix Diagram

Appendix: Glossary

LMS: Abbreviation for Learning Management System; a software system that assists educators, for use by both instructors, students, and institutions.

Mutation: Code that has been modified by the system for generated problems.

Parson's Problems: Computer programming problems based on modifying, reordering, or selecting lines of code.

PPALMS: Abbreviation for the name of the system (Parson's Problems Appliance for Learning Management Systems).

Regex: (1) Regular expression. (2) The regular expression libraries used to check expressions for equivalency and correctness.

.qti: (Question and Test Interoperability) A file type containing generated problems, able to be imported into an LMS.

Variation: (see Mutation).