Final Project Report: Spooky Quiz Game

COMP 3663

March 20th 2024

Authors:
Noah Baker
Jakob Legere
Riley Meyers

Table of Contents

Introduction

Project Overview

Purpose

Create a fun Spooky Quiz game that demonstrates our understanding of design patterns and how to implement them. The ‘Spooky’ part comes from the chance that a random effect will be applied to the question scenes without the user knowing. Our main goal with the design patterns we implemented was to make it easy to add more features, question categories, and random effects.

Functionalities

Leaderboard
3 Categories of Questions
Dynamic Scoring System
Random Effects applied to quiz questions

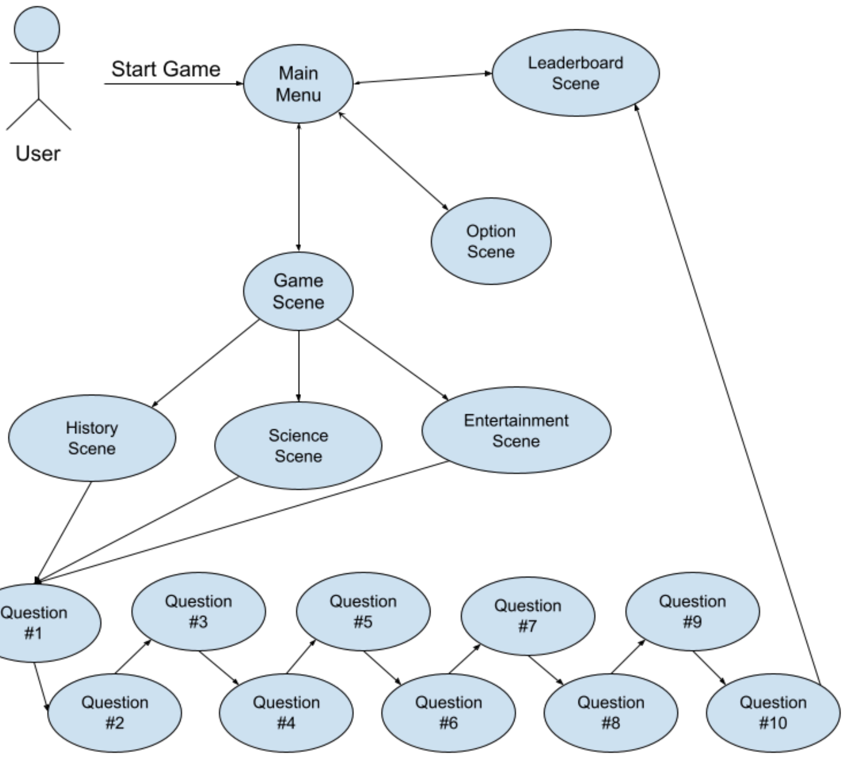
System Description

The user starts the game on the main menu, where they can choose from the Leaderboard, the options, or the category selection screen. The only way to progress forward is to choose the category selection, then the user can choose between History, Science, or Entertainment.

Once the user chooses a category and starts the game, they will see 10 multiple choice trivia questions in a row. For each question, the user has to choose one of the options and submit their answer. If they don’t choose an answer, they will not be allowed to go to the next question.

After answering all 10 questions, the user will be told their score and how long they took to finish the quiz. Then the user will be asked to enter a name for their score on the leaderboard. If the user's score is in the top 10 scores, then their name and score will be added to the leaderboard display, and the user can go back to the main menu and start a new game or exit the application.

The user flow is represented in the diagram below:



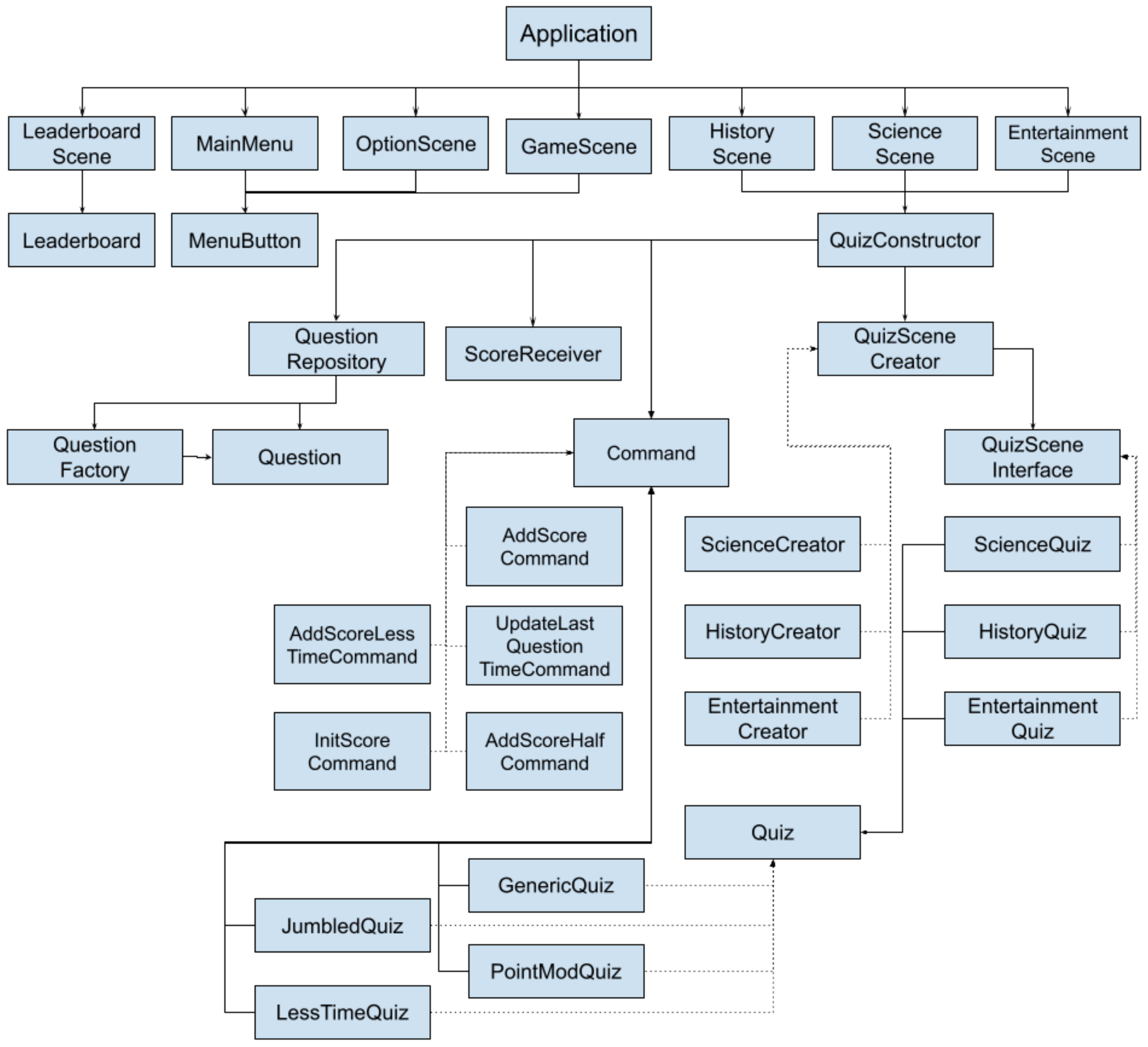
Architecture Overview

The architecture of our application is represented in the simplified diagram below. Each rectangle represents one of the classes in our game, and the arrows represent dependency (e.g. the LeaderboardScene class is dependent on the Leaderboard class).

The Application class creates the main scenes required for the UI to function and sets them up when the application is started. The question scenes aren't created until the user actually chooses a category and starts the games. Then the QuizConstructor class is responsible for pulling the appropriate questions from the Question Repository, creates the 10 question scenes (using the QuizFactory) and adds them to the parent panel.

When constructing the 10 questions for the quiz, the concrete products for each category generate a random number and use that number to choose which strategy to implement in each question. The different strategies represent the 'Spooky' aspect of our game, by modifying the user experience of the quiz.

Below is a simplified version of our class diagram, meant to show general dependencies in our program. **For our full UML class diagrams, see the Appendix.**



Technical Specifications

Java Swing UI Library:

Java Swing UI Library was used for the creation of all UI components on the project. Swing handles the window/frame creation and all aspects related to the UI itself. It has tools that were used like buttons, ‘scene’ switching and styling. Swing is a key component as it allows the user to have a visual component to interact with our system. The system is built on the basis of the component “JPanel” and our system uses this to add to the Parent JPanel to create a dynamic changing view.

Leaderboard Implementation:
The leaderboard is a static class that is initialized when the Application is started. System.getProperty is used to see if the user is on windows or mac, then we use System.get to get the file path and use that to create the scores.txt file (if it doesn't already exist). The scores file is created with 10 lines that say 'Empty 0', which is the same format we save new names into the scores file.

When the user finishes a round of the quiz game, they will be prompted to enter their name for the leaderboard. If their score is in the top 10 it gets added to the leaderboard. This is done by doing an insertion sort on the existing list, and chopping off the 11th entry. After the score.txt file is updated by calling addNewScore, the leaderboard scene is updated.

The leaderboard scene is displayed to the user by parsing the previously created scores.txt document and displaying it in a Swing text component. After the user completes the game, they are prompted to input a name for the leaderboard. This will then be stored and sent to the leaderboard along with their score. The main limitation of the leaderboard is that the scores are locally stored in a text file, rather than a database with other users. This was to keep simplicity in a part of the system that we felt did not need to be complicated. Due to this decision, your leaderboard is initialized with no entries, however scores will persist between play sessions.

Game Mechanics

Quiz Specifics:
There are 3 categories of quizzes within our application, those being Science, History, and Entertainment. Each category has 10 questions each, with 4 multiple choice options and 1 correct option among them. Quiz questions are displayed one at a time and cannot be passed until an answer has been provided by the user. If the correct option has been supplied by the user, then a certain amount of points will be awarded to the user and the next question will be subsequently loaded. If the correct option is not chosen, then the user will not be awarded any points. After 10 questions, the user will be told their final score and the quiz will end.

Spooky Modes:
Certain questions have become haunted, and will take on abnormal traits. The types of question variants that can be found in the game are as follows:

- Generic - Nothing spooky about this one, this is the base quiz that offers no modifiers.
- Jumbled - The spooky modifier reverses all text answer options in the question. There is no point manipulation for this modifier.
- LessTime - The hidden time counter is halved for this question, only giving users 50 seconds to achieve points instead of the typical 100.
- PointMod - All points awarded on this question variant are halved before they are added to the running total.

Most of the question variants are hidden from the user, minus the jumbled variant, so users may not know why they're getting the point results they are until the quiz has ended, or if they're aware of the haunted quiz they're participating in.

Leaderboard:
The leaderboard stores the top 10 names and scores. The leaderboard is updated when the user completes a game, and the user is sent to the Leaderboard after the 10th question.

Scoring specifics:
The score for each question is based on how long the user takes to answer the question. The theoretical max score for a single question is 100 points, subtracted by 10 for each second the user takes to answer the question. There is a minimum score of 1 if the user takes longer than 10 seconds but still gets the question right. If the user gets the question wrong, they get no points for that question.

Scoring is modified by some of our different quiz strategies that implement the 'Spooky' part of our quiz game. This is supported by our Command pattern, as we implement a different command for each type of scoring. AddScoreHalfCommand will divide the points for that question by 2, and AddScoreLessTimeCommand will give the users only 5 seconds to get more than 1 point for the question.

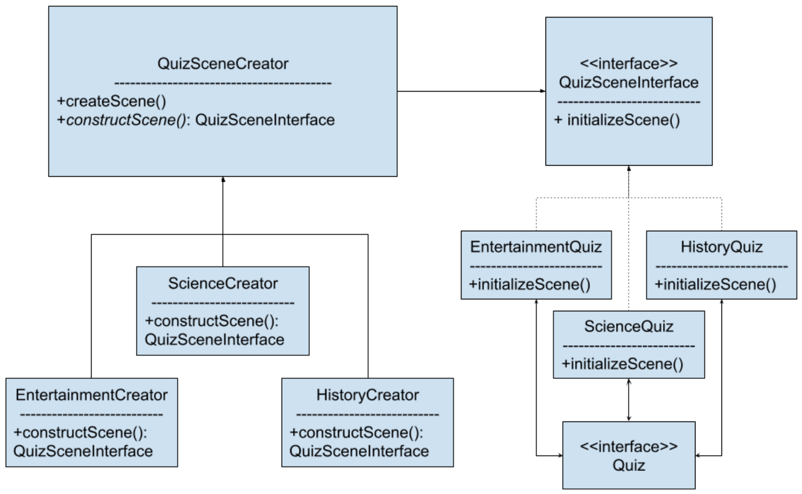
Design Patterns Used

Several different design patterns were considered for our project, but we ultimately decided on using the 4 following patterns to shape our project into what it is now. The following will be discussed in detail within the following section.

List of Design Patterns:

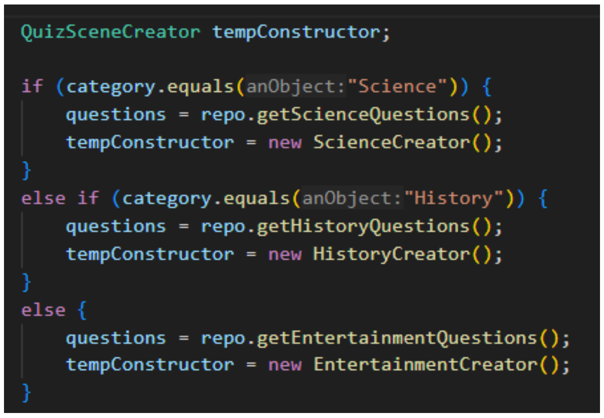
- Factory Pattern
- Command Pattern
- Singleton Pattern
- Strategy Pattern

Factory Pattern



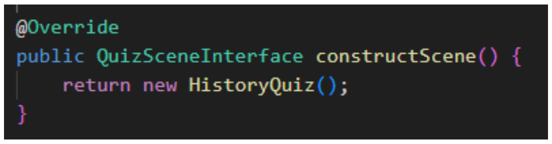
Purpose and Use:
The factory method was especially fitting for our application because of our choice of UI framework for java: Javax Swing. Swing involves creating scenes (aka Panels) and adding them to the parent panel. We need to create the scenes in a different way depending on the category selected, but we may also want to add more categories in the future. So the factory method became the obvious choice for creating the question scenes. The method createScene() calls the appropriate constructScene(), which returns a concrete product object. Then we call initializeScene() on the concrete product to get the JPanel for the question.

Implementation Details:
Interacting with the Factory
The QuizConstructor.java class is responsible for creating the 10 question scenes using our QuizSceneFactory package. QuizConstructor uses the QuizSceneCreator interface as a single access point for creating the different categories of quizzes. Then the appropriate constructor (between ScienceCreator.java, EntertainmentCreator.java, and HistoryCreator.java) is initialized using the interface.


```java
QuizSceneCreator tempConstructor;

if (category.equals(anObject:"Science")) {
    questions = repo.getScienceQuestions();
    tempConstructor = new ScienceCreator();
}
else if (category.equals(anObject:"History")) {
    questions = repo.getHistoryQuestions();
    tempConstructor = new HistoryCreator();
}
else {
    questions = repo.getEntertainmentQuestions();
    tempConstructor = new EntertainmentCreator();
}
```

Concrete Constructors
We have 3 concrete constructors (one for each category), and each one is fairly simple and similar. The constructors implement QuizSceneInterface


```java
@Override
public QuizSceneInterface constructScene() {
    return new HistoryQuiz();
}
```

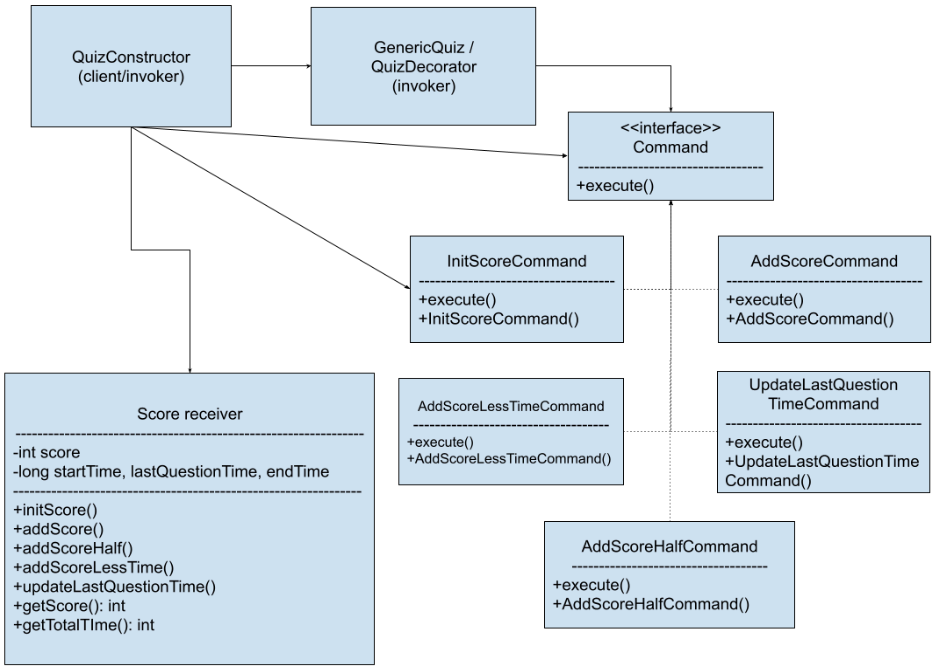
and they create the appropriate product object and return it to QuizConstructor.java.

Concrete Products
The concrete product generates a random number to decide which strategy to implement in each individual quiz scene, then uses the Quiz interface to actually construct the Question scenes by calling the quizConstructor method with all of the required information. The concrete products implement QuizSceneInterface.java to provide a single access point to QuizConstructor.java.

Factory Pattern Classes
- Products
    - QuizSceneInterface.java
        - ScienceQuiz.java
        - EntertainmentQuiz.java
        - HistoryQuiz.java
- Creators
    - QuizSceneCreator.java
        - ScienceCreator.java
        - EntertainmentCreator.java
        - HistoryCreator.java

Benefits:
The quiz creation code makes it easy to reuse the User interface code, since Swing requires a lot of code to create and format the elements on the UI. This follows the single responsibility, since we could move the object creation code into one place. Implementing the factory pattern also follows the open / closed principle. To add a new category for the quiz, you only need to add a ConcreteCreator and ConcreteProduct (very simple classes), then add the questions to the question code and QuizConstructor. An arbitrary number of categories could be implemented without making the codebase more complex, since the creation of quiz scenes is centralized by the factory pattern.

Command Pattern

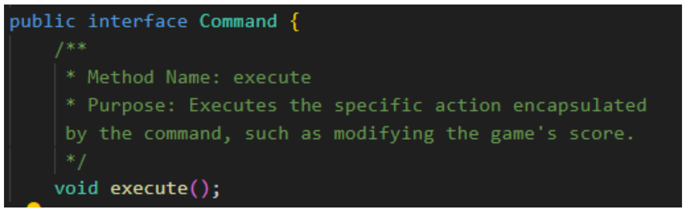


Purpose and Use:

The command pattern was selected to make it easier to add additional ‘Spooky’ mechanics to the game, such as our AddScoreHalfCommand. We are also able to change which command is being called during runtime, which keeps the application flexible and in keeping with the spooky theme. Implementing the command pattern enforced a specific way of increasing the score, which is even more important when we are changing the scoring method for each question.

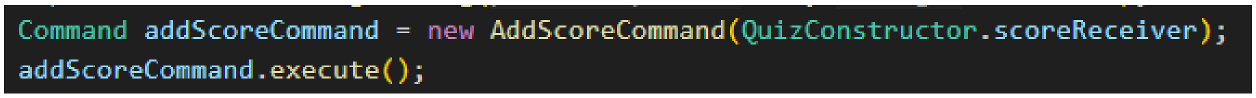
Implementation Details:

By implementing all of the commands through the command interface, we were able to provide a single point of access for all of the quiz scenes to interact with our receiver Score Receiver.java. Each of the concrete commands implement our command interface, which serves as a single point of access for the rest of our application to work with command objects.


```
public interface Command {
    /**
     * Method Name: execute
     * Purpose: Executes the specific action encapsulated
     by the command, such as modifying the game's score.
     */
    void execute();
```

The commands call specific functions in our receiver class, Score receiver.java. Each command has its own associated method in the receiver. Currently our score variable is set to static, since only one game can be running at a time. The receiver uses system time to calculate how long the player has taken for each question and the overall quiz.

We have numerous invokers because we need QuizConstructor to invoke the score initialization for each new quiz, but we also need specific questions to implement different commands due to the spooky theme of the game. So each of the different quiz strategies invoke a different set of commands to update the score.



```
Command addScoreCommand = new AddScoreCommand(QuizConstructor.scoreReceiver);
addScoreCommand.execute();
```

The appropriate command for that strategy (this is the generic strategy) is associated with the score receiver object created and initialized with a score of zero before the quiz.

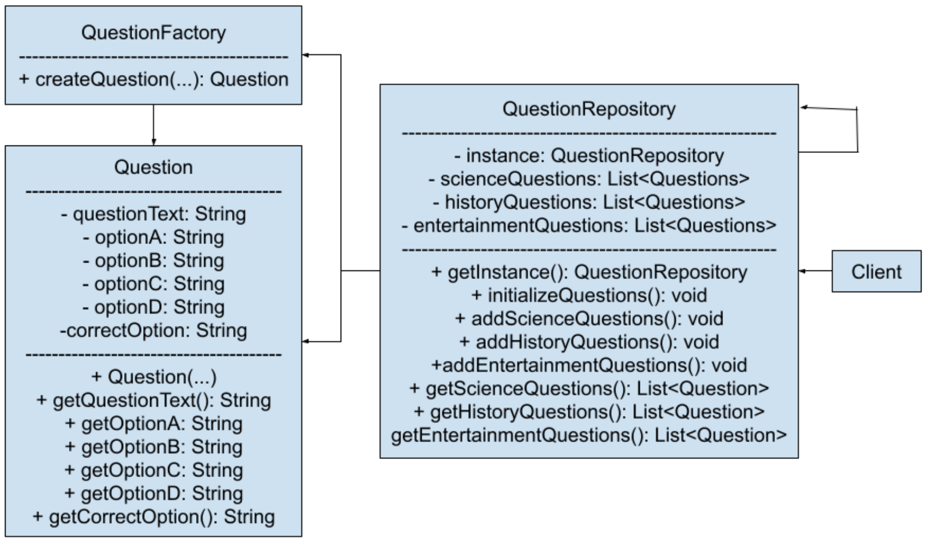
Specific Classes:

- Retriever
    - ScoreReceiver.java
- Command Interface
    - Command.java
        - AddScoreCommand.java
        - AddScoreHalfCommand.java
        - AddScoreLessTimeCommand.java
        - InitScoreCommand.java
        - UpdateLastQuestionTimeCommand.java
- Invokers
    - GenericQuiz.java
    - JumbledQuiz.java
    - LessTimeQuiz.java
    - PointModQuiz.java
    - QuizConstructor.java

Benefits:

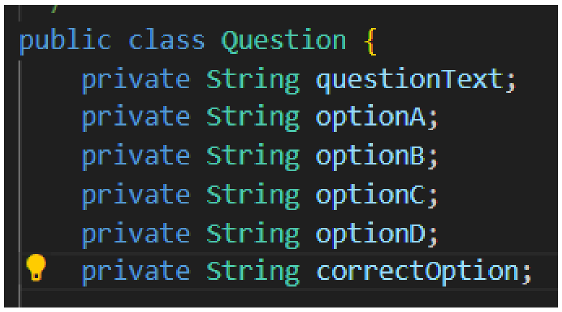
The command pattern makes it easier to scale up our code and support numerous additional features and functionalities for the spooky quizzes. It enforces a specific way of interacting with the score receiver which makes it easier to maintain the existing code base as it is added upon. By implementing the Command pattern, we also separate the request sender from the request executor, which makes it simpler to change how commands are processed in the future. Implementing the command pattern supported our ultimate goal of making it easier to extend and build on top of our quiz game application.

Singleton Pattern


QuestionFactory
+ createQuestion(...): Question
Question
- questionText: String
- optionA: String
- optionB: String
- optionC: String
- optionD: String
-correctOption: String
+ Question(...)
+ getQuestionText(): String
+ getOptionA: String
+ getOptionB: String
+ getOptionC: String
+ getOptionD: String
+ getCorrectOption(): String
QuestionRepository
- instance: QuestionRepository
- scienceQuestions: List<Questions>
- historyQuestions: List<Questions>
- entertainmentQuestions: List<Questions>
+ getInstance(): QuestionRepository
+ initializeQuestions(): void
+ addScienceQuestions(): void
+ addHistoryQuestions(): void
+addEntertainmentQuestions(): void
+ getScienceQuestions(): List<Question>
+ getHistoryQuestions(): List<Question>
getEntertainmentQuestions(): List<Question>
Client

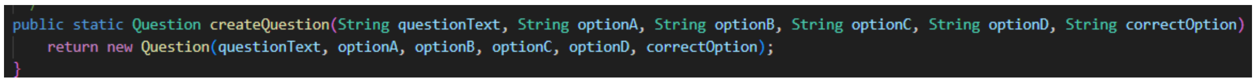
Purpose and Use:
The Singleton pattern was used in order to create each question and hold each question. Singleton was used instead rather than other creational design patterns due to the necessity and importance of ensuring that a question is only created once, by a universal creator. The singleton design pattern ensures that we have one instance of each question that we can use in our quiz. The singleton creational pattern is the core to our project.

Implementation Details:
Every Question is defined as a Question class, which is designed to represent a single question. The question class defines 6 string variables, questionText, optionA, optionB, optionC, optionD, and correctOption. The class also contains a constructor and 6 getter methods, one for each variable.



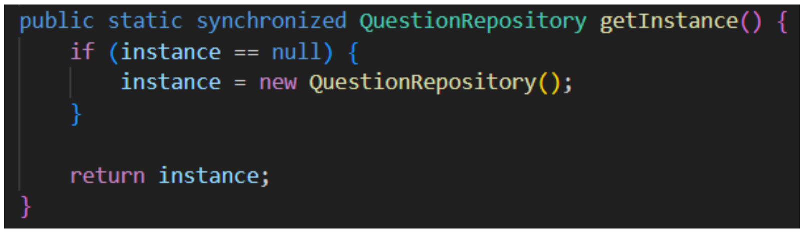
```
public class Question {
    private String questionText;
    private String optionA;
    private String optionB;
    private String optionC;
    private String optionD;
    private String correctOption;
```

The Questions are created by the QuestionFactory class, which contains one method: createQuestion, which accepts an entry for all the variables needed for the Question class and organizes them into a new question.



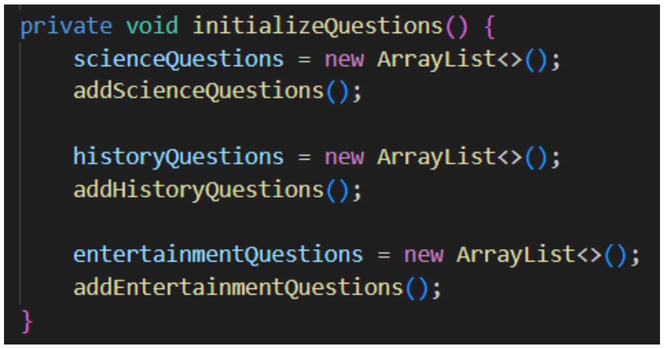
```
public static Question createQuestion(String questionText, String optionA, String optionB, String optionC, String optionD, String correctOption)
    return new Question(questionText, optionA, optionB, optionC, optionD, correctOption);
}
```

Finally, QuestionRepository.java is where the magic happens. If there has not been an instance of the QuestionRepository created yet, getInstance() will create a new Question Repository and return it.



```java
public static synchronized QuestionRepository getInstance() {
    if (instance == null) {
        instance = new QuestionRepository();
    }

    return instance;
}
```

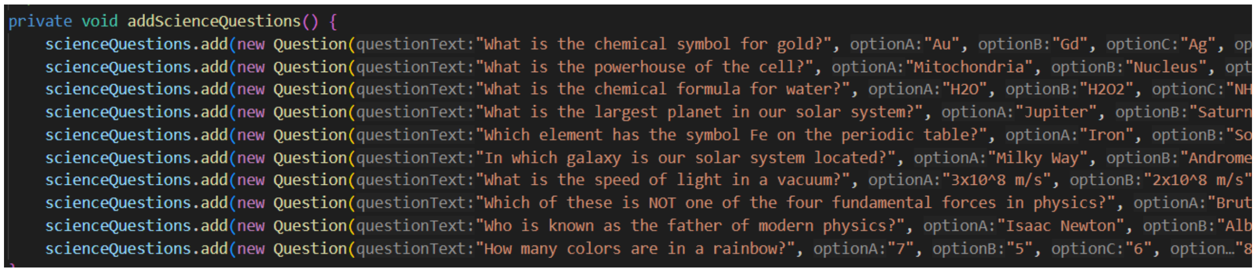


```java
private void initializeQuestions() {
    scienceQuestions = new ArrayList<>();
    addScienceQuestions();

    historyQuestions = new ArrayList<>();
    addHistoryQuestions();

    entertainmentQuestions = new ArrayList<>();
    addEntertainmentQuestions();
}
```

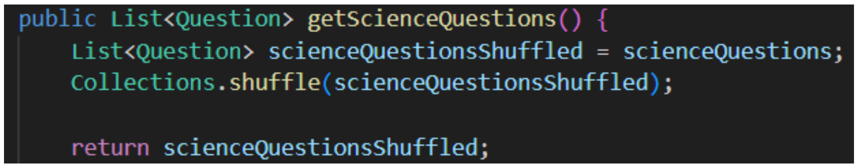
Creating a new instance of QuestionRepository will call initializeQuestions(), which generates 3 lists, one for Science, one for History, and one for Entertainment. Each list is then filled with pre-generated questions using the add__Questions() method, with “__” being the respective quiz category.



```java
private void addScienceQuestions() {
    scienceQuestions.add(new Question(questionText:"What is the chemical symbol for gold?", optionA:"Au", optionB:"Gd", optionC:"Ag", op
    scienceQuestions.add(new Question(questionText:"What is the powerhouse of the cell?", optionA:"Mitochondria", optionB:"Nucleus", opt
    scienceQuestions.add(new Question(questionText:"What is the chemical formula for water?", optionA:"H2O", optionB:"H2O2", optionC:"NH
    scienceQuestions.add(new Question(questionText:"What is the largest planet in our solar system?", optionA:"Jupiter", optionB:"Saturn
    scienceQuestions.add(new Question(questionText:"Which element has the symbol Fe on the periodic table?", optionA:"Iron", optionB:"So
    scienceQuestions.add(new Question(questionText:"In which galaxy is our solar system located?", optionA:"Milky Way", optionB:"Androme
    scienceQuestions.add(new Question(questionText:"What is the speed of light in a vacuum?", optionA:"3x10^8 m/s", optionB:"2x10^8 m/s"
    scienceQuestions.add(new Question(questionText:"Which of these is NOT one of the four fundamental forces in physics?", optionA:"Brut
    scienceQuestions.add(new Question(questionText:"Who is known as the father of modern physics?", optionA:"Isaac Newton", optionB:"Alb
    scienceQuestions.add(new Question(questionText:"How many colors are in a rainbow?", optionA:"7", optionB:"5", optionC:"6", option…"8
```

Each Category also contains a getter method that shuffles the quiz questions from their standard order before delivering the question list to the client.

Specific Classes:
- Question.java
- QuestionFactory.java
- QuestionRepository.java

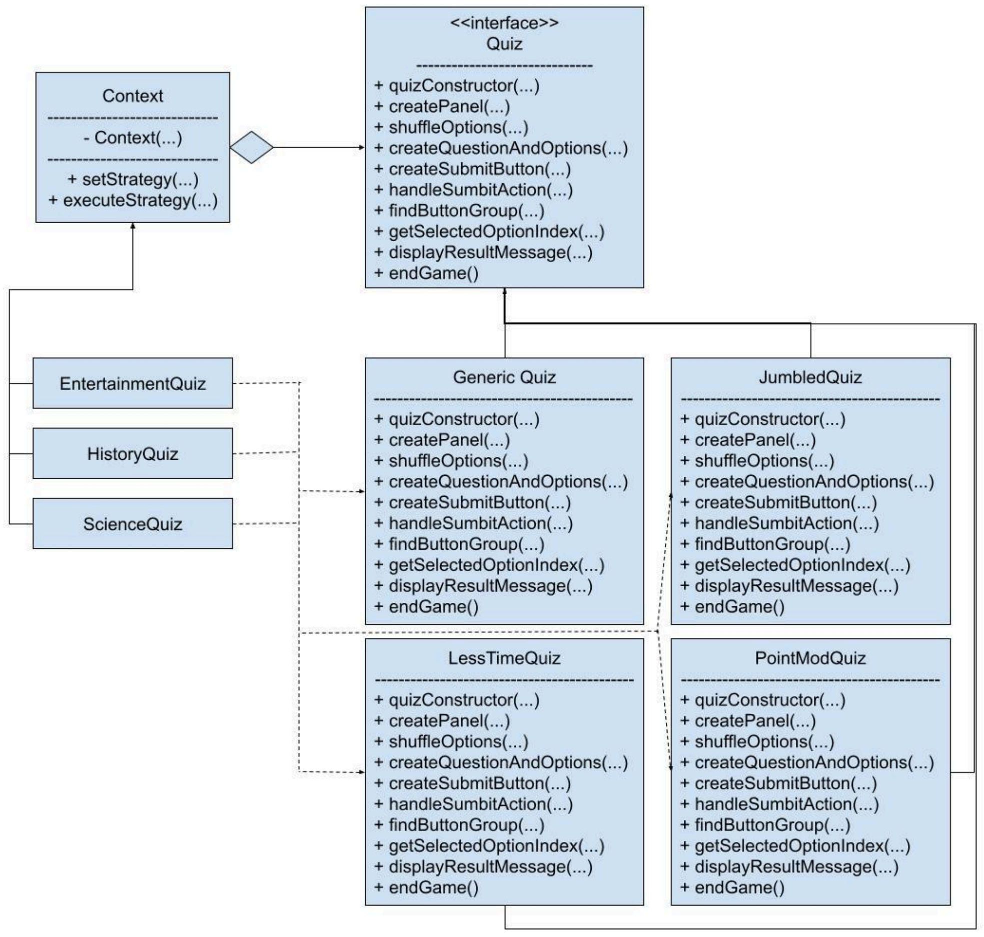


```java
public List<Question> getScienceQuestions() {
    List<Question> scienceQuestionsShuffled = scienceQuestions;
    Collections.shuffle(scienceQuestionsShuffled);

    return scienceQuestionsShuffled;
```

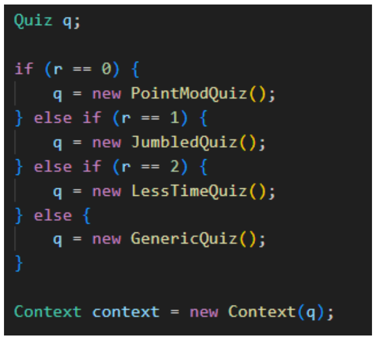
Benefits: The benefits of using singleton for the system enables us to only have one instance of a (question) at a time. It creates all questions in a neat and orderly fashion upon being called for the first time, and stores them universally under one location. Making sure that every quiz calls from the same quiz questions ensures that we don’t have multiple instances of identical question sets floating around and taking up space while the application is running. The scalability of this design pattern in this system is vital, as it allows for us to create even more questions and ensure that they can only be created through this pattern. Another benefit of this pattern is that we know exactly where the questions are coming from, and how they are created. This ensures that in our use case it is maintainable and we understand where question related bugs are originating. We are aware however that the use of Singleton violates the single responsibility principles, but for our use case it makes the most sense.

Strategy Pattern -



Purpose and Use:
The strategy design pattern was chosen after it was determined that the wrapper and template design patterns were not the best option for the system. The strategy design pattern is perfect for enabling the 'spooky' aspect of the system. It allowed us to change the behavior dynamically at runtime. We chose Strategy over Template for that reason, template works at the class level meaning it's static. Template is extending an algorithm whereas Strategy allows us to alter an object (GenericQuiz.java) behavior and supply it with a different behavior (JumbledQuiz.java).

Implementation Details: Utilizing the Strategy pattern in our project has allowed us to easily change the behavior of a quiz question at runtime by having a randomizer choose between several "Quiz" objects. Entertainment.java, Science.java, and History.java each contain a randomizer within their initializeScene() method that decides which variant of the Quiz this particular question will adopt. The chosen strategy is then assigned to the context object for that question. The context class is only aware of the quiz interface, and will execute the chosen strategy to create the quiz scene with all the required resources.

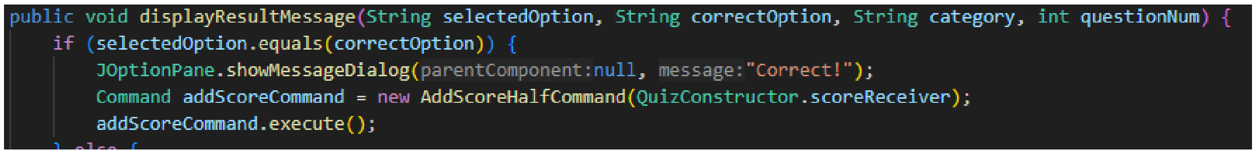

```java
Quiz q;

if (r == 0) {
    q = new PointModQuiz();
} else if (r == 1) {
    q = new JumbledQuiz();
} else if (r == 2) {
    q = new LessTimeQuiz();
} else {
    q = new GenericQuiz();
}

Context context = new Context(q);
```

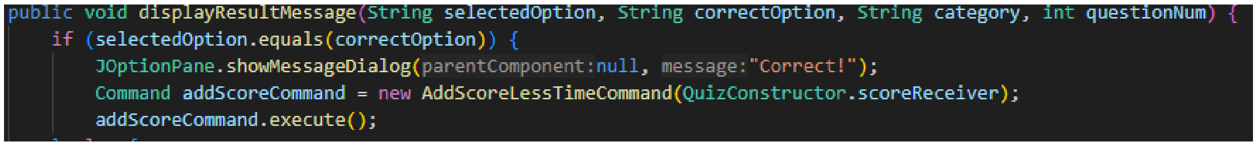
All of these quizzes have the same methods, but there are slight variations in some methods. There are 4 variants within our finished project:

- GenericQuiz has no special “haunted” traits and is the most common type of quiz question
- PointModQuiz modifies the points awarded by answering the question correctly by spitting the number in half before adding it to the total points. This change comes in the displayResultMessage() method, in which AddScoreHalfCommand() is called instead of AddScoreCommand().



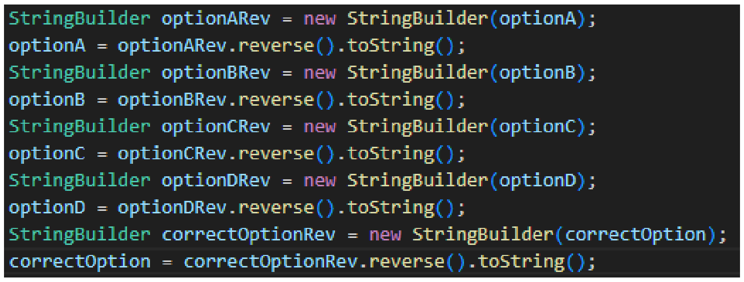
```java
public void displayResultMessage(String selectedOption, String correctOption, String category, int questionNum) {
    if (selectedOption.equals(correctOption)) {
        JOptionPane.showMessageDialog(parentComponent:null, message:"Correct!");
        Command addScoreCommand = new AddScoreHalfCommand(QuizConstructor.scoreReceiver);
        addScoreCommand.execute();
```

- LessTimeQuiz also modifies how much time the user has access to by reducing the time limit from 100 seconds to 50 seconds, overall reducing the number of points the question will award. This change, like the PointModQuiz, alters the displayResultMessage() method, where AddScoreLessTimeCommand() is called instead of AddScoreCommand().



```java
public void displayResultMessage(String selectedOption, String correctOption, String category, int questionNum) {
    if (selectedOption.equals(correctOption)) {
        JOptionPane.showMessageDialog(parentComponent:null, message:"Correct!");
        Command addScoreCommand = new AddScoreLessTimeCommand(QuizConstructor.scoreReceiver);
        addScoreCommand.execute();
```

- JumbledQuiz takes all of the options and the correct answer received from the question information and reverses the string’s order before presenting it to the user. This in turn makes the question confusing, and makes the user have to stop and read the options backwards. This Strategy modifies the quizConstructor() method, adding several lines of code to the start of the method that utilize the StringBuilder class to easily reverse the string’s orders.



```java
StringBuilder optionARev = new StringBuilder(optionA);
optionA = optionARev.reverse().toString();
StringBuilder optionBRev = new StringBuilder(optionB);
optionB = optionBRev.reverse().toString();
StringBuilder optionCRev = new StringBuilder(optionC);
optionC = optionCRev.reverse().toString();
StringBuilder optionDRev = new StringBuilder(optionD);
optionD = optionDRev.reverse().toString();
StringBuilder correctOptionRev = new StringBuilder(correctOption);
correctOption = correctOptionRev.reverse().toString();
```

Specific Classes:

- Context.java
- Quiz.java
    - GenericQuiz.java
    - JumbledQuiz.java
    - LessTimeQuiz.java
    - PointModQuiz.java

The benefits of using the strategy design pattern far outweigh the potential negatives. The Strategies can be swapped and reused across different game scenarios. Strategy can also not be used during the runtime and that still causes no issues on the system. Best of both worlds. Using the strategy for this system allows us to add or remove haunted aspects with ease. The strategy pattern also allows this system to scale comfortably, as they are not always ran, and when they are called to run it is only at runtime and only for the aspects we want. This promotes faster runtimes and not bloating resource use.

Using strategy allows us to pinpoint bugs and problems with specific 'haunted' aspects of the quiz, like we were able to do during development. For example; we knew there was a problem with the "Jumbled" haunted aspect, as it caused the scene to look and behave differently from the generic quiz. This led to easy debugging and problem solving; which will continue throughout future maintenance. The strategy also helped us follow the important principle that classes should be open to extension but closed to modification. It does this by ensuring that the genericQuiz class is extended, but not directly modified by another aspect.

Conclusion

Summary of Findings

While it was difficult to implement the design patterns initially, now we can see that it will be easy to add more features and maintain the application. By implementing the factory pattern, we made it easier to scale our application in the future, adding more categories easily and at a low cost. The command pattern makes it easier to maintain and reuse our code in the future, but may cause some limitations to the types of 'Spooky' features we could implement in the future. This limitation is part of why the strategy pattern was such a good fit for us, since it allows us to override all of the question scene logic if needed to implement a specific Spooky feature.

In theory, we could have developed this software without using any design patterns, and it would have probably worked, but would've been very difficult to improve and extend in the future, not to mention that the project would've gotten harder to manage further down the development path. Instead, the development got easier and easier as the implemented design patterns created a solid foundation of code that was easy to understand and connect to.

Future Work

- Add additional categories and spooky features
- Improve UI design
- Add more questions to each category
- Extend factory pattern to abstract factory pattern
- Add more options

Final Thoughts

Some of the design patterns we implemented were straightforward choices due to the GUI framework that we selected. The factory method was a great fit because of how JPanels need to be created and added to the parent panel. Other design patterns weren't as obvious at first though, like the command pattern for tracking the score. We initially planned to use the observer pattern, but after trying to draw a diagram that would work with our system, we realized that the observer pattern wouldn't work and we had to look through all the design patterns to find one that would work for us.

While it's important to plan ahead for projects like this, we also needed to stay flexible when we realized that our initial plans wouldn't work. Focusing on the design patterns was an interesting way to design software that we hadn't experienced before, and it forced us to take a more rigorous approach to our code. This didn't just provide the benefits of the design patterns themselves, but also improved the quality of our code across the board, since we had to ensure that the other parts of our application could easily interface with the implemented design patterns.

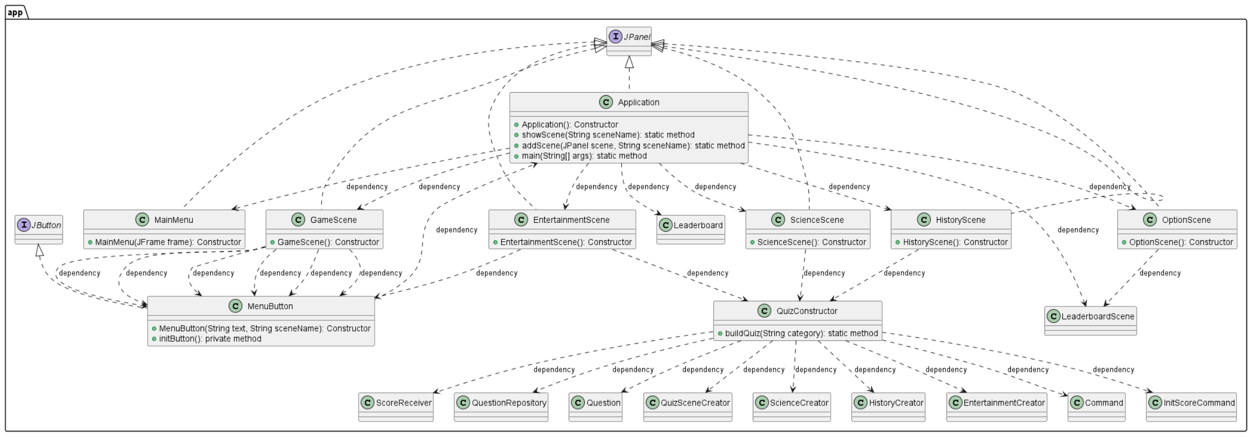
In conclusion, implementing design patterns is vital for any projects above a certain size, complexity, or development time. Designing software in terms of design patterns results in software that is easier to maintain, scale, and reuse, even though it may take more effort upfront. Implementing design patterns in your software is an investment that will save you time in the long run.

Appendices

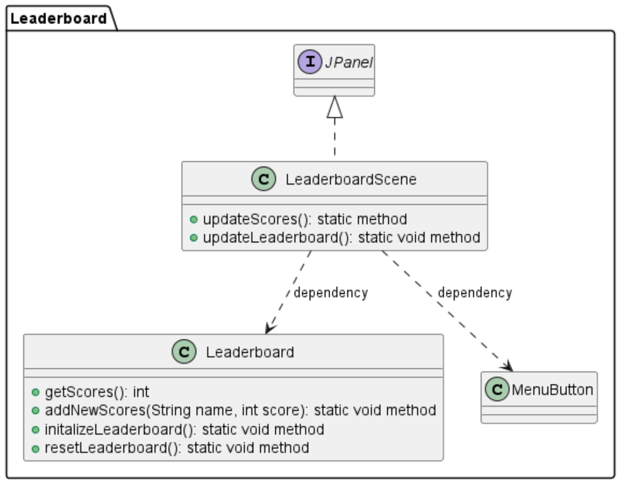
UML Diagrams

Diagrams were created with PlantUML, each diagram is a different package in our program. See file versions in the Additional Resources folder or the link here: https://drive.google.com/drive/folders/1yXDLGXcvQ2T56DERx-M0nV50-1lwHNS8?usp=sharing

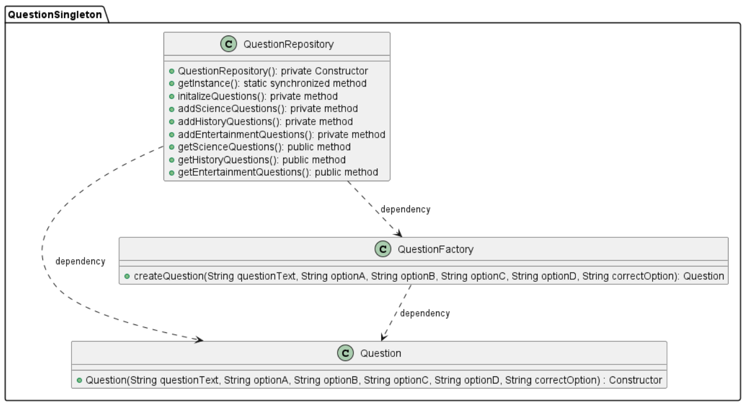
App package


app
JPanel
Application
Application(): Constructor
showScene(String sceneName): static method
addScene(JPanel scene, String sceneName): static method
main(String[] args): static method
JButton
MainMenu
MainMenu(JFrame frame): Constructor
GameScene
GameScene(): Constructor
EntertainmentScene
EntertainmentScene(): Constructor
Leaderboard
ScienceScene
ScienceScene(): Constructor
HistoryScene
HistoryScene(): Constructor
OptionScene
OptionScene(): Constructor
MenuButton
MenuButton(String text, String sceneName): Constructor
initButton(): private method
QuizConstructor
buildQuiz(String category): static method
LeaderboardScene
ScoreReceiver
QuestionRepository
Question
QuizSceneCreator
ScienceCreator
HistoryCreator
EntertainmentCreator
Command
InitScoreCommand
dependency

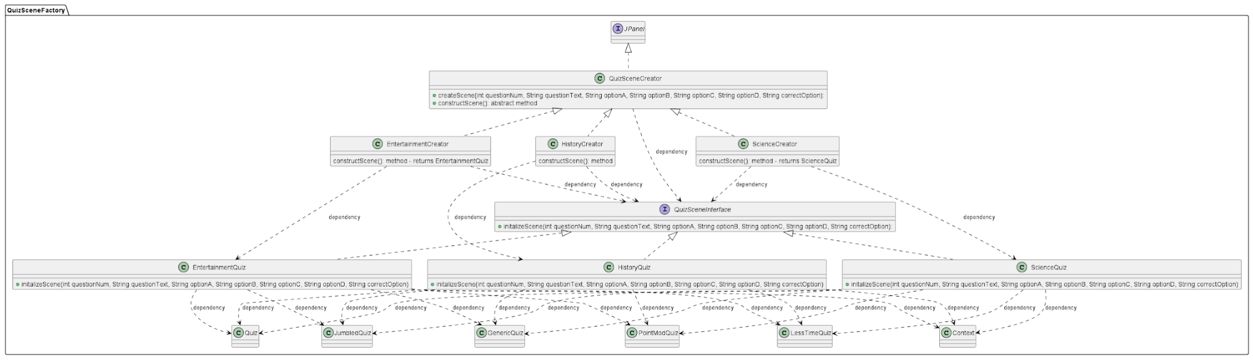
Leaderboard Package


Leaderboard
JPanel
LeaderboardScene
updateScores(): static method
updateLeaderboard(): static void method
dependency
dependency
Leaderboard
getScores(): int
addNewScores(String name, int score): static void method
initalizeLeaderboard(): static void method
resetLeaderboard(): static void method
MenuButton

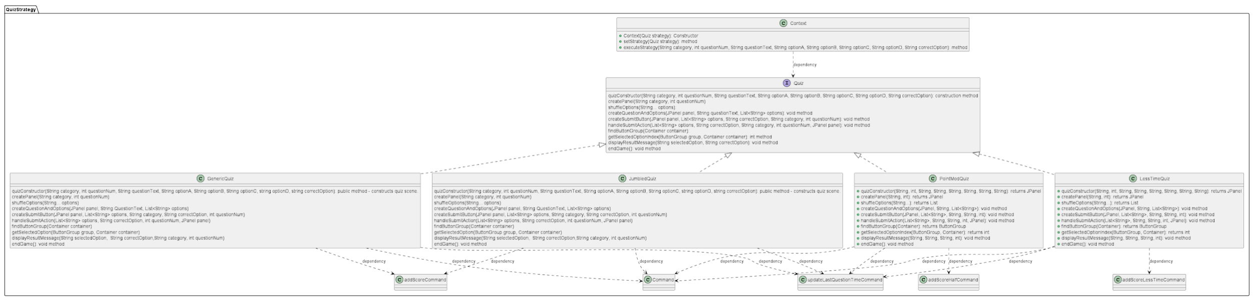
QuestionSingleton Package


QuestionSingleton
QuestionRepository
QuestionRepository(): private Constructor
getInstance(): static synchronized method
initalizeQuestions(): private method
addScienceQuestions(): private method
addHistoryQuestions(): private method
addEntertainmentQuestions(): private method
getScienceQuestions(): public method
getHistoryQuestions(): public method
getEntertainmentQuestions(): public method
dependency
QuestionFactory
createQuestion(String questionText, String optionA, String optionB, String optionC, String optionD, String correctOption): Question
dependency
dependency
Question
Question(String questionText, String optionA, String optionB, String optionC, String optionD, String correctOption) : Constructor

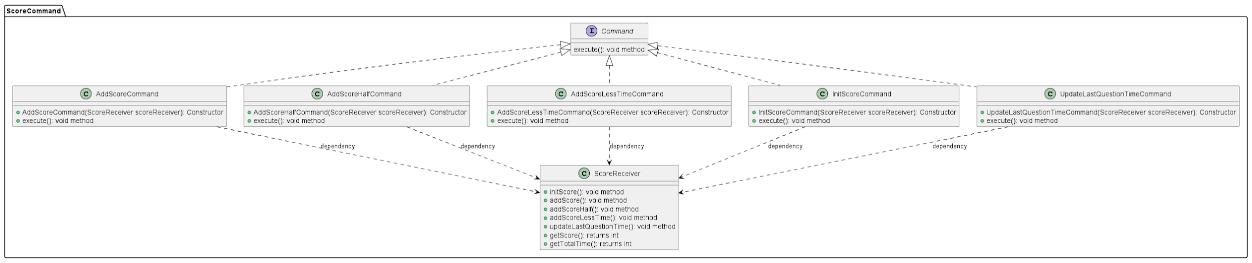
QuizSceneFactory Package


QuizSceneFactory
JPanel
QuizSceneCreator
createScene(int questionNum, String questionText, String optionA, String optionB, String optionC, String optionD, String correctOption):
constructScene(): abstract method
EntertainmentCreator
constructScene(): method - returns EntertainmentQuiz
HistoryCreator
constructScene(): method
ScienceCreator
constructScene(): method - returns ScienceQuiz
QuizSceneInterface
initalizeScene(int questionNum, String questionText, String optionA, String optionB, String optionC, String optionD, String correctOption):
EntertainmentQuiz
initalizeScene(int questionNum, String questionText, String optionA, String optionB, String optionC, String optionD, String correctOption)
HistoryQuiz
initalizeScene(int questionNum, String questionText, String optionA, String optionB, String optionC, String optionD, String correctOption)
ScienceQuiz
initalizeScene(int questionNum, String questionText, String optionA, String optionB, String optionC, String optionD, String correctOption)
dependency
Quiz
JumbledQuiz
GenericQuiz
PointModQuiz
LessTimeQuiz
Context

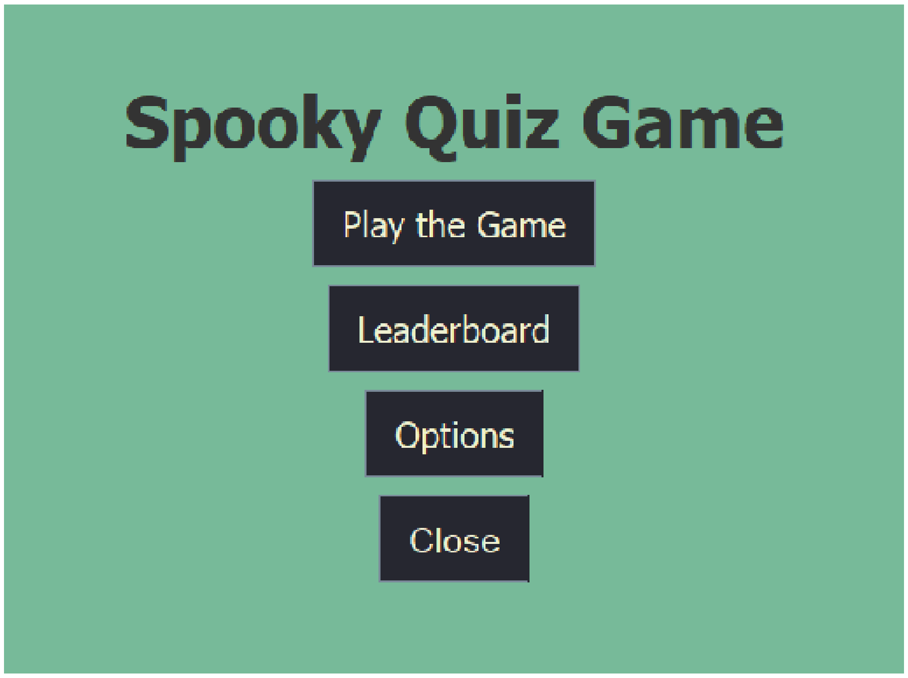
QuizStrategy Package


QuizStrategy
Context
Context(Quiz strategy): Constructor
setStrategy(Quiz strategy): method
executeStrategy(String category, int questionNum, String questionText, String optionA, String optionB, String optionC, String optionD, String correctOption): method
dependency
Quiz
quizConstructor(String category, int questionNum, String questionText, String optionA, String optionB, String optionC, String optionD, String correctOption): construction method
createPanel(String category, int questionNum)
shuffleOptions(String... options):
createQuestionAndOptions(JPanel panel, String questionText, List<String> options): void method
createSubmitButton(JPanel panel, List<String> options, String correctOption, String category, int questionNum): void method
handleSubmitAction(List<String> options, String correctOption, String category, int questionNum, JPanel panel): void method
findButtonGroup(Container container):
getSelectedOptionIndex(ButtonGroup group, Container container): int method
displayResultMessage(String selectedOption, String correctOption): void method
endGame(): void method
GenericQuiz
quizConstructor(String category, int questionNum, String questionText, String optionA, String optionB, String optionC, string optionD, string correctOption): public method - constructs quiz scene.
createPanel(String category, int questionNum)
shuffleOptions(String... options)
createQuestionAndOptions(JPanel panel, String QuestionText, List<String> options)
createSubmitButton(JPanel panel, List<String> options, String category, String correctOption, int questionNum)
handleSubmitAction(List<String> options, String correctOption, int questionNum, JPanel panel)
findButtonGroup(Container container)
getSelectedOption(ButtonGroup group, Container container)
displayResultMessage(String selectedOption, String correctOption,String category, int questionNum)
endGame(): void method
JumbledQuiz
quizConstructor(String category, int questionNum, String questionText, String optionA, String optionB, String optionC, string optionD, string correctOption): public method - constructs quiz scene.
createPanel(String category, int questionNum)
shuffleOptions(String... options)
createQuestionAndOptions(JPanel panel, String QuestionText, List<String> options)
createSubmitButton(JPanel panel, List<String> options, String category, String correctOption, int questionNum)
handleSubmitAction(List<String> options, String correctOption, int questionNum, JPanel panel)
findButtonGroup(Container container)
getSelectedOption(ButtonGroup group, Container container)
displayResultMessage(String selectedOption, String correctOption,String category, int questionNum)
endGame(): void method
PointModQuiz
quizConstructor(String, int, String, String, String, String, String, String): returns JPanel
createPanel(String, int): returns JPanel
shuffleOptions(String...): returns List
createQuestionAndOptions(JPanel, String, List<String>): void method
createSubmitButton(JPanel, List<String>, String, String, int): void method
handleSubmitAction(List<String>, String, String, int, JPanel): void method
findButtonGroup(Container): returns ButtonGroup
getSelectedOptionIndex(ButtonGroup, Container): returns int
displayResultMessage(String, String, String, int): void method
endGame(): void method
LessTimeQuiz
quizConstructor(String, int, String, String, String, String, String, String): returns JPanel
createPanel(String, int): returns JPanel
shuffleOptions(String...): returns List
createQuestionAndOptions(JPanel, String, List<String>): void method
createSubmitButton(JPanel, List<String>, String, String, int): void method
handleSubmitAction(List<String>, String, String, int, JPanel): void method
findButtonGroup(Container): returns ButtonGroup
getSelectedOptionIndex(ButtonGroup, Container): returns int
displayResultMessage(String, String, String, int): void method
endGame(): void method
dependency
addScoreCommand
Command
updateLastQuestionTimeCommand
addScoreHalfCommand
addScoreLessTimeCommand

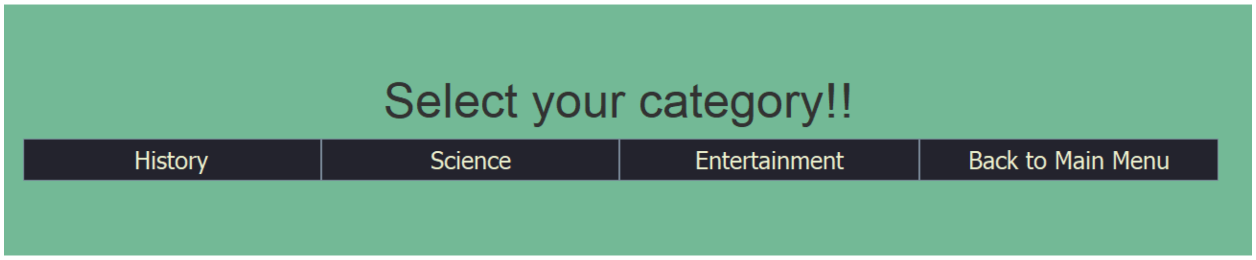
ScoreCommand Package


ScoreCommand
Command
execute(): void method
AddScoreCommand
AddScoreCommand(ScoreReceiver scoreReceiver): Constructor
execute(): void method
AddScoreHalfCommand
AddScoreHalfCommand(ScoreReceiver scoreReceiver): Constructor
execute(): void method
AddScoreLessTimeCommand
AddScoreLessTimeCommand(ScoreReceiver scoreReceiver): Constructor
execute(): void method
InitScoreCommand
InitScoreCommand(ScoreReceiver scoreReceiver): Constructor
execute(): void method
UpdateLastQuestionTimeCommand
UpdateLastQuestionTimeCommand(ScoreReceiver scoreReceiver): Constructor
execute(): void method
dependency
ScoreReceiver
initScore(): void method
addScore(): void method
addScoreHalf(): void method
addScoreLessTime(): void method
updateLastQuestionTime(): void method
getScore(): returns int
getTotalTime(): returns int

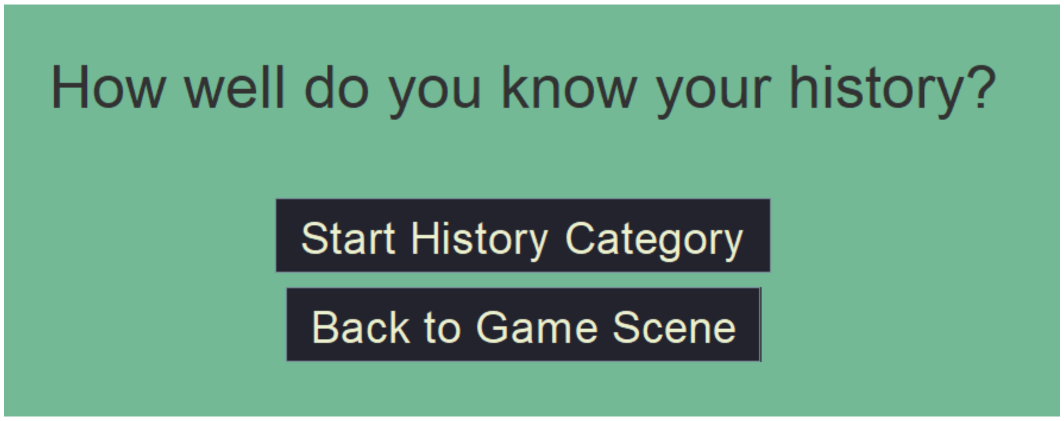
Screenshots

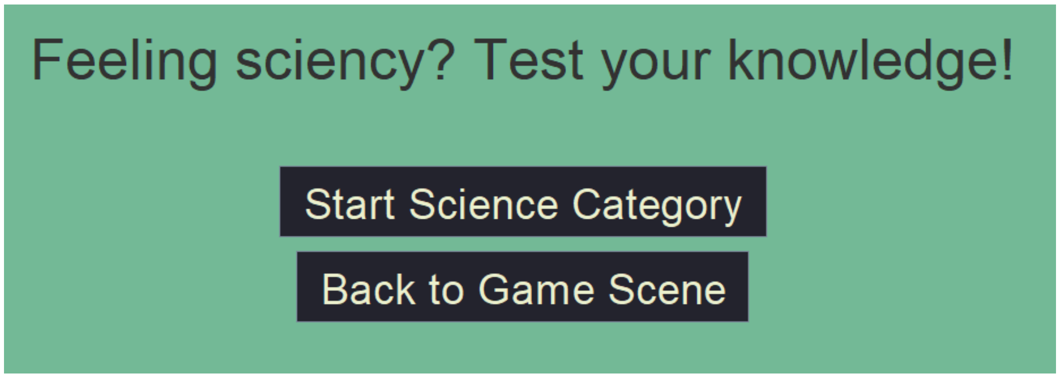


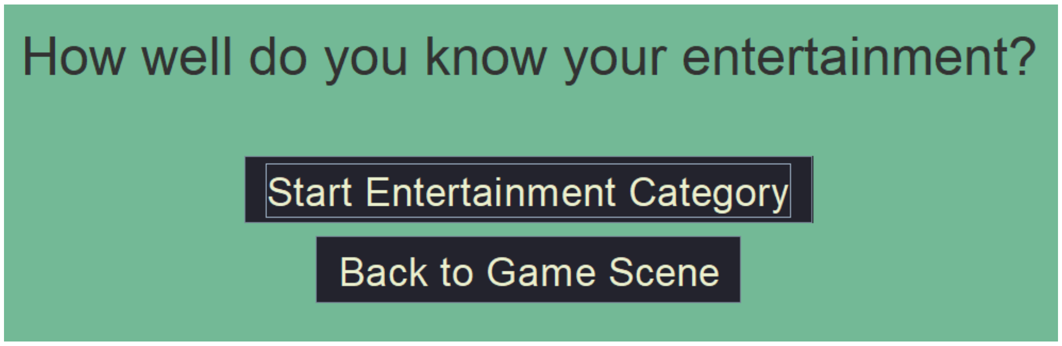
The Homepage of our application. “Play the game” brings you to the Category Select screen, “Leaderboard” brings you to the leaderboard display, “options” allows you to clear the leaderboard, and “close” closes the application



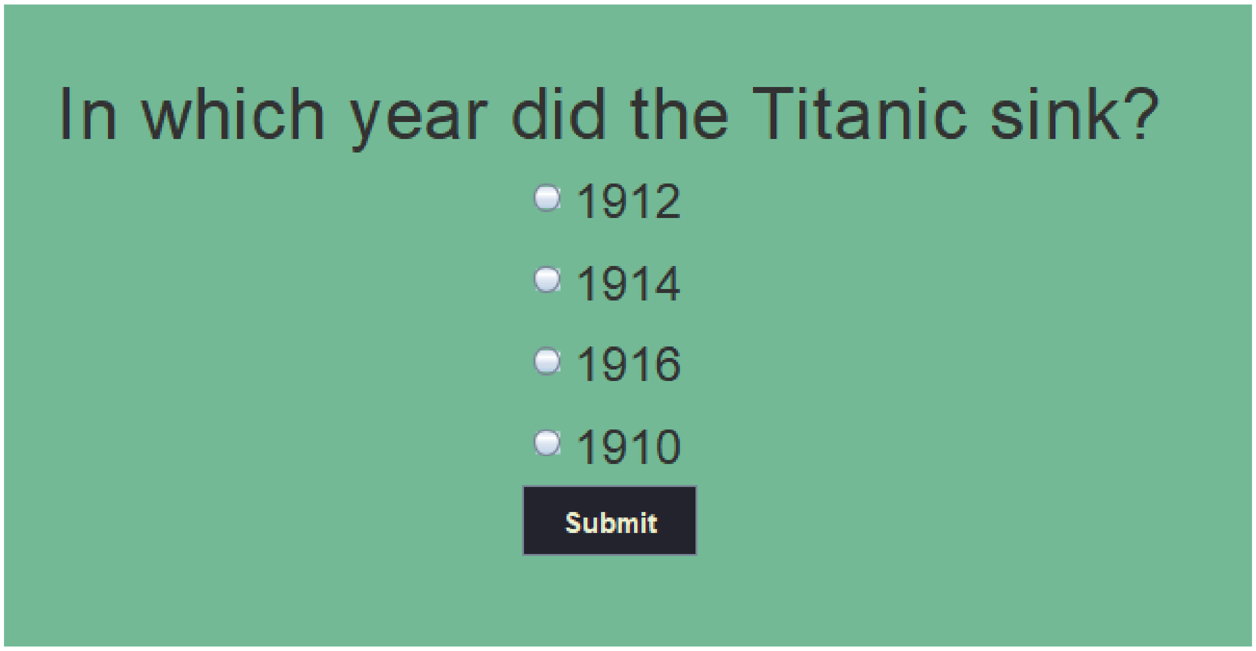
After selecting the “Play the Game” button, you are brought to the Category Select screen. History, Science, and Entertainment each will bring you to the respective quiz, and “Back to Main Menu” returns you to the previous menu screen.






How well do you know your entertainment?
Start Entertainment Category
Back to Game Scene

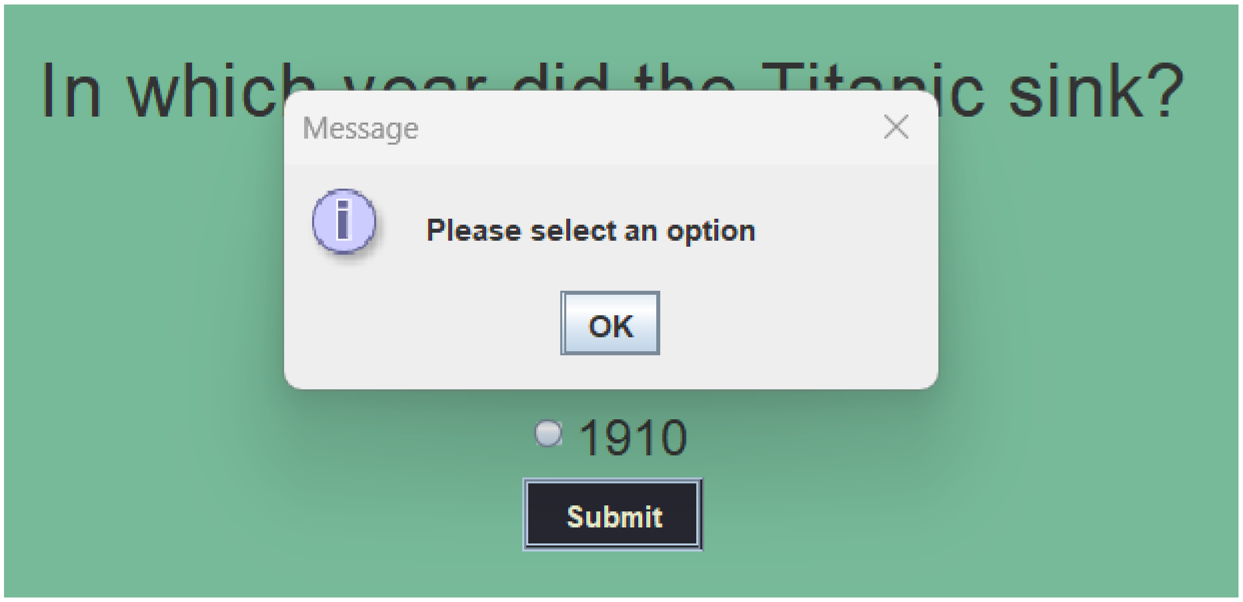
History, Science, and Entertainment all have very similar starting screens. The first button will begin the respective quiz, and the “Back to Game Scene” returns you to the category select screen.



In which year did the Titanic sink?

- 1912
- 1914
- 1916
- 1910

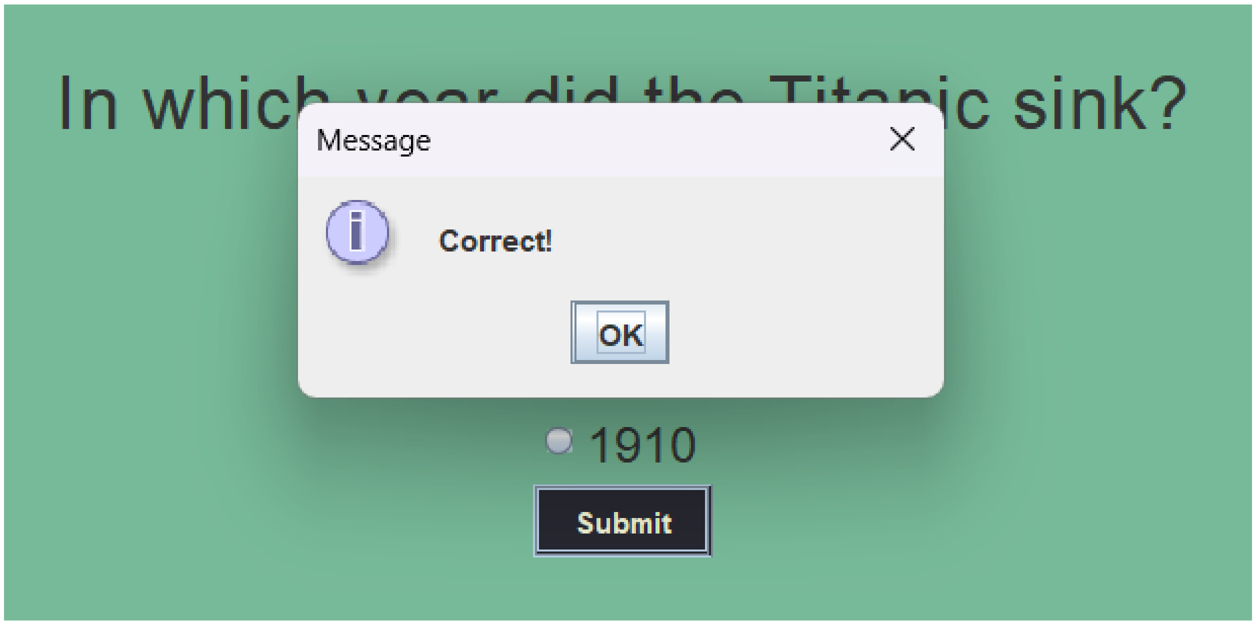
A sample question from the History section. Only one of the options can be selected at once, and submit will compare the selected answer to the correct one.



Message

Please select an option

Pressing Submit before selecting an option will not allow the quiz to proceed, but will instead spawn a pop up window requesting you go back to select an option. Pressing okay will close this pop up window.

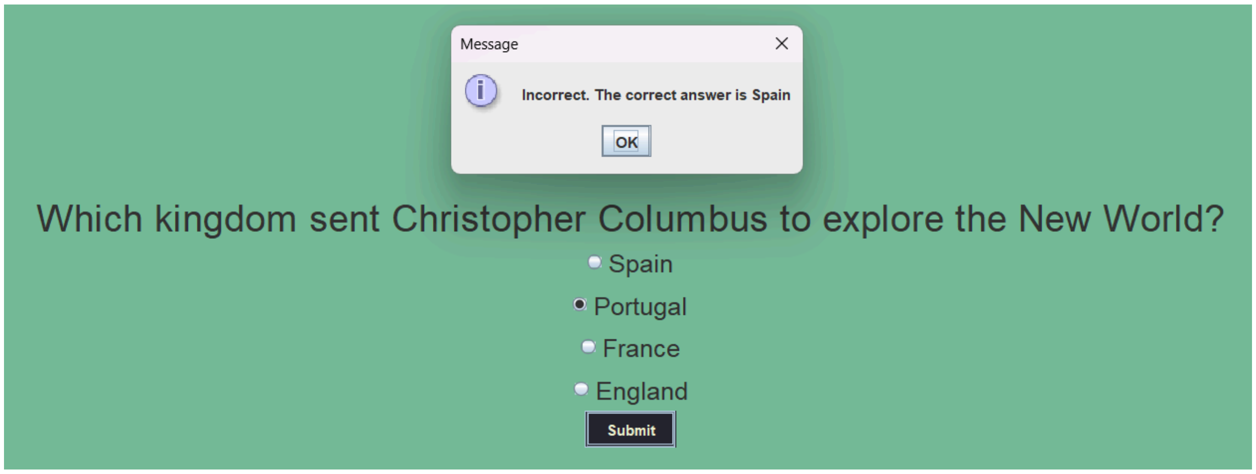


In which year did the Titanic sink?

Correct!

1910

Choosing the correct options spawns a pop up window telling you that you chose the correct answer. Pressing okay will close the window, and replace the question data with the next in queue.

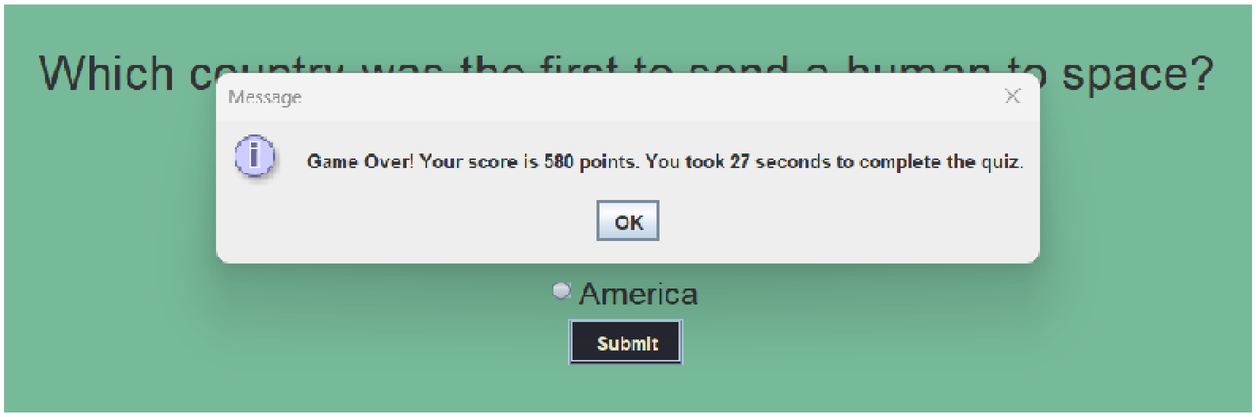


Incorrect. The correct answer is Spain

Which kingdom sent Christopher Columbus to explore the New World?

- Spain
- ● Portugal
- France
- England

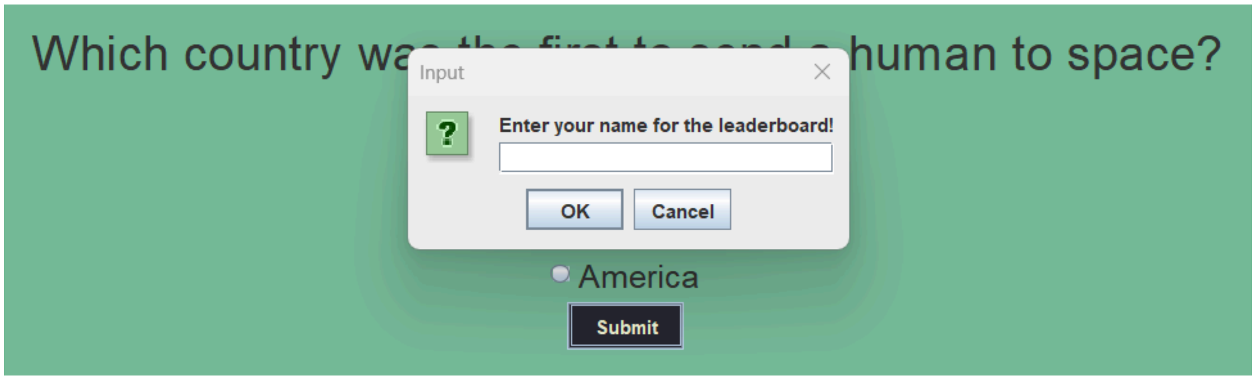
Selecting the wrong option will spawn a pop up window telling you your answer was incorrect, and informing you of the correct answer. Pressing okay will close the window, and replace the question data with the next in queue.


Which country was the first to send a human to space?

Message

Game Over! Your score is 580 points. You took 27 seconds to complete the quiz.

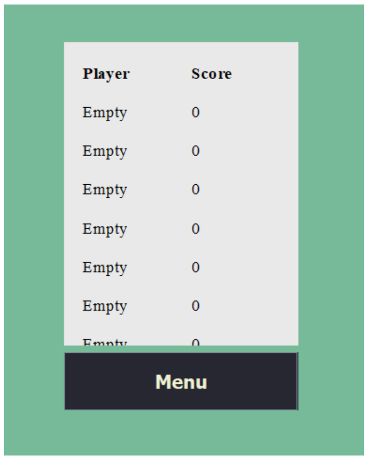
When all 10 questions have been answered, a pop up will inform the user the quiz is over instead of loading another question. It will tell the user their score, as well as their time. Pressing okay will load another pop up to ask the user to input their name for the leaderboard.


Which country was the first to send a human to space?

Input

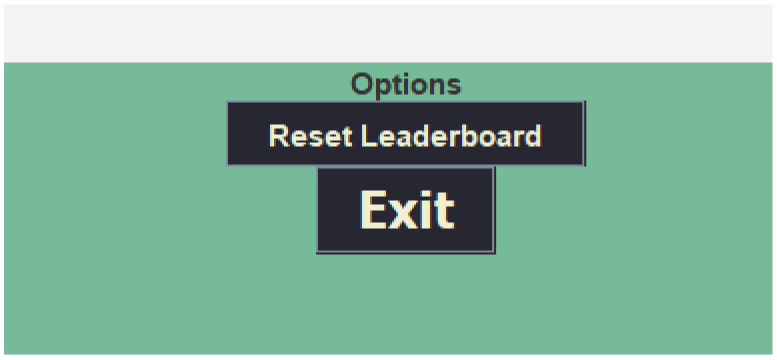
Enter your name for the leaderboard!

Pressing okay when there is no input, or pressing cancel will not save your data to the leaderboard, and close the pop up. Entering a string and pressing OK will submit your results to the leaderboard, and close the pop up. When this pop up is closed, you will be brought to the leaderboard screen



Player	Score
Empty	0
Empty	0
Empty	0
Empty	0
Empty	0
Empty	0
Empty	0

The leaderboard stage shows all of the highscores achieved by players. When there is no data, it displays Empty as the name, and 0 as the score. This screen can be accessed at any time using the leaderboard button on the home screen. Pressing Menu will bring you back to the main menu.



The options screen presents 2 buttons. “Reset Leaderboard” will clear all entries on the leaderboard when pressed. “Exit” will bring you back to the main menu.