Spooky Quiz

•••

Software Engineering 2 Final Project

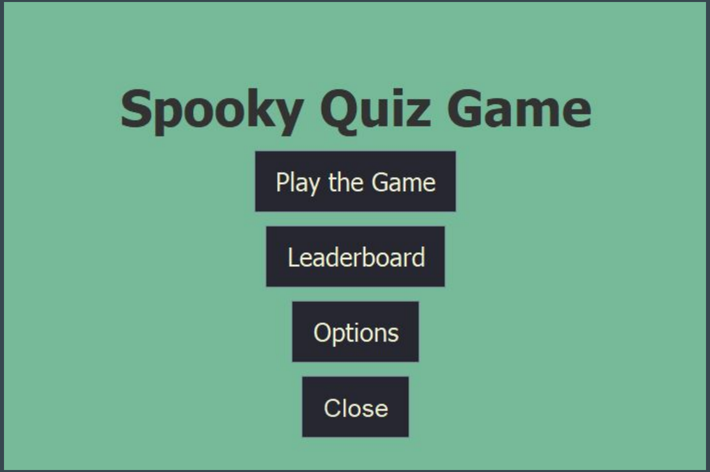
~

Noah Baker
Jakob Legere
Riley Meyers

Overview

Introduction to Spooky Quiz

- Quiz game created in Java
- Haunted by an evil spirit that causes questions to occasionally act up
- Complete all the questions in the quiz in record time and go for the high score



Purpose & Function

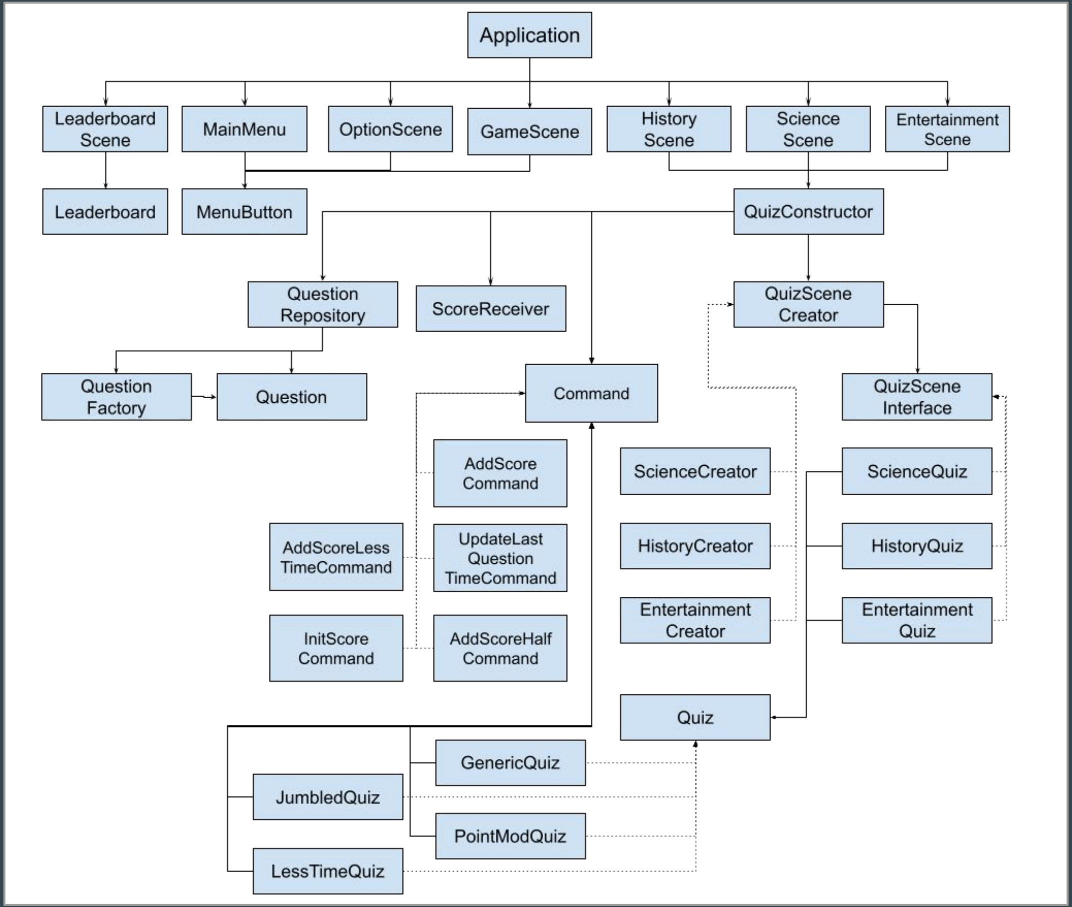
- Create a system that demonstrates our understanding of design patterns
- Choose patterns that made our code easy to modify, as well as add extra features and content
- Make something fun :)

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

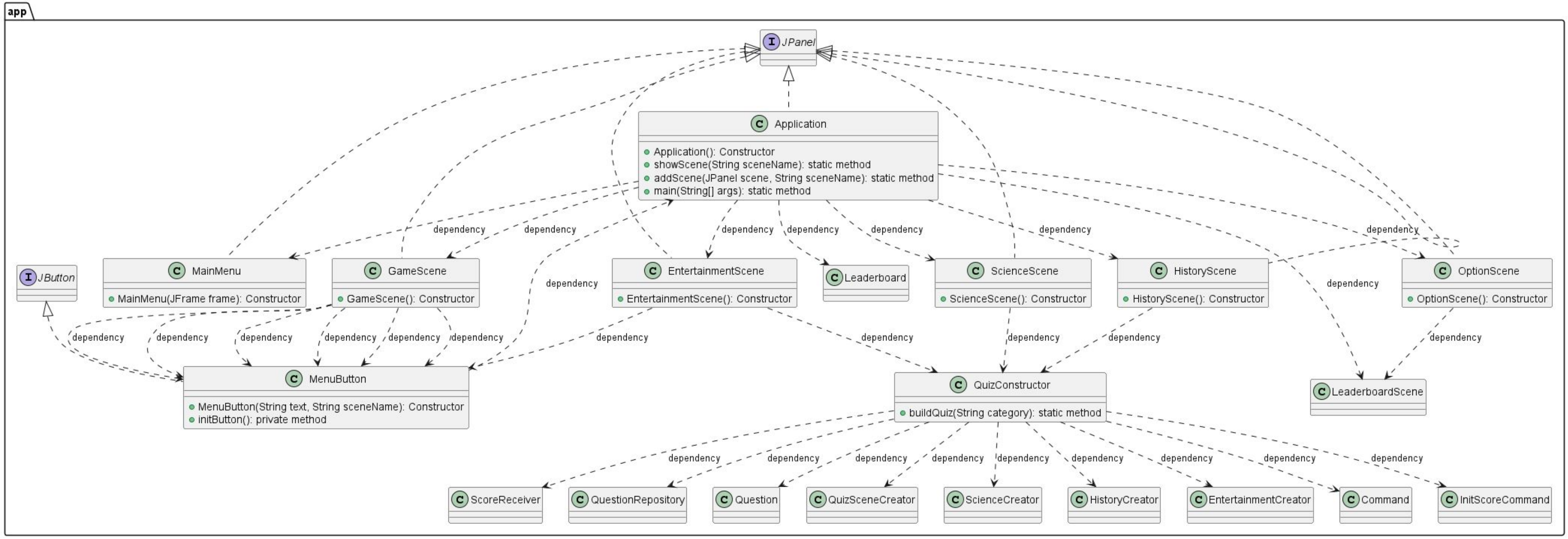
Programming Language

- Why Java?
  - The OOP language
  - Strictness - Doesn't let us get away with anything
- We all had a *little* knowledge.
- Lots of examples for design patterns; very documented

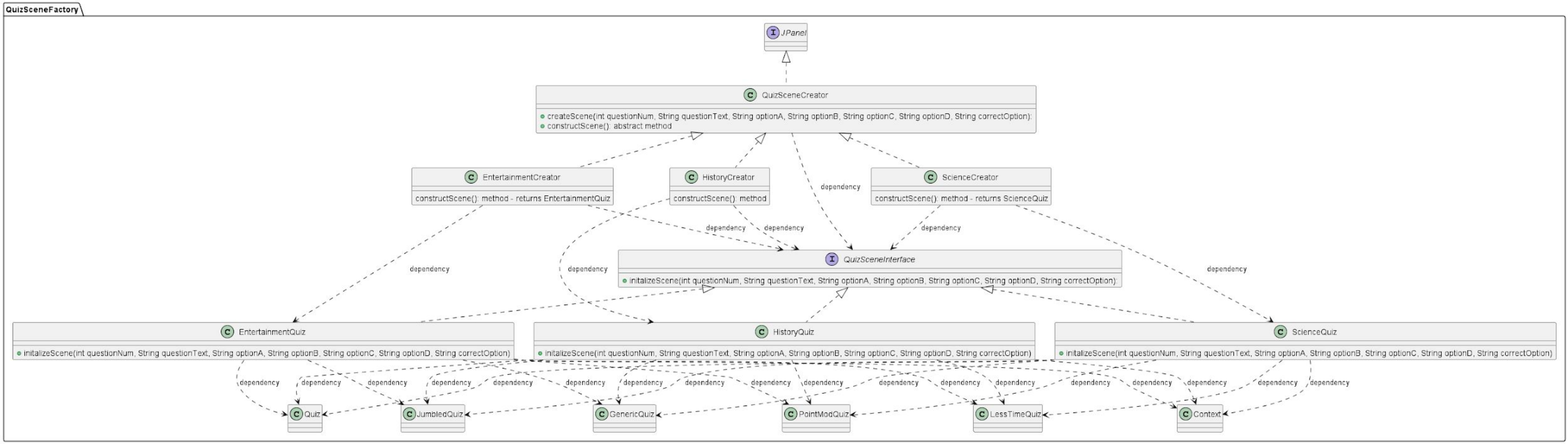
System Model

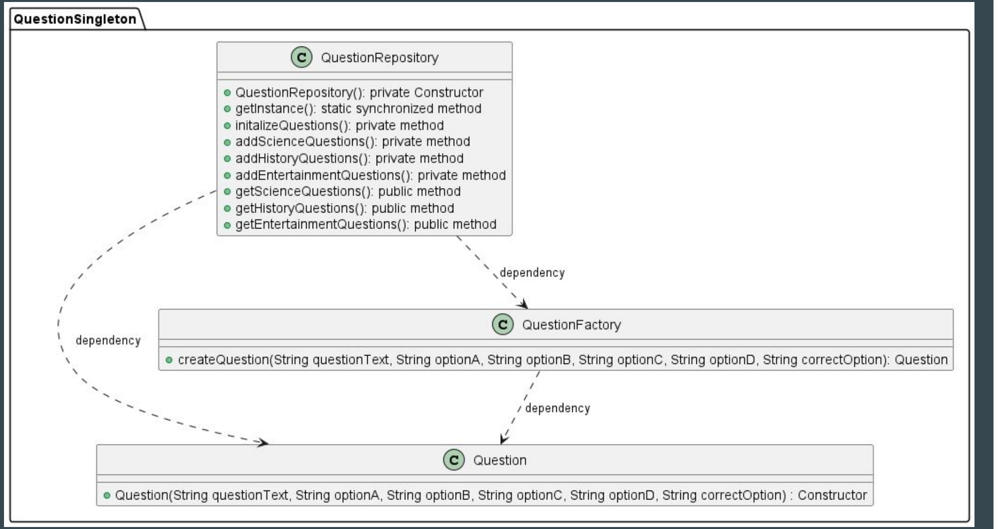
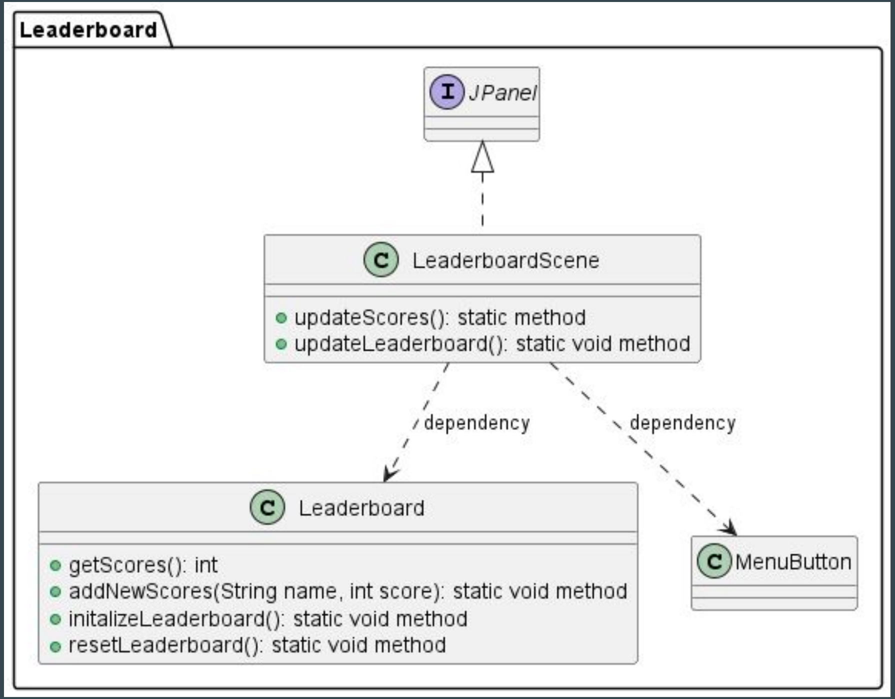


App Package

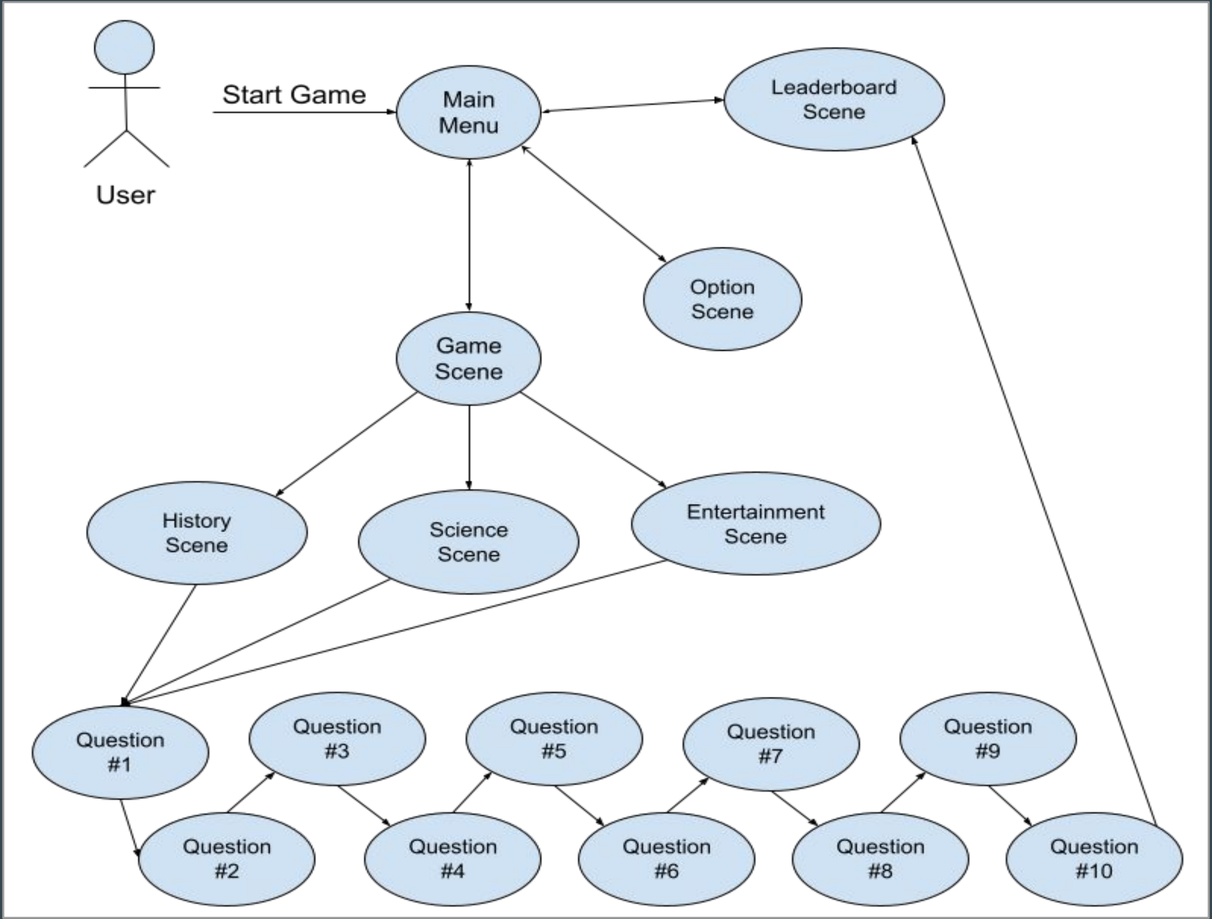

app
JPanel
Application
Application(): Constructor
showScene(String sceneName): static method
addScene(JPanel scene, String sceneName): static method
main(String[] args): static method
JButton
MainMenu
MainMenu(JFrame frame): Constructor
GameScene
GameScene(): Constructor
EntertainmentScene
EntertainmentScene(): 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

QuizSceneFactory

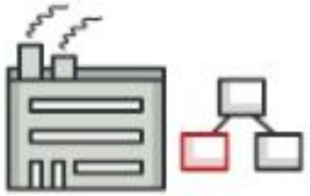

QuizSceneFactory
JPanel
QuizSceneCreator
createScene(int questionNum, String questionText, String optionA, String optionB, String optionC, String optionD, String correctOption):
constructScene(): abstract method
EntertainmentCreator
constructScene(): method - returns EntertainmentQuiz
HistoryCreator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


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

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

User Flow

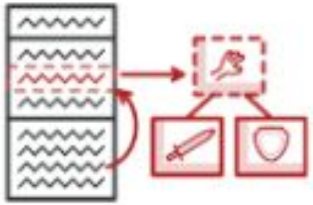

User
Start Game
Main Menu
Leaderboard Scene
Option Scene
Game Scene
History Scene
Science Scene
Entertainment Scene
Question #1
Question #2
Question #3
Question #4
Question #5
Question #6
Question #7
Question #8
Question #9
Question #10

Design Patterns


Factory Method

Command

Singleton


Strategy

Factory Method

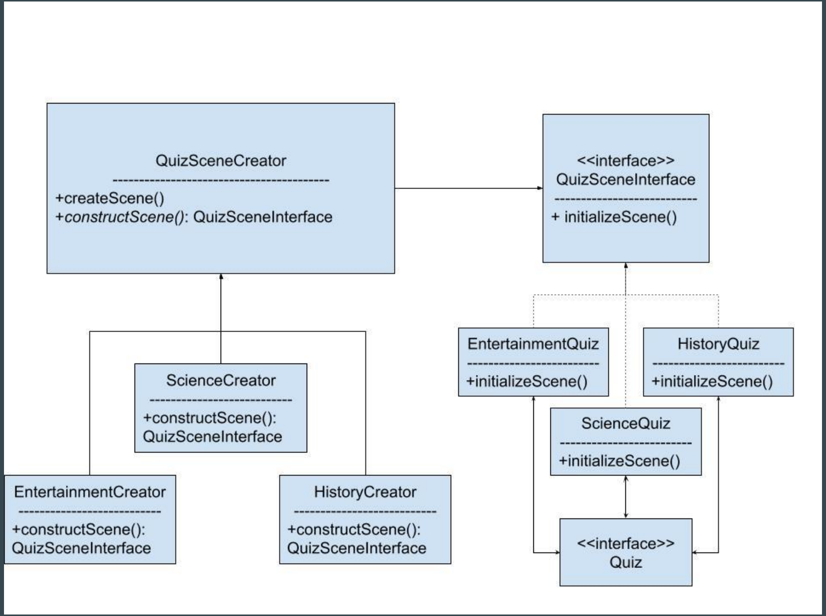
-Javax Swing requires the creation of scenes

-We need to create the question scenes in a slightly different way depending on the category

-Our factory pattern returns JPanel objects that we can add to the main panel

- We have a concrete product and concrete constructor for each category

- Allows us to add more categories easily (open / closed principle) and provide single place for object creation code



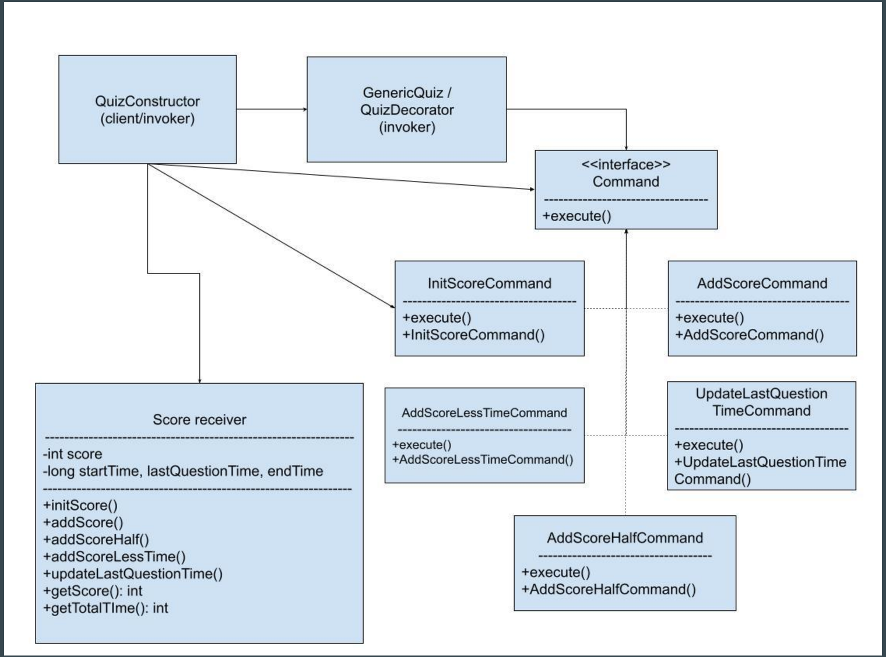
Command

-Chose command to enforce a specific way of increasing the score

-Especially important due to our Spooky features that modify how score for each question is calculated

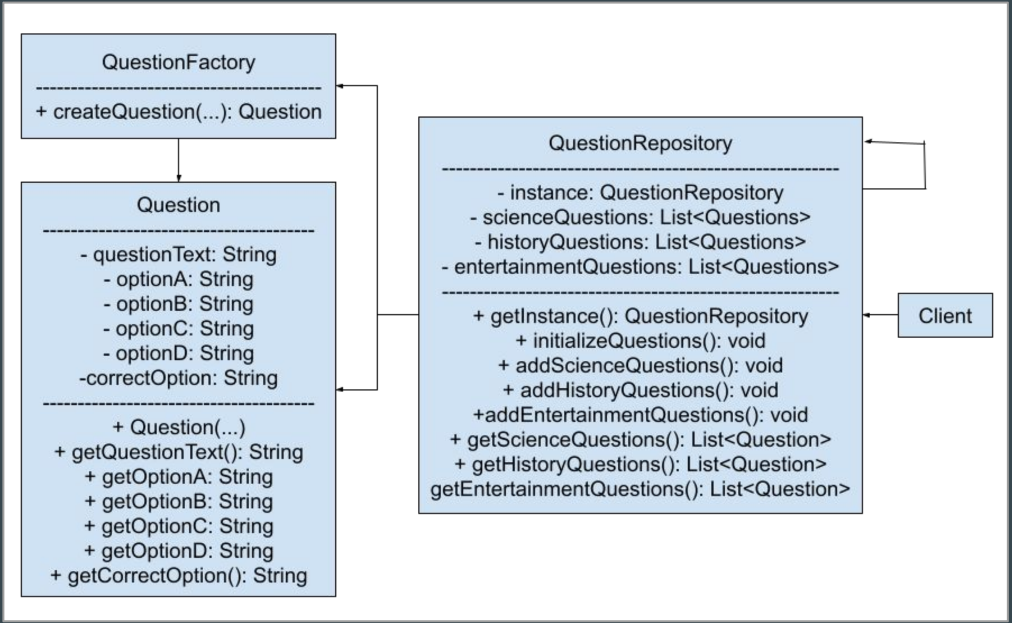
-Command interface allows the rest of the application to work with Command objects interchangeably by calling execute();

-Implementing the command pattern makes it easier to extend our spooky features in the future without breaking other code



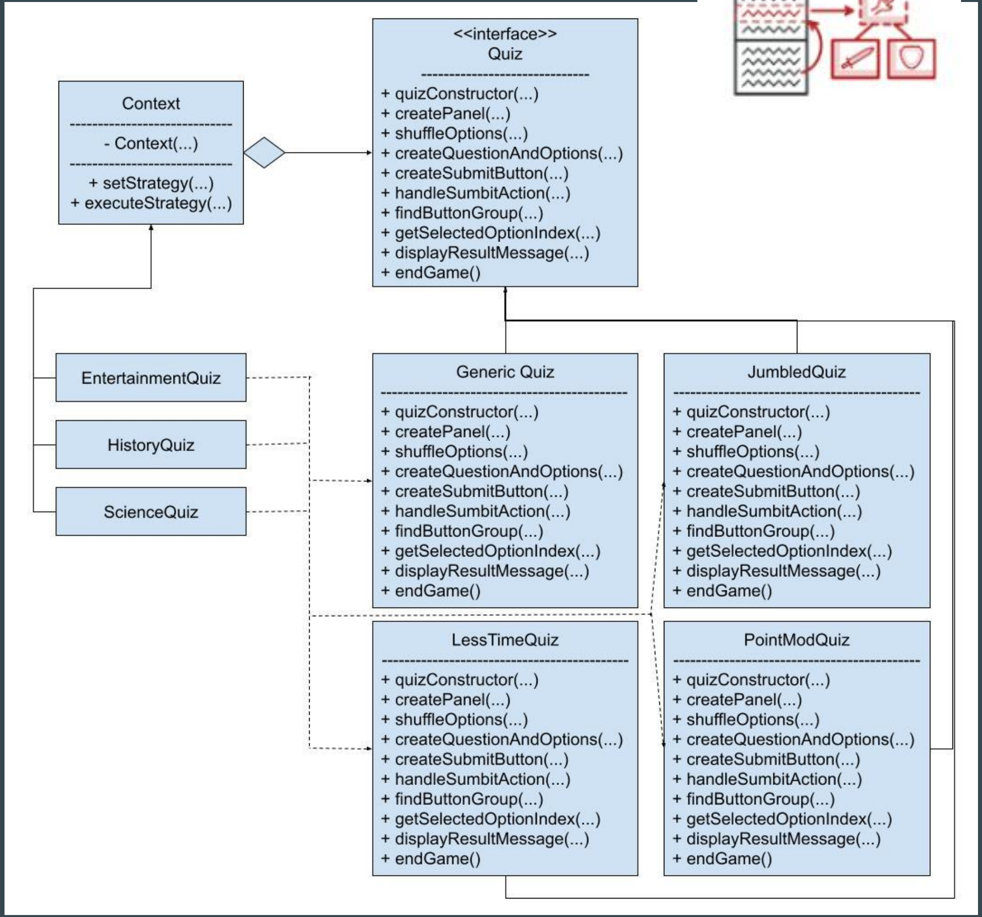
Singleton

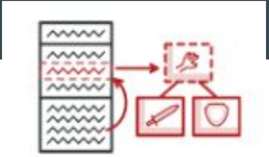
- Generates Questions as well as Question lists for each category
- Singleton ensures that only one instance of the questions lists exists at once
- QuestionRepository also contains the getter methods for the question lists
    - Gives us a consistent and reliable way to call in the questions


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

Strategy

- Used to dynamically alter the traits of individual questions, giving them their “Haunted” traits
- Why Strategy over other patterns?
  - We considered both Template and Wrapper patterns
  - Utilized Strategy in the end because it was more dynamic, worked at runtime, and was based around modifying object behaviour
- 4 Types of Concrete Strategies (Or question variations) exist in our final code





Conclusion - Summary

- Difficult at first
- We now appreciate OOP a lot more
- We realize how programming in this way is beneficial for the long-term maintenance and scalability of any program
- Adding the later design patterns were easier with the solid foundation using our previous design patterns.

Conclusion - Future Work

- UI design
- More categories/Questions (thanks design patterns!)
- More haunted aspects
- QOL on details - input checking, more options, audio, etc
- Extending factory pattern to abstract factory pattern

Conclusion - Final Thoughts

- Some design patterns were a straightforward pick for us due to our use of Swing.
  - Factory for Jpanels
- We needed to stay flexible
  - Some patterns that we thought looked good on paper, didn't work with our existing system well when trying to create UML diagrams.
- **We learned a valuable lesson about design patterns, and how the upfront short-term time commitment to ensuring quality code is written is well worth it for the long term maintainability and scalability of the codebase.**