Assignment 7

I. Project Details

Title: Love Island Game Show Simulator

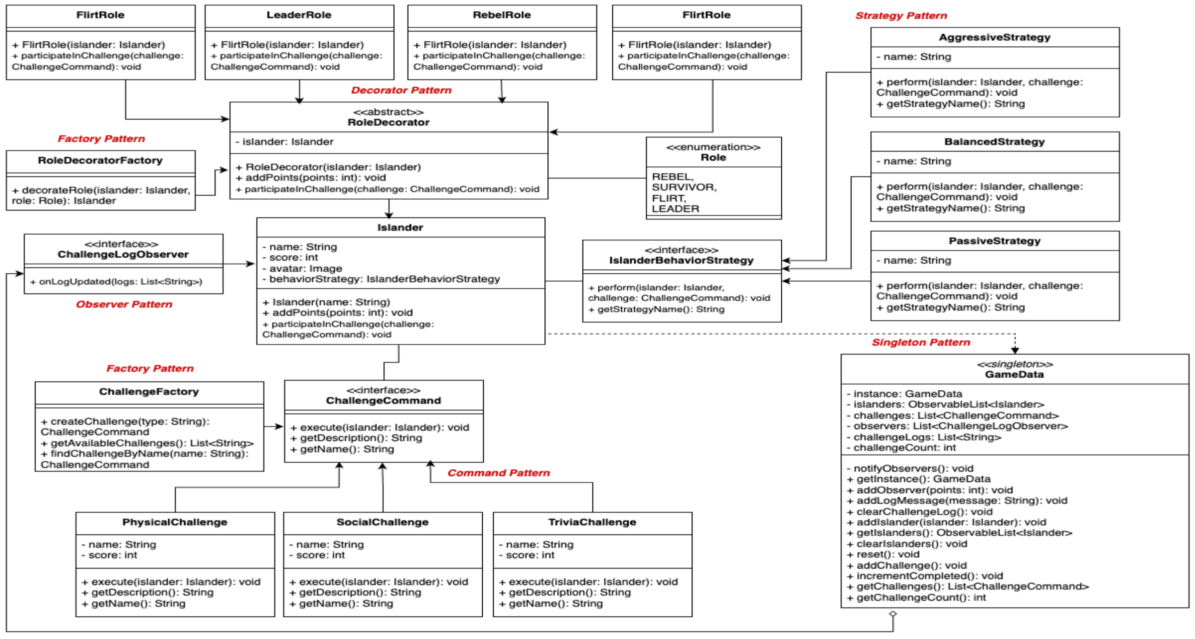
Team: Alexis Davidson, Tiara Gibson, Khushi Jani

II. Final State of System

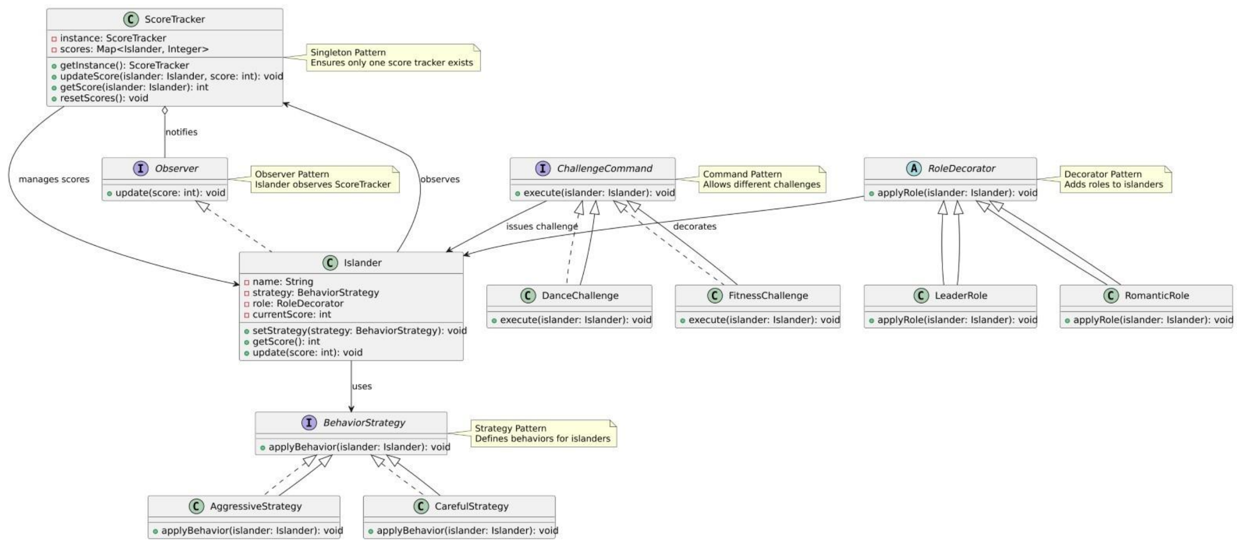
The Love Island Game Show Simulator is a simulation of a reality game show where contestants ("islanders") interact, participate in challenges, and earn scores based on their performance and behaviors. As a part of the simulation, the user can initialize the islanders, assigning each with a unique name and a behavior strategy for each. The user can manually enter each islander's name and behavior strategy, or the information can be randomized. Before each challenge is simulated, the user can select role(s) for each islander that influences their performance on the upcoming challenge. A minimum of three challenges must be performed before officially ending the game simulation and viewing the winners (the top 3 islanders by score).

III. Final Class Diagram and Comparison

Final Class Diagram


FlirtRole
+ FlirtRole(islander: Islander)
+ participateInChallenge(challenge: ChallengeCommand): void
LeaderRole
+ FlirtRole(islander: Islander)
+ participateInChallenge(challenge: ChallengeCommand): void
RebelRole
+ FlirtRole(islander: Islander)
+ participateInChallenge(challenge: ChallengeCommand): void
FlirtRole
+ FlirtRole(islander: Islander)
+ participateInChallenge(challenge: ChallengeCommand): void
Strategy Pattern
AggressiveStrategy
- name: String
+ perform(islander: Islander, challenge: ChallengeCommand): void
+ getStrategyName(): String
Decorator Pattern
<<abstract>>
RoleDecorator
- islander: Islander
+ RoleDecorator(islander: Islander)
+ addPoints(points: int): void
+ participateInChallenge(challenge: ChallengeCommand): void
Factory Pattern
RoleDecoratorFactory
+ decorateRole(islander: Islander, role: Role): Islander
<<enumeration>>
Role
REBEL,
SURVIVOR,
FLIRT,
LEADER
BalancedStrategy
- name: String
+ perform(islander: Islander, challenge: ChallengeCommand): void
+ getStrategyName(): String
Islander
- name: String
- score: int
- avatar: Image
- behaviorStrategy: IslanderBehaviorStrategy
+ Islander(name: String)
+ addPoints(points: int): void
+ participateInChallenge(challenge: ChallengeCommand): void
<<interface>>
ChallengeLogObserver
+ onLogUpdated(logs: List<String>)
Observer Pattern
<<interface>>
IslanderBehaviorStrategy
+ perform(islander: Islander, challenge: ChallengeCommand): void
+ getStrategyName(): String
PassiveStrategy
- name: String
+ perform(islander: Islander, challenge: ChallengeCommand): void
+ getStrategyName(): String
Singleton Pattern
<<singleton>>
GameData
- instance: GameData
- islanders: ObservableList<Islander>
- challenges: List<ChallengeCommand>
- observers: List<ChallengeLogObserver>
- challengeLogs: List<String>
- challengeCount: int
- notifyObservers(): void
+ getInstance(): GameData
+ addObserver(points: int): void
+ addLogMessage(message: String): void
+ clearChallengeLog(): void
+ addIslander(islander: Islander): void
+ getIslanders(): ObservableList<Islander>
+ clearIslanders(): void
+ reset(): void
+ addChallenge(): void
+ incrementCompleted(): void
+ getChallenges(): List<ChallengeCommand>
+ getChallengeCount(): int
Factory Pattern
ChallengeFactory
+ createChallenge(type: String): ChallengeCommand
+ getAvailableChallenges(): List<String>
+ findChallengeByName(name: String): ChallengeCommand
<<interface>>
ChallengeCommand
+ execute(islander: Islander): void
+ getDescription(): String
+ getName(): String
Command Pattern
PhysicalChallenge
- name: String
- score: int
+ execute(islander: Islander): void
+ getDescription(): String
+ getName(): String
SocialChallenge
- name: String
- score: int
+ execute(islander: Islander): void
+ getDescription(): String
+ getName(): String
TriviaChallenge
- name: String
- score: int
+ execute(islander: Islander): void
+ getDescription(): String
+ getName(): String

Project 5 Class Diagram


ScoreTracker
instance: ScoreTracker
scores: Map<Islander, Integer>
getInstance(): ScoreTracker
updateScore(islander: Islander, score: int): void
getScore(islander: Islander): int
resetScores(): void
Singleton Pattern
Ensures only one score tracker exists
notifies
manages scores
observes
Observer
update(score: int): void
Observer Pattern
Islander observes ScoreTracker
ChallengeCommand
execute(islander: Islander): void
Command Pattern
Allows different challenges
RoleDecorator
applyRole(islander: Islander): void
Decorator Pattern
Adds roles to islanders
issues challenge
decorates
Islander
name: String
strategy: BehaviorStrategy
role: RoleDecorator
currentScore: int
setStrategy(strategy: BehaviorStrategy): void
getScore(): int
update(score: int): void
DanceChallenge
execute(islander: Islander): void
FitnessChallenge
execute(islander: Islander): void
LeaderRole
applyRole(islander: Islander): void
RomanticRole
applyRole(islander: Islander): void
uses
BehaviorStrategy
applyBehavior(islander: Islander): void
Strategy Pattern
Defines behaviors for islanders
AggressiveStrategy
applyBehavior(islander: Islander): void
CarefulStrategy
applyBehavior(islander: Islander): void

Key Changes

The final system for the Love Island Game Show Simulator is overall pretty similar to our intended plans from the first assignment, but there were a few additions made along the way. The singleton *GameData* was added once it was realized that the *ScoreTracker* that we originally planned to add did not actually capture all the different cases we wanted to track. Expanding on the functionality of *ScoreTracker* by creating *GameData* instead allowed us to not only track the scores of the islanders, but to also track the islanders, the observers for the logging, as well as the challenges that were being simulated.

As the application was being developed, we also recognized places where the factory design pattern could be implemented, which was also something that was added to the final system that was missing from our original UML diagrams. *RoleDecoratorFactory* and *ChallengeFactory* were both added to the final system, as the factory pattern allowed for a centralized location for the creation for the roles and the challenges. We realized that adding these factories would improve the overall flexibility of the application.

IV. Third-Party code vs. Original Code Statement

A lot of the code that was utilized in our project was based off what we had implemented for our Zoo Simulation assignment. The implementation of the design patterns was all based off our class notes as well as that initial project that was completed.

However, for utilizing JavaFX, that was where a bit of research had to come into play to correctly implement our intended functionality. While we did not spend too much time on the styling of the JavaFX project, there was definitely some references that needed to be made to tutorials and other third-party resources. The following are the resources that were referenced while implementing the GUI for this application:

- Geeks for Geeks Tutorial (Various Implementations) https://www.geeksforgeeks.org/javafx-tutorial/
- Tutorials Point Guide (Various Implementations) https://www.tutorialspoint.com/javafx/javafx_application.htm
- Official JavaFX Documentation https://docs.oracle.com/javafx/2/api/overview-summary.html
- Stack Overflow – Popup Window Implementation https://stackoverflow.com/questions/22166610/how-to-create-a-popup-window-in-javafx
- Stack Overflow – Listening to Scene Changes https://stackoverflow.com/questions/59686969/javafx-listen-to-scene-changes

Also, in order to generate the images of each “islander” as a part of our UI, we utilized DALL-E image generator, as that allowed us to get consistent styling images quickly, as our focus was on the implementation rather than the styles.

V. Statement on the OOAD process for your Overall Project

One key challenge that our team experienced was planning for the UI portion of the project. While we outlined the major workflows for our user interface in the beginning design phase of the project, we could have benefitted a bit more from spending time describing the structure of the project and how the backend would be able to communicate with the front end. Our intention was to have the main screens set up by the halfway point of the project to give ourselves enough time to clean it up and enhance the styling, however that was missed because we underestimated the amount of time the initial setup of the JavaFX scenes/controllers was going to take.

Along similar lines but in a positive light, the detail that was included in our initial project proposal for the backend implementation of the project was extremely helpful in the execution of the assignment. There really weren’t any issues when implementing the different design patterns and the overall flows of the project due to the detail included in planning, which was helpful because it did allow some more time to be spent on the UI portion that was a little more ambiguous.

Overall, our process was effective in completing the majority of what we had initially set out to accomplish. With the amount of information that we had put into the project proposal and other planning, it helped us stay on the same page when it came to the completion of these items and how we expected the system to work in general. I do think that the inclusion of some actual task planning like a Kanban board maty have helped us in the prioritization of development.