OOP Project : Snake Game - Insights

Relevant links

- The repo: https://github.com/mohitdmak/OOP-Assignment
- The video: https://drive.google.com/file/d/1NbJlScEJIN2VEpsowfjPSNuUx3oUtWJy/view?usp=sharing

Purpose of project / What does the project do?

✓ Our project, written completely in Java, and using all the object-oriented programming principles and practices taught in CS - F213 Course tries to replicate the popular "Snake and Food" game. (Details on how to run the project in the repo's README)

✓ When running the code, it generates a Graphical User Interface (which we implemented using Java. Fx and Java Swing modules) which asks the user to create a player account by signing up and then logging in to the app. In the home menu, the user chooses his preferred difficulty level (which has options "Easy", "Medium" and "Hard", and have varying speeds of the snake) and Board Dimensions (8 x 8, 10 x 10, and 12 x 12). The Game then starts and also keeps track of the user's high score in each board type upon game end. The user can also access the leaderboard for each board type, and know his rank among many other users. (NOTE: The high score is calculated based on the length of the snake at the time of the Game end and on a factor that is different for each difficulty level).

Analysis of Source code based on OOP Principles:

1.) Usage of abstraction over concrete classes :

- If we observe the requirements of our project, there are 4 main classes that need to be created without any use of abstraction (Snake, Board, Square, and Game). Thus there isn't any scope of having them based on some abstraction.
- However, we have a lot of GUI Classes that could have been based on some abstraction. For eg: The `signupGUI` and `loginGUI` classes have the most graphical elements in common (i.e A username entry, a password entry, submitting button, and a button to move to either the Signup Page or Login Page).
- These elements were created, defined, and used twice in both these classes, which could have been prevented by having them abstract some `" Authenticate "` class having those buttons and labels created.
- Thus only the code to be executed upon entering those details would need to be separately implemented in these classes since it is the only logic flow that is different for signup and login processes.

2.) Preferring usage of Interfaces over direct Implementations:

- Another major area upon which we could have improved is in the boardGUI class. While making the GUI code, the file became quite messy and large, due to our individual implementations of the GUI code for the 3 different board sizes. We took the preferred board size input from the user, and accordingly created the buttons, UI, and their integration with the functions in our "core" classes (Snake, Game, Board, and Square).
- This could have been improved by having an interface that has methods to create button objects, layout, and UI depending on the parameters supplied, and our 3 classes (boardGUI_small, boardGUI_medium, boardGUI_big for example) implementing this interface would thus generate the board GUI simply by supplying different parameters

3.) Encapsulating what varies in the code:

- This is a concept which we probably applied well enough in our code, by encapsulating objects like high scores, rankings, player accounts, into easily accessible and modifiable data structures. We extensively used Maps to map 2 related objects and to directly access one from the other instead of separately obtaining and operating on them. (For eg: mapping BoardSize enumerations with Set of Players rank list, or BoardSize with player's high scores, etc)

4.) Loose coupling between objects that interact:

- A lot of our objects depended on other objects being made in order for it to be made. For example, the Game object needs both a Snake and a Board object. We could have made them independent of each other, minimizing dependencies.

Analysis using Design Patterns (Observer Pattern)

- The observer pattern generally is useful to apply in cases where the state or properties of multiple classes depend or change based directly on the changes in one particular class of object.
- Our project would thus have been better made by applying this principle to all such sets of classes depending on a particular property or class.
- For eg: The property of Each and every square on the board GUI (i.e Color) depends on the snake's movements and user inputs. Thus upon each movement of the snake, the surrounding square objects must be notified of color change (i.e changing among colors of empty, food, snake node or snakehead squares), if any.
- We implemented this to some extent, by updating the Game class in every movement, and further checking all square objects for changes.