Project 7 - Final Report - Snakey 2D

Trevor Green, Amanda Smith, Connor Tribbett

Repo: https://github.com/trgr5899/Snakey-2D

Final Status:

Overall, the final state of our system is almost fully implemented from a functionality standpoint, but we were unable to implement some patterns we originally planned to due to design changes and lack of time/need. To begin, our game is fully functional with three snake game modes (Normal, Challenge, Multiplayer) with power ups including the supporting menus and functionality like leaderboards and account management. This includes the database for our data model and unseen functionality like a debug logger, which are all the functional features we originally planned for. The patterns we were able to implement include observer, singleton, and abstract factory for our loggers, game database, and power ups respectively. On the other hand, we did not implement the template pattern for our menus as originally planned in project 5 and realized this before project 6. The template pattern did not fit well because JavaFX allowed us to create menus with a single function that wasn't similar enough between menus. Furthermore, we did not end up implementing the command pattern to send user input and game commands to the snake object. This is mainly because the pattern ended up not being entirely necessary (but potentially could have given us performance benefits) and we did not have enough time to justify implementing it. Overall though, our final project ended in a great state and was able to leverage OOAD with patterns and general OO concepts.

UML Class Diagram:

Old Diagram:
https://drive.google.com/file/d/1FUfX6ekqt55ID9v1izWlae2r8BE9NxuU/view?usp=sharing

New Diagram:
https://drive.google.com/file/d/1ABJTsWco9AEZahmhr-TvSreLbO6FyMO4/view?usp=sharing

Changes: In terms of game functionality the only changes that were made to remove the ChallengeMode class and just create a function for it within the SinglePlayerGame class. Then we added a few more functions to the Grid and Game class. Most of these functions were just utilized for making the game physics/functionality easier.

Third-Party Code Statement:

Most third-party code we used came from the JavaFX and sqlite-JDBC frameworks. These provided basic functionality for displaying 2d graphics and connecting to SQLite

databases in java respectively. These libraries helped abstract hard, OS-specific tasks but still required us to build our entire project with the building blocks provided.

On top of this, we used some of the plentiful JavaFX documentation from oracle and other sites to guide us in implementing things like menus and the general structure of a JavaFX app. Specific documentation pages used are listed in the code, and the overall documentation links are provided below. Examples used in this case were often heavily modified to fit our needs.

Finally, we used an online tutorial for Snake in JavaFX as a guide for game development within JavaFX. This implementation was very short and not written in an object oriented manner, so it was used mainly as a reference for things like user input and game/drawing logic structure. These are all very general topics related to game development, but the reference provided us with how to implement these ideas in JavaFX. Overall though, our game implementation is much more complex, object-oriented, and well designed.

JavaFX References:
https://www.tutorialspoint.com/javafx/
https://docs.oracle.com/javase/8/javase-clienttechnologies.htm
Snake Reference:
https://github.com/Gaspared/snake

OOAD Process Statement:

1. One key design issue we had was a lack of understanding the underlying technology we were using (JavaFX) and how that would impact our proposed UML design. This arose because even if our proposed implementation of a pattern was sound in theory, JavaFX had oddities and extra functionality we did not anticipate that removed the need for it. For example, we originally planned to implement our menus in a template pattern, but JavaFX provided an easy way to create a menu in one function that didn't lend itself to this pattern.
2. One positive of our design process was leveraging interfaces and class inheritance to allow the independent development of large portions of our game that were easily tied together. For example, we were easily able to develop important objects like the Database and many menus independently and tie them all together in minutes. This helped development, and showed low coupling in our design.
3. Another positive aspect of our design process was incorporating a number of observers into the code from the start. Specifically, having a debug logger to report errors with detailed information helped speed up development time and allowed us to better perform integration testing when combining large sub-systems like the database and menus.