COMPENG 2SH4 Project – Peer Evaluation [25 Marks]

Your Team Members Kun Xing, Ibtisam Alhasoon

Team Members Evaluated Joshua Guo, Suchir Ladda

Provide your genuine and engineeringly verifiable feedback. Ungrounded claims will lead to deductions. Completing the peer code evaluation on time will earn your team a total of **25 marks**. Do not exceed 2 paragraphs per question.

Peer Code Review: OOD Quality

1. **[3 marks]** Examine the main logic in the main program loop. Can you easily interpret how the objects interact with each other in the program logic through the code? Comment on what you have observed, both positive and negative features.

   The main program loop effectively demonstrates object-oriented design, with clear and straightforward interactions between the `Player` and `GameMechs` objects. Each object has a well-defined role, making it easy to see how they work together. `GameMechs` handles tasks like managing the game state, setting up the board, and keeping track of the score, while `Player` focuses on movement, collision detection, and interacting with the board. This separation of responsibilities keeps the logic clean and easy to follow.
   One notable strength is how the `Player` object uses methods from `GameMechs`, such as `getInput` and `setExitTrue`, to handle interactions without duplicating data. The use of `objPos` and `objPosArrayList` enhances organization, particularly for tracking positions on the game board. The program loop is well-structured, and the object-oriented approach makes the logic easy to interpret and understand. It effectively showcases the principles of good OOD design.

2. **[3 marks]** Quickly summarize in point form the pros and cons of the C++ OOD approach in the project versus the C procedural design approach in PPA3.

   Pros of C++ OOD:

   - Encapsulation keeps the code modular and organized by grouping related data and methods into classes.
   - Classes are reusable and easy to extend, making the program scalable for future features.
   - Code is more readable, with clear roles for each class, like Player handling movement and GameMechs managing the game state.
   - Dynamic memory allows flexible data structures and efficient resource management.
   - Changes to specific parts of the program can be made without affecting other areas, improving maintainability.

   Cons of C++ OOD:

- The use of dynamic memory and class structures can add complexity and make debugging harder.
- Memory management requires caution to avoid leaks or dangling pointers.
- Concepts like constructors and destructors may be more difficult compared to simpler procedural code.

C++ OOD vs C Procedural Design:

- C's procedural approach is simpler and easier for smaller programs but lacks the structure and scalability of OOD.
- Without encapsulation, the code can become harder to manage as complexity increases.

Peer Code Review: Code Quality

1. **[3 marks]** Does the code offer sufficient comments, or deploys sufficient self-documenting coding style, to help you understand the code functionality more efficiently? If any shortcoming is observed, discuss how you would improve it.

   The code provides a good level of comments throughout, which helps in understanding its functionality. For instance, most functions are accompanied by a brief explanation of their purpose, and there are detailed comments within the GameMechs class to describe the logic behind board initialization and boundary setup. However, there are some areas where comments could be improved. For example, in the Player class, certain sections of the movePlayer function could use additional clarification, especially regarding the wraparound logic for the snake's movement. Additionally, the RunLogic function in the main file could benefit from more descriptive comments about how the individual components interact to achieve the game loop.

2. **[3 marks]** Does the code follow good indentation, add sensible white spaces, and deploys newline formatting for better readability? If any shortcoming is observed, discuss how you would improve it.

   Overall, the code demonstrates good indentation and reasonable use of whitespace, which enhances readability. Nested structures, such as loops and conditional statements, are well-aligned, making it easier to follow the flow of logic. The use of newline formatting, particularly in the game loop and board update functions, helps keep the code visually organized. That said, there are minor areas for improvement. For example, in the RunLogic function, the sequence of method calls appears slightly cluttered, and additional blank lines between logical sections would improve visual clarity. Similarly, the movePlayer function's switch-case structure could be spaced out slightly more to distinguish between cases.

Peer Code Review: Quick Functional Evaluation

1. **[3 marks]** Does the Snake Game offer smooth, bug-free playing experience? Document any buggy features and use your COMPENG 2SH4 programming knowledge to propose the possible root cause and

the potential debugging approaches you'd recommend the other team to deploy. (NOT a debugging report, just technical user feedback)

The Snake Game does offer a smooth, bug-free playing experience. When running the code, the wraparound logic around the border and food consumption logic were checked. When running the code, the snake was able to increase length by 1 when consuming normal food and implementing the bonus food without any issues. After consuming food, the newly generated food was able to generate at random spots within the game that was not occupied by the snake. We were able to play the game up to a score of over 20 without any issues. While there are no bugs game-wise, we would recommend adding a menu page describing how to play the game as that might be unclear to first time players without experience. In conclusion, the game aspect did not demonstrate any bugs as all features worked perfectly, however a menu page could be added to improve user experience.

2. **[3 marks]** Does the Snake Game cause memory leak? If yes, provide a digest of the memory profiling report and identify the possible root cause(s) of the memory leakage.

   The Snake Game does not cause memory leak, when running the memory profiling report, it indicates that there are 0 leaks for 0 total leaked bytes. While there was no memory leak, there are a few areas where issues could occur. In the destructors of Player and objPosArrayList, pointers are being deleted but not set to null afterwards, risking dangling pointers if accessed afterwards. Overall, the Snake Game does not cause memory but a few areas could use precautionary code to avoid any issues that could be caused in code extension.

Project Reflection

Recall the unusual objPos class design with the additional Pos struct. After reviewing the other team's implementation in addition to your own, reflect on the following questions:

1. **[3 marks]** Do you think the compound object design of objPos class is sensible? Why or why not?

   The compound object design of the objPos class with the additional Pos struct is sensible as it promotes modularity and code clarity. By grouping x and y into a single structure, operations on positions, such as comparisons or updates, are more intuitive and less error prone. For instance, the isPosEqual method directly compares two Pos struct, simplifying logic and improving readability. This approach allows future extensions, such as adding additional attributes like z for a 3D coordinate system, without significant changes to the class structure. This design aligns with principles of object-oriented programming by treating position as a cohesive unit, making it easier to extend the codebase. Additionally, the Pos struct improves the abstraction by encapsulating positional data, reducing redundancy when passing or storing coordinates. When passing parameters, instead of passing separate x and y values through multiple methods, a single Pos object can be passed, which makes the code cleaner and minimizes any chance of mix-ups between parameters. However, the use of dynamically allocated pointers (Pos* pos) introduces complexity and potential memory management risks, which may not be necessary in the case of 2D coordinates where the positions are simple data types. An additional complexity is when trying to access the Pos struct in other classes, which would require multiple calls to access the x and y data members. Overall, while there are some issues associated with additional complexity, we believe for this project of 2D coordinates,

the design is still sensible but not as much as if we were to extend it, where it would be much more sensible.

2. **[4 marks]** If yes, discuss about an alternative objPos class design that you believe is relatively counterintuitive than the one in this project. If not, explain how you'd improve the object design. <u>You are expected to facilitate the discussion with UML diagram(s).</u>

   Instead of having a separate struct Pos, the implementation would be to create two pointers on the heap, one pointing to the y coordinate and the other pointing to the y coordinate. For this assignment, this would reduce the complexity of accessing coordinate member variables in other classes but may not reduce the complexity if we were to add a third coordinate.

objPos
+int* x; +int* y; +char symbol;
+objPos() +objPos(xPos: int, yPos: int, sym: char) +setObjPos(o: objPos): void : : Rest of objPos methods

   This design uses x and y as variables within the objPos class itself instead of the original design which used a Pos struct outside of the objPos class.

   Another possible design is to set the Pos struct as a private data member and create getters and setters that get the x and y data members of the struct instead of the current design which only gets the objPos object.

objPos
−Pos* pos; +char symbol;
+objPos() +objPos(xPos: int, yPos: int, sym: char) +setObjPos(o: objPos): void : : Rest of objPos methods +getX(): int Xcoord; +getY() :int Ycoord;