COMPENG 2SH4 Project – Peer Evaluation [25 Marks]

Your Team Members ___Milana Kalinic 400503797___ _____Noor Azam 400503826_____

Team Members Evaluated ___bacopulj 400505130____ ___assafa2 400522187____

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.

Positive features:

- Distinct steps are clearly present- input handling, logic updates, drawing, and delay. This improves readability and interpretation of the code
- Comments increase code legibility, making it easy to understand each object's role (ie GameMechs manages game state (board dimensions, exit flags, scoring), Player manages player direction and movement, Food manages food generation and position)
- Object interactions are contained within their own classes, improving code organization
- Interpretation of object interaction is easy and intuitive. For example, is clear that mainGameMechs monitors the game's exit condition using mainGameMechs->getExitFlagStatus(), acting as the central controller for the game's state. DrawScreen() redraws the screen depending on positions of the player, food, and walls. Player's positions are retrieved using player->getPlayerPos() and drawn on the board, as well as Food positions via food->getFoodPos(). Player and food then dynamically interact through positional logic in the line if (i == player->getPlayerPos()->getElement(k).pos->y && j == player->getPlayerPos()->getElement(k).pos->x), as it is checked if a position matches either a player's position or a food's position.

Negative features:

- Nested loops in DrawScreen() may become inefficient for large boards because they check every position on the board for matches with Player or Food.
- Performance bottlenecks may arise from repeated calls to methods (ie player->getPlayerPos()->getElement()) because of the redundancy of computations
- Additional checks for successful memory allocation could be implemented to enhance error handling (ie could check if inputs are valid or if dynamic memory allocations succeed)

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.

Some pros of the C++ OOD approach in the project versus the C procedural approach in PPA3 are that classes have clear and intuitive responsibilities (ie Player, Food, GameMechs) which improves the maintainability of the code as well. An OOD approach also supports feature expansion, which is evident in the incorporation of special foods (advanced feature). The dynamic tracking of the snake's body and food through objPosArrayList simplifies the logic as opposed to having static arrays in a C procedural design as in PPA3. Some cons of a C++ OOD approach is that dynamic allocation such as in objPosArrayList increases the risk of memory leaks, requiring implementation of destructors or else memory leakage occurs. Additionally, implementing rule of 6 increases code complexity and length compared to procedural functions in PPA3 which are a lot simpler.

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.

Sufficient comments were provided throughout the code to support the reader's understanding of its functionality. They were provided to explain function methods when needed. For example, the line objPosArrayList *Player::getPlayerPos() const has a comment explaining that its purpose is to get the player's position list and return the reference to the playerPos array list through return playerPosList.

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.

Appropriate use of indentation, white spaces, and newline formatting were employed. For example, the switch cases within Player.cpp are appropriately indented and newline formatting/line break was used such that it was easily legible. Curly brackets were also entered on a new line from constructor/member function names, to improve legibility, as seen in the below screenshot. A line break is also utilized to separate game mechanic (input, exitFlag, loseFlag, and score) from size/dimension (boardSizeX, boardSizeY) member variables for improved legibility and organization.

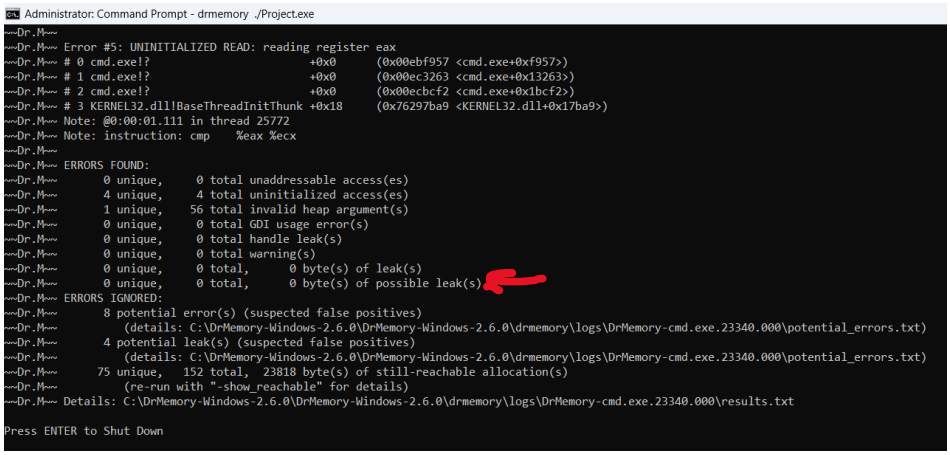
```cpp
GameMechs::GameMechs(int boardX, int boardY)
{
    input = 0;
    exitFlag = false;
    loseFlag = false;
    score = 0;

    boardSizeX = boardX;
    boardSizeY = boardY;
}
```

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 offers a smooth, bug-free playing experience. There were no noticeable bugs even after running the game multiple times, including when the game is exited by pressing escape. The score correctly tracks the collection of food by incrementing points when the snake consumes food. When I press escape mid-game, an exit message of "Press ENTER to Shut Down" pops up. When I lose the game by touching the snake, an exit message of "You lost! Your score was: x" with x being the player's final score pops up. The code executes smoothly as it should.

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.



As seen in the screenshot of a memory profiling above, the game causes zero leaks (0 unique, 0 total, 0 byte(s) of possible leak(s)).

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?

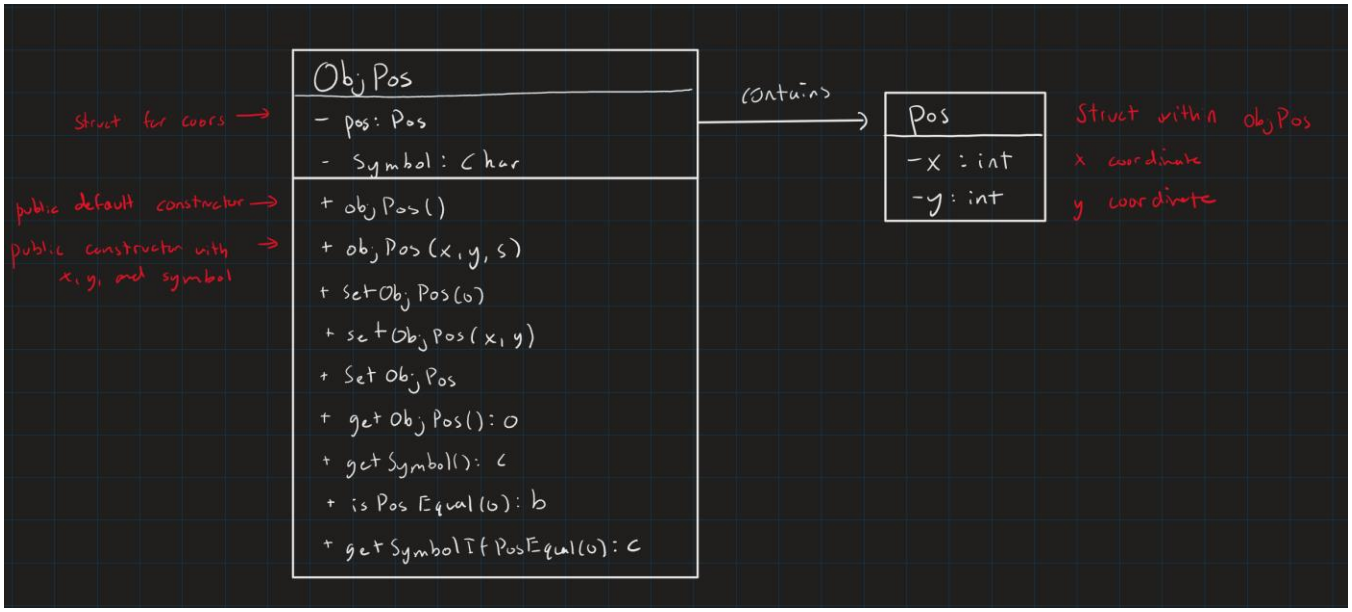
The design of the objPos class includes a Pos struct that is dynamically allocated on the heap. This design is not necessary and adds complexity that is not needed. Even though It makes the coding logic clearer by grouping the x and y coordinates, it can be coded without using dynamic memory allocation and be just as functional. Using memory allocation risks leakage and forces extra code; implementation of memory management using the rule of minimum four/six. For a simple class such as objPos, this design is inefficient as the extra computation required for memory allocation and deallocation on the heap outweigh the benefit of making it more organized.

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. You are expected to facilitate the discussion with UML diagram(s).

What would be more efficient than dynamically allocating Pos in the constructor using "new", we can use a struct acting as a member variable to avoid the potential issues mentioned above. It could be coded as follows:

```
class objPos {
    private:
        struct Pos {
        int x;
        int y;
        } pos;
        char symbol;
```

The UML Diagram would be:



The updated UML Diagram maintains clarity and improves efficiency by eliminating dynamic memory allocation while grouping the x and y coordinates within a private struct which ensures they can only be accessed through public getter methods. Using this method instead avoids any potential memory leakage and overhead while simplifying the process of programming since we do not have to account for memory management.