COMPENG 2SH4 Project – Peer Evaluation **[25 Marks]**

Your Team Members ahmedt52, Akrama7

Team Members Evaluated Bakry1, sheikhzs

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.

Overall, the main program loop's main logic was easy to interpret, and the objects were easy to follow. The best cases of the use of objects were the interactions between the gamemechanics and playercharacter objects. This can be seen in many places. In input detection, *gamemechanics->setInput(MacUILib_getChar())* got the input and *playercharacter->updatePlayerDir()* changed the direction based on this input. In food collision and generation, *gamemechanics->getFood()->generateFoodBucket()* generated the food on the board while using *playercharacter->getPlayerPos()* to avoid generation on the snake. Plus, *playercharacter->checkFoodCollision()* checks for collision, and updates game mechanics respectively. In addition the big program loop *while(!gamemechanics->getExitFlagStatus())* keeps the game running until *playercharacter->checkSelfCollision()* determines whether the lose condition needs to be updated.

The objects had proper uses alone as well. When looking at gamemechanics object, input detection was easily identifiable due to the use of MacUILib, along with the difficulty being set in run logic. When drawing the screen, the gamemechanics object is made of proper use through its respective functions, including board dimensions, game difficulty, food generation, score incrementation, and win conditions. When looking at the playercharacter object, it seems logical how the algorithm of the run logic function works, when looking at the update of the player direction, move player, and food collision functions respectively. Even in the draw screen, using the array list and indexing, it made sense how different parts of the snake were printed (head and tail). The only drawbacks are that there are some repeated parts of code. For example, in food rendering there was repetition due to use of many else if statements with repeated logic. Perhaps a helper function with switch case can be used. Similarly when looking at player collision, perhaps a helper function that renders the output symbol to be printed instead of *playercharacter->getPlayerPos()->getElement(j).symbol*.

----

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.

OOD

- Firstly, there was a lot of reusability of classes, mainly of objPos.

- In addition, having these individual classes meant that it was easy for more than one person to work on the entire project; one person can develop game mechanics and another can develop the player class.
- Plus, this approach makes it so that it is easier to develop the project further by adding more features, which can be done by modifying classes without drastically changing existing logic and code.
- One con was the simple fact that OOD is more complex, which make it initially difficult to understand interactions between objects.
- Adding on to this, debugging any errors is much harder especially with object interactions.

Procedural

- Firstly, procedural is generally more simpler, making it easy to follow the flow.
- Also, not as much memory is needed to allocate, as well as no class inheritance, increasing performance.
- There are many cons, one being that there is less reusability due to no class inheritance and polymorphism.
- Also there is no encapsulation, meaning that there may be unintended reference to variables.
- It is harder to collaborate with others in procedural.

<u>Peer Code Review: Code Quality</u>

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.

   Yes, the peer code offers sufficient comments at almost every single class and function. Although it does not go into the depths of each and every line of code, it does a very good job at showcasing the functions of bigger code snippets. To further improve this, I would break down and explain what some of the more complicated functions do line by line although that might cause issues with code readability.

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.

Yes, the code was very well organized and is easy to read. The functions are separated by good amounts of spaces (not too much and not too little) along with indentations for loops, classes etc. The comments are placed well and do not interfere with the readability of the code.

<u>Peer Code Review: Quick Functional Evaluation</u>

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)

For the most part the snake game is very smooth on my computer, although I do see a bit of lag when the snake gets big, and my score is high. This may be due to rendering issues and the delay settings which can be changed to achieve a smoother gameplay at higher scores. Apart from this there were no bugs at all in the game.

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.

   Upon running Dr. Memory using command prompt and Visual Studio Code, the program resulted in 0 bytes of memory leak.

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:

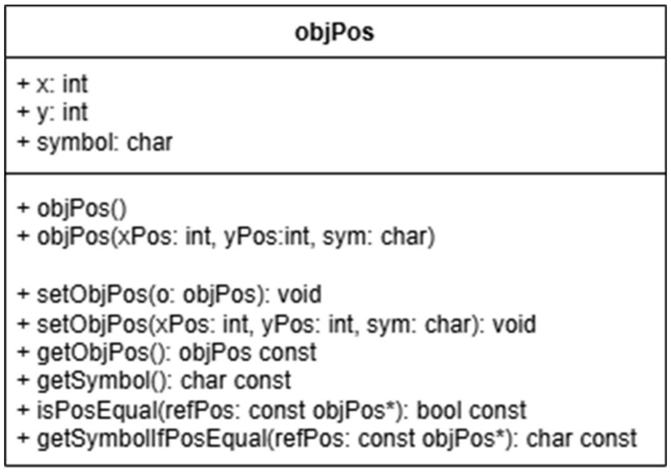
1. **[3 marks]** Do you think the compound object design of objPos class is sensible? Why or why not?

Yes, this compound object design of objPos is sensible, especially in the context of this project. Firstly, there is a pointer that makes reference to x and y position instance on the heap, which conserves memory. Also, this class is relatively simple having the position and symbol, which is encapsulated therefore self-contained. Adding on to this, it is easy to reuse in other parts of the program, as seen with drawing contents on the board.

Of course, having the pointer means taking in account for memory allocation, and may cause memory leaks or errors if not handled correctly. However, with proper practice, this can be avoided.

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).

Here is alternative design of objPos with the UML diagram. Essentially the struct does not require the encapsulation, and variables x, y, and symbol are members directly made public. There is no pointer pointing to the Pos.


objPos
+ x: int
+ y: int
+ symbol: char
+ objPos()
+ objPos(xPos: int, yPos:int, sym: char)
+ setObjPos(o: objPos): void
+ setObjPos(xPos: int, yPos: int, sym: char): void
+ getObjPos(): objPos const
+ getSymbol(): char const
+ isPosEqual(refPos: const objPos*): bool const
+ getSymbolIfPosEqual(refPos: const objPos*): char const

Perhaps the biggest issue which makes this counterintuitive is the fact that it violates the concept of encapsulation due to public data members. This makes getter functions redundant. Also, it would be more tedious to implement changes due to the absence of the Pos struct. Had there been the Pos struct, only that would be modified, but its absence would require more changes across the code. In addition, there is more room for error because of the chance of directly modifying x and y.