# COMPENG 2SH4 Project – Peer Evaluation

Your Team Members Ross Anderson anderr26, Nikolai Jagessar jagessan.

Team Members Evaluated Max F fangm16, Jacky N namj10.

Provide your genuine and engineeringly verifiable feedback. Ungrounded claims will lead to deductions.

## Part I: OOD Quality

1. **[6 marks]** OOD is about sensible code modularization. Looking at the header files of each object, can you easily interpret the possible behaviours of the objects involved in the program, and how they would interact with each other in the program? Comment on what you have observed, both positive and negative features.
    a. **GameMechs.h**
        i. **Positives**
            1. The inclusion of header guards (#ifndef, #define, and #endif) prevents multiple inclusions and potential compilation errors. This would ensure that the header file is only included once in the compilation process.
            2. Functions like getInput, getBoardSizeX, getBoardSizeY, etc., suggest what they do without needing extra comments, aiding in understanding the intended behaviour of these functions/methods.
        ii. **Negatives**
            1. The inclusion of "using namespace std;" in the header file is discouraged, especially within header files, as it can cause namespace pollution and conflicts when used in larger projects. It's not technically incorrect, but preferable to avoid namespace declarations at this scope.
            2. The declaration of the Food class within GameMechs.h / GameMechs.cpp is unprofessional and goes against the project criteria. While this would not prevent the program from running, it creates confusion for larger projects and is considered better practice to separate it into Food.h / Food.cpp files.
    b. **objPos.h**
        i. **Positives**
            1. The use of #ifndef, #define, and #endif ensures header guards are in place, preventing multiple inclusions and potential compilation issues. This practice ensures the header file is included only once during the compilation process, maintaining code integrity.
            2. The objPos class encapsulates data related to an object's position (x, y, symbol). Encapsulation helps in organizing related data and functions together, adhering to the principles of OOD.
            3. Functions like setObjPos, getObjPos, isPosEqual, etc., provide clear indications of their purpose and functionality, making it easier to understand the intended behavior of the objPos class.

ii. **Negatives**

1. The copy constructor objPos(objPos &o); should ideally take its parameter as const objPos &o to prevent accidental modification of the passed object. This ensures that the passed object remains unchanged while being used within the copy constructor.
2. Setter Functions: The usage of setter functions like setObjPos could be enhanced by following a more standard naming convention. For example, having a function like setObjPosition might improve clarity and consistency.
3. Implicit Default Constructor: While there is an explicit default constructor (objPos();), it might be beneficial to document its purpose or add comments explaining its default behaviour to avoid confusion or assumptions.

c. **player.h**

i. **Positives**

1. The Player class encapsulates player-related functionalities like position management and direction. This encapsulation contributes to modularity by isolating player behaviors within its own class.
2. The Player class constructor utilizes dependency injection by accepting references to GameMechs and Food. This allows the Player object to interact with these components without directly creating or managing them. Dependency injection promotes loose coupling and facilitates testing and reusability.
3. The enum Dir within the Player class allows clear representation of the player's direction state, facilitating control over the player's movement within the game.

ii. **Negatives**

1. While the class mainly focuses on player behaviors, you also incorporate checking map positions with hardcoded magic numbers for board width and height, violating OOP principles. Ensure that the Player class remains dedicated to managing player behaviors exclusively.
2. The movePlayer() method contains references to actions taken when certain food is eaten. This works, but also muddies the water and contains no additional comments to clarify that this feature is located here. Consider moving this functionality out of movePlayer() because checking what food is eaten isn’t “moving the player”.

2. **[6 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.
    a. **Positives**
        i. The main loop manages the game logic sequentially: getting input, updating game state, rendering the screen, and applying a delay. This sequence makes logical sense and thus enhances code readability and understanding.
        ii. The code demonstrates interaction between the Player object, Food object, and GameMechs object. For example:
            1. Player object interacts with GameMechs to get user input and update its position based on user commands.
            2. GameMechs manages game state, including tracking the score and determining game termination conditions.
            3. Food object generates food positions for the player.
        iii. The objects (Player, GameMechs, Food) seem to encapsulate their functionalities, abstracting away implementation details and providing a clear interface for interaction.
    b. **Negatives**
        i. Direct access to objects' internal states in the DrawScreen function, like accessing Player positions and Food positions, violates encapsulation. It could lead to issues if the internal structure of these objects changes in the future.
        ii. The conditional statements in DrawScreen lead to complex rendering logic. Simplifying and modularizing this logic by breaking it into smaller, more manageable functions or classes could improve code readability and maintainability.
        iii. Some commented-out code and portions like the debugging section are commented but not utilized. There's a lack of descriptive comments explaining the rationale behind certain decisions or the purpose of specific code blocks. Consider removing these entirely when creating a finished project.

3. **[5 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.

   The pros of C++ OOD compared to C procedural are as follows:

   - Increased Modularity – C++ allows for the encapsulation of functions within classes which increases the modularity of logical blocks.
   - Inheritance – This makes code reuse easier as a common class can be created then the features can be inherited by a sub class of the parent class making it easier to specialize functions.
   - Abstraction – This reduces the complexity of the main project file as most of the logic is hidden away in a class.

   However, there are some cons such as,

   - C procedural is simpler to grasp as all the code is clearly laid out in one file compared to the C++ approach.

## Part II: Code Quality

1. **[5 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.

- While the code deploys some level of self-documentation through the structure of the code files, explaining the main features of the program, the level of explanation of the logic under the 'hood' could be improved, where there is a lot going on with little to no explanation through comments.
- Additionally, there are sections in the main project file that could have been hidden from the final 'client' by including them in the game mechanics' class. One such example is the input processing that could have been hidden away within the getInput function of the GameMechs class instead of doing it in the main project file.
- Another thing that could be improved is breaking up functions into multiple sub-functions so that the intent of any given function is clear just from glancing at it. This would also make the debugging process a lot easier. One example of this is the collision logic with the player, which is wrapped up in the move player function. If a separate function were created for the collision logic that could be called in the move player function, its intention would be clear, and the code would be easier to read and understand.

2. **[4 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.

- The code generally followed a consistent indentation style through the multiple files.
- In addition, the use of white spaces around operator ensuring that they could be seen and not right after the text.
- Finally, the newline formatting which makes the code easy to read, as there is clear new lines between each logic block.

## Part III: Quick Functional Evaluation

1. **[8 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 a technical user feedback)

The game is almost bug free, but there are some small issues like unpredictable snake length growth when eating food.

Possible causes:

a. Inconsistent Body Segment Update:
   i. Root Cause: The logic responsible for updating the snake's body segments might be inconsistent. There might be cases where new body segments are added incorrectly to either the head or the tail as seen when playtesting.
   ii. Debugging Approach: Review the code responsible for adding new body segments. Check for any conditional statements or logic that might lead to unpredictable segment updates or accidental loops.

b. Variable or State Mismanagement:
   i. Root Cause: Mismanagement of variables or states related to the snake's growth could lead to unpredictable growth patterns.
   ii. Debugging Approach: Check how variables (like the aList number) related to the snake's size, growth, or movement direction are managed. Verify that they are correctly updated and handled throughout the game.

Recommendations for Debugging:

a. Implement extensive logging and tracing mechanisms in the code to track the state of the snake, its body segments, growth triggers, and position updates during gameplay. This helps identify the points where inconsistencies occur.

b. Use a debugger (like gdb!!!!) to step through the code, especially during the snake's growth or update processes. This allows for real-time inspection of variable values and code execution to pinpoint where the issue arises.

c. Collaborate with peers to perform a thorough code review, specifically focusing on the segments related to snake growth and body updates. Additional perspectives may identify overlooked bugs or inconsistencies.

d. Consider refactoring complex or convoluted logic related to the snake's growth into more manageable and structured functions. Simplifying code can often reveal hidden bugs and make the logic easier to debug.

2. **[6 marks]** Does the Snake Game cause memory leak?  If yes, provide a digest of the memory profiling report and identify the possible root cause of the memory leakage.

   The snake game does not cause a memory leak, as verified by running Dr. Memory. This is because there are correctly implemented destructors for all heap members used. I.e., every time the new keyword is called, an associated delete is called.

## Part IV: Your Own Collaboration Experience (Ungraded)

1. Tell us about your experience in your first collaborated software development through this project – what was working and what wasn't. If you are a one-person team, tell us what you think may work better if you had a second collaborator working with you.

   Overall, we both had a positive experience. However, there were some minor bumps along the way, which is inevitable in group projects. What was nice was the constant communication throughout the whole project, so even if one of us was working on one feature, the other partner could always just text the other and ask what was going on. As we both got familiar with each other's classes (like partner A vs. partner B responsibilities), we were able to help each other debug and troubleshoot issues the other overlooked, often very simple fixes. On the flip side, this flip-flop between partners A and B caused us to have to learn and thoroughly understand what the other was doing, which caused friction at times, especially when first trying to implement the game. Mechs/players/food classes together for the first time, but overall, the experience was enjoyable as it was a sandbox experience for the actual industry.