**Project Sprint #3**

The SOS game is described in CS449HomeworkOverview.docx. You should read the description very carefully.

Your submission must include the GitHub link to your project and you must ensure that the instructor has the proper access to your project. You will receive no points otherwise.

**GitHub link:**

Implement all the features that support a human player to play a simple or general SOS game against a human opponent and refactor your existing code if necessary. The minimum features include **choosing the game mode (simple or general), choosing the board size, setting up a new game, making a move (in a simple or general game),** and **determining if a simple or general game is over**. The following is a sample GUI layout. It is required to use a class hierarchy to deal with the common requirements of the Simple Game and the General Game. **If your code for Sprint 2 has not considered class hierarchy, it is time to refactor your code**.

|  |  |  |
| --- | --- | --- |
| SOS Icon  Description automatically generated Simple game Icon  Description automatically generated General game Board size  8 | | |
| Blue player  Icon  Description automatically generated S  Icon  Description automatically generated O | Chart, line chart  Description automatically generated | Red player  Icon  Description automatically generated S  Icon  Description automatically generated O |
|  | Current turn: blue (or red) | New Game |

Figure 1. Sample GUI layout of the working program for Sprint 3

**Deliverables: expand and improve your submission for sprint 2.**

1. **Demonstration (9 points)**

Submit a video of no more than five minutes, clearly demonstrating the following features.

1. A simple game that the blue player is the winner
2. A simple draw game with the same board size as (a)
3. A general game that the red player is the winner, and the board size is different from (a)
4. A general draw game with the same board size as (c)
5. Some automated unit tests for the simple game mode
6. Some automated unit tests for the general game mode

In the video, you must explain what is being demonstrated.

1. **Summary of Source Code (1 points)**

|  |  |  |
| --- | --- | --- |
| Source code file name | Production code or test code? | # lines of code |
|  |  |  |
|  |  |  |
| Total | |  |

**You must submit all source code via github to get any credit for this assignment.**

1. **Production Code vs User stories/Acceptance Criteria (3 points)**

Summarize how each of the user story/acceptance criteria is implemented in your production code (class name and method name etc.)

|  |  |
| --- | --- |
| **User Story ID** | **User Story Name** |
| 1 | Choose a board size |
| 2 | Choose the game mode of a chosen board |
| 3 | Start a new game of the chosen board size and game mode |
| 4 | Make a move in a simple game |
| 5 | A simple game is over |
| 6 | Make a move in a general game |
| 7 | A general game is over |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **User Story ID** | **AC ID** | **Class Name(s)** | **Method Name(s)** | **Status (complete or not)** | **Notes (optional)** |
| 1 | 1.1 |  |  |  |  |
|  | 1.2 |  |  |  |  |
|  | … |  |  |  |  |
| 2 | 2.1 |  |  |  |  |
|  | … |  |  |  |  |

1. **Tests vs User stories/Acceptance Criteria (2 points)**

Summarize how each of the user story/acceptance criteria is tested by your test code (class name and method name) or manually performed tests.

|  |  |
| --- | --- |
| **User Story ID** | **User Story Name** |
| 1 | Choose a board size |
| 2 | Choose the game mode of a chosen board |
| 3 | Start a new game of the chosen board size and game mode |
| 4 | Make a move in a simple game |
| 5 | A simple game is over |
| 6 | Make a move in a general game |
| 7 | A general game is over |

4.1 Automated tests directly corresponding to some acceptance criteria

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **User Story ID** | **Acceptance Criterion ID** | **Class Name (s) of the Test Code** | **Method Name(s) of the Test Code** | **Description of the Test Case (input & expected output)** |
| 1 | 1.1 |  |  |  |
|  | 1.2 |  |  |  |
|  | … |  |  |  |
| 2 | 2.1 |  |  |  |
|  | … |  |  |  |

4.2 Manual tests directly corresponding to some acceptance criteria

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **User Story ID** | **Acceptance Criterion ID** | **Test Case Input** | **Test Oracle (Expected Output)** | **Notes** |
| 1 | 1.1 |  |  |  |
|  | 1.2 |  |  |  |
|  | … |  |  |  |
| 2 | 2.1 |  |  |  |
|  | … |  |  |  |

4.3 Other automated or manual tests not corresponding to the acceptance criteria

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Number** | **Test Input** | **Expected Result** | **Class Name of the Test Code** | **Method Name of the Test Code** |
|  |  |  |  |  |
|  |  |  |  |  |

1. **(Part 1) Describe how the class hierarchy in your design deals with the common and different requirements of the Simple Game and the General Game.**

**At least 1/2 page, excluding screenshots/diagrams (12pt, single-spaced), is required.** **(3 Points)**

In this SOS game design, the class hierarchy is set up so that both the Simple and General game modes share common features while keeping their differences separate. The **SOSGame** class is the base, meaning both game modes inherit from it. This makes sense because no matter which mode you're playing, some things are always the same, like setting up the board, switching turns, and making sure moves are valid. Instead of writing the same code twice, this base class handles all the shared stuff, and SimpleSOS and GeneralSOS just build on top of it. The **SimpleSOS** class is pretty straightforward. It follows the rule that as soon as an SOS is formed, the game ends right away, and the winner is declared. It doesn’t care about who has more SOSs, just that someone made one first. On the other hand, **GeneralSOS** has a more competitive system where every SOS is counted, and the game only ends when the board is full. This means GeneralSOS needs extra tracking to keep up with the number of SOS formations for each player. So, while both classes rely on the same core game functions, GeneralSOS adds an **SOS counter** to keep track of scores, while SimpleSOS doesn’t. This structure makes it way easier to maintain and expand the game later. If a new mode were added, it could just extend SOSGame without breaking the existing ones. The whole design keeps things clean and avoids repeating code where it’s not necessary. Instead of cramming everything into one giant class, separating Simple and General into their own classes makes it way easier to manage the differences without messing up the parts that are the same.

**(Part 2) Demonstrate how you use LLM (ChatGPT or other) to analyze how well your code adheres to the design principles discussed in class - modularity, cohesion, coupling, and encapsulation – using the definitions provided in class. Provide screenshots of your interactions with the LLM, showing the prompts used and the responses received.**

**Interpret the LLM’s feedback, refine your prompts if needed to obtain more relevant responses, and discuss any changes made to your code based on the analysis. Explain how these refinements improve adherence to the design principles.**

**At least 1/2 page is required, excluding screenshots (12pt, single-spaced).** **(2 Points)**

I wanted to make sure my SOS game design actually followed good programming principles, so I used ChatGPT to analyze it. The main things I focused on were **modularity** and **cohesion. B**asically, making sure my code was well organized and that each part had a clear purpose. ChatGPT pointed out that my class hierarchy is solid. I’ve got a **base class (SOSGame)** that handles all the common stuff, and then two subclasses, **SimpleSOS** and **GeneralSOS**, that add their own rules. This makes the code more **reusable** and keeps things clean. It also follows **encapsulation** since each method takes care of a specific job, like switching turns or checking for an SOS. One issue, though, is that my makeMove() method is doing too much. It **validates moves, updates the board, checks for SOS formations, and switches turns** all in one go. That kinda breaks the. A fix for this would be breaking it into smaller methods inside **SOSGame**, so both game modes can just reuse them instead of repeating code.

