

### 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: [smyGitt/CS-449-Software-Engineering-Foundations](#)**

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.**

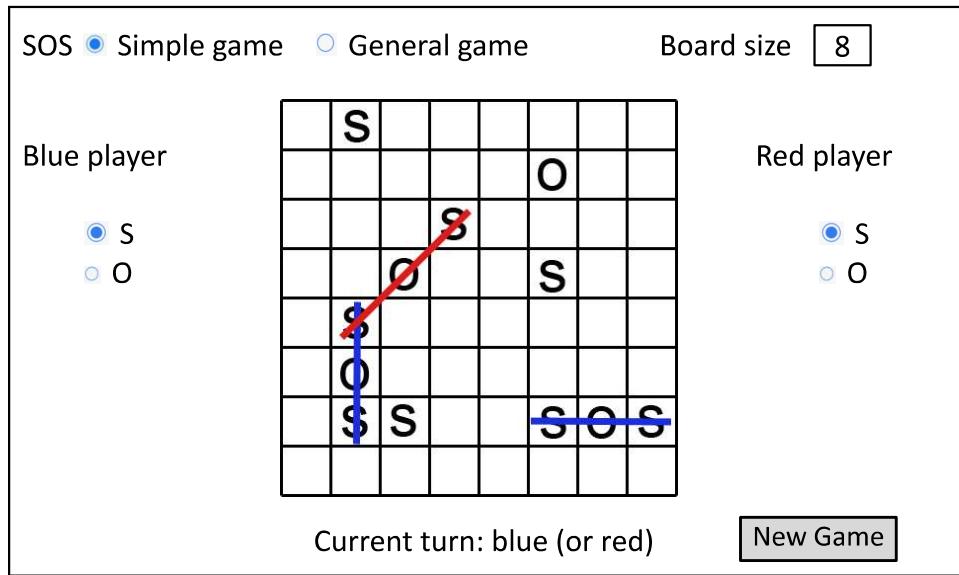


Figure 1. Sample GUI layout of the working program for Sprint 3

**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.

- (a) A simple game that the blue player is the winner

- (b) A simple draw game with the same board size as (a)
  - (c) A general game that the red player is the winner, and the board size is different from (a)
  - (d) A general draw game with the same board size as (c)
  - (e) Some automated unit tests for the simple game mode
  - (f) Some automated unit tests for the general game mode

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

VIDEO LINK: <https://youtu.be/ZHmkmoLHbBY>

## 2. Summary of Source Code (1 points)

Source code file name	Production code or test code?	# lines of code
main.py	production	250
game_logic.py	production	381
unit_testing	Test code	362

### 3. 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	MainApplication, Tile	MainApplication: __title_screen(), game_board()  Tile: set_button_instance()	inProgress	The board is able to change sizes, however requires starting a new game window.
	1.2	SOSGameLogic	dimension_validate()	Complete	-
	1.3				
2	2.1	MainApplication	title_screen()	Complete	-
	2.2				
3	3.1	MainApplication	title_screen() game_board() __player_tab() board_control_tab()	Complete	-
	3.2		title_screen(), game_board()	inProgress	The “Start!” button still opens a new window instead of using the current window.
4	4.1	SOSGameLogic	on_tile_click()	Complete	-
	4.2				
5	5.1	SOSGameLogic	on_tile_click() game_over() reset_state() __move_analysis()	Complete	There was an error with the US where one of the AC needed the “Competitive” / “General” game to end immediately after a single point.
	5.2				
	5.3				
6	6.1	SOSGameLogic	on_tile_click() switch_player()	Complete	-
	6.2		on_tile_click() move_analysis()		
	6.3		on_tile_click() update_button_state()		
7	7.1	SOSGameLogic	on_tile_click() game_over()	inProgress	No max score set yet.
	7.2				



#### 4. 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	TestFunctions	test_ac_5_1()	Input board size: 10, expected output: True
	1.2		test_ac_5_2()	Input board size: 2, expected output: False
	1.3		test_ac_5_3()	Input board size: 16, expected output: False
2	2.1	Manual test required.	-	-
	2.2		-	-
3	3.1		-	-
	3.2		-	-
4	4.1	TestFunctions	__test_ac_4_1()	Input: Empty cell on the board clicked, expected output: "S" cell in a player's owned tiles dictionary.
	4.2		__test_ac_4_2()	Input: Occupied cell on the board clicked, expected output: occupied button does not change, and turn also does not switch to other user.
5	5.1	TestFunctions	__test_ac_5_1()	Input: 'Casual' match, winning S-O-S move made, expected output: game ends immediately, and "Red Player Wins!" message shown.
	5.2		__test_ac_5_2()	Input: 'Competitive' match, S-O-S scoring move made, expected output: game continues and Red player score increases, and turn also does not switch to other user.
	5.3		__test_ac_5_3()	Final empty cell clicked with no S-O-S formed, expected output: game ends, and "The game is a draw" message shown.
6	6.1	Manual test required.	-	-
	6.2		-	-
	6.3		-	-
7	7.1	Not implemented.	-	-
	7.2		-	-

#### 4.2 Manual tests directly corresponding to some acceptance criteria

User Story ID	Acceptance Criterion ID	Test Case Input	Test Oracle (Expected Output)	Notes
2	2.1	'Casual' radio button clicked when 'Competitive' is active.	'Casual' button becomes active, and 'Competitive' button becomes inactive.	-
	2.2	'Competitive' radio button clicked when 'Casual' is active.	'Competitive' button becomes active, and 'Casual' button becomes inactive.	-
3	3.1	"Start!" button pressed with set size and mode	board is cleared, scores are 0:0, and turn indicator shows "Blue player's turn".	-
	3.2	"New Game" button pressed during an ongoing game, expected	board is cleared and scores reset, and new game starts with current settings.	-
6	6.1	Red player places a letter with no S-O-S formed	turn indicator switches to "Blue player's turn"	-
	6.2	Red player places 'O' completing a diagonal S-O-S	Red player's score increases, and turn also does not switch to other user.	-
	6.3	Occupied cell on the board clicked	occupied button does not change, and turn also does not switch to other user.	-
7	7.1	Not implemented	NOT implemented	-
	7.2			

5. (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)

The hierarchy in the code is “linear”, in that it does not extend any one class (outside of unittest class). How the code handles the different win conditions is through a variable that stores the currently chosen state of the game in Game Logic, which is then used in specific scenarios where differing game logic is required to separate the different logic. For example: point counting and win conditions are dependent on a if statement where a ‘casual’ game will send the game over signal when a single point is reached, whereas the ‘competitive’ (or general) game will continue to go on until the entire board has been filled, and the scores keep counting. The common logic is shared between them by default, as the game logic is a single instance created when the main file, which is the GUI file, is initialized. I found this to be a good solution for games with such similar logic except for a few critical areas, and even though I did not do any hierarchies, logically I see it as an equivalent of creating a child class with those functions with conditional statements based on chosen game mode being overridden by the child class’ own definition.

**(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)

Hello! As requested, I have analyzed your `main.py` and `game_logic.py` files based on the four design principles you provided.

Here is my analysis of how well your code adheres to these principles.

### 1. Modularity (High Adherence)

You have done an excellent job with modularity.

- **Strengths:** The code is split into two distinct modules, `main.py` and `game_logic.py`. Their responsibilities are clearly separated:
  - `main.py` handles all GUI presentation and user interaction (the "View" and "Controller").
  - `game_logic.py` handles all game rules, state, and data (the "Model").
- This separation makes the project much easier to understand and maintain. The dependency is also correct: `main.py` (the GUI) imports `game_logic.py` (the engine), but the logic module does not import the GUI module.

### 2. Cohesion (High Adherence)

Both modules demonstrate high cohesion.

- `game_logic.py`: All classes within this file (`SOSGameLogic`, `Player`, `Tile`) are directly related to the single purpose of defining and executing the rules of the SOS game.
- `main.py`: The `MainApplication` class is also highly cohesive. Its methods are all focused on the single purpose of building, displaying, and managing the Tkinter GUI components (title screen, game board, player tabs, etc.).

### 3. Coupling (Poor Adherence / High Coupling)

This is the most significant area for improvement. The modules are tightly coupled.

- **Problem:** Your "pure" logic module, `game_logic.py`, imports and directly uses the `tkinter` library.
- **Examples:**
  1. `SOSGameLogic` uses `tk.IntVar`, `tk.StringVar`, and `tk.BooleanVar` to store its own state (e.g., `game_board_dimension_variable`, `current_player_name_variable`).
  2. The `Player` class stores its score in a `tk.IntVar`.
  3. The `Tile` class holds a direct reference to its corresponding `tk.Button` instance.
  4. `SOSGameLogic` methods (like `__update_button_color`) directly call

- `tile.button_instance.config(...)`, manipulating GUI elements from within the logic module.
- Impact:** This tight coupling means your game logic is completely dependent on `tkinter`. You cannot reuse `game_logic.py` for a different GUI framework (like PyQt), a web application, or a simple console version without significant refactoring.

#### 4. Encapsulation (Weak Adherence)

The code's encapsulation is weak, as the modules reach directly into each other's internal attributes instead of using methods.

- Problem:** The `MainApplication` class in `main.py` repeatedly accesses attributes deep inside the `game_logic` object.
- Examples:**
  - `textvariable=self.game_logic.game_board_dimension_variable`
  - `variable=self.game_logic.config_do_ai_opponent`
  - `textvariable=self.game_logic.player_dict[player_number].score_variable`
  - `command=(lambda tile :`  
`self.game_logic.gameboard_tile_instance_dict[row_index][column_index]: ...)`
- Impact:** This breaks encapsulation because the `MainApplication` (the GUI) "knows" exactly how `SOSGameLogic` and `Player` store their data (e.g., as a `tk.IntVar` named `score_variable`). If you ever wanted to change how the score is stored in the `Player` class, you would also have to find and fix every part of the GUI that accesses it.

#### Summary and Recommendations

Your code has excellent modularity and cohesion. To improve coupling and encapsulation, I recommend the following:

- Decouple `game_logic.py`:** Remove all `import tkinter` references from `game_logic.py`.
  - Store state as plain Python types (e.g., `self.score = 0` instead of `tk.IntVar()`).
  - The `Tile` class should only store its state (e.g., `self.letter = "S"`), not a `tk.Button`.
- Strengthen Encapsulation:** Have `main.py` manage all `tkinter` variables.
  - When a player makes a move, `main.py` should call a simple method like `self.game_logic.make_move(row, col)`.
  - After the move, `main.py` should ask the logic for the new state using getter methods (e.g., `new_score = self.game_logic.get_player_score(1)`) and then update its own `tk.IntVar` for the score label.

This approach (often part of a Model-View-Controller pattern) will make your game engine truly independent and your overall design much more robust and maintainable.

While I will not make changes now, I do agree that I was a bit hesitant to include `tkinter` library in the `gamelogic` file, as it meant that the game logic is also handling some GUI elements. However, I could not figure out how to resolve this, and continue to develop the game with it still present. The LLM's suggestion is excellent, and what I was basically planning to do but didn't know the exact details. It also adheres to the design principals as it completely separates the `gamelogic` and `GUI`, making those independent from each other (reduces coupling) and therefore increases modularity, and also makes it easier to maintain. I also agree there

are some redundant code left over from the development process and I plan to remove all unnecessary code to simplify and make the code at least more readable and purposeful.