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: https://github.com/jsb58p/CS449---Software-Engineering-Project

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**.

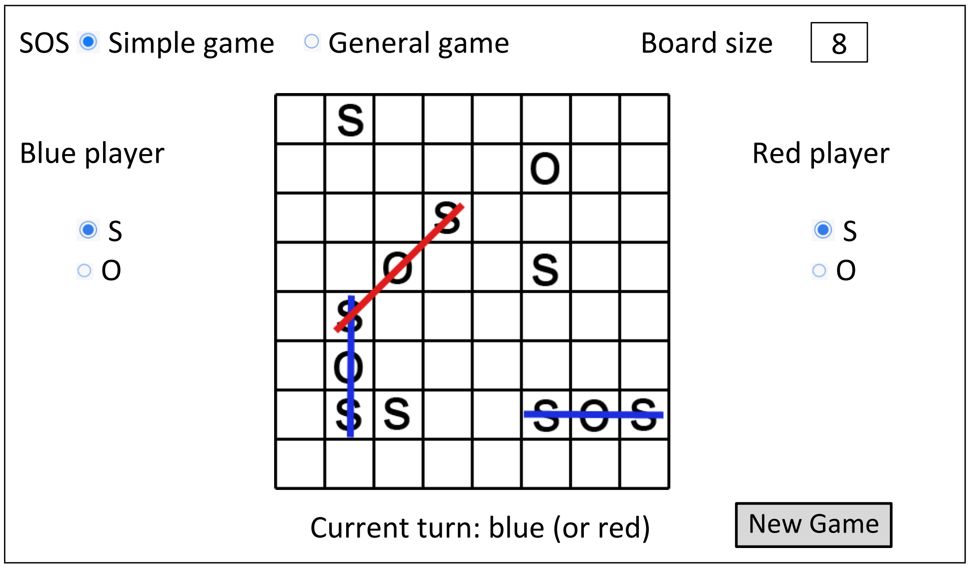

Figure 1. Sample GUI layout of the working program for Sprint 3

Deliverables: expand and improve your submission for sprint 2.
https://umsystem.hosted.panopto.com/Panopto/Pages/Viewer.aspx?id=96994e70-b8dc-497a-9a43-b38c002c78b1

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.

2. Summary of Source Code (1 points)

Source code file name	Production code or test code?	# lines of code
RadioButtonGroup.java	Production code	71
BoundaryCheck.java	Production code	72
GameModeLogic.java	Production code	89
SimpleMode.java	Production code	69
GeneralMode.java	Production code	65
SOSGameGUI.java	Production code	270
SOSGame.java	Production code	189
TestSOSGame.java	Test code	110
	Total	935

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

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 and Name	AC ID	Class Name(s)	Method Name(s)	Status (complete or not)	Notes (optional)
1 Choose a board size	1.1	SOSGame	SOSGame(int boardSize, GameMode gameMode SOSGameGUI gui) constructor, getBoardSize() setBoardSize(int boardSize)	Complete	Board size is set via constructor and can be set or retrieved.
		SOSGameGUI	start(Stage primaryStage), updateBoardDisplay()	Complete	Spinner in the GUI allows user to select board size 3-10.
2 Choose the game mode of a chosen board	2.1	SOSGame	GameMode enum (SIMPLE, GENERAL), SOSGame(int boardSize, GameMode gameMode, SOSGameGUI gui) constructor setGameMode(GameMode gameMode)	Complete	GameMode enum defines the two game modes. Constructor accepts and stores the selected mode.

		RadioButtonGroup	RadioButtonGroup(String label1, String label2, Boolean isVertical) constructor, getSelectedButton(), selectFirst()	Complete	Used to create radio buttons for “Simple game” and “General game” with toggle group to ensure only one is selected.
		SOSGameGUI	start(Stage primaryStage), getSelectedButton()	Complete	Instantiates RadioButtonGroup with game mode options. When “New Game” is clicked, passes current selection to SOSGame constructor.
3 Start a new game of the chosen board size and game mode	3.1	SOSGame	SOSGame(int boardSize, GameMode, SOSGameGUI gui) constructor, getBoard(), getBoardSize()	Complete	Constructor initializes the board array with empty cells determined by spinner value.
		SOSGameGUI	start(Stage primaryStage), updateBoardDisplay()	Complete	“New Game” button event handler uses board size and game mode selections, calls updateBoardDisplay() to show board.
	3.2	N/A	N/A	N/A	Acceptance criteria 3.2 is no longer relevant, board size and game mode is selected when application loads and cannot be unselected.
4 Make a move in a simple game	4.1	SOSGame	isCellEmpty(int row, int col), makeMove(int row, int col, char letter)	Complete	Validates that selected cell is empty before allowing move. Then update the board array with the chosen letter, ‘S’ or ‘O’.
		RadioButtonGroup	getSelectedButton()	Complete	Returns which radio button is selected (‘S’ or ‘O’) for the current player.
		SOSGameGUI	updateBoardDisplay()	Complete	When an empty vcell is clicked, updates button display with current player’s chosen letter ‘S’ or ‘O’ from RadioButtonGroup.
	4.2	SOSGame	getCurrentPlayer(), switchPlayer()	Complete	Track which player’s turn it is and switch after each valid move.
		SOSGameGUI	updateBoardDisplay()	Complete	Calls switchPlayer() and updates the “Current Turn” label.
5 A simple game is over	5.1	SOSGame.java	makeMove(int row, int col, char letter),	Complete	Determines in the final move that the

			isGameOver(), getWinner()		game is over and gets the winner.
		SOSGameGUI.java	endGameDisplay(Player playerColor, int blueScore, int redScore)	Complete	Sets the end-game display.
		GameModeLogic.java	isGameOver(), getWinner()	Complete	Abstract method declarations that subclasses implement.
		SimpleMode.java	isGameOver(), getWinner()		Determines if the game is over and calculates the winner.
6 Make a move in a general game	6.1	SOSGame	isCellEmpty(int row, int col)**,** makeMove(int row, int col, char letter)	Complete	Validates that selected cell is empty before allowing move. Then update the board array with the chosen letter, 'S' or 'O'.
		RadioButtonGroup	getSelectedButton()	Complete	Returns which radio button is selected ('S' or 'O') for the current player.
		SOSGameGUI	updateBoardDisplay()	Complete	When an empty vcell is clicked, updates button display with current player's chosen letter 'S' or 'O' from RadioButtonGroup.
	6.2	SOSGame	getCurrentPlayer()**,** switchPlayer()	Complete	Track which player's turn it is and switch after each valid move.
		SOSGameGUI	updateBoardDisplay()	Complete	Calls switchPlayer() and updates the "Current Turn" label.
7 A general game is over	7.1	SOSGame.java	makeMove(int row, int col, char letter SOSGame.Player player), isGameOver(), getWinner(),	Complete	Determines in the final move that the game is over and gets the winner.
		SOSGameGUI.java	endGameDisplay(Player playerColor, int blueScore, int redScore)	Complete	Sets end-game display.
		GameModeLogic.java	isGameOver(), getWinner().	Complete	Abstract method declarations that subclasses implement.
		GeneralMode.java	isGameOver(), getWinner()	Complete	Determines if the game is over and calculates the winner.

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 and Name	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 Choose a board size	1.1	TestSOSGame	testChooseBoardSize()	**Input:** Create SOSGame with size 5 and another with size 7. **Expected Output:** getBoardSize() returns 5 for first game and 7 for second game
2 Choose the game mode of a chosen board	2.1	TestSOSGame	testChooseGameMode()	**Input:** Create SOSGame with GameMode.SIMPLE and another with GameMode.GENERAL **Expected Output:** getGameMode() returns SIMPLE for first game and GENERAL for second game
3 Start a new game of the chosen board size and game mode	3.1	TestSOSGame	testStartNewGameState()	**Input:** Create new SOSGame with size 5 **Expected Output:** All board cells contain ‘ ‘ (empty), getCurrentPlayer() returns Player.BLUE
4 Make a move in a simple game	4.1	TestSOSGame	testMakeMoveInSimpleGame()	**Input:** Check if cell (1,1) is empty, make move at (1,1) with ‘S’. **Expected Output:** isCellEmpty(1,1) returns true before move, board[1][1] contains ‘S’ after move.
	4.2	TestSOSGame	testMakeMoveInSimpleGame()	**Input:** Make a move, then call swithcPlayer()

				Expected Output: getCurrentPlayer() changes from BLUE to RED after switch
5 A Simple game is over	5.1	TestSOSGame	testSimpleGameIsOver()	**Input: A** 5x5 game board with 'O' in every cell. **Expected Output:** isGameOver() returns true.
6 Make a move in a general game	6.1	TestSOSGame	testMakeMoveInGeneralGame()	**Input:** Make a move, then call switchPlayer() **Expected Output:** isCellEmpty(2,2) returns true before move, board[2][2] contains 'O' after move
	6.2	TestSOSGame	testMakeMoveInGeneralGame()	**Input:** Make a move, then call switchPlayer() **Expected Output:** getCurrentPlayer() changes from BLUE to RED after switch
7 A General game is over	7.1	TestSOSGame	testSimpleGameIsOver()	**Input:** A 7x7 game board where all but one cell are filled using makeMove() and the last cell is then set to 'S' **Expected Output:** isGameOver() returns true after the last cell is filled

4.2 Manual tests directly corresponding to some acceptance criteria

User Story ID and Name	**Acceptance Criterion ID**	**Test Case Input**	**Test Oracle (Expected Output)**	**Notes**
1 Choose a board size	1.1	Change spinner value to 8 and click "New Game"	8x8 grid displays with 64 empty cells	The grid displays as expected.
2 Choose the game mode of a chosen board	2.1	Select "General game" radio button and click "New Game"	Game starts in general mode (Shown in console output)	The console output verifies the chosen game mode is selected.
3 Start a new game of the chosen board size and game mode	3.1	Set board size to 5, select "Simple game", and click "New Game".	5x5 grid displays, "Current Turn: Blue Player" displays in blue text, console output shows game size and simple mode.	The grid displays as expected and the console outputs the game size and mode as expected.
4 Make a move in a simple game	4.1	Blue player selects 'S' radio button and clicks on empty cell at (0,0).	'S' appears in cell (0,0)	The selected letter appears in the correct cell.
	4.2	Blue player makes move.	"Current Turn: Red Player" displays in red text.	The Current Turn label changes as expected.

5 A simple game is over	5.1	In Simple 5x5 game: Blue plays S(0,0), Red plays 0(0,1), Blue plays S(0,2)	After Blue's second S, blue horizontal line through the placed letters, instruction label shows "Blue Player Wins!" in blue, board is disabled.	The lines are drawn, the board grid and instruction label updates as expected.
	5.2	In Simple 5x5 game, fill entire board with 'O' letters to avoid SOS pattern.	After all cells are filled, instruction label show "Draw" in black text, board is disabled, score is 0 for Red and Blue.	The board grid and instruction label updates as expected
6 Make a move in a general game	6.1	Red player selects 'O' and clicks empty cell at (2,3).	'O' appears in cell (2,3).	The selected letter appears in the correct cell.
	6.2	Red player makes move.	"Current Turn: Blue Player" displays in blue text.	The Current Turn label changes as expected.
7 A general game is over	7.1	In General 3x3 game, place 'S' until the board is filled..	After board is filled, the instruction label shows "Draw." And the board grid is disabled.	The instruction label shows "Draw." And the board grid is disabled as expected.

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**

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 class hierarchy in my design deals with the common and different requirements of Simple Game and General Game by implementing an abstract class GameModeLogic as discussed in class. The two subclasses, SimpleMode and GeneralMode, both handle the mode-specific requirements, while GameModeLogic handles the common elements. The method handleMove(int row, int col, char letter, SOSGame.Player player) performs the move sequence. It calls game.CheckScore(row, col, letter) to count how many SOS patterns the move created, storing the results in sosFormed. It then calls the private method updateScore(SOSGame.Player player, int points) which updates the appropriate player's score based on who made the move. It then evaluates shouldPlayerContinue(sosFormed) to determine if the current player gets another turn. In SimpleMode this will always return false since a single score ends the game. GeneralMode implements the functionality for a player to continue if they score an SOS. If the abstract method shouldPlayerContinue(sosFormed) returns false, it calls game.switchPlayer() to change turns.

The updateScore() method is private, ensuring that only handleMove() can modify scores. It simply assigns points to blueScore if the player is BLUE, or to redScore if the player is RED.

The isGameOver() method determines when the game should end. In SimpleMode, this occurs when a player has scored or the board is completely full. This handles the win conditions and draw conditions.

GeneralMode compares both final scores when the board is full. It determines based on redScore and blueScore, deciding whether there was a winner or whether the game was a draw.

(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)

SCREENSHOTS ON NEXT PAGES.

ChatGPT assessed that my code adheres to the principles discussed according to the definitions provided in class regarding modularity, cohesion, coupling, and encapsulation. It noted that each Java file defines a distinct, self-contained component; dependencies are limited and primarily directional, each class performs a single, well-defined responsibility, and that encapsulation is strong and properly maintained.

The assessment seems accurate to me, though I may be biased. I made sure to separate different responsibilities into different classes such as bounds checks, game modes, and radio button groups. For example, the BoundaryCheck class along with the Bound enums makes it easier to use the right boundary checks when placing a letter in a more readable way. The boundary checks can be implemented in SOSGame.java without having to remember or know the code that goes into them.

Another example is RadioButtonGroup. There are pairs of Radio Buttons that are needed in multiple aspects of the GUI. I created a class to be able to easily create new pairs of radio buttons as needed.

Each module addresses a distinct functional concern with no redundant overlap or cross-dependence. There are no cyclic dependencies in my code. There is low coupling in the code. Each module is made to represent a very specific functionality. All of the elements in them belong together. Access modifiers prevent direct external modifications of internal state. Implementation details are encapsulated within each module.

The initial analysis wasn't very in-depth, so I pressed ChatGPT for more detail. It made a table describing how all of the classes adhere to the principles discussed in class. The LLM identified strong modularity in regard to the game logic, mode-specific rules, and GUI. IT said that there was high cohesion, using examples such as BoundaryCheck, which exclusively handles array boundary validation, and RadioButtonGroup, which only manages pairs of radio buttons. It showed that coupling was mostly weak, and only moderate where necessary. It confirmed that encapsulation was strong through logic abstraction, minimal exposure of interfaces, and hidden implementation of SimpleMode and General mode with the GameModeLogic base type.

GeneralMode.java
File

RadioButtonGroup.java
File

SimpleMode.java
File

SOSGame.java
File

SOSGameGUI.java
File

BoundaryCheck.java
File

GameModeLogic.java
File

Modularity: the degree to which the components of a system are separated

Coupling: degree of interdependence between modules
 or the strength of the relationships between modules

Cohesion: the degree to which the elements inside a module belong together
◦ High (low) cohesion: the elements inside the module have a high (low) degree of connectedness
 Good design aims for high cohesion!
◦ Method: statements belong together (to one function)
◦ Class: public constructors & methods belong together
◦ Rule of thumb: a descriptive name

Encapsulation: bundling of data with the methods that operate on that data in one module
◦ Representing the essential features of something

Analyze how well your code adheres to the design principles above- modularity, cohesion, coupling, and encapsulation – using the definitions provided.

Only address these issues. Be objective. No random suggestions.

Analysis of adherence to design principles (based on provided definitions):

---

1. Modularity

Each Java file defines a distinct, self-contained component:

- `SOSGame` encapsulates overall game state and core logic.
- `GameModeLogic`, `SimpleMode`, and `GeneralMode` define behavior variations.
- `BoundaryCheck` handles positional validation.
- `RadioButtonGroup` and `SOSGameGUI` isolate user interface elements.

These modules are clearly separated by purpose and demonstrate **strong modularity**.

---

2. Coupling

Dependencies are limited and primarily directional:

- `GameModeLogic` depends on `SOSGame`, but not vice versa.
- `GeneralMode` and `SimpleMode` inherit from `GameModeLogic`.
- UI classes (`RadioButtonGroup`, `SOSGameGUI`) reference model classes without circular dependencies.

This reflects **low coupling** and appropriate dependency flow (model → GUI, not reversed).

---

3. Cohesion

- Each class performs a single, well-defined responsibility:
  - `BoundaryCheck`: validation of move legality.
  - `SOSGame`: management of board, turns, and game state.
  - `SimpleMode` / `GeneralMode`: implement specific rule sets.
  - `SOSGameGUI`: manages user interaction and layout.

Class methods relate closely to their respective class responsibilities, showing **high cohesion**.

---

4. Encapsulation

- Classes restrict data with `private` and `protected` fields (`blueScore`, `redScore`, `boardSize`).
- Interaction occurs through constructors and controlled public methods.
- Internal details (like boundary logic or scoring mechanisms) are not exposed outside the class interfaces.

Encapsulation is **strong** and properly maintained.

Overall assessment:

The system exhibits **high modularity**, **low coupling**, **high cohesion**, and **strong encapsulation** in full compliance with the provided definitions.

Go into more exact detail.

Detailed technical analysis of adherence to modularity, coupling, cohesion, and encapsulation
(Strictly objective evaluation based on the definitions provided.)

---

1. Modularity

Definition: Degree to which system components are separated.

Evaluation:

Class	Responsibility	Separation from Others	Assessment
`SOSGame`	Core logic: manages game state, board, players, and rules.	Does not depend on UI classes; only logic-related.	High modularity.
`GameModeLogic`	Abstract layer for mode-specific rule handling.	Isolated logic abstraction; no UI interaction.	High modularity.
`SimpleMode`	Implements "first SOS wins" rule.	Extends `GameModeLogic`; no external dependencies.	High modularity.
`GeneralMode`	Implements "most SOS wins" rule.	Same dependency structure as `SimpleMode`.	High modularity.
`BoundaryCheck`	Performs array boundary validation for SOS moves.	Independent utility; no reverse dependencies.	High modularity.
`RadioButtonGroup`	Manages two UI radio buttons and toggle group.	Confined to UI layer; no link to logic.	High modularity.
`SOSGameGUI`	Main UI container controlling layout, display, and interaction.	Separates presentation from logic classes.	High modularity.

Result:
Each module addresses a distinct functional concern with no redundant overlap or cross-dependence. Responsibilities are clearly separated.
→ Adherence: Strong

---

2. Coupling

Definition: Degree of interdependence between modules.

Evaluation:

Relationship	Direction	Type of Coupling	Assessment
`SimpleMode` → `GameModeLogic`	Inheritance	Weak (interface-level)	Acceptable
`GeneralMode` → `GameModeLogic`	Inheritance	Weak (interface-level)	Acceptable
`GameModeLogic` → `SOSGame`	Composition	Moderate; necessary for rule enforcement.	Acceptable
`BoundaryCheck` → `SOSGame.Bound`	Type reference only; no functional dependency.	Weak	Acceptable
`SOSGameGUI` → `SOSGame`, `GameModeLogic`	Reference for updating state or display.	Moderate; typical MVC coupling.	Acceptable
`RadioButtonGroup` → `JavaFX` library	Framework dependency.	Controlled external coupling.	Acceptable

Result:
No cyclic dependencies. No class exposes another's internal representation. Dependencies are directional (logic below, UI above).
→ Adherence: Strong (low coupling)

---

3. Cohesion

Definition: Degree to which elements inside a module belong together.

Evaluation:

Class	Internal Method/Field Relationship	Evidence of Cohesion	Assessment
`SOSGame`	Fields (`boardSize` , mode, turn tracking) and methods (play move, score tracking) all manage core game state.	High cohesion; all members serve a single purpose.	High
`GameModeLogic`	Fields (`blueScore` , `redScore` , `game`) and abstract/utility methods all relate to scoring and move logic.	High	High
`SimpleMode`	Overrides or extends scoring/win detection relevant to simple rule.	High	High
`GeneralMode`	Overrides or extends scoring/win detection relevant to general rule.	High	High
`BoundaryCheck`	Contains only logic ensuring valid move coordinates.	High	High
`RadioButtonGroup`	Contains two buttons and one toggle group; all relate to UI control grouping.	High	High
`SOSGameGUI`	All members deal with layout, event handling, and rendering.	High	High

Result:
Each method and field directly supports the class's stated function; no evidence of unrelated utilities or responsibilities mixed within.
→ Adherence: High cohesion throughout

---

4. Encapsulation

Definition: Bundling of data and methods operating on that data; hiding internal state.

Evaluation:

Class	Visibility Modifiers	Data Access	Encapsulation Enforcement
`SOSGame`	Most state fields private/protected; access through methods.	Exposes minimal interface.	Strong
`GameModeLogic`	Fields protected; accessible only to subclasses.	Abstracts logic; hides implementation details.	Strong
`SimpleMode` , `GeneralMode`	Use parent's protected interface; no public data exposure.	Implementation hidden behind base type.	Strong
`BoundaryCheck`	Fields private; encapsulated boundary logic.	No external field exposure.	Strong
`RadioButtonGroup`	UI components private; managed through internal getters/setters.	Encapsulation maintained.	Strong
`SOSGameGUI`	Keeps layout objects local/private; methods coordinate GUI behavior.	Encapsulation respected.	Strong

Result:
Access modifiers consistently prevent direct external modification of internal state. State manipulation occurs through controlled methods. No class exposes raw internal data structures.
→ Adherence: Strong encapsulation

Summary Table

Principle	Adherence	Supporting Evidence
Modularity	Strong	Classes cleanly separated by role (logic, rules, UI, validation).
Coupling	Low	Dependencies directional and minimal; no cycles.
Cohesion	High	Each class's members align with a single responsibility.
Encapsulation	Strong	Use of private/protected fields and restricted interfaces.

Overall conclusion:

The codebase adheres closely to all four design principles.

- **Modularity:** Each file represents a discrete subsystem.
- **Coupling:** Dependencies are minimal and properly directed.
- **Cohesion:** Each class's internal logic is purpose-specific and consistent.
- **Encapsulation:** Implementation details are well-contained within each module.