****

Laboratory Manual

for

SC1003

Introduction to Computational Thinking and Programming

Practical Exercise #6:

Procedure Abstraction (Battleship Game)

**COLLEGE OF COMPUTING AND DATA SCIENCE**

**NANYANG TECHNOLOGICAL UNIVERSITY**

# Learning Objectives

# Apply procedural abstraction to break down the Battleship game into smaller, reusable functions.

# Encapsulate tasks such as initializing the board, placing ships, and checking attacks into separate functions.

# Continue building on the code developed in Ex. 5 – Data Abstraction.

# Equipment

# PC/notebook with Python compiler (e.g., Python IDLE)

# 1. Introduction

# In Ex. 5, you used data abstraction with strings, tuples, lists, and dictionaries to store and organize game data. In this exercise, we will modularize the Battleship game using procedural abstraction by creating functions for:

# Initializing the game board

# Getting ship positions from the user

# Placing ships on the board

# Displaying the board

# Checking attack coordinates

# 2. Practical Task

# We will re-structure the existing Battleship code into smaller functions. Each function should perform one task and can be called from the main program.

# Hint Code with TODO Tasks

# BOARD\_SIZE = 10

# # Function 1: Initialize an empty board

# def init\_board():

# ## TODO: Return a BOARD\_SIZE x BOARD\_SIZE list of lists with all cells set to 0

# pass

# # Function 2: Get starting position for a ship

# def get\_ship\_position(ship\_name):

# ## TODO: Prompt the user for coordinates in "row,col" format

# ## Convert to tuple of integers and return

# pass

# # Function 3: Place ship on the board

# def place\_ship(board, start\_pos, length, symbol):

# ## TODO: Place the ship horizontally from start\_pos for 'length' cells

# pass

# # Function 4: Display the board

# def display\_board(board):

# ## TODO: Print each row of the board

# pass

# # Function 5: Check attack validity

# def check\_attack(board, row, col):

# ## TODO: Return True if coordinates are within range, else False

# pass

# # -----------------------

# # Main Program

# # -----------------------

# # Initialize board

# game\_board = init\_board()

# # Dictionary to store ships: name, length, symbol

# ships = {

# "Carrier": {"length": 5, "symbol": "C"},

# "Submarine": {"length": 3, "symbol": "S"}

# }

# # Get and place each ship

# for name, details in ships.items():

# start\_pos = get\_ship\_position(name)

# place\_ship(game\_board, start\_pos, details["length"], details["symbol"])

# # Display the final board with ships placed

# display\_board(game\_board)

# # Example attack check

# attack\_row = int(input("Enter attack row (0-9): "))

# attack\_col = int(input("Enter attack column (0-9): "))

# print("Valid attack?", check\_attack(game\_board, attack\_row, attack\_col))

# Sample Output

# Please enter start\_position of Carrier (row,col): 4,1

# Please enter start\_position of Submarine (row,col): 5,2

# Enter attack row (0-9): 6

# Enter attack column (0-9): 1

# Valid attack? True

# 0 0 0 0 0 0 0 0 0 0

# 0 0 0 0 0 0 0 0 0 0

# 0 0 0 0 0 0 0 0 0 0

# 0 0 0 0 0 0 0 0 0 0

# 0 C C C C C 0 0 0 0

# 0 0 S S S 0 0 0 0 0

# 0 1 0 0 0 0 0 0 0 0

# 0 0 0 0 0 0 0 0 0 0

# 0 0 0 0 0 0 0 0 0 0

# 0 0 0 0 0 0 0 0 0 0

**Optional Extension – Complete the Battleship Game**

*(For advanced students / after Lab 6)*

Once you have completed the procedural abstraction tasks in Ex. #7, extend your program to make it into a simple playable Battleship game.

**Additional Requirements**

1. **Refactor Attack Logic into Functions**
   * Create get\_attack() to prompt the user for a target (row, col) or quit.
   * Create apply\_attack(board, row, col) to:
     + Return "hit", "miss", "repeat", or "out\_of\_bounds".
     + Mark hits with "X", misses with "o".
2. **Track Ship Status**
   * Store each ship’s cells in a dictionary (or list of tuples).
   * After each hit, check if a ship is completely sunk. Print a message like "Carrier sunk!".
3. **Loop Until Game Ends**
   * Keep asking for attacks until all ships are sunk or the player quits.
   * After each turn, display the updated board.
4. **Optional Tie-Break Rule**
   * If there are multiple valid next moves in AI extension (if implemented later), choose based on a consistent rule (e.g., smallest row number first).