CST-250 Activity 2: Developing a Chessboard Application Using 2D Arrays and Multilayer Design

[Overview 2](#_Toc199410695)

[Part 1 – Board Class Library 5](#_Toc199410696)

[Class Library Challenges 23](#_Toc199410697)

[Part 2 – Console Application 25](#_Toc199410698)

[Console App Challenges 35](#_Toc199410699)

[Part 3 – Windows GUI Application 36](#_Toc199410700)

[GUI App Challenges 49](#_Toc199410701)

[Check for Understanding 50](#_Toc199410702)

[Deliverables 55](#_Toc199410703)

# Overview

This activity will build on the practices of separation of concerns to integrate data structures and user interfaces in a C# application.

**Learning Objectives**: The goals of this exercise are to:

1. Create an application that uses a 2D array game board.
2. Create methods that manipulate the squares according to game rules.
3. Implement the design practice of separation of backend logic from front-end interface code.
4. Apply abstraction and encapsulation.

**Details:** Create two versions of an app, one console and one GUI, to allow a user to see all the possible legal moves for a chess piece placed on a board. This will need three projects:

1. Chess Class Library – used in both the console and GUI applications.
   1. Board Model class – contains the 2D array composed of cells to represent the chessboard.
   2. Cell Model class – contains properties for each cell.
   3. Board Logic class – business access layer class for board methods.
2. Console App – text-based chessboard to see the next possible moves for each chess piece.
3. Windows Forms App – graphical user interface to perform the same functionality as the console app.

**The GCU College of Engineering and Technology Coding Guidelines and Best Practices for C#**, available in the Class Resources, must be followed in all CST-250 projects to ensure consistency, readability, and professionalism in student code. These guidelines promote a clear separation of concerns by organizing code into the Models, BusinessLogicLayer, and DataAccessLayer folders. The document also outlines essential standards for casing, naming conventions, code structure, commenting, and organization. Additionally, it emphasizes best practices in object-oriented design, exception handling, and the use of design patterns. Adhering to these expectations is required for successful completion of course assignments.

**Data Structure:** specialized format for organizing, processing, and storing data so it can be used efficiently. In programming, data structures are essential for managing and manipulating data logically and systematically. In this activity, data structures are introduced and used in the following ways:

**1. Two-Dimensional Arrays**

* **Used in**: BoardModel class
* **Details**:
  + A CellModel[,] Grid represents the chessboard.
  + This 2D array is a core data structure to map each cell on the board by row and column.
  + It models the physical structure of a chessboard and supports logic operations like movement checking.

**2. Classes as Custom Data Structures**

* **Classes used**:
  + CellModel – represents a single square on the board (contains Row, Column, PieceOccupyingCell, IsLegalNextMove).
  + BoardModel – contains the grid of CellModel objects and the board size.
  + These classes structure the data for each chess scenario and support encapsulation of attributes and behaviors.

**3. Dictionary**

* **Used in**: UpdateButtons() method in the GUI portion
* **Details**:
  + A Dictionary<string, string> called pieceMap is used to map chess piece abbreviations (e.g., "K", "Q") to their full names ("King", "Queen").
  + It enables quick key-based lookup, a hallmark of the dictionary data structure.

**4. Tuple**

* **Used in**: Utility.GetRowAndCol() method
* **Details**:
  + A Tuple<int, int> is returned to represent the selected row and column.
  + This temporarily stores and passes two related values (coordinates) efficiently.

**5. Array of Buttons**

* **Used in**: Button[,] \_buttons in the GUI project
* **Details**:
  + Mirrors the logical board (CellModel[,] Grid) in the visual interface.
  + Each button corresponds to a cell, allowing interactive UI logic based on board data.

# Part 1 – Board Class Library

**Objectives:**

* Create classes in C# that will be usable as a backend for the two versions of the chess application.
* Understand how abstraction is applied by exposing only the essential features of a class while hiding the unnecessary internal details.
* Understand how encapsulation helps in bundling data and methods that operate on that data, and restricting access to some components.

1. Create a new Class Library project in Visual Studio as shown in Figure 1.

A screenshot of a computer

Description automatically generated

Figure 1: Creating a new class library

1. Name the project ChessBoardClassLibrary as shown in Figure 2 and choose the most recent long-term support .NET framework.

A screenshot of a computer program

Description automatically generated

Figure 2: Create the Chess Board Class Library project



* Add the project to source control.
* See the Class Resources for a detailed tutorial on GitHub source control.

1. Delete Class1 by right-clicking the file and selecting Delete as shown in Figure 3.

A screenshot of a computer

Description automatically generated

Figure 3: Deleting Class1 from the solution explorer

1. Create two new folders in the project named Models and Services by right-clicking on the project as shown in Figure 4. Inside the Services folder, add a folder named BusinessLogicLayer. Figure 5 shows the solution explorer with the correct folder structure. This step adds abstraction by organizing the project in logical layers. By placing data definitions in the **Models** folder, business rules in the **BusinessLogicLayer**, and data handling code in the **DataAccessLayer**, you separate **what** each part of the application does from **how** it does it.

A screenshot of a computer

Description automatically generated

Figure 4: Adding a new folder to the ChessBoardClassLibrary

A screenshot of a computer

Description automatically generated

Figure 5: Solution Explorer with the new folder structure

1. We will create two classes to represent a chessboard: a CellModel class and a BoardModel class. First, we will create the CellModel class. It will contain a Row and a Column. Additionally, the CellModel class will have properties for PieceOccupyingCell and IsLegalNextMove to see if the CellModel currently has a chess piece on it and if the CellModel is a legal move for the current scenario. Create a new class in the Models folder and name it CellModel.cs as shown in Figure 6.

A screenshot of a computer

Description automatically generated

Figure 6: Creating the CellModel class

1. Update the access specifier of the class to public so classes outside of the project will be able to access the class, and add a citation to the top of the class as shown in Figure 7: Citation for CellModel class.

A computer screen shot of text

AI-generated content may be incorrect.

Figure 7: Citation for CellModel class

1. Then, add four properties to the class: an int Row, an int Column, a string PieceOccupyingCell, and a bool IsLegalNextMove as shown in Figure 8. Apply encapsulation to the Row and Column properties. Specifically, set their setters to private so that only the CellModel class itself can modify these values. This protects the internal state of the object and prevents unintended changes from outside the class.

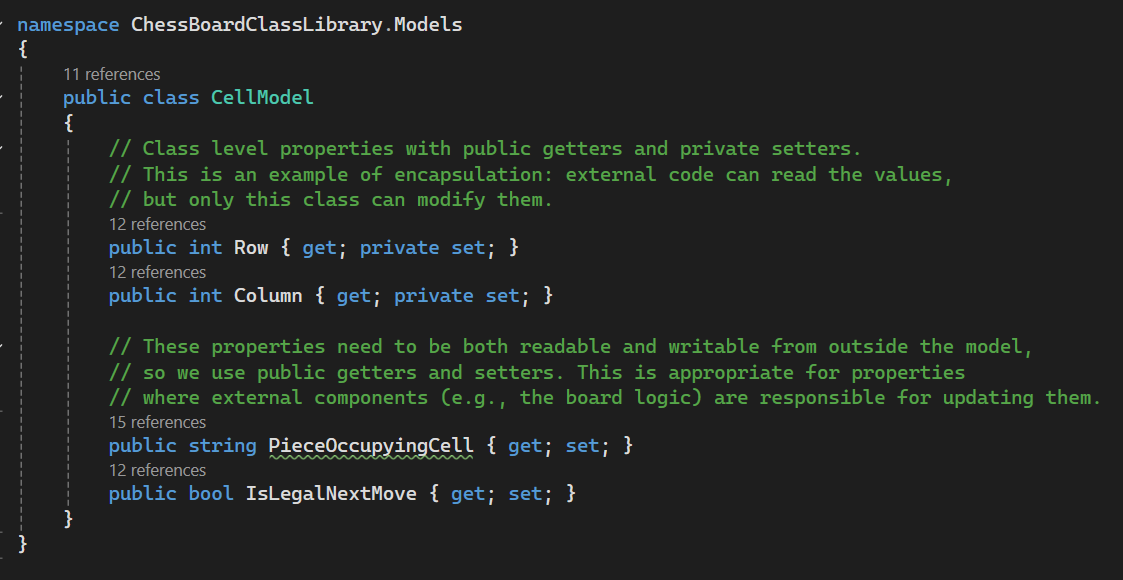


Figure 8: Properties for CellModel

1. Create a parameterized constructor to initialize the properties. We will only include the row and column of the cell in the constructor. You can right-click inside the class and select "Quick Actions and Refactoring" > "Generate Constructor…" to have Visual Studio create the constructor for you. Make sure the only select Row and Column when generating the parameterized constructor. Add default values for PieceOccupyingCell and IsLegalNextMove as well as a summary comment to the constructor so it mirrors Figure 9.

A screen shot of a computer code

Description automatically generated

Figure 9: Parameterized Constructor for CellModel class

1. Next, create a second class within the Models folder named BoardModel.cs and change the access specifier to public as shown in Figure 10. Add a citation to the top of the class.

A computer screen with text

AI-generated content may be incorrect.

Figure 10: Creating the BoardModel class with a citation

1. The BoardModel class will contain two properties: an int Size and a 2D array of CellModels named Grid as shown in Figure 11. The size property will dictate the size of the board, which will be square. The grid property will represent the chessboard, with every space containing a CellModel. The 2D array acts as a list of lists with each CellModel having a specific address. To access a CellModel, you must specify two indexes, the array, and the element inside the array. However, it is generally easier to think of the indexes as a row and a column for each CellModel. Apply encapsulation to both properties. Specifically, set their setters to private so that only the BoardModel class itself can modify these values. This protects the internal state of the object and prevents unintended changes from outside the class.

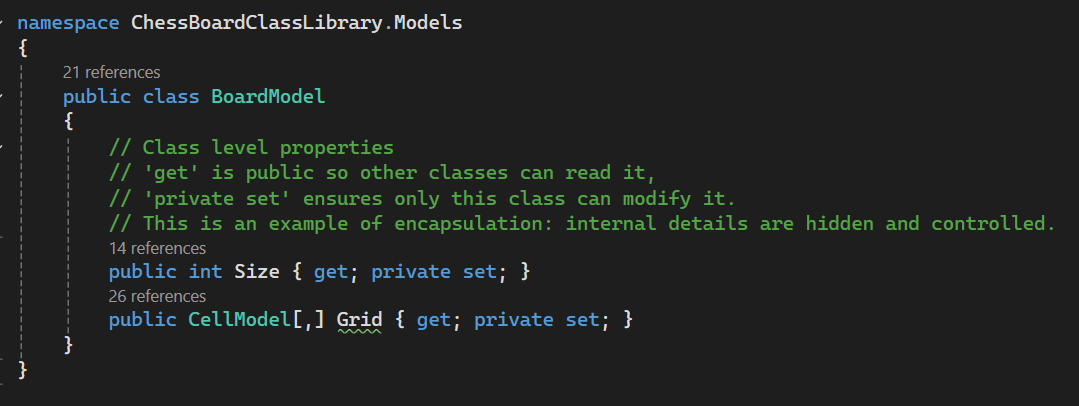


Figure 11: Class level properties for the board model class

1. The BoardModel class will have a parameterized constructor that takes the size property for the board. We will need to initialize the board as well as set each index of the 2D array to a new CellModel object. We will do this using two nested for loops, one for the rows and one for the columns. Also remember to add a summary comment to the constructor as shown in Figure 12.

A screenshot of a computer program

AI-generated content may be incorrect.

Figure 12: Parameterized Constructor for board model class

1. Since the for loops for initializing the board cause the parameterized constructor to be messy, we will refactor the BoardModel class so that the loops are in a separate method. We can then call that method within the constructor as shown in Figure 13.

Encapsulation is implemented by defining access modifiers in the class member as private. The internal details of how a BoardModel grid is initialized (like using nested loops) are hidden inside this private helper method.

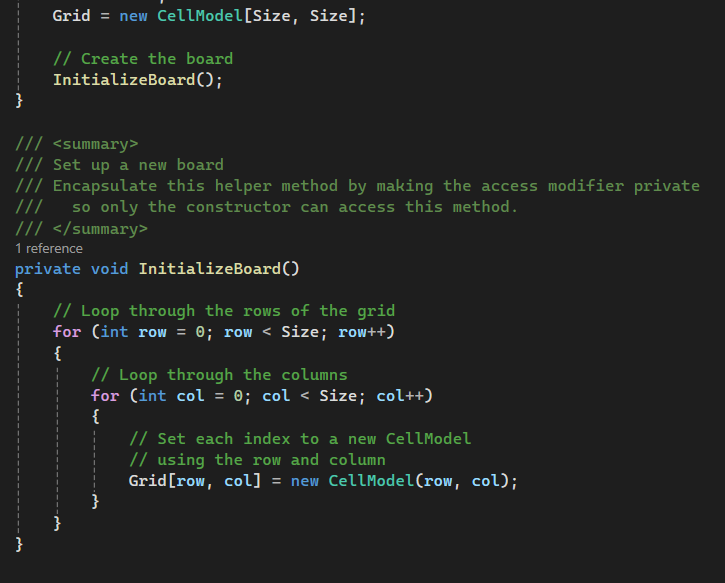


Figure 13: Cleaning up the board model constructor and encapsulating InitializeBoard



* Commit all changes in the project to source control.
* Commit Message: Create the CellModel and BoardModel classes.

1. Now that we have completed both of our models, we need to create the business logic layer class for our project. In the BusinessLogicLayer folder we created earlier, add a new class named BoardLogic.cs and change the access specifier to public as shown in Figure 14. Remember to add a citation to the top of the class.

A computer screen shot of a program

AI-generated content may be incorrect.

Figure 14: Creating the board logic class

1. This class will contain three methods that we will use for chessboard logic in the console and GUI applications. Since the BoardModel object will be created and held in the presentation layer, each method will need to take the BoardModel object as a parameter and return it back to the presentation layer at the end of the method. First, create a method to reset the BoardModel by setting the IsLegalNextMove and PieceOccupyingCell properties back to their defaults. Instead of using the nested for loops we used while initializing the board, we will use a foreach loop that will allow us to iterate over each CellModel in the board parameter as shown in Figure 15.

Encapsulate the ResetBoard method to hide internal mechanics and expose only the necessary functionality to the user.

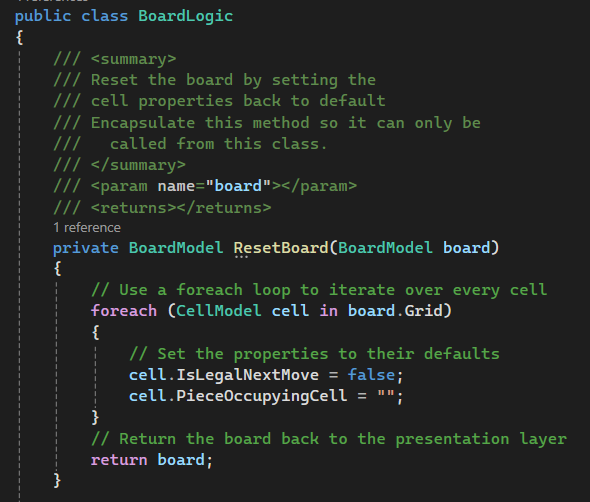


Figure 15: Method to reset the board for the next chess move

1. Next, we will create a method to check if a row/column location is on the grid. We will need this method to ensure we do not identify locations off the grid for the piece to move to. Since this method does not update the BoardModel, we do not need to return the board model. Instead, we can return a Boolean to say if the location is on the grid or not. The method will check the row and column individually, then combine the result to return as shown in Figure 16.

Encapsulate IsOnBoard method by making it private, as it is only used within this class and does not need to be exposed publicly.

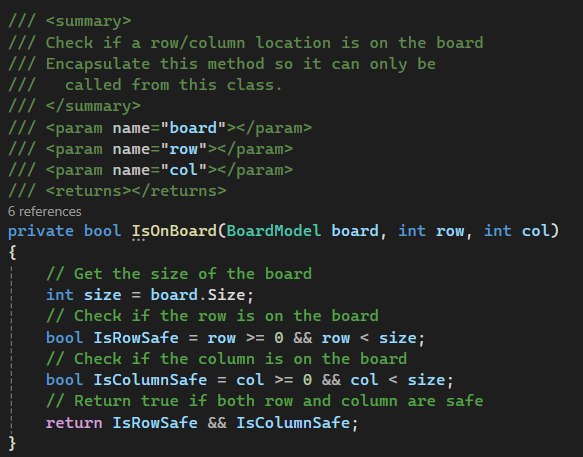


Figure 16: Method to check if a location is on the board



* Commit all changes in the project to source control.
* Commit Message: Add ResetBoard and IsOnBoard methods to BoardLogic class.

1. The third method we will need to create will mark the legal moves of a chess piece based on its location on the board. To do this, we will pass in the BoardModel, the CellModel occupied, and a string for the current chess piece. First, we will set up the method and add a summary and end-of-method comment as shown in Figure 17.

A screen shot of a computer program

Description automatically generated

Figure 17: Setting up the mark legal moves method

1. Next, we can use a switch statement based on the chess piece to determine which cells are legal next moves. We can use the ToLower method so that casing does not matter for the piece name. First, reset the BoardModel and set up the switch statement with different chess pieces as shown in Figure 18.

A screenshot of a computer program

AI-generated content may be incorrect.

Figure 18: Setting up the switch statement for MarkLegalMoves

1. In chess, the knight moves in an L pattern of two squares in any direction, then one square perpendicularly, as shown in Figure 19. We could check each of the eight moves individually, but that would be a lot of very similar code that would become hard to debug in the future. Instead, we will set up two 1D arrays of integers that will contain the possible row and column moves for the knight. Using the list of moves, we can use a for loop to check all eight moves as shown in Figure 20. Additionally, we need to set the PieceOccupyingCell property for the current cell to "N" for knight ("K" will be used for the king).

A chess board with a horse head

Description automatically generated

Figure 19: Chess Knight Moves

A screen shot of a computer program

Description automatically generated

Figure 20: Switch case for the knight piece

1. As the moves for the other pieces are added, this method will begin to look very crowded. To fix this, we can move the logic for each piece into its own method. We can create a method called MarkValidKnightMoves to encapsulate the knight's movement. This method will need to have the current CellModel and BoardModel passed in and will return the BoardModel. Since nothing outside of this class will need to access these methods, we can set the access specifier to private. Then, move the two arrays and the for loop into the method, and add a return statement for the board parameter. Your new method should mirror Figure 21.

Encapsulation is implemented by defining access modifiers in the class members.

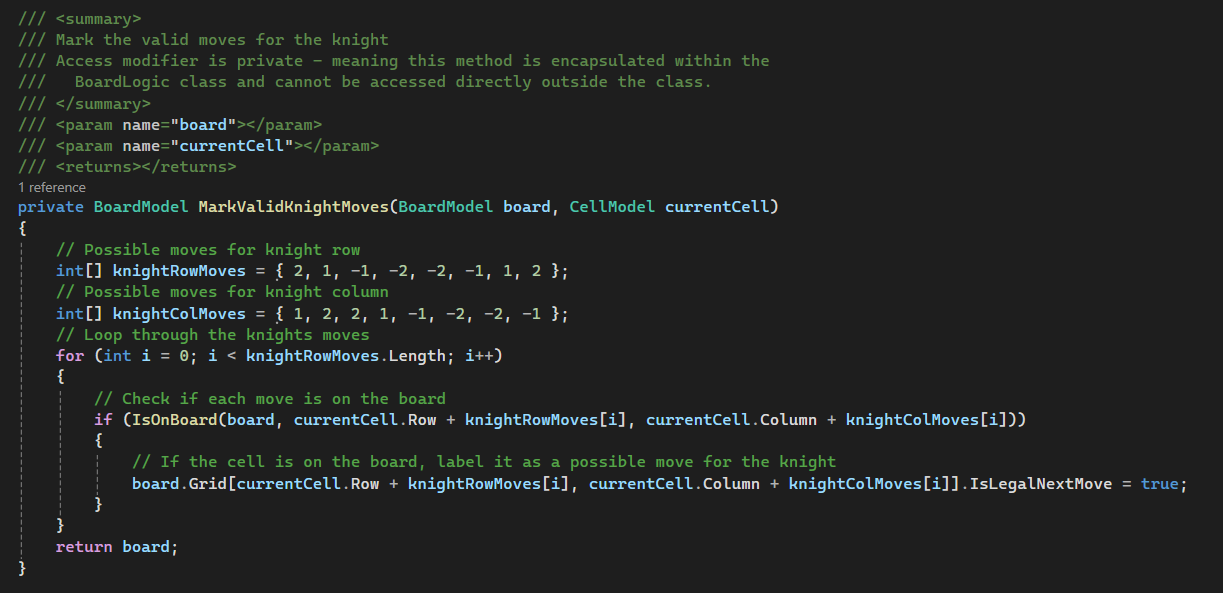


Figure 21: Moving the knight logic to a separate method and encapsulating it

1. Then, we can move back to our switch statement to call our new method. Remember to assign the board variable to the return from MarkValidKnightMoves as shown in Figure 22.

A screen shot of a computer code

Description automatically generated

Figure 22: Calling the MarkValidKnightMoves method



* Commit all changes in the project to source control.
* Commit Message: Add logic to mark knight moves as valid.



* Take a screenshot of the citations for CellModel, BoardModel, and BoardLogic. Make sure to include the class declarations in the images.
* Take a screenshot of the CellModel and BoardModel constructors and InitializeBoard method in the BoardModel class.
* Take a screenshot of the ResetBoard, IsOnBoard, MarkLegalMoves, and MarkValidKnightMoves methods in BoardLogic.
* Take a screenshot of the UML for the CellModel, BoardModel, and BoardLogic classes.
* Paste the images into a Word document.
* Put a caption below each image explaining what is being demonstrated.

## Class Library Challenges

Add the following features to the program:

1. Finish the rook case for in switch statement in its own method. The rook can move vertically or horizontally on the board as shown in Figure 23.

A black and white checkerboard with a chess piece in the middle

Description automatically generated

Figure 23: Chess Rook Moves

1. Finish the bishop case for in switch statement in its own method. The bishop can move diagonally on the board as shown in Figure 24.

A black and white checkerboard with a penguin in the center

Description automatically generated

Figure 24: Chess Bishop Moves

1. Finish the queen case for in switch statement in its own method. The queen has the combined moves of the rook and the bishop. She can move vertically, horizontally, or diagonally on the board as shown in Figure 25.

A game board with a checkerboard and a crown

Description automatically generated

Figure 25: Chess Queen Moves

1. Finish the king case for in switch statement in its own method. The king can move one square in any direction on the board as shown in Figure 26.

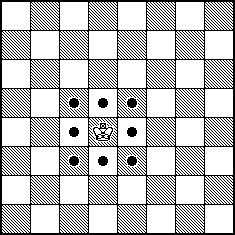


Figure 26: Chess King Moves



* Take a screenshot of each method added or updated to implement the features.
* Take a screenshot of the updated UMLs for any classes with added methods.
* Paste the images into a Word document.
* Put a caption below each image explaining what is being demonstrated.



* Commit the challenges to source control with a commit message

# Part 2 – Console Application

In this section, we will create a console text-only interface to play the game. The user will be asked for the piece as well as the row and column coordinates for the location of the chess piece. The program will then print a board with all the possible destinations for that piece.

1. Right-click on the solution and add a new project. Create a new Console App project named ChessBoardConsoleApp as shown in Figure 27.

A screenshot of a computer

Description automatically generated

Figure 27: Creating a new console app

1. In the Additional Information page, unselect "Do not use top-level statements" as shown in Figure 28. This means the main method will not be explicitly defined in the program. Figure 29 shows what the solution explorer should look like after the new project is created.

A screenshot of a computer

Description automatically generated

Figure 28: Uncheck "Do not use top-level statements"

A screenshot of a computer program

Description automatically generated

Figure 29: Solution explorer with two projects

1. Because the class library has no main method, it is not a runnable project and will error if you attempt to run it. So, we need to set the startup project to the console app. Right-click the ChessBoardConsoleApp and select "Set as Startup Project" as shown in Figure 30.

A screenshot of a computer program

Description automatically generated

Figure 30: Setting a new startup project

1. Now, if we run the program, we should see the "Hello, World!" message that is in our Program.cs file as shown in Figure 31.

A screenshot of a computer screen

Description automatically generated

Figure 31: Console running after startup project is configured

1. To access the class library classes and methods we just wrong, we will need a reference to the class library. Right-click on ChessBoardConsoleApp > Add > Project Reference. In the reference manager, check the ChessBoardClassLibrary project as shown in Figure 32 and select OK.

A screenshot of a computer

Description automatically generated

Figure 32: Adding a project reference to ChessBoardClassLibrary



* Commit all changes in the project to source control.
* Commit Message: Add a console app to the class library.

1. Next, go to your Program.cs file. Delete the contents and add comments to delimit where your main method code will go as shown in Figure 33. You can also add a welcome message for the user and a citation.

A screenshot of a computer program

AI-generated content may be incorrect.

Figure 33: Comments to mark the main method for Program.cs and a citation

1. Instead of putting functions directly into the Program.cs file. we should create a utility class to contain methods for our console application. Create another delimiting comment as well as a declaration for the public, static Utility class as shown in Figure 34.

A screenshot of a computer program

AI-generated content may be incorrect.

Figure 34: Creating a utility class for the console application

1. Declare an internal, static PrintBoard method with a summary comment and end of method comment inside the Utility class as shown in Figure 35. By declaring the method as static, it means that the method belongs to the class instead of an object instance. This allows us to call the method using Utility.PrintBoard instead of needing to create an instance of Utility.

A screen shot of a computer program

AI-generated content may be incorrect.

Figure 35: PrintBoard method in the Utility class

1. Add two nested for loops to the print board method to iterate over the board parameter. Then, add a series of if-statements to print out the BoardModel as shown in Figure 36. To move to the next line after every row of symbols, we can use the Console.WriteLine method to add a return character.

A screen shot of a computer program

AI-generated content may be incorrect.

Figure 36: Completed print board method in the Utility class



* Commit all changes in the project to source control.
* Commit Message: Add a Utility class for utility methods and a PrintBoard method for the console app.

1. Next, we can go back up to our main method and add guiding comments for what we want our program to do. Your main method should mirror Figure 37 after the comments have been added. These comments will template out the rest of our Program.cs class.

A screenshot of a computer program

AI-generated content may be incorrect.

Figure 37: Planning comments for the main method

1. Create the new board to have a size of 8, which is a standard chessboard. Then, use the Utility class to print the empty board as shown in Figure 38.

A screen shot of a computer code

AI-generated content may be incorrect.

Figure 38: Printing an empty board



* Take a screenshot of the application running at this point.
* Paste the image into a Microsoft Word document.
* Put a caption below the image explaining what is being demonstrated.

1. Prompt the user to choose the piece they are placing using a Console.ReadLine as shown in Figure 39. We can declare and initialize the piece variable at the top of our Main method.

A computer screen shot of a game

AI-generated content may be incorrect.

Figure 39: Add logic for user to select piece

1. Back in the Utility class, create a new internal method, GetRowAndCol, that will prompt the user for the row and column for their piece. This method will return two integer values using a tuple object as shown in Figure 40. The tuple allows us to combine two different objects into a single variable that we can return and use.

A computer screen with colorful text

AI-generated content may be incorrect.

Figure 40: Using a tuple to get the row and column from the user

1. Declare the tuple at the top to receive the data from the method as shown in Figure 41. Next, call the new method from the main method as shown in Figure 42.

A screenshot of a computer program

AI-generated content may be incorrect.

Figure 41: Declaring a Tuple

A screen shot of a computer screen

AI-generated content may be incorrect.

Figure 42: Calling the GetRowAndCol method



* Commit all changes in the project to source control.
* Commit Message: Add user input from chess pieces to the console app.

1. Now that we have the piece and location, we should create an instance of the BoardLogic class as shown in Figure 43 so we can mark the legal moves.

A screen shot of a computer code

AI-generated content may be incorrect.

Figure 43: Declare and Initialize the board logic class

1. Call the MarkLegalMoves method in boardLogic using the user's input as shown in Figure 44. Save the result back to the board variable

A computer screen shot of text

AI-generated content may be incorrect.

Figure 44: Marking the legal moves using the user input

1. Lastly, print the board out again with the new moves marked as shown in Figure 45.

A screen shot of a computer

AI-generated content may be incorrect.

Figure 45: Reprinting the board for the user



* Commit all changes in the project to source control.
* Commit Message: Finish functionality to mark the legal moves and show the board.



* Take a screenshot of the application running at this point with a queen placed at 1, 1.
* Take a screenshot of the citations for Program.cs.
* Take a screenshot of the PrintBoard and GetRowAndCol methods in the Utility class.
* Take a screenshot of the UMLs for the Utility class.
* Paste the images into a Word document.
* Put a caption below each image explaining what is being demonstrated.

## Console App Challenges

The program works but is not yet finished. Here are some things you need to complete:

1. **Fix Out-of-Bounds Errors** – Make sure that the user does not enter a number that is smaller or larger than the size of the grid.
2. **Fix Input Errors –** The user is supposed to type in various information about the location and type of chess piece. If the user tries to put in letters or numbers outside of the acceptable options, the program will crash. Verify the input from the user before proceeding with the rest of the program.
3. **Board Printing Outlines –** The program shows only a single character for each board cell. Modify the output to print a board outline like the example shown here in Figure 46. This shows outlines for each square, column headers, and row numbers.

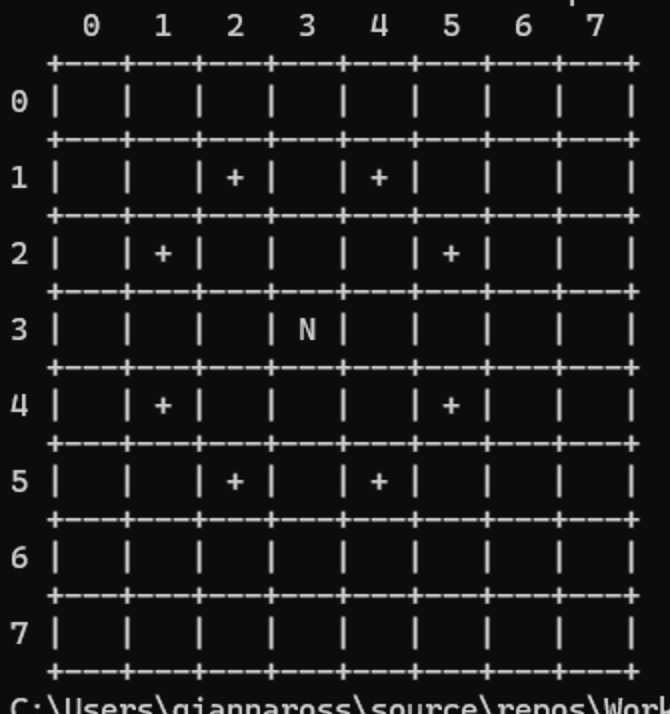


Figure 46 "Pretty" printing for the chessboard



* Commit the challenges to source control with a commit message



* Take a screenshot of the application running after the new features have been added.
* Take a screenshot of each method added or updated to implement the features.
* Take a screenshot of the updated UML diagrams for the updated classes.
* Paste the images into a Word document.
* Put a caption below each image explaining what is being demonstrated.

# Part 3 – Windows GUI Application

In this section, we are going to repurpose the ChessBoardClassLibrary by combining it with a GUI front-end. The function of the application will be the same, but the visual appeal will be much higher.

1. Add a new project to the solution by right-clicking the ChessBoardClassLibrary solution and choosing Add > New Project…. Create a new Windows Forms App as shown in Figure 47. Name the project ChessBoardGUIApp as shown in Figure 48 and create the new application.

A screenshot of a computer

Description automatically generated

Figure 47: Creating a new Windows Forms App

A screenshot of a computer

Description automatically generated

Figure 48: Configuring ChessBoardGUIApp



* Commit all changes in the project to source control.
* Commit Message: Add a GUI app to the project.

1. Rename Form1 to FrmChessBoard as shown in Figure 49 by right-clicking on Form1.cs in the solution explorer and selecting rename.

A screenshot of a computer

Description automatically generated

Figure 49: Renaming FrmChessBoard

1. Next, change the text in the top left corner of FrmChessBoard to "Chess Board" by navigating to the properties section of the form and updating the Text property as shown in Figure 50. Additionally, change the Name property to FrmChessBoard to match the form's name.

A screenshot of a computer

AI-generated content may be incorrect.

Figure 50: Changing the name of the form

1. Add four controls to FrmChessBoard: a combo box, a panel, and two labels as shown in Figure 51. The combo box is a dropdown that will be used to select the chess piece. Change the name of the combo box to cmbChessPieces. Additionally, we can update the items property by selecting the ellipsis button next to the word collection as shown in Figure 52 to add a list of the possible chess pieces: King, Queen, Bishop, Knight, Rook, as shown in Figure 53.

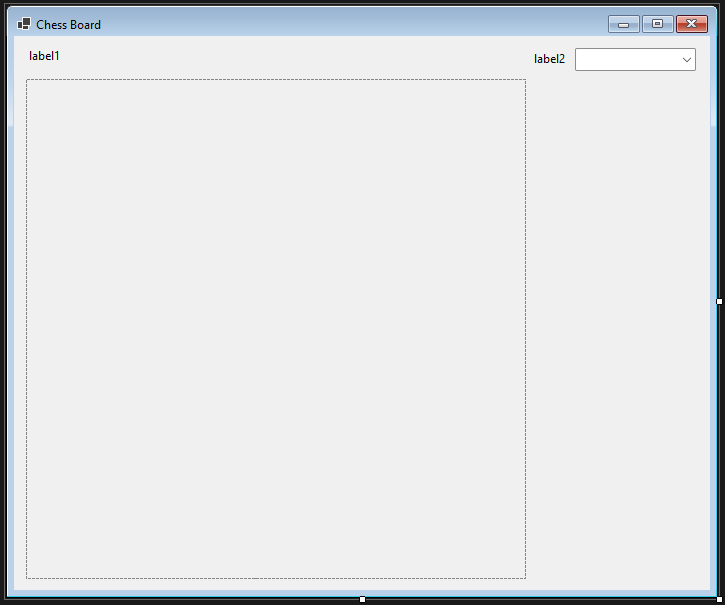


Figure 51: Four controls on FrmChessBoard

A screenshot of a computer

Description automatically generated

Figure 52: Adding pieces to the Items Collection

A screenshot of a computer

Description automatically generated

Figure 53: Collection for cmbChessPieces

1. Next, resize the panel control to be 500×500 using the Size property as shown in Figure 54. This may involve resizing the form. Then, rename the panel to pnlChessBoard.

A screenshot of a computer program

AI-generated content may be incorrect.

Figure 54: Update the size of the panel

1. Lastly, change the text for the two labels. Label1 should say "Select a chess piece and its location on the board and see the legal moves." Label2 should say "Pieces:" as shown in Figure 55.

A screenshot of a computer

Description automatically generated

Figure 55: Labels for the chessboard



* Commit all changes in the project to source control.
* Commit Message: Finish the UI setup for the GUI chessboard.

1. Add a project reference to the ChessBoardClassLibrary. Right-click on the ChessBoardGUIApp and choose Add > Project Reference. In the Reference Manager, check the ChessBoardClassLibrary project as shown in Figure 56 and select OK.

A screenshot of a computer

AI-generated content may be incorrect.

Figure 56: Adding a reference to the ChessBoardClassLibrary project

1. Open the code for FrmChessBoard by right-clicking and selecting View Code. Add a citation to the top of the class as shown in Figure 57.

A computer screen shot of a program

AI-generated content may be incorrect.

Figure 57: FrmChessBoard citation

1. Create three class level variables: a private BoardModel object named \_board, a private BoardLogic object named \_boardLogic, and a private 2D array of Buttons named \_buttons as shown in Figure 58. We will use the buttons array to set up the chessboard on our GUI. In the FrmChessBoard constructor, initialize all three of the variables. The \_board variable should have a size of 8 and the buttons array will be 8×8. Remember to add a summary comment to the constructor.

A computer screen shot of a computer program

AI-generated content may be incorrect.

Figure 58: Class level variables for FrmChessBoard

1. Create a new method beneath the FrmChessBoard constructor named SetUpButtons as shown in Figure 59. The method will have a private access specifier and return nothing. Remember to add summary and end of method comments.

A screenshot of a computer program

AI-generated content may be incorrect.

Figure 59: Method to set up the button grid in the GUI

1. This method will be used to populate the GUI panel with button controls. First, we need to calculate the size of the buttons based on the panel's size. Then, make the panel square using the Height and Width attributes so the buttons fit nicely in the panel. We can then loop through the BoardModel's Grid property using nested for loops to set up the individual buttons as shown in Figure 60.

A computer screen shot of a program code

AI-generated content may be incorrect.

Figure 60: Set up button method before the individual button logic has been added

1. Assign each index in the button array to a new button and set the width and height to the variable we created earlier. Then, calculate the position of the button based on the left and top sides of the control. We can calculate the left position by multiplying the row number by the button size. Likewise, we can calculate the top position by multiplying the column number by the button size. Attach an event handler to the button named BtnSquareClickEH. This will create an error, as we will create this method next.

In Windows Forms apps, we can use the Tag property of a control to store commonly needed information where it is easy to access. In this case, we will store a Point object in the Tag property of each button to keep track of the location of the button. A Point object allows us to store a pair of x-y coordinates in a single object. We can also set the Text of the button to show its coordinates. Lastly, use the panel's Controls property to add the created button to the panel as shown in Figure 61.

A screen shot of a computer program

AI-generated content may be incorrect.

Figure 61: Setting up the individual buttons in SetUpButtons

1. Call the SetUpButtons method in the FrmChessBoard constructor as shown in Figure 62.

A computer screen shot of a program code

AI-generated content may be incorrect.

Figure 62: Calling the SetUpButtons method



* Commit all changes in the project to source control.
* Commit Message: Setup a grid of buttons on the GUI.

1. To create the BtnSquareClickEH method, you can type the method signature manually, or hover over the error and select "Show Potential Fixes" > "Generate Method 'BtnSquareClickEH'" as shown in Figure 63. If you choose to automatically generate the method, make sure to delete the nullable reference for the object parameter and remove the "throw new NotImplementedException()" code as shown in Figure 64. Add a summary comment to the new method.

A screenshot of a computer program

AI-generated content may be incorrect.

Figure 63: Generating the BtnSquareClickEH

A computer screen with text on it

AI-generated content may be incorrect.

Figure 64: Empty BtnSquareClickEH method

1. Change the startup project to the GUI app by right-clicking the ChessBoardGUIApp and select "Set as Startup Project" as shown in Figure 65.

A screenshot of a computer

AI-generated content may be incorrect.

Figure 65: Set the GUI project as the startup project

1. If you run the program, you should see that the buttons have been created in the panel control with the row and column numbers listed on each button.



* Take a screenshot of the application running at this point.
* Take a screenshot of the citation for FrmChessBoard. Make sure to include the class declaration in the image.
* Take a screenshot of the FrmChessBoard constructor and SetUpButtons methods.
* Take a screenshot of the UMLs for the FrmChessBoard class.
* Paste the images into a Word document.
* Put a caption below each image explaining what is being demonstrated.

1. Next, create a new method that will update the text for each button based on the BoardModel grid. The method will have a private access specifier with the name UpdateButtons and return nothing. Remember to add a summary comment as shown in Figure 66. Optionally, you can add an end-of-method comment to keep track of the last curly brace for this method.

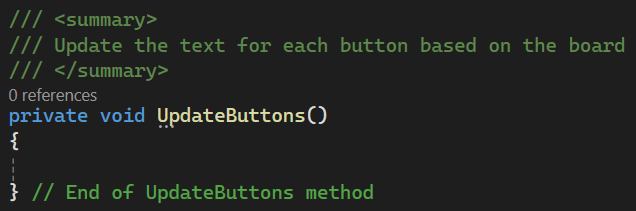


Figure 66: Method to reset the text for each button

1. Set up a new Dictionary object named piece map in the UpdateButtons method as shown in Figure 67. This will store the chess notations that we used in StoreLogic and map them to the full names for the chess pieces. The dictionary data structure stores key-value pairs. Each entry consists of two parts: they key and the value. The key is used to uniquely identify the entry, and the value is the data associated with that key. Similarly to the list structure, the dictionary also needs to be defined with a set to types, such as <string, string> or <int, decimal>. To get a value from the dictionary we can use the following format: dictionaryName[key], like an array. Then, we can loop through the grid to update the text based on each cell.

A computer screen shot of a program

AI-generated content may be incorrect.

Figure 67: Update the text for each button using a dictionary

1. Lastly, we will populate the BtnSquareClickEH method to show the legal moves for each piece. To get the button that triggered the click event handler, we can use the sender parameter and cast the object to a button. Then, we can get the location of the button by casting the Tag property to a point object as shown in Figure 68. Show a message box to the user to say which cell they clicked. Next, get the name of the selected piece out of the combo box and send it to the MarkLegalMoves method and store the return in the board variable. Finally, use the method we just created to update the buttons.

A computer screen shot of a program code

AI-generated content may be incorrect.

Figure 68: Set up the click event handler for the buttons

1. Run the application and place each piece on the board to make sure they work.



* Take five screenshots of the application running at this point, one with each chess piece: King, Queen, Bishop, Knight, Rook.
* Take a screenshot of the BtnSquareClickEH and UpdateButtons methods.
* Take a screenshot of the updated UML for the FrmChessBoard class.
* Paste the images into a Word document.
* Put a caption below each image explaining what is being demonstrated.



* Commit all changes in the project to source control.
* Commit Message: Finish the GUI by marking the legal moves.

## GUI App Challenges

Add the following features to enhance your program:

1. **Error checking**

Double check to make sure all cases work as designed. Check for out of bounds errors. Update any logic necessary to make sure that the program does not error out for the user.

1. **Color Theme Options**

Create a list of color theme options for the user to pick from or have the users pick their own set of custom colors.

Possible ideas are cool colors, warm colors, crazy colors, checkerboard pattern, gradient colors (cells could fade from one color to another from top to bottom, for example), neon colors, pastel colors, nature colors, etc. If you want to generate custom color palettes for the activity, you could use [Adobe Color](https://color.adobe.com/), [Coolors](https://coolors.co/), [Color Hunt](https://colorhunt.co/), [Canva Color Palette Generator](https://www.canva.com/colors/color-palette-generator), [Paletton](https://paletton.com/), or any other websites.

As a tip, you can create arrays of color objects to store the chosen colors based on theme.

Include all color themes in your word document screenshots.



* Commit the challenges to source control with a commit message.



* Take a screenshot of the application running after the new features have been added.
* Take a screenshot of each method added or updated to implement the features.
* Take a screenshot of the updated UMLs for any classes with added methods.
* Paste the images into a Word document.
* Put a caption below each image explaining what is being demonstrated.

# Check for Understanding

These quiz questions will help you review important concepts from the lesson. These are not graded. You do not have to submit your score.

1. What is the primary benefit of separating the chessboard logic and models into their own class library in the project?
   1. To make the user interface more responsive
   2. To ensure that the application can only be run on Windows
   3. To allow the logic to be reused by multiple applications
   4. To increase the processing speed of the chess application
2. Why is it important to maintain a separation of concerns between the user interface and the backend logic in software development?
   1. It makes the program run faster.
   2. It simplifies debugging and maintenance.
   3. It is required by all programming languages.
   4. It prevents users from interacting with the application.
3. In the context of the chessboard application, which C# construct is best suited for use in a flexible and reusable method to check if a chess piece's move is within the board limits?
   1. A for loop
   2. An if statement
   3. A switch statement
   4. A while loop
4. What is the role of the CellModel class in the Chess Board project?
   1. To manage user inputs for the chess piece positions
   2. To represent each square on the chessboard, holding properties like occupancy and legal move status
   3. To handle the graphical rendering of the chessboard
   4. To store the high scores of players
5. Which element in the application allows the user to select a chess piece type to place on the board?
   1. A textbox control
   2. A label control
   3. A combo box control
   4. A panel control
6. What does the declaration "Button[,] Buttons = new Button[8, 8];" represent in the chessboard application?
   1. It initializes a two-dimensional array of button objects.
   2. It declares an array of eight button objects.
   3. It creates a single button object with two properties.
   4. It sets eight different properties on a single button object.
7. How does the BoardModel class primarily function within the chessboard application?
   1. It serves as a layout manager for the GUI elements.
   2. It provides networking capabilities for multiplayer sessions.
   3. It encapsulates the entire game board as a grid of cell objects.
   4. It handles sound effects and animations in the GUI.
8. In the context of the chessboard application, why is a two-dimensional array used for both Button[,] buttons and Cell[,] Grid in the Board class?
   1. Because it simplifies the storage and access of elements arranged in a grid pattern, reflecting the physical layout of a chessboard.
   2. Because it allows for faster computations and better performance.
   3. Because two-dimensional arrays require less memory than one-dimensional arrays.
   4. Because it provides automatic sorting of chess pieces.
9. What does the "Point" structure specifically represent in this chessboard application?
   1. The coordinates of the chess piece image
   2. The size of the chessboard
   3. The row and column indices for a button
   4. The location of the button on the form
10. Which C# data structure is used to map chess piece abbreviations to their full names?
    1. A list
    2. An array
    3. A queue
    4. A dictionary
11. What is the purpose of getting the "Tag" property in the BtnSquareClickEH method?
    1. To determine the size of the button
    2. To retrieve the coordinates stored in the button
    3. To change the color of the button
    4. To display the text on the button
12. Which method is called to update the text on each button based on the game state in the chessboard application?
    1. UpdateButtons
    2. BtnSquareClickEH
    3. SetUpButtons
    4. MarkLegalMoves
13. What does the dictionary object pieceMap directly associate in its key-value pairs?
    1. Button text and color
    2. Chess notation and corresponding full name of the piece
    3. Button coordinates and piece position
    4. Button color and its visibility
14. Why is a dictionary object used in the UpdateButton method instead of a list or an array?
    1. Because it allows quicker access to elements by a key
    2. Because it automatically sorts the keys
    3. Because it can only store string types
    4. Because it uses less memory

Answer Key:

1. What is the primary benefit of separating the chessboard logic and models into their own class library in the project?

c. To allow the logic to be reused by multiple applications

1. Why is it important to maintain a separation of concerns between the user interface and the backend logic in software development?

b. It simplifies debugging and maintenance

1. In the context of the chessboard application, which C# construct is best suited for creating a flexible and reusable method to check if a chess piece's move is within the board limits?

b. An if statement

1. What is the role of the CellModel class in the Chess Board project?

b. To represent each square on the chessboard, holding properties like occupancy and legal move status

1. Which element in the application allows the user to select a chess piece type to place on the board?

c. A combo box control

1. What does the declaration "Button[,] Buttons = new Button[8, 8];" represent in the chessboard application?

a. It initializes a two-dimensional array of button objects.

1. How does the BoardModel class primarily function within the chessboard application?

c. It encapsulates the entire game board as a grid of cell objects.

1. In the context of the chessboard application, why is a two-dimensional array used for both Button[,] buttons and Cell[,] Grid in the Board class?

a. Because it simplifies the storage and access of elements arranged in a grid pattern, reflecting the physical layout of a chessboard.

1. What does the "Point" structure specifically represent in this chessboard application?

c. The row and column indices for a button

1. Which C# data structure is used to map chess piece abbreviations to their full names?

d. A dictionary

1. What is the purpose of getting the Tag property in the BtnSquareClickEH method?

b. To retrieve the coordinates stored in the button

1. Which method is called to update the text on each button based on the game state in the chessboard application?

a. UpdateButtons

1. What does the dictionary object pieceMap directly associate in its key-value pairs?

b. Chess notation and corresponding full name of the piece

1. Why is a dictionary object used in the UpdateButton method instead of a list or an array?

a. Because it allows quicker access to elements by a key

# Deliverables

You must submit **two separate files**:

* One **PDF** containing Parts 1–3.
* One **zip file** containing the complete codebase for Part 4.

Part 1: Activity Submission

1. Cover Page
   * Use APA formatting for the cover page.
2. GitHub Repository Link
   * At the top of the second page, provide a link to your private GitHub repository.
   * Be sure to add your instructor and grading assistant (if applicable) as collaborators.
3. Updated Flowchart
   * Include the latest flowchart reflecting your current application design.
4. Updated UML Diagram(s)
   * Provide UML diagrams that reflect the current implementation.
5. Base Application Screenshots
   * Include screenshots as required in the activity instructions.
   * Each screenshot must have a clear caption directly below the image explaining what is being demonstrated.
   * All code must be properly commented.
6. Section Headers
   * Use clear headers as outlined in these deliverables to identify each part of the assignment within the PDF.
7. Citations
   * Include a citation at the top of each page containing code, flowcharts, or UML diagrams.
8. File Naming
   * Name the PDF according to the activity title and include your full name.
9. Coding Standards Compliance
   * Ensure GCU College of Engineering and Technology Coding Guidelines and Best Practices, found in the Class Resources, have been followed to ensure consistency, readability, and professionalism in student code.

Part 2: Activity Challenges

1. Challenge Screenshots
   1. Include screenshots that demonstrate completion of the challenge requirements as described in the activity guide.
2. Minimum Features
   1. Console App Challenges
      1. Implement and document all challenges.
   2. GUI App Challenges
      1. Implement and document all challenges.

Part 3: Planning for Topic 3 Activity 3

1. Flowchart for Topic 3 Activity 3
   1. Using the requirements as outlined in Topic 3 Activity 3, create detailed flowcharts that can be used to implement the design of the following apps:
      1. Count To One
      2. Factorial
      3. Greatest Common Divisor
      4. Flood Fill
2. UML Diagram(s) for Topic 3 Activity 3
   1. Provide UML diagram(s) based on the same requirements for each of the following apps:
      1. Count To One
      2. Factorial
      3. Greatest Common Divisor
      4. Flood Fill

Part 4: Code Submission (Separate Zip File)

1. Zip Your Project Folders
   1. Compress the full folder containing all source code and necessary files into a .zip file.
2. Executable Submission
   1. Ensure the application runs correctly. The zip file will be unzipped and tested during grading, so include all dependencies.
3. File Naming
   1. Name the zip file according to the activity and include your full name.

**Submission Reminder**

**Do not submit a single zip file that contains both the PDF and the code.** Each file must be uploaded separately.