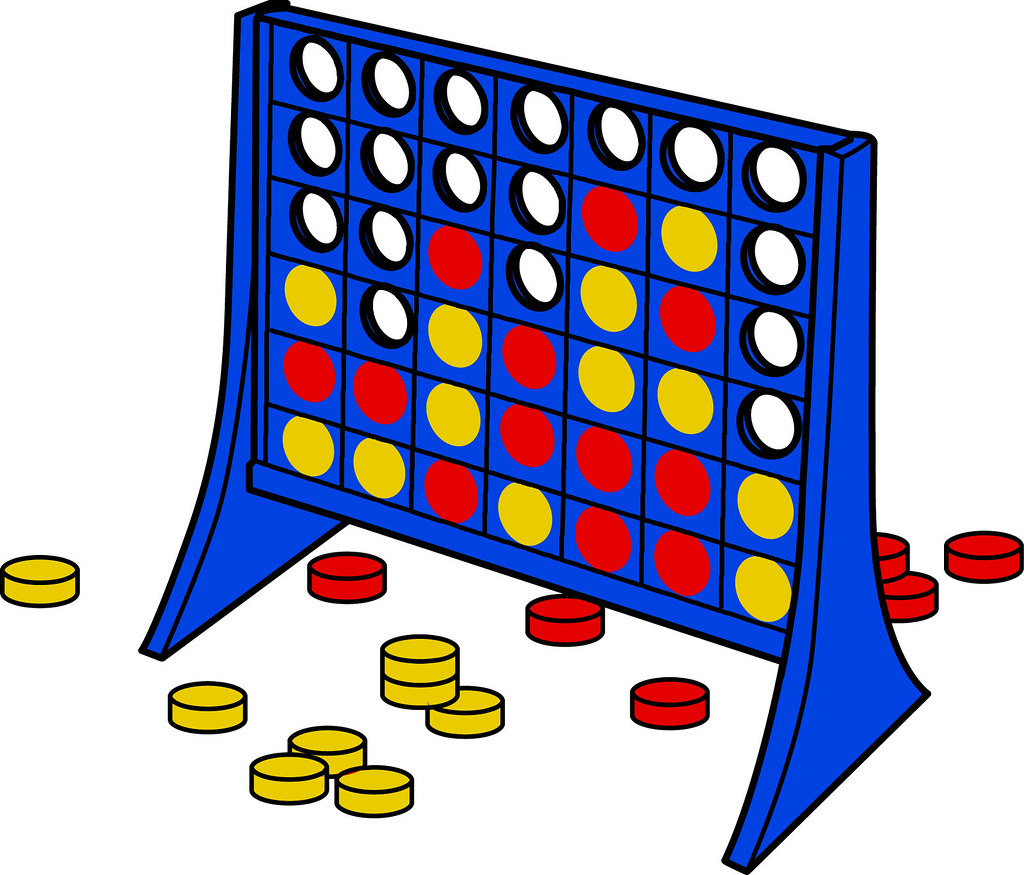
|  |  |
| --- | --- |
| Connect Four Game Prototype |  |



|  |  |  |
| --- | --- | --- |
| University  Birmingham City University | Project Instructor  Prof. Emmet Cooper | Subject  Computer Mathematics  & Declarative Programming |
| Team Member 1  Nisarg Patel | Team Member 2  Anirudh |  |

Table of Contents

[Introduction 3](#_Toc161592567)

[Overview of the Project 3](#_Toc161592568)

[Purpose of Report 3](#_Toc161592569)

[Scope of the Project 3](#_Toc161592570)

[Project Goals 4](#_Toc161592571)

[Planning and Designing a Program (T1) 5](#_Toc161592572)

[Methods of User Interaction 5](#_Toc161592573)

[Gherkin Specifications 5](#_Toc161592574)

[Feature: Connect Four Game in Terminal 5](#_Toc161592575)

[Data Model Design 9](#_Toc161592576)

[Player Data Model 9](#_Toc161592577)

[Game Board Data Model 9](#_Toc161592582)

[Turn Data Model 10](#_Toc161592587)

[Directions Data Model 10](#_Toc161592589)

# Introduction

## Overview of the Project

This project is aimed to recreate the Connect Four game in the form of an interactive command-line program.

According to [Wikipedia](https://en.wikipedia.org/wiki/Connect_Four), Connect Four (also known as Four Up, Plot Four, Find Four, Captain's Mistress, Four in a Row, Drop Four, and Gravitrips in the Soviet Union) is a game in which the players choose a color and then take turns dropping colored tokens into a six-row, seven-column vertically suspended grid. The pieces fall straight down, occupying the lowest available space within the column. The objective of the game is to be the first to form a horizontal, vertical, or diagonal line of four of one's own tokens.

Also, The Connect Four game prototype project was managed using GitHub, a collaborative platform for version control and project management. Here’s the project repository link [Connect-Four-Terminal-Game](https://github.com/NisargMukeshbhaiPatel/Connect-Four-Terminal-Game).

## Purpose of Report

This report aims to outline the process of designing and implementing the Connect Four game prototype, as mentioned in coursework. It will cover the planning, design, development & testing of the command-line version of the game.

## Scope of the Project

Implementing a Connect Four command-line program using Python with curses lib for the user interface. Unit testing will be done with Python's built-in unittest framework, while Behave will be used for Behavior-Driven Development (BDD) testing.

# Project Goals

The primary objective of the project is to create a small-scale prototype of Connect Four game implemented as a command-line program.

1. Basic Gameplay Features:
   * Initiate 2-player games and enable turn-based token placement.
   * Implement single-player mode against CPU.
   * Visually appealing user interface/experience in the terminal using ASCII art and terminal colors.
2. User Interaction:
   * Define user interaction methods using Gherkin specifications and Hoare logic.
   * Implement screen routing to allow switching between different screens within the game.
   * Develop key handlers.
   * Handle key events when actions are performed for each screen.
3. Data Model Design:
   * Design data model for the game state.
4. Implementation:
   * Use Python to create different data components.
   * Write functions, modules, etc, to define behaviours for various program features and interactions.
5. Testing
   * Manual testing will be done according to Gherkin specifications.
   * Utilize a [behave](https://behave.readthedocs.io/en/stable/) library for automated behavioural tests, and use python built in [unittest](https://docs.python.org/3/library/unittest.html) module for automated unit tests.

# Planning and Designing a Program (T1)

## Methods of User Interaction

### Gherkin Specifications

We decided to put all scenarios in one feature for our Connect Four game. Since the game's mechanics are pretty simple, like starting the game, taking turns, and winning, keeping them all together seemed like a practical choice.

But if the game evolves to include more features or becomes more complex over time, we'll consider splitting the scenarios into separate features.

|  |  |
| --- | --- |
| Feature: Connect Four Game in Terminal | |
| **Scenario:** Game window initializes correctly | |
| **Given** a new Connect Four game | |
| **Then** the game should display the start screen menu | |
|  | |
| **Scenario:** Game starts correctly with empty board | |
| **Given** a Connect Four game with player1 and player2 | |
| **Then** the game board should be empty | |
|  | |
| **Scenario**: Player’s switch turns dropping pieces | |
| **Given** a Connect Four game with player1 and player2 | |
| **When** player1 drops a piece in column 1 | |
| **Then** it should be player2 turn |

|  |
| --- |
| **Scenario**: Taking Turns - Player vs Computer |
| **Given** a Connect Four game with player1 and computer |
| **Then** it should be computer turn |
| **When** the computer automatically drops a piece |
| **Then** the computer should drop a piece in a valid column on the  game board |
| **And** it should be player1 turn |
|  |
| **Scenario**: Game ends in a draw |
| **Given** a Connect Four game with players player1 and player2 |
| **When** players drop pieces on the game board in the following  Pattern  X 0 0 0 X 0 X  0 X X X 0 X 0  X 0 0 0 X 0 X  0 X X X 0 X 0  X 0 0 0 X 0 X  0 X X X 0 X 0 |
| **Then** the game should end in a draw |

|  |
| --- |
| **Scenario**: Player wins diagonally |
| **Given** a Connect Four game with players player1 and player2 |
| **When** players drop pieces on the game board in the following  pattern  - - - - - - -  - - - - - - -  - - - - - - X  - - - - - X 0  - - - - X 0 0  - 0 0 X 0 0 0 |
| **Then** player2 should win the game with the last dropped row 3 and  column 7 |
|  |
| **Scenario**: Player wins horizontally |
| **Given** a Connect Four game with players player1 and player2 |
| **When** players drop pieces on the game board in the following  pattern  - - - - - - -  - - - - - - -  - - - - - - X  - - - - - X 0  - - - - X X 0  - 0 0 0 0 X 0 |
| **Then** player1 should win the game with the last dropped row  6 and column 2 |

|  |
| --- |
| **Scenario**: Player wins vertically |
| **Given** a Connect Four game with players player1 and player2 |
| **When** players drop pieces on the game board in the following pattern  - - - - - - -  - - - - - - -  - - - - - X X  - - - - - X 0  - - - - X X 0  - X 0 0 0 X 0 |
| **Then** player2 should win the game with the last dropped row 3  and column 6 |
|  |
| **Scenario**: Invalid Move when column is full |
| **Given** a Connect Four game with players player1 and player2 |
| **When** players drop pieces on the game board in the following  pattern  - - - - - - X  - - - - - - X  - - - - - 0 X  - - - - - X 0  - - - - X X 0  - X 0 0 0 X 0 |
| **Then** dropping a piece in column 7 should return -1 |

## Data Model Design

|  |
| --- |
| Player Data Model |
| Set Definition: P represent the set of all players in the game. **Elements**: Each player, denoted by *pi*, is represented as tuple containing their name and color.  *pi*​ = { *name*: string, *color* ∈ *C* } Constraints:The name and color attributes of a player are of string type.*C* represents the set of available colors in the curses library. |
| Game Board Data Model |
| Definition: Let G represent the set of all cells in the game board. **Elements**: The game board is represented as a set of tuples where each tuple contains the coordinates of the cell and the player's move represented by the player index in *P*. A value of -1 indicates an empty cell.  numRows = 6  numCols=7  G={ (i , j, p) ∣ i, j ∈ W, 0≤i<numRows, 0≤j<numCols, p ∈ {−1,0,1} } |

|  |
| --- |
| Constraints:Dimensions: The game board has 6 rows and 7 columns, both being non-negative integers.Cell Values: Each cell can contain the index representing a player's index or -1. |
| Turn Data Model |
| Definition: Let T represents the set of all turns in the game. **Elements**: Each turn, denoted by *t*, represents the index of the current player whose turn it is in the players array.  numPlayers = 2  T={t: t ∈ W ∣ t ≥ 0 ∧ t < numPlayers }  **Constraints:**   * Turns must be non-negative integers * Each turn is the index of the current player in the players array. |
| Directions Data Model |
| Definition: Let D represent the set of all possible directions in the game board.Elements: Each direction, denoted by *di* , is represented as a tuple containing the horizontal and vertical direction vectors. D={(dx, dy) ∣ dx, dy ∈ { −1,0,1 }, (dx, dy) ≠ (0,0) }  **Constraint:** Direction must not be the zero vector, meaning it must have both horizontal and vertical components. |
| Game State Data Model |
| Definition: Let S represent the set of all possible game states.Elements: Each game state, denoted by si , is a combination of players, current player’s turn, and the game board. si = { players: P, turn: t, game\_board: g }  **Constraint:** Player set P, current turn t, and game board g must follow their respective data model constraints discussed above. |

### Type Definitions for Data Models in Python

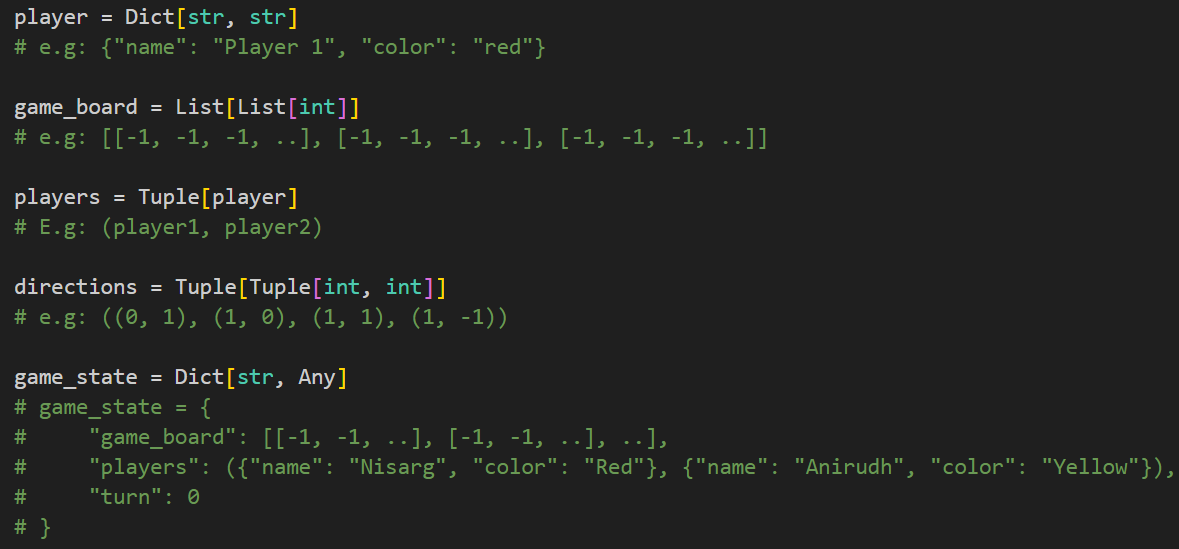


Fig1: type definitions of data models in python

### Type Definitions Justification

|  |
| --- |
| **Player**:   * Using a dictionary for player allows easy access to player attributes by key. * Strings Provide clear and readable player name and color storage.   **Game Board:**   * A 2D list provides a structured way to represent the game board, with rows and columns. * As player information is stored separately in the players, using index on the game board simplifies access and manipulation of players.   **Game State:**   * A dictionary allows for easy access to different components of the game state.   **Directions:** It is defined as a tuple of tuples, where each inner tuple represents dx and dy. Adding dx to the row index and dy to the column index easily gives the direction. |

# Implementing Program (T2)

## Justification for Technology Choices

Python is chosen as primary language for this project because of its simplicity, extensive lib/modules support, and cross-platform compatibility, so our Connect Four game can run on multiple platforms.

Curses library enables lightweight, terminal-based interface, optimizing resource usage and offering precise control over output. Additionally, while curses is built-in for Python on Linux, it’s available as an external module, [windows-curses](https://github.com/zephyrproject-rtos/windows-curses), on Windows.

The Behave framework aligns with the project's Behavior-Driven Development approach with clear & maintainable test cases.

## Screen Rendering Utilities

### Overview

The code, located in the **utils/ui.py**, contains utility functions for rendering grids/boards and rectangles on the screen, primarily targeting UI elements for board-based games, making it generic beyond just game like Connect Four.

|  |
| --- |
| Function draw\_board() |
| Draws a grid representing the game board  Parameters:   * rows, cols: Number of rows and columns in the grid. * width, height: Width and height of the grid on the screen. * win: The window object where the grid will be drawn. * delay: Optional parameter for introducing delays between drawing for animating effect. * color: Optional to set the board color (default: Blue).   Returns: List containing the horizontal and vertical gap sizes, it’s useful for other parts of the program that need to interact with the grid and wants to know the gaps calculated to make the grid.  Pure Function: Accepts input parameters and produces output without modifying external state.  Modularity: function has single responsibility is to draw a grid and is generic.    Code:  Fig2: draw\_board function code |
| Function draw\_rect\_inside() |
| Draws a rectangle inside the grid at a specified row and column.  Parameters:   * row, col: Row and column indices where the rectangle starts. * w, h: width and height of the rectangle. * color: Color attribute for the rectangle. * win: The window object where the rectangle will be drawn.   Returns: None  Pure Function & Modular  Code:    Fig3: draw\_rect\_inside function code |
| Function draw\_rect\_around() |
| Draws a rectangle around a specified position in the grid.  Parameters:   * col: Column index where the rectangle starts. * w, h: width and height of the rectangle. * color: Color attribute for the rectangle. * win: The window object where the rectangle will be drawn.   Returns: None  Produces output without side effects & focuses on a specific task.  Code:    Fig4: draw\_rect\_around function code |
| Function add\_multi\_line\_str() |
| Adds a multi-line string of python on the screen at a given position  Parameters:   * win: The window object where the string will be added. * y, x: Coordinates for the string's starting position. * str: The multi-line string to be added. * color: Optional color for the string.   Returns: None  Code:    Fig5: add\_multi\_line\_str function code |

## ANSI Color Handling Utilities

### Overview

The ANSI utility file **(utils/ansi.py)** provides essential functionality for handling ANSI colors and displaying ASCII art images in the terminal. It defines functions for initializing ANSI color pairs, converting images to ASCII art, and rendering them with appropriate colors. It uses a generic approach, allowing its reuse in other terminal-based application too.

|  |
| --- |
| ANSI\_COLORS |
| ANSI\_COLORS is a tuple that stores predefined ANSI color codes for text coloring in the terminal.    Fig6: ANSI\_COLORS data |

|  |
| --- |
| Function set\_init\_pair\_ansi\_index() |
| * **set\_init\_pair\_ansi\_index()** is a function responsible for initializing curses color pairs for the terminal. * It first calls **curses.use\_default\_colors()** to use the default background color of terminal. * The color pairs are set for both foreground and background colors using **curses.init\_pair().**     Fig7: set\_init\_pair\_ansi\_index function |
| Function add\_img\_with\_ansi() |
| It is a function responsible for adding an image converted to ASCII art to the terminal with ANSI colors.  Parameters:   * path: Path to the image file. * x: X-coordinate where the image should start drawing. * y: Y-coordinate where the image should start. * cols: Number of columns to fit the image. * Stdscr/win: The standard screen object provided by curses.     More Explanation:   * It first converts the image to ASCII art using the ascii\_magic library and retrieves the output. * Within each line, it splits the line based on ANSI escape sequences to extract color information. * It utilizes stdscr.addch() to add each character of the image to the terminal with the appropriate color, based on the ANSI color code.     Fig8: add\_img\_with\_ansi function |

## Start Screen

### Overview

The **Start Screen** is the initial screen/point-of-interaction for players, it renders a logo, menu options, and provides instructions for navigation and interaction using keyboard as mouse interactions are not supported yet.

|  |
| --- |
| Function render\_start\_screen() |
| The “render\_start\_screen” function is responsible for displaying the start screen of the game  Parameters:   * win (curses.window): The window object where the start screen will be rendered. * state (Optional, Any): The state of the game or additional information needed for rendering the start screen. Default is None.   Return Values: String or None   * Returns the selected menu option (PLAY, PLAY\_COMP, HOW\_TO\_PLAY) when the user presses ENTER. If the user presses 'q' to quit, it returns None. * It specifies the route chosen by the user. Which will be used to render another screen (more info in Routing for Screens) |

|  |
| --- |
| Window Setup |
| Implementation details:   * Initializes the window for rendering the start screen using curses. * Determines if the terminal size is sufficient to display the start screen. If the height is less than 25 lines, a warning is given, indicating that the screen height is insufficient for the start screen. * Calculates padding based on the available height, it’s adjusted to a minimum of 3 for larger terminal heights. * Defines the x and y coordinates from which the screen will start drawing.     Fig9: Window Setup |
| Menu Configuration |
| Implementation:   * Defines the menu options and their positions on window. * Initializes the “menu\_focus” variable to indicate the currently selected menu item, allowing navigation through the menu options.     Fig10: Menu Configuration |
| Rendering Functions |
| Implementation:   * Defines functions for rendering various elements such as the logo, menu, and information text. * It Encapsulates rendering logic within separate functions (render\_info\_text, * render menu) making them reusable.     Fig11: Rendering Functions |
| Background Animation |
| Implementation:   * Animates the background by cycling through a sequence of frames. * Uses a while loop to iterate through a series of frames and updates the background accordingly using add\_img\_with\_ansi function. * After rendering the animation, the function enters a loop with a specified delay (0.6 seconds) to handle user input which we’ll discuss after this.     Fig12: Background Animation |
| Text Animation |
| Implementation:   * The function toggles between two versions of the logo (CONNECT\_FOUR\_LOGO and CONNECT\_FOUR\_LOGO2) at regular intervals, creating a flickering effect. * Similar to logo animation, the function alternates the appearance of the currently focused menu items. * It uses a toggle mechanism to switch between versions each time the animation loop iterates.     Fig13: Logo and menu animation logic |
| Delay and Event Handling |
| Delay Handling:   * It controls the speed of animation transitions. * After each iteration of the animation loop, the function waits for a predefined delay period before proceeding to the next iteration.   Input Handling:   * While waiting for animation delays, the function detects user for input. This continuous detection ensures uninterrupted user interaction with screen.     Fig14: Delay and Event Handling |

## Game Screen

### Overview

|  |
| --- |
| Function render\_game\_screen() |
| This function is responsible for rendering the main game in a terminal. It sets up the game board, initializes player information, handles user input, and updates the screen based on player actions. This function encapsulates the entire user interface and game interaction logic.  Parameters:   * win: The curses window object where the game screen will be rendered. * state: Optional parameter representing the current state of the game. * vs\_comp: Boolean flag indicating whether the game is played against the computer or 2 players.   Return Values: Tuple: (next\_screen, winner)   * next\_screen: String representing the next screen the game should route to. * winner: Player object representing the winner of the game, if any. This value may be None if the game ended in a draw or if the player chooses to quit. |
| Game Initialization |
| * The terminal window is set up using the curses library. * Dimensions for the game board and UI elements are calculated based on the terminal size. * Player information, including names and colors, is initialized. In single-player mode, player names are set to "Player" and "Computer". we are not supporting custom player names yet.     Fig15: Game screen initialization |
| Show Player Turn |
| * **render\_player\_turn** function displays the current player's turn on the screen. * The function takes the player's name and color as arguments and updates the UI accordingly.     Fig16: render\_player\_turn() to Show Player |
| Column Selection |
| * Players navigate through columns using arrow keys to select the column for dropping their piece. * Current column selection is visually highlighted using draw\_rect\_around.     Fig17: Column Selection    Fig18: Column Selection key Handling |
| Piece Dropping |
| 1. Player selects a column:   * The player navigates through columns using arrow keys and selects the column where they want to drop their piece.   2. Player drops the piece:   * Upon pressing Enter, the game logic checks if the selected column is valid for dropping a piece. * If the column is full or not valid, the game ignores the input. * If the col is valid, the piece is dropped into the lowest row of that column.   3. Checking for Win:   * After the piece is dropped, the game logic checks for a win condition. * It checks if the recently dropped piece creates a connect-four sequence vertically, horizontally, or diagonally. * If a win condition is met, the game ends, and the winning cells flashes.   4. Checking for Draw:   * If there is no win condition after the piece is dropped, the game logic checks for a draw condition. * If the game board is filled without a winner, all cells flash briefly before routing to start screen.   5. Updating Game State: After checking win or draw conditions, the game state is updated accordingly.  6. Switching Turns :   * If the game is not in vscomp mode, the turn is switched to the next player. * The UI is updated to reflect the current player's turn. * In vscomp mode, if the current turn is the computer's turn, it calculates its move. the computer's move is simulated using a simple random strategy to select a column for now we’ll add ai in future.     Fig19: Piece dropping key handling |
| Quitting Game |
| To quit at any time by press 'q'. Upon quitting, the game returns to the start screen.    Fig20: game quitting key handling |

## Routing for Screens

### Overview

The screen routing system manages different screens within the Connect Four game. It uses a screen map dictionary and a main loop to facilitate navigation between screens.

|  |
| --- |
| Screen Map |
| * It’s a dictionary mapping screen identifiers to rendering functions. * Each screen identifier linked to a specific rendering function.     Fig21: Screen Mapping with render functions |
| Main Loop |
| * Main loop executes continuously, handling screen transitions and user interactions. * Clears the screen, refreshes display, and renders current screen based on screen map. * Waits for user input/actions triggering transitions to other screens based on returns of rendering function with the next screen and state to be passed. * Data passed between facilitates communication between different screens used yet but will be very useful for ending screen (not yet done).     Fig22: Main loop for routing |

# Testing and Verifying the Program

## Manual Testing

## Unit Tests

### Overview

It involves ensuring that each component functions correctly and also to verify the correctness of individual functions and modules in program.