Computer Mathematics and Declarative Programming: Mastermind Command Line Game

Angelo Luis Lagdameo | Imogen Dicen

S2 2024/25

CMP5361: Computer Mathematics and Declarative Programming

Contents

T1 – Planning and Designing

Project Choice & Project Description

Project Members:

- Angelo Luis Lagdameo
- Imogen Dicen

Project Choice: Mastermind Command Line Game

Project Description - Mastermind

This is a "Design from scratch" task and is intended for those students who would like the freedom (and challenge) of speccing out their system from the ground up.

Based on the Mastermind game, your company wants you to create a small-scale prototype for the Mastermind game where it will take the form of an interactive command line program.

The program should aim to implement the following capabilities:

- Allow a player to start a new 2-player game between a human codemaker and a human codebreaker
- Allow a player to start a new single-player game against the CPU where the CPU is the code maker and the code to break is algorithmically generated
- Allow a player to start a new single-player 'campaign' against the CPU where the CPU acts as the code maker but the codes are already predefined (acting a separate "levels" of the game)

Using your developing understanding of how to plan out a program, you should plan out the program using appropriate techniques, considering the following key areas:

- Specify, explain, and justify appropriate methods of user interaction for the program using suitable techniques such as the use of Gherkin specifications based in Hoare logic
- Specify, explain, and justify an appropriate data model for the various kinds of data the program will deal with, supported by appropriate data type and data structure definitions based in Set theory. This should be accompanied by appropriate Type definitions within the programming language based on these Set theory definitions.
- Specify, explain, and justify an appropriate set of behaviours for the program, supported by appropriate definitions based in Mathematical Relations, Mathematical Functions, and Graph Theory.

User / Stakeholder Identification & Analysis

Age Range

As stated in the project description, we are designing the small-scale prototype of the Mastermind game for a company. A company is filled with a range of personnel at different stages of their life. On average, company staff will be around the age of 23+, whilst considering that they are individuals that can play a boardgame.

Considering the company, people from all age ranges can play the game, as long as they are capable of understanding and playing the game.

Technical Capability

As the game is intended for command-line usage, the user must be able to read and type into the command line. Interface complexity should be straightforward and comprehensible to eliminate the difficulty in the case that the user is not widely computer command prompt oriented. To increase readability and ease of playing the game, colours may be used for output in the command line.

Assuming the company is game-development oriented, we would expect that the main field of staff would be capable of having the basic understanding of the Command Prompt nature and the capability of operating a computer as a piece of hardware.

Motivation

Board games in general are played for entertainment purposes, as well as for stimulating the brain. Mastermind is a board game in itself, allowing one or two people to enjoy its challenges.

As the target user specified in the project description is the company. Considering that the company has drawn out the following capabilities expected out of the development of the game program, they will mainly be testing and validating the iterations of the Mastermind implementation so that it behaves soundly to their ideal vision of how the finished product should look and behave like.

Additionally, considering external users involved in the category of everyone, we could infer that some may be using it in the case of analysing for literature review such that they are interested in understanding the mathematic aspect behind the gameplay.

Initial Analysis of requirements

Explicit Req. ID	Name	Description
1	Small-Scale Prototype	Development of a working Mastermind Command Line Game at minimum should carry out the main stated capabilities in order to leave room for necessary testing and the program refinement.
2	Mastermind Game	Program should aim to behave in accordance to how the original boardgame works.
3	Interactive Command-Line Program	Gameplay of the game program will be brought out on the Python IDLE Shell to mimic the Command Line style of an interface.
4	Single Player Mode	Single Player mode should be available to start a game between the Human Code Breaker and the CPU Code Maker where the code to be guessed is algorithmically generated before starting a game.
5	Multiplayer Mode	Multiplayer mode should be available to start a game between a Human Code Breaker and Human Code Maker in which the Code to be guessed is generated by the Human Code Maker before the game starts.
6	Campaign Mode	Single Player Campaign mode should be available to start a game between the Human Code Breaker and the CPU Code Maker. • Levels contain pre-defined codes and a different amount of guess attempts in accordance with their difficulty o 1st Level (Easy) has a Normal Four Colour Pattern o 2nd Level (Medium) has a Four Colour Pattern containing colour duplicates o 3rd Level (Hard) has a Four Colour Pattern.

Implicit

Implicit Req. ID	Name	Description
1	Small board	As we are developing a small-scale prototype of the game, it is implied that the game board will also be small.
2	Main menu	To allow the user to select a game option, there must be a main game menu to choose these options from.
3	Redirection to the main menu	It would be sensible to redirect users to the main menu after a game has ended, rather than closing the program on game completion.
4	User input	As the game is to be played from the command-line, it is expected that the user is to input text to navigate the main menu and play Mastermind.
5	Resignation	There should be an option for the user to end a current game if they intend to go back to the main menu to select another game option.

6	Command line friendly board layout	As the game is to be played from the command-line, the board will be made up from characters instead of being drawn.
7	Display of previous attempts	After every guess attempt, when the board is updated for the next turn, the board will display previous rows of guesses with feedback.

Level of Interactivity

As a command-line board game, the program will not be a chatty program, only prompting the user for their options and stating the user's choices for confirmation, as well as stating the winner of the game, and any error messages for incorrect input.

This Mastermind prototype will work based on very little input from the user, in which options will be given a respective number value designated for a specific selection.

For the marking of Codebreaker guesses, feedback will automatically be generated and will be displayed on an updated version of the game board.

Method of Interactivity

The user will be able to interact with this Mastermind prototype via the command line, as stated in the project description.

We will be using numbers as a form of user input that correspond to choices from a menu displayed (e.g. main menu, code peg choice menu).

The program's / CPU's output will be presented to the user in the command line interface, and the game board will be updated and displayed after every turn.

User Story Specifications

This section provides Gherkin specifications for our system and its behaviours.

Interactable Components of the System

- Allow a player to choose the game mode and start a new game (multiplayer, single player against CPU, single player campaign against the CPU).
- Allow a player to choose a coloured peg to be placed.
- Allow a player to confirm their guess containing the coloured pegs placed.

Happy Paths

This section entails the scenarios of the Mastermind Command Line Game that occur when the intended paths of the program are taken due to valid user input.

Feature: Choosing a game option

Narrative: As a command line user I want to be able to select the game option in the main menu So, I can play a Mastermind game
Scenario 1: Player selects 1st game option Given I have specified an argument of *1* to the command line When I press Enter Then 'SINGLEPLAYER GAME MODE.' should be output And the game board is displayed And the command line prompts the CodeBreaker for their guess
Scenario 2: Player selects 2nd game option Given I have specified an argument of *2* to the command line When I press Enter Then the command line prompts the CodeMaker to form the secret code
Scenario 3: Player selects 3rd game option Given I have specified an argument of *3* to the command line When I press Enter Then 'CAMPAIGN GAME MODE STAGE 1.' should be output And the game board is displayed And the command line prompts the CodeBreaker for their guess
Scenario 4: Player selects 4th game option Given I have specified an argument of *4* to the command line When I press Enter Then a 'Exiting Mastermind…' message should be output And program is exited

Feature: Choosing a peg colour to put in the peg slot

Narrative: As a command line user I want to be able to select the peg colour So, I can place it in a peg slot to guess the secret code
Scenario: Player selects peg colour Given the program has output a list of colours I can choose from And I have specified an argument of **colour** When I press Enter Then 'You have chosen a **colour** peg.' should be output

Feature: Choosing a Confirmation Option for setting the Secret Code

Narrative: As the Codemaker I want to be able to select a confirmation option So, I can confirm my choice of secret code for the CodeBreaker to guess
Scenario: Codemaker proceeds to confirm their secret code Given the user has selected their peg colours for each slot of their secret code And the program prompts whether I want to confirm my secret code When I enter an argument of *"y"* to the command line prompt Then 'Choice Confirmed.' should be output And the program should proceed on to letting the Codebreaker make their 1st guess
Scenario: Codemaker proceeds to cancel their secret code Given the user has selected their peg colours for each slot of their secret code And the program prompts whether I want to confirm my secret When I enter an argument of *"n"* to the command line prompt Then 'Choice Cancelled' should be output And the program re-prompts the Codemaker to set their secret code

Feature: Choosing a Confirmation Option for a Guess

Narrative: As a command line user I want to be able to select a confirmation option So, I can confirm my choice of pegs for my guess
Scenario: Player proceeds to confirm their guess Given the user has selected their peg colours for each slot of their guess And the program prompts whether I want to confirm my guess When I enter an argument of *"y"* to the command line prompt Then 'Choice Confirmed.' should be output And the program should display the update board
Scenario: Player proceeds to cancel their guess Given the user has selected their peg colours for each slot of their guess And the program prompts whether I want to confirm my guess When I enter an argument of *"n"* to the command line prompt Then 'Choice Cancelled.' should be output And the current game board is displayed And the command line re-prompts for the 1st peg colour option of their guess

Sad Paths

This section entails the scenarios of the Mastermind Command Line Game that occur when the unintended paths of the program are taken due to invalid user input.

Feature: Choosing a game option

Narrative: As a command line user I want to be able to select the game option in the main menu So, I can play a Mastermind game
Scenario 1: Player selects an Invalid Game Option Given I have specified an argument of **invalid input** to the command line prompt When press Enter Then 'That is an invalid main menu choice.' should be output to specify an invalid input And the main menu reloads the main menu interface And the command line re-prompts for a Valid Game Option

Feature: Choosing a peg colour to put in the peg slot

Narrative: As a command line user I want to be able to select the peg colour So, I can place it in a peg slot to guess the secret code
Scenario: Player selects invalid peg colour option Given the program has output a list of colours I can choose from And the program has selected the peg slot for me to fill And I have specified an argument of **invalid input** When I confirm my choice of peg colour Then 'That is an invalid code peg choice.' should be output And the command line re-prompts for a Valid Colour peg Option

Feature: Choosing a Confirmation Option for setting the Secret Code

Narrative: As the Codemaker I want to be able to select a confirmation option So, I can confirm my choice of secret code to guess
Scenario: Codemaker enters an invalid confirmation option Given the user has selected their peg colours for each slot of their secret code And the program prompts whether I want to confirm my secret code When I enter an argument of **invalid input** to the command line prompt Then 'That is an invalid confirmation choice.' should be output And the command line re-prompts for a Valid Confirmation peg Option

Feature: Choosing a Confirmation Option for a Guess

Narrative: As a command line user I want to be able to select a confirmation option So, I can confirm my choice of pegs for my guess
Scenario: Player enters an invalid confirmation option Given the user has selected their peg colours for each slot of their guess And the program prompts whether I want to confirm my guess When I enter an argument of **invalid input** to the command line prompt

Then ‘That is an invalid confirmation choice.’ should be output And the command line re-prompts for a Valid Confirmation Option

Data Model

Types

Menu

Explanation:

Menu will represent the option chosen by the human Player from a menu displayed. Created as a Class, it will utilise Enum to set the corresponding Option Values (e.g. Singleplayer initiated by 1).

We will be using Enums to represent this type, to ease handling user input (they will just input the menu number instead of a word matching the option).

Type definition:

$$let\ Menu = \{\ (Singleplayer, 1), (Multiplayer, 2), (Campaign, 3), (Exit, 4)\ \}$$

Type definition in Python:

```
class Main_Menu_Option(Enum):
    Single_Player = 1
    Multiplayer = 2
    Campaign = 3
    Exit = 4

Menu: TypeAlias = Main_Menu_Option
```

ConfirmationOption

Explanation:

ConfirmationOption will represent the option chosen by the human Player to confirm their selection.

Type definition:

$$let\ ConfirmationOption = \{\ (Yes, y), (No, n)\ \}$$

Type definition in Python:

```
class Confirmation_Option(Enum):
    Yes = 'y'
    No = 'n'

Confirm: TypeAlias = Confirmation_Option
```

Code

Explanation:

Code pegs will represent the individual pegs used by the CodeMaker or CPU to make the secret code and by the CodeBreaker to break the secret code.

We will be using Enums to represent this type, to ease handling user input.

The choices of colour are orange, green, blue, yellow, purple and brown; the empty peg is used to define currently empty peg slots in the guess section of a game board row.

Type definition:

$$let\ Code = \{\ (\text{Empty}, 0), (Orange, 1), (Green, 2), (Blue, 3), (Yellow, 4), (Purple, 5), (Brown, 6)\ \}$$

Type definition in Python:

```
class Code_Peg_Option(Enum):
    Empty = 0
    Orange = 1
    Green = 2
    Blue = 3
    Yellow = 4
    Purple = 5
    Brown = 6

Code: TypeAlias = Code_Peg_Option
```

Hint

Explanation:

Hint pegs will represent the individual pegs used by the CPU / program to give feedback on whether the guess made by the CodeBreaker matches the secret code, or not.

The white hint peg infers that a certain coloured code peg has been chosen correctly but has been placed incorrectly on the game board; a red hint peg infers that a certain coloured code peg matches its placement in the secret code. The empty hint peg is used to define currently empty peg slots in the feedback section of a game board row.

Type definition:

$$let\ Hint = \{\ Empty, White, Red\ \}$$

Type definition in Python:

```
class Hint_Peg(Enum):
    Empty = 0
    White = 1
    Red = 2

Hint: TypeAlias = Hint_Peg
```

SecretCode

Explanation: SecretCode represents the combination of code pegs used by the CodeMaker or CPU to create the code to be guessed by the CodeBreaker to win the game.
Type definition: $$let\ SecretCode = Code * Code * Code * Code$$
Type definition in python: `Secret: TypeAlias = tuple[Code, Code, Code, Code]`

Guess

Explanation: Guess represents the combination of code pegs used by the CodeBreaker to guess the secret code.
Type definition: $$let\ Guess = Code * Code * Code * Code$$
Type definition in Python: `Guess: TypeAlias = tuple[Code, Code, Code, Code]`

Feedback

Explanation: Feedback represents the combination of marker pegs used by the CodeMaker to indicate whether the Guess of the CodeBreaker matches the Secret Code.
Type definition: $$let\ Feedback = Hint * Hint * Hint * Hint$$
Type definition in Python: `Feedback: TypeAlias = tuple[Hint, Hint, Hint, Hint]`

Row

Explanation: Row represents a row on the game board where the left section of the row is the CodeBreaker's Guess and the right section of the row is the CodeMaker's Feedback on the Guess.
Type definition: $$let\ Row = Guess * Feedback$$
Type definition in Python: `Row: TypeAlias = tuple[Guess, Feedback]`

Board

Explanation:

Board represents the Mastermind game board, made up of rows. We have chosen to represent our game board using a tuple of 6 rows, indicating the 6 attempts in a normal game; for a harder game, we would represent the game board as a tuple of 4 rows. A board type can either be a normal board or a hard board.

Type definition:

$$let\ NormalBoard = Row * Row * Row * Row * Row * Row$$

$$let\ HardBoard = Row * Row * Row * Row$$

$$let\ Board = NormalBoard \cup HardBoard$$

Type definition in Python:

```
Normal_Board: TypeAlias = tuple[Row, Row, Row, Row, Row, Row]
Hard_Board: TypeAlias = tuple[Row, Row, Row, Row]
Board: TypeAlias = Normal_Board | Hard_Board
```

Player

Explanation:

Player represents a participant of a Mastermind game. A Player could be the human CodeBreaker, a human CodeMaker or the CPU (a CodeMaker).

Type definition:

$$let\ Player = \{CodeMaker, CodeBreaker, CPU\}$$

Type definition in Python:

```
@dataclass(eq=True, frozen=True)
class CodeMaker:
    pass

@dataclass(eq=True, frozen=True)
class CodeBreaker:
    pass

@dataclass(eq=True,frozen=True)
class CPU:
    pass

Player : TypeAlias = CodeMaker | CodeBreaker | CPU
```

Players

Explanation:

Players represents the pair of players that play a Mastermind game. A Players pair can either be a CodeMaker and CodeBreaker, or a CodeBreaker and CPU.

Type definition:

$$let\ Players = (CodeBreaker, CodeMaker) \cup (CodeBreaker, CPU)$$

Type definition in Python:

```
Players : TypeAlias = tuple[CodeBreaker, CodeMaker] | tuple[CodeBreaker, CPU]
```

Axiomatic Definitions and Functions

Stage 1 – Planning Phase

Stage 1 will walk through our initial attempt of planning and designing functions for our Mastermind program. This includes the way in which gameplay and user input was handled, as well as how feedback was generated in response to certain input.

Main

The **main** construct serves as the starting point to where the program begins with calling **receiveMainMenuInput** where the running prompt for Main Menu option is initiated.

receiveMainMenuInput

Explanation:

A running prompt for the Main Menu Option to be selected enables the User to enter their choice what game mode they want to play or if they want to simply exit the program. The choice is passed into the appropriate parser to return the corresponding main menu option.

Depending on what main menu option has been selected, startGameplay is called with the appropriate arguments passed to it. Arguments such as the board(s), players, and secret code(s) will vary on the game mode selected.

Axiomatic definition:

$$receiveMainMenuInput: void$$

$$\begin{aligned} &let\ parseMainMenuOption = \{x: MainMenuOption \mid x == SinglePlayer \cdot startGameplay(x, Board, tuple[Player, Player], normalSecretCode)\} \cup \\ &\{x: MainMenuOption \mid x == Multiplayer \cdot startGameplay(x, Board, tuple[Player, Player], makeSecretCode)\} \cup \\ &\{x: MainMenuOption \mid x == Campaign \cdot startGameplay(x, tuple[Board, Board, Board], tuple[Player, Player], tuple[normalSecretCode, hardSecretCode, hardSecretCode]\} \end{aligned}$$

Graph representation:

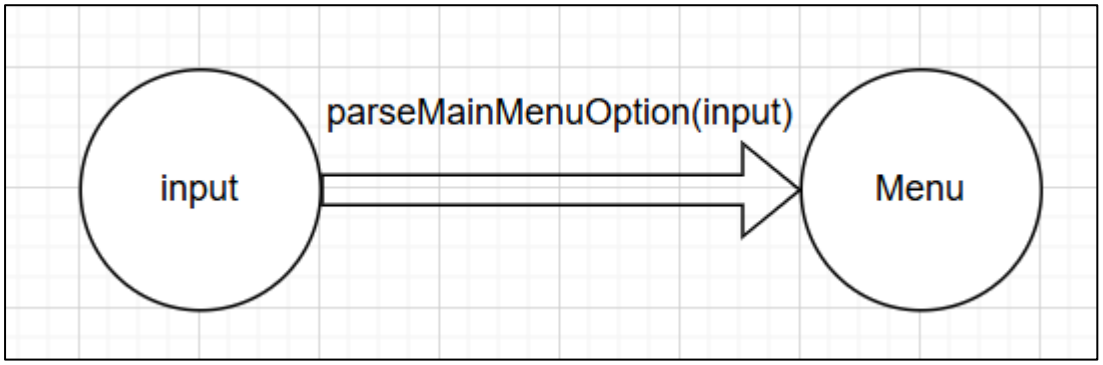


Figure 1 Data transformation graph of when receiveMainMenuInput is called.

parseMainMenuOption

Explanation:

When the Player makes a choice in the main menu and it is received, we need a function to turn their string input into a valid Menu type.

Axiomatic definition:

$$parseMainMenuOption: string \rightarrow Menu$$

$$let\ parseMainMenuOption = \{(\text{"0"}, \text{Single Player}), (\text{"1"}, \text{Multiplayer}), (\text{"2"}, \text{Campaign}), (\text{"3"}, \text{Exit})\} \cup \{x: string \mid x \notin \{\text{"0"}, \text{"1"}, \text{"2"}, \text{"3"}\}(x, None)\}$$

receiveConfirmationInput

Explanation:

In the case of making a Guess or a custom Secret Code, this function prompts for the confirmation option, enabling decision making logic external from this function to decide whether to proceed with a certain next set of instructions or re-prompt the user. The choice is passed into the appropriate parser to convert the choice into its corresponding value to return the appropriate Boolean.

Axiomatic definition:

$$receiveConfirmInput: Union[Guess] \rightarrow bool$$

$$let\ receiveConfirmInput = \{x: ConfirmationOption \mid x == Yes \cdot True\} \cup \{x: ConfirmationOption \mid x == No \cdot False\}$$

Graph representation:

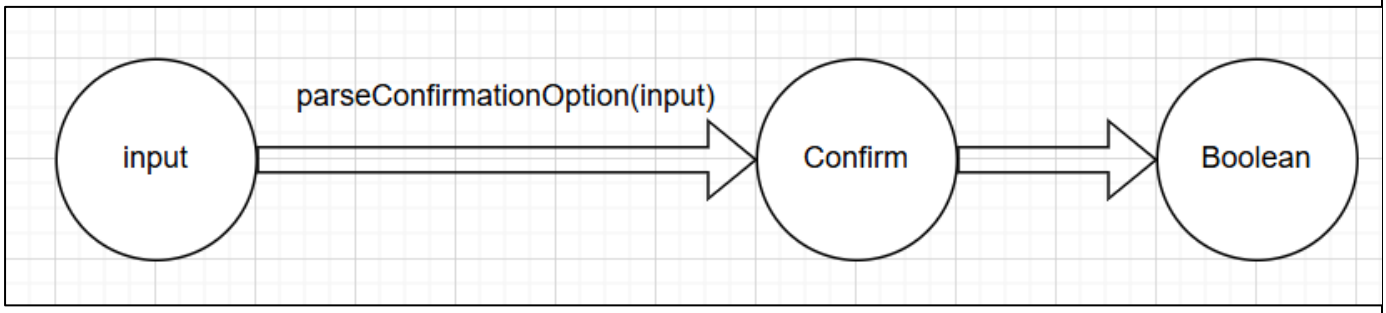


Figure 2 Data transformation graph of when receiveConfirmationInput is called.

parseConfirmationOption

Explanation:

Once a Player has made a Guess or Secret code, we want to confirm the Player that they are happy with their choice. After their confirmation is received, we need a function to turn their string into a valid Confirm type.

Axiomatic definition:

$$parseConfirmationOption: string \rightarrow Confirm$$

$$let\ parseConfirmationOption = \{(\text{"y"}, Yes), (\text{"n"}, No)\} \cup \{x: string \mid x \notin \{\text{"Y"},\ \text{"y"},\ \text{"N"},\ \text{"n"}\} \cdot (x, None)\}$$

receiveCodePegInput

Explanation:

This function is responsible for receiving the user's input for a Code peg and parsing it by calling parseCodePegOption. It will display a message regarding what Code peg the Player has chosen, and return the Code peg. Until the Player chooses a valid Code peg, this function will not be exited.

Axiomatic definition:

$$receiveCodePegInput: void \rightarrow Code$$

Graph representation:

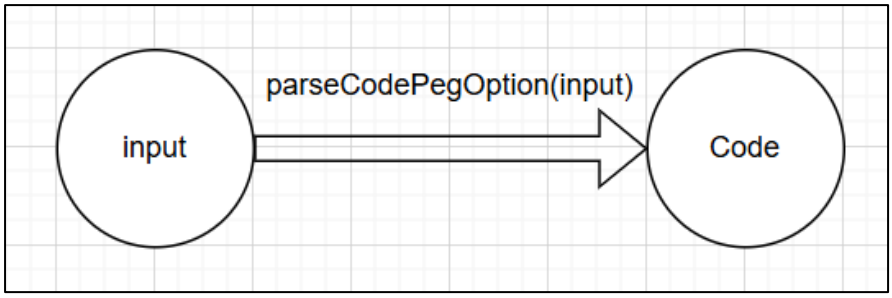


Figure 3 Data transformation graph of when receiveCodePegInput is called.

parseCodePegInput

Explanation:

After a Code peg is chosen by the Player and received, their string input must be turned into a valid Code type.

Axiomatic definition:

$$parseCodePegOption: string \rightarrow Code$$

$$let\ parseCodePegOption = \{(\text{"1"}, \text{Orange}), (\text{"2"}, \text{Green}), (\text{"3"}, \text{Blue}), (\text{"4"}, \text{Yellow}), (\text{"5"}, \text{Purple}), (\text{"6"}, \text{Brown})\} \cup \{x: string \mid x \notin \{\text{"1"}, \text{"2"}, \text{"3"}, \text{"4"}, \text{"5"}, \text{"6"}\} \cdot (x, None)\}$$

str

Explanation:

Throughout gameplay, we would want to print the string representation of each Code peg, Hint peg, and Player.

Axiomatic definition:

$$str: Code \cup Hint \cup Player \rightarrow string$$

generateSecretCode

Explanation:

This function takes in the MainMenuOption in order to govern how the SecretCode is formulated. Using the available Code Pegs, Singleplayer would generate a normal combination, Multiplayer prompts for 4 Code values through receiveCodePegInput, and Campaign will generate 3 normal combinations and return them as a tuple due to the staged nature of the gameplay.

Axiomatic definition:

$$generateSecretCode: MainMenuOption \rightarrow Union[SecretCode, tuple[SecretCode, SecretCode, SecretCode]]$$

startSinglePlayer

Explanation:

Creating a separate function for Single Player would begin with generating the appropriate Secret Code. The guessing process is then initiated to prompt for the 4 Code Pegs, which will then be used to generate Feedback and the game's finished state. The Board is then updated and displayed with the recent guess and feedback made. Using the game's finished state, decision making to whether the whole sequence of instructions is to be repeated is governed or if the game is ended early to announce the overall winner.

startMultiplayer

Explanation:

Similarly to how startSinglePlayer, however, when it comes to generating the Secret Code, the user input is required.

getFeedback

Explanation:

In a Mastermind game, along with the CodeBreaker's Guess, the Guess's Feedback is also shown on the game Board. In getFeedback, we want to create a tuple of four Code pegs, consisting of Red, White, and Empty hint pegs depending on how close the CodeBreaker's Guess is to the Secret code.

Axiomatic definition:

$$getFeedback: Guess, SecretCode \rightarrow (bool, Feedback)$$

$$let\ getFeedback = \{a: Guess; b: SecretCode \mid a == b \cdot (True, Feedback)\}$$
$$\cup\ \{a: Guess; b: SecretCode \mid a\ != b \cdot (False, Feedback)\}$$

displayBoard

Explanation:

This function displays the board of the current game. It will take in the Board and output it in the terminal, not returning anything.

Axiomatic definition:

$$displayBoard : Board \rightarrow void$$

Display:

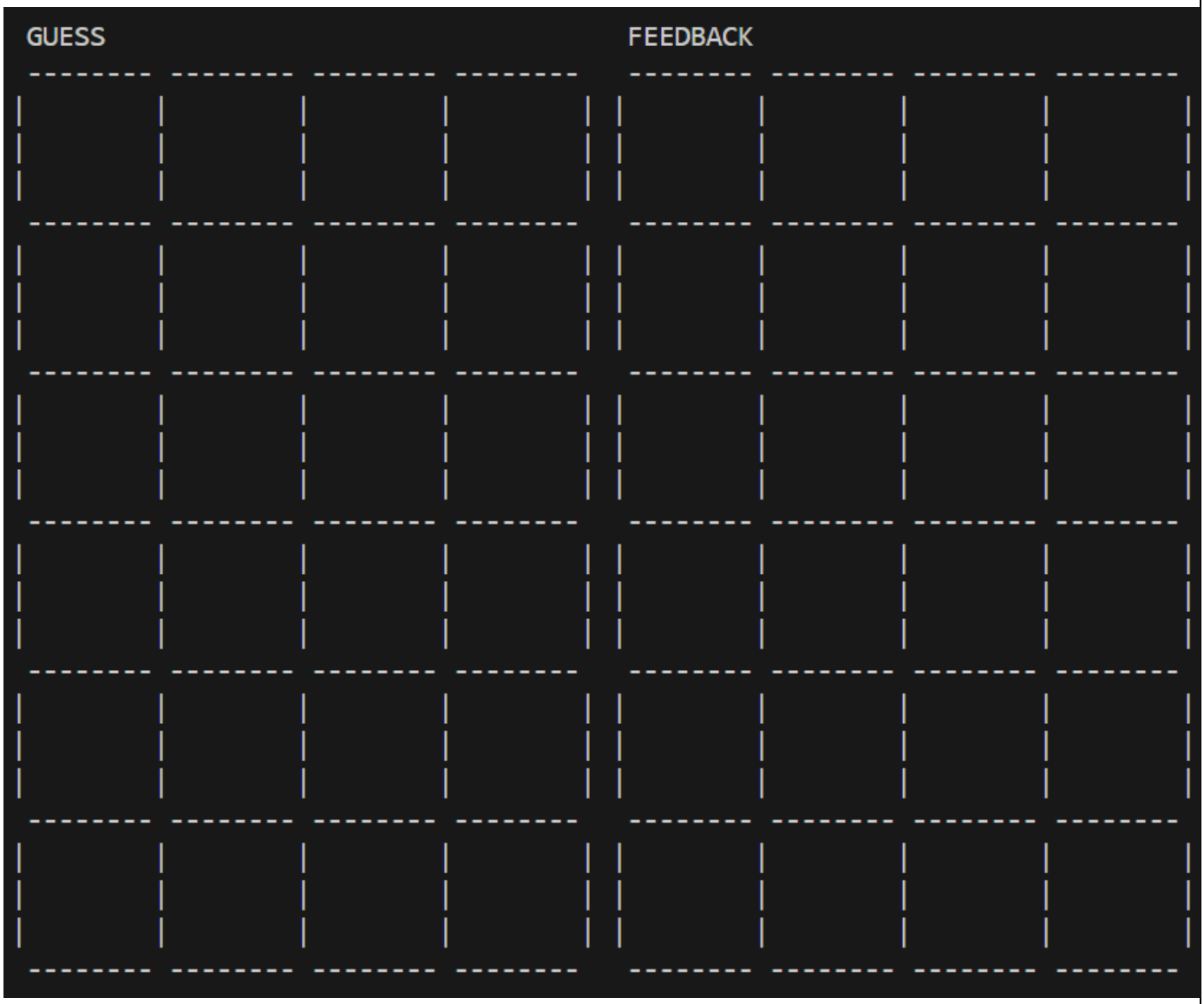


Figure 4 What we want the board to look like when displayed.

printInColour

Explanation:

This function returns a string containing the corresponding Code or Hint peg and their ANSI escape code colour's string. For Empty Code and Hint pegs, we don't want any string printed when displayBoard is called, so we assign them a blank.

Axiomatic definition:

$$printInColour: Code \cup Hint \rightarrow string$$

$$\begin{aligned} let\ printInColour = \{x: Code \cup Hint \mid x = Orange \cdot \text{"\033[38;5;208morange\033[0m"} \\ \vee x = Green \cdot \text{"\033[38;5;82mgreen\033[0m"} \\ \vee x = Blue \cdot \text{"\033[28;5;12mblue\033[0m"} \\ \vee x = Yellow \cdot \text{"\033[28;5;184myellow\033[0m"} \\ \vee x = Purple \cdot \text{"\033[28;5;134mpurple\033[0m"} \\ \vee x = Brown \cdot \text{"\033[28;5;94mbrown\033[0m"} \\ \vee x = White \cdot \text{"\033[28;5;15mwhite\033[0m"} \\ \vee x = Red \cdot \text{"\033[28;5;124mred\033[0m"} \\ \vee x = Empty \cdot \text{" "}\} \end{aligned}$$

Display:

Figure 5 What we want the return strings to look like when printed.

updateBoard

Explanation:

Using the Guess made by the CodeBreaker and generated Feedback, a Row can be created in the form of a tuple containing them. This Row can be appended on to the existing game board we pass in.

To maintain the declarative aspect of the program, an integer type parameter is used to decide what index of the game board that the new Row should be added to, without having to use a variable and changing its state through appending and re-declaration of the variable.

Axiomatic definitions:

$$updateBoard: Board, Guess, Feedback \rightarrow Board$$
$$updateBoard: Board, Guess, Feedback, int \rightarrow Board$$

$$\begin{aligned} let\ receiveConfirmInput = \{x: ConfirmationOption \mid x == Yes \cdot True\} \cup \\ \{x: ConfirmationOption \mid x == No \cdot False\} \end{aligned}$$

Stage 2 – Improvement Phase

Stage 2 was our revised plan for our Mastermind program. A second stage of planning and design was needed after discussion with a Tutor, prompting improvement on certain aspects of our implementation, as well as the need to ensure our program was aligned with the style of declarative and functional programming.

In terms of gameplay, separate functions for game modes has been revised to cluster their functionalities all into one function instead and will initiate the decision making process.

Initially, we had planned to have huge chunks of code made for each game mode of our Mastermind program, however, after discussion with a Tutor, it was realised that a better approach would be to make separate functions that could be reused within the whole program. Instead of having three functions like SinglePlayerSecret, MultiPlayerSecret, and CampaignSecret, we chose to create individual functions based on the different ways a Secret code could be made:

- normalSecretCode for automatically generated Secret codes with 4 unique Code pegs
- makeSecretCode for a CodeMaker-made Secret code
- hardSecretCode for automatically generated Secret codes with 2 unique Code pegs and another Code peg pair

normalSecretCode

Explanation: For the SinglePlayer mode and stage 1 of the Campaign mode, we want to create a Secret code with no duplicates, that is easy to guess.
Axiomatic definition: $$normalSecretCode: void \rightarrow Secret$$

makeSecretCode

Explanation: Prompting for a Secret Code, receiveCodePegInput will prompt for an individual Code Peg choice. Until there the size of the Secret Code being formulated is of a value of four Code Pegs, the receiveCodePegInput will be called to prompt for the final Code Peg choice.
Axiomatic definition: $$makeSecretCode: int \rightarrow SecretCode$$

hardSecretCode

Explanation: For stages 2 and 3 of the Campaign mode, we want to increase the difficulty of guessing the Secret code. So, for a harder Secret code, we will create a Secret code with 2 unique Code pegs and a Code peg pair of the same, 3rd unique, colour.
Axiomatic definition: $$hardSecretCode: void \rightarrow Secret$$

startGameplay

Explanation:

Prior to deciding the amount of guesses available to be made in a single game for a game mode, startGameplay decides on the flow of how many games are played for a specific game mode. In the case that Single Player or Multiplayer is selected, only one game is played, whilst Campaign will govern whether or not another game is played, which depends on the game state of a previous game.

Axiomatic definition:

$$\frac{\begin{array}{c} startGameplay: MainMenuOption, \\ Union[Board, tuple[Board, Board, Board]], tuple[Player, Player], \\ Union[SecretCode, tuple[SecretCode, SecretCode, SecretCode]], bool, int \rightarrow None \end{array}}{\begin{array}{l} let\ startGameplay = \{x: MainMenuOption \mid x == SinglePlayer \vee Multiplayer \cdot \\ playGame(x, Board, \\ tuple[Player, Player], SecretCode)\} \cup \\ \{x: MainMenuOption; y: bool \mid x == Campaign \wedge y == True \cdot playGame(x, tuple[Board], \\ tuple[player], tuple[SecretCode], 1, int)\} \end{array}}$$

playGame

Explanation:

Depending on the game mode chosen, playGame decides whether to let the CodeBreaker make another Guess whilst governing the limit to the number of guesses that can be within a singular game mode. Campaign mode uses the stage currently being played do decide when to reduce the number of guesses available when it comes to the final stage.

Axiomatic definition:

$$\begin{array}{c} playGame: MainMenuOption, Board, tuple[Player, Player], \\ SecretCode, int, int \rightarrow tuple[bool, Board] \end{array}$$

playRound

Explanation:

Assuming we have either started the game or have made a Guess previously, playRound is called to carry out the set of instructions to make and confirm a Guess to attempt at guessing the Secret Code. When a Guess is being made, a running prompt occurs to obtain a Code Peg choice through getGuess. We can confirm the Guess made by calling receiveConfirmationInput, which will run getGuess again if not confirmed. Confirming will generate Feedback, update the existing Board, display it, and return the current state of the Board and the game.

Axiomatic definition:

$$\frac{playRound: Board, Secret, int \rightarrow tuple[Board, bool]}{\begin{array}{l} let\ playRound = \{x: ConfirmationOption \mid x == y \cdot tuple[Board, bool]\} \cup \\ \{x: ConfirmationOption \mid x == n \cdot getGuess\} \end{array}}$$

endGame

Explanation:

This function will reveal the Secret Code and who has won the game. Using the game's finished state, we can govern which of the players of the game session is displayed as winner.

Considering the way in which Campaign mode works, whilst displaying the winner of the overall game, using the game mode, the current stage being played, and the game's finished state, we can display appropriately a message to indicate if a stage has been completed, the campaign game being completed, or failure to completing the campaign.

Axiomatic definition:

$$endGame: MainMenuOption, int, bool, Players, Secret \rightarrow void$$

$$let\ endGame = \{x: bool; y \in Players \mid x == True \cdot print(y)\} \cup \{x: bool; y \in Players \mid x == True \cdot print(y)\}$$

$$let\ endGame = \{x: MainMenuOption; y: int; z: bool \mid x == Campaign \wedge y == 2 \wedge z == True \cdot print()\} \cup \{x: MainMenuOption; y: int; z: bool \mid x == Campaign \wedge y < 2 \wedge z == True \cdot print()\} \cup \{x: MainMenuOption;\ z: bool \mid x == Campaign \wedge z == True \cdot print()\}$$

During our first stage of implementation, we realised we needed to ensure that our program stuck to the declarative and functional programming style, causing us to start a second planning stage. After this, we agreed that getFeedback would be made up of smaller functions:

- getRedHints
- getWhiteHints
    - checkIfDupe
    - checkGuessedCorrectly
    - checkAlmostGuessed
    - occursOnce
    - occursTwice
    - checkThroughGuess
- joinHints
- sortHints

getRedHints

Explanation:

This function will return a list of tuples which contain a Boolean value, indicating whether or not the peg was correctly guessed, as well as the peg that was guessed, for every peg in the CodeBreaker's Guess.

Axiomatic definition:

$$getRedHints: Guess, SecretCode \rightarrow [(bool, Code)]$$

$$let\ getRedHints = \{x: Guess, y: Secret \mid \forall i\ \in \{0,1,2,3\}\ \cdot ((x[i] = y[i] \Rightarrow outputList[i] = (True, x[i])) \wedge (x[i] \neq y[i] \Rightarrow outputList[i] = (False, x[i])\ \cdot outputList\}$$

getWhiteHints

Explanation:

This function recursively assigns Hint pegs to the CodeBreaker's Guess to get Feedback. It returns a list of tuples of Boolean and Code, just like the getRedHints function, but for White Hint pegs.

The base case of this function is when there are no elements left in the tuple guessLeft, and the list runningFeedback is returned.
If the base case has not yet been reached, this function returns a call of itself with the parameters of the tuple guessLeft, the list runningFeedback (the building of a list containing which pegs can be assigned a White Hint peg), the Secret code, and the list redPegs.

The 1st list being used as y in this definition is the list runningFeedback.

Axiomatic definition:

$$checkThroughGuess: tuple, list, Secret, list \rightarrow list$$

$$let\ checkThroughGuess = \{x: tuple, y: list \mid x = () \cdot y\} \cup \{x: tuple, y: list, z: Secret, a: list \mid x \neq () \cdot checkThroughGuess(x, y, z, a)\}$$

$$getWhiteHints: Guess, SecretCode, list \rightarrow [(bool, Code)]$$

When deciding on which Code pegs can be assigned White Hint pegs, there are many factors to consider in regard to our current plan:

- Is the Code peg currently being looked at a duplicate in a Secret code generated by hardSecretCode?
- Has the current Code peg already been assigned a Red Hint peg?
- Has the current Code peg already been assigned a White Hint peg?
- If the Code peg occurs twice in the Secret code, have both already been assigned a non-Empty Hint peg?

Consequently, to ensure that this function leans towards declarative and functional programming, getWhiteHints is made up of smaller functions:

- checkIfDupe
- checkGuessedCorrectly
- checkAlmostGuessed
- occursOnce
- occursTwice

checkIfDupe

Explanation:

This function returns a Boolean value based on whether the current Code peg being checked from the CodeBreaker's Guess is the duplicated peg of a hard mode Secret code (the Secret code in the 2nd and 3rd stage of a Campaign).

Axiomatic definition:

$$checkIfDupe: Code, Secret \rightarrow bool$$

$$let\ checkIfDupe = \{x: Code, y: Secret \mid |(\forall i \in \{0,1,2,3\} \mid x[i] = y[i])| = 2 \cdot True\} \cup \{x: Code, y: Secret \mid |(\forall i \in \{0,1,2,3\} \mid x[i] = y[i])| \neq 2 \cdot False\}$$

checkGuessedCorrectly

Explanation:

This function ensures that a White Hint will not be assigned to a Code peg that has already been assigned a Red Hint in the list redPegs. It counts how many times the current Code peg has been correctly matched to the SecretCode.

Axiomatic definition:

$$checkGuessedCorrectly: Code, list \rightarrow int$$

$$let\ checkGuessedCorrectly: \{x: (True, Code), y: list \mid x \in y \cdot |(\forall i \in \{0,1,2,3\} \mid x[i] = y[i])|\} \cup \{x: (True, Code), y: list \mid x \notin y \cdot 0\}$$

checkAlmostGuessed

Explanation:

This function ensures that a White Hint will not be assigned to a Code peg that has already been assigned a White Hint in the list runningFeedback. It counts how many times the current Code peg has been almost matched to the SecretCode in the running feedback of the checkThroughGuess recursive function.

Axiomatic definition:

$$checkAlmostGuessed: Code, list \rightarrow int$$

$$let\ checkAlmostGuessed: \{x: (True, Code), y: list \mid x \in y \cdot |(\forall i \in \{0,1,2,3\} \mid x[i] = y[i])|\} \cup \{x: (True, Code), y: list \mid x \notin y \cdot 0\}$$

occursOnce

Explanation:

This function checks whether a White Hint peg can be assigned to a Code peg from the CodeBreaker's Guess if it only occurs once in the Secret code.

If the Code peg has not already been assigned a Red Hint peg, and has not already been assigned a White Hint peg, it is valid to assign a White Hint peg to it for feedback.

The first list to be input to this function is the redPegs list containing whether the Code pegs can be assigned Red Hint pegs or not, and the second list represents the runningFeedback formed within checkThroughGuess.

Axiomatic definition:

$$occursOnce: Code, list, list \rightarrow (bool, Code)$$

$$let\ occursOnce: \{x: Code, y: list, z: list \mid checkGuessedCorrectly(x, y) = 0 \wedge checkAlmostGuessed(x, z) = 0 \cdot (True, x)\} \cup \{x: Code, y: list, z: list \mid \neg\ (checkGuessedCorrectly(x, y) = 0 \wedge checkAlmostGuessed(x, z) = 0 \cdot (False, x)\}$$

occursTwice

Explanation:

This function checks whether a White Hint peg can be assigned to a Code peg from the CodeBreaker's guess if it occurs twice in the SecretCode (in this case, this will mean it is part of the hard Secret code).

If the peg has not already been assigned two Red Hint pegs or two White Hint pegs or one Red Hint and one White Hint peg, it is valid to assign a White Hint peg to it for feedback.

The first list to be input to this function is the redPegs list containing whether the Code pegs can be assigned Red Hint pegs or not, and the second list represents the runningFeedback formed within checkThroughGuess.

Axiomatic definition:

$$occursTwice: Code, list, list \rightarrow (bool, Code)$$

$$\frac{}{let\ occursOnce: \{x: Code, y: list, z: list \mid checkGuessedCorrectly(x, y) + checkAlmostGuessed(x, z) < 2 \cdot (True, x)\} \cup \{x: Code, y: list, z: list \mid \neg\ (checkGuessedCorrectly(x, y) + checkAlmostGuessed(x, z) < 2 \cdot (False, x)\}}$$

formatRow

Explanation:

This function was created in the second stage of planning; it aims to format a given Row from a Board so it can be displayed by the displayBoard function.

Axiomatic definition:

$$formatRow: Row \rightarrow list[string]$$

Display:

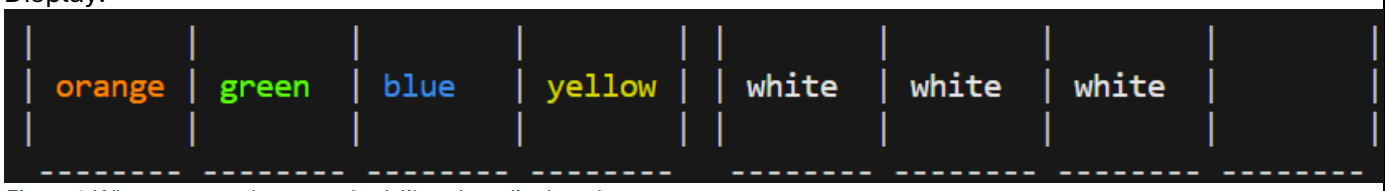


Figure 6 What we want the row to look like when displayed.

formatPeg

Explanation:

This function was also created in the second stage of planning; it aims to format a given Code or Hint peg so it can be displayed by the displayBoard function.

Axiomatic definition:

$$formatPeg: Code\ \cup Hint \rightarrow string$$

$$\frac{}{let\ formatPeg: \{x: Code\ \cup Hint \mid |str(x)| = 6 \cdot ("|"\ +\ printInColour(x) + "\ ")\} \cup \{x: Code\ \cup Hint \mid |str(x)| \neq 6 \cdot ("|"\ +\ printInColour(x)\ + ("\ " * (6 - |str(x)|)) + "\ ")\}}$$

Display:

Figure 7 What the formatted peg would look like when displayed.

The Mastermind Function Dependency Graph

Our implementation of a Mastermind program can be represented by this function dependency graph.

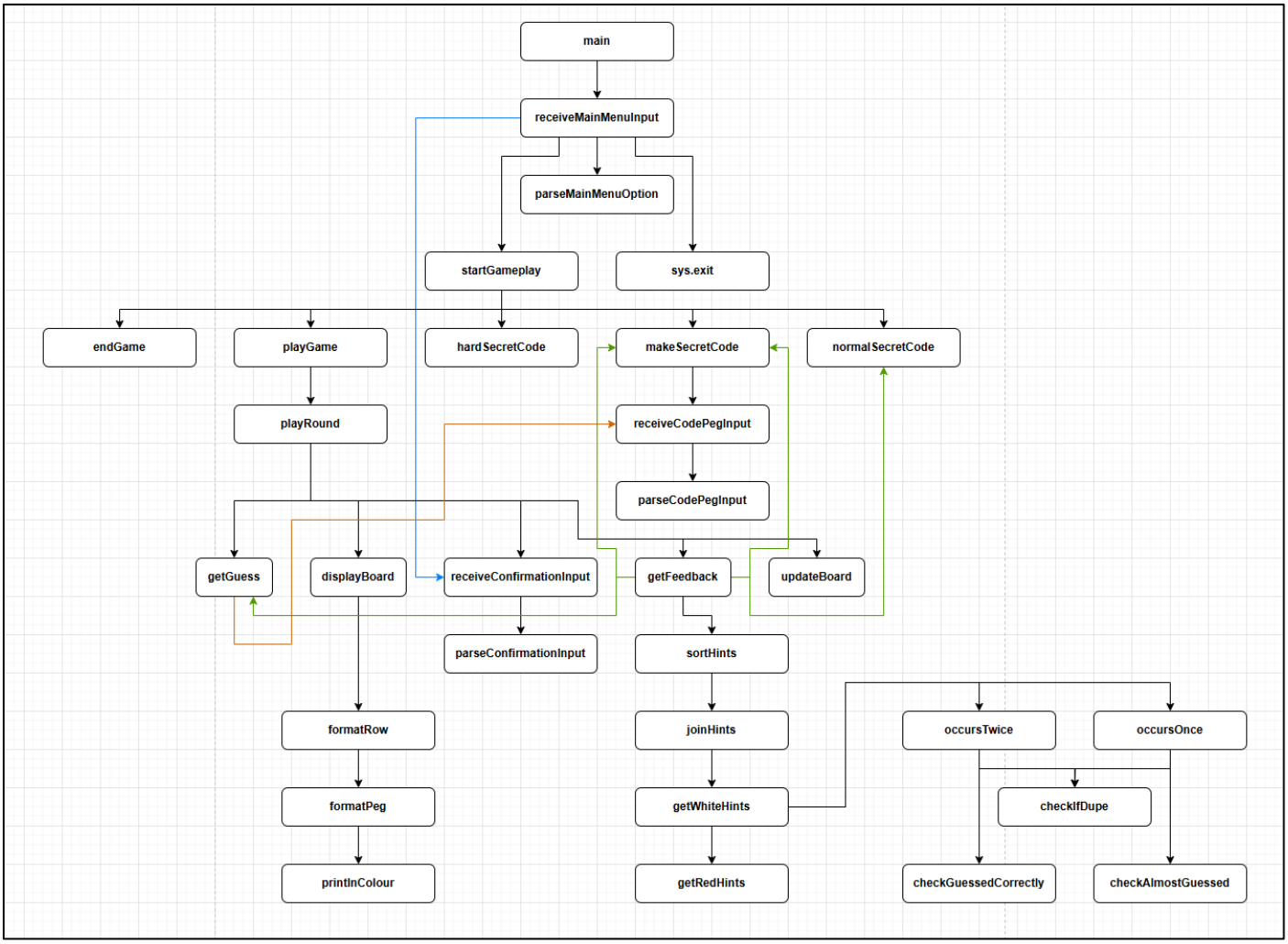


Figure 8 A dependency graph indicating which functions are needed to fully execute a certain function.

T2 – Program Implementation

Imports

The following imports were made:

Module Name	Purpose
Sys	Call for system exit.
Random	Generates random samples for **Secret** codes.
Dataclass	Ensure that some type classes such as **Player** types are immutable.
Enum	To enumerate type classes such as **Code**, **Hint**, **Confirmation**, and **Menu**.
Type, TypeAlias, Union	Aids with Type hinting.

```
from __future__ import annotations
import sys, random
from dataclasses import dataclass
from enum import Enum
from typing import Type, TypeAlias, Union
```

Types

For all the types in T1, we stuck to our plan. Therefore, the Python code snippets in T1 are the same as you will see in T2.

Menu

For the game options / main menu options, we decided to use enums to allow for an easy error handling of input – the **Player** only needs to input 1, 2, 3, or 4 when prompted in the main menu to select a **Menu** option.

```
class Main_Menu_Option(Enum):
    Single_Player = 1
    Multiplayer = 2
    Campaign = 3
    Exit = 4

Menu: TypeAlias = Main_Menu_Option
```

Confirm

Again, we did the same for the Confirm type. In **parseConfirmationOption**, we use the built-in strip method to aid in easy error handling. So, the **Player** can input Y, N, y, n for a valid **Confirm** option.

```
class Confirmation_Option(Enum):
    Yes = 'y'
    No = 'n'
```

```
Confirm: TypeAlias = Confirmation_Option
```

Code

We used enums to allow the user to only input 1, 2, 3, 4, 5, 6 when selecting a **Code** peg for their **Guess** or **Secret** code choice.

```
class Code_Peg_Option(Enum):
    Empty = 0
    Orange = 1
    Green = 2
    Blue = 3
    Yellow = 4
    Purple = 5
    Brown = 6

Code: TypeAlias = Code_Peg_Option
```

Hint

We also used enums for **Hint** pegs, although after some reflection, we could have used a normal class, like we did with the **Player** type.

```
class Hint_Peg(Enum):
    Empty = 0
    White = 1
    Red = 2

Hint: TypeAlias = Hint_Peg
```

Player, and Players

For **Player**, we used frozen data classes to ensure that they are immutable. We also made the **Players** type immutable by defining them using tuples.

```
@dataclass(eq=True, frozen=True)
class CodeMaker:
    pass

@dataclass(eq=True, frozen=True)
class CodeBreaker:
    pass

@dataclass(eq=True,frozen=True)
class CPU:
    pass

Player : TypeAlias = CodeMaker | CodeBreaker | CPU
Players : TypeAlias = tuple[CodeBreaker, CodeMaker] | tuple[CodeBreaker, CPU]
```

Secret

We composed **Secret** code using four **Code** peg types, encasing them within a tuple, making them unchangeable.

```
Secret: TypeAlias = tuple[Code, Code, Code, Code]
```

Guess

Like for **Secret**, we made **Guess** in the same way, using a tuple of four **Code** peg types.

```
Guess: TypeAlias = tuple[Code, Code, Code, Code]
```

Table 1 The Python implementation of the Guess type.

Feedback

Similarly, we made **Feedback** using a tuple of four **Hint** peg types.

```
Feedback: TypeAlias = tuple[Hint, Hint, Hint, Hint]
```

Table 2 The Python implementation of the Feedback type.

Row

To represent a **Row**, we merged **Guess** and **Feedback** in a tuple, just like they are displayed on the game **Board**.

```
Row: TypeAlias = tuple[Guess, Feedback]
```

Table 3 The Python implementation of the Row type.

NormalBoard, HardBoard, and Board

In our program, we have a **NormalBoard**, and a **HardBoard**. A **NormalBoard** is a tuple made up of six **Rows**; a **HardBoard** is a tuple made up of four **Rows**.

A **Board** type can either be a **NormalBoard** or a **HardBoard**.

```
Normal_Board: TypeAlias = tuple[Row, Row, Row, Row, Row, Row]
Hard_Board: TypeAlias = tuple[Row, Row, Row, Row]
Board: TypeAlias = Normal_Board | Hard_Board
```

Table 4 The Python implementation of the NormalBoard, HardBoard, and Board types.

Program Functions

main

Explanation:
The construct ensures that the beginning of all instructions to be followed occur within it.
The **mastermind_intro** is displayed along with **receive_main_menu_input**, which is constantly called to initiate prompting for the **Main_Menu_Option** This is so that even after a **Mein_Menu_Option** is selected and has finished executing the **start_gamepla**, it reverts back to prompting for the **Main_Menu_Option**.

Code Snippet:

```
if __name__=="__main__":
    print(mastermind_intro)

    while True:
        receive_main_menu_input()
```

receiveMainMenuInput

The function runs an ongoing input prompt for the User to enter a value.

Decision Logic Step	Description
1	If a value of 1 for **Single_Player** is the **selected_option**, **start_gameplay** is called passing **selected_option**, **empty_normal_board**, a tuple containing players **CodeBreaker** and **CPU**, and a **normal_secret_code** call.
2	If a value of 2 for **Multiplayer** is the **selected_option**, **make_secret_code** is called to prompt the Code Maker. Once a **Secret** is returned, **receive_confirmation_input** is called passing the **custom_secret_code** to it.
2.1	If **confirmation_choice** is True, **check_two_dupe_secret** and **check_one_dupe_secret** are called to check the Secret Code before calling **start_gameplay** to pass the **selected_option**, **empty_normal_board**, a tuple containing **CodeBreaker** and **CodeMaker**, and custom_secret_code.
2.2	If **confirmation_choice** is False, **make_secret_code** is called again until **confirmation_choice** is True.
3	If a value of 3 for **Campaign** is the **selected_option**, **start_gameplay** is called passing **selected_option**, a tuple containing two **empty_normal_board** and **empty_hard_board**, a tuple containing players **CodeBreaker** and **CPU**, and a tuple containing a **normal_secret_code** call and two **hard_secret_code** calls.
4	If a value of 4 for **Exit** is the **selected_option**, **sys.exit()** is called and ends the whole program.

```
def receive_main_menu_input() -> None:
    print(main_menu_options)
    selected_option = Main_Menu_Option.parse_main_menu_option(input("> "))
    print()

    match selected_option:
        case Main_Menu_Option.Single_Player:
            start_gameplay(selected_option, empty_normal_board, (CodeBreaker(), CPU()),
normal_secret_code())
        case Main_Menu_Option.Multiplayer:
            while True:
                print("\nCODEMAKER: ENTER SECRET CODE ---------------------- ")
                custom_secret_code: Secret = make_secret_code()
                confirmation_choice: bool =
receive_confirmation_input(custom_secret_code)
                if confirmation_choice == True:
                    if check_two_dupe_secret(custom_secret_code) == True:
                        print(f"\nINVALID MESSAGE INPUT --------------------\nYou can't
have more than two of the same Code Peg in the Secret Code.")
                    else:
                        if check_one_dupe_pair_secret(custom_secret_code) == True:
                            print(f"\nINVALID MESSAGE INPUT --------------------\nYou
can't have more than one pair of Duplicates in the Secret Code.")
                        else:
                            break
            start_gameplay(selected_option, empty_normal_board, (CodeBreaker(),
CodeMaker()), custom_secret_code)
        case Main_Menu_Option.Campaign:
            start_gameplay(selected_option, (empty_normal_board, empty_normal_board,
empty_hard_board), (CodeBreaker(), CPU()), (normal_secret_code(), hard_secret_code(),
hard_secret_code()))
        case Main_Menu_Option.Exit:
            print("Exiting Mastermind...")
            sys.exit()
```

parseMainMenuOption

Explanation:

This function is a method of the MainMenuOption class that parses the user's input, ensuring they input a '1', '2', '3', '4', when prompted in **receiveMainMenuInput**. The return value is then checked in **receiveMainMenuInput**.

Code Snippet:

```
@classmethod
    def parse_main_menu_option(cls : Type[Main_Menu_Option], input : str) ->
Main_Menu_Option:
        try:
            main_menu_option: Main_Menu_Option = Main_Menu_Option(int(input))
            return main_menu_option
        except ValueError:
            print("That is an invalid main menu choice.")
            return None
```

receiveConfirmationInput

Explanation:
This function displays the Guess or Secret Code made before calling this function and runs an ongoing input for the **Confirmation_Option** to be entered.
If **selected_option** is Yes, Boolean True is returned to indicate the Guess or Secret Code made is to be confirmed. Alternatively, if No, Boolean True is returned to indicate a decline to the Guess or Secret Code made.

Code Snippet:

```
def receive_confirmation_input(choice: Union[Guess, Secret]) -> bool:

    """
    receives the Player's parsed Confirm_Option and prints out the appropriate message
to the command-line

    Parameters:
        choice (Union[Guess, Secret]) - The user's choice of Guess / Secret code

    Returns:
        bool - Whether the Player chose Yes or No

    """

    while True:
        print(confirmation_options)
        print("Your Choice: ")
        print(*(print_in_colour(peg) for peg in choice))
        print("Are you sure you want to continue?")
        selected_option = Confirmation_Option.parse_confirmation_option(input("> "))
        print()

        match selected_option:
            case Confirmation_Option.Yes:
                print("\nChoice Confirmed.")
                return True
            case Confirmation_Option.No:
                print("\nChoice Cancelled.")
                return False
```

parseConfirmationOption

Explanation:
This function displays the Guess or Secret Code made before calling this function and runs an ongoing input for the **Confirmation_Option** to be entered.
If **selected_option** is Yes, Boolean True is returned to indicate the Guess or Secret Code made is to be confirmed. Alternatively, if No, Boolean True is returned to indicate a decline to the Guess or Secret Code made.

Code Snippet:

```
@classmethod
    def parse_confirmation_option(cls : Type[Confirmation_Option], input : str) ->
Confirmation_Option:
        try:
            lower_confirmation_input: str = input.lower().strip()
            confirmation_option: Confirmation_Option =
Confirmation_Option(lower_confirmation_input)
            return confirmation_option
        except ValueError:
            print("That is an invalid confirmation choice.")
            return None
```

receiveCodePegInput

Explanation:

This function receives the **Player**'s parsed **Code** peg input and prints the appropriate message to the command-line, returning the **Code** peg type to its parent function.
A while loop runs until the Player chooses a valid CodePegOption (not None).

Code Snippet:

```
def receive_code_peg_input() -> Code:
    while True:
        print(code_peg_options)
        selected_option = Code_Peg_Option.parse_code_peg_option(input("> "))

        if selected_option:
            match selected_option:
                case Code_Peg_Option.Orange:
                    print("You have chosen an " + print_in_colour(selected_option) + "
peg.")
                case _:
                    print("You have chosen a " + print_in_colour(selected_option) + "
peg.")
            return selected_option
        else:
            print("Invalid peg choice.")
```

parseCodePegOption

Explanation:

This function is a method of the CodePegOption class that parses the user's input, ensuring they input a '1', '2', '3', '4', '5', '6' when prompted for their Code peg input. The return value is then checked in **receiveCodePegInput**.

Code Snippet:

```
@classmethod
    def parse_code_peg_option(cls : Type[Code_Peg_Option], input : str) ->
Code_Peg_Option:
        try:
            code_peg_option: Code_Peg_Option = Code_Peg_Option(int(input))
            match code_peg_option:
                case Code_Peg_Option.Empty:
                    return None
```

```
                case _:
                    return code_peg_option
        except ValueError:
            print("That is an invalid code peg choice.")
            return None
```

__str__

For the types Code, Hint and Player, throughout the program we have to reference to them by their string representations, hence, within their respective classes we made a __str__ function.

```
def __str__(self) -> str:
        return self.name.lower()
```

```
def __str__(self) -> str:
        return self.name.lower()
```

```
def __str__(self) -> str:
        return "Code Maker"
```

```
def __str__(self) -> str:
        return "Code Breaker"
```

```
def __str__(self) -> str:
        return "CPU"
```

normalSecretCode

Explanation:

This is used to generate a **SecretCode** for SingePlayer, Multiplayer and round 1 of Campaign game modes. It stores all possible **Code** pegs that can be chosen from (from 1-6) in a variable called **validCodePegs** and chooses 4 unique random **Code** pegs from that list to make up the new **SecretCode**.

List comprehension is used to list which **Code** pegs are valid to choose from, being stored in **validCodePegs**.

Random is then used from import to generate a sample of 4 unique **Code** pegs from the **validCodePegs** list, being stored as a tuple in **newSecretCode** to form a **Secret** code.

Code Snippet:

```
def normal_secret_code() -> Secret:
    valid_code_pegs: list[Code] = [peg for peg in Code if str(peg) != "empty"]
    newSecretCode: Secret = tuple(random.sample(valid_code_pegs, k=4))
    return newSecretCode
```

makeSecretCode

Explanation:

This function prompts for four **Code_Peg_Option** by calling **receive_code_peg_input** within a tuple and joining the return value of recursively calling **make_secret_code** with **secret_size** incremented by 1 passed into it.
If the **secret_size** equals 4, **receive_code_peg_input** will be called for the fourth and final time before returning the **Secret**.

Code Snippet:

```
def make_secret_code(secret_size: int = 1) -> Secret:
    print()
    print(f"---------- CODE PEG CHOICE NO.#{secret_size} ----------")
    if secret_size == 4:
        return (receive_code_peg_input(),)
    else:
        return (receive_code_peg_input(),) + make_secret_code(secret_size+1)
```

hardSecretCode

Explanation:
This is used to generate a **SecretCode** in rounds 2 and 3 of the Campaign mode, consisting of 2 unique **Code** pegs, and a single pair of duplicate **Code** pegs.

Again, list comprehension is used to list which **Code** pegs are valid to choose from, being stored in **validCodePegs**.
Random is used from import to generate a sample of 3 unique **Code** pegs and stores this in the list **noDuplicateSample**.
The third peg in **noDuplicateSample** is taken and added onto **noDuplicateSample**, stored in **duplicateCode**.
This list is shuffled by random from import, utilising the sample function, and turned into a tuple to form a **Secret** code in **newSecretCode**.

Code Snippet:

```
def hard_secret_code() -> Secret:
    valid_code_pegs: list[Code] = [peg for peg in Code if peg != Code(0)]
    no_duplicate_sample: list[Code] = random.sample(valid_code_pegs, k=3)
    duplicate_code: list[Code] = no_duplicate_sample + [no_duplicate_sample[2]]
    newSecretCode: Secret = tuple(random.sample(duplicate_code, k=4))
    return newSecretCode
```

startGameplay

Explanation:
This function decides on the way secret code is generated as well as the amount of games sessions to be played. Singleplayer and Multiplayer executes only one game session before displaying the final game results, whereas Campaign will go through the process of checking the success status returned to **game_stage** in order to decide whether another game session is to be executed or to display the final game results.

Steps	Description
1	**IF MainMenuOption** is **Single_Player OR Multiplayer**:
1.1	The **playGame** function is called within a variable passing a **game_mode**, **game_board**, tuple containing two **Player** types, and **secret_code**.

1.2	Assuming that we have the a **Boolean** type value returned from calling **playGame**, we then call **end_game** passing the **Boolean** type value and the **players**.
1.3	The Terminal will display the overall winner of that gameplay.
1.4	The **startGameplay** function ends.
2	**IF MainMenuOption** is “Campaign” and **campaign_flag** == True:
2.1	The **playGame** function is called within a variable passing a **game_mode**, a specific **Board** from **game_board** using **current_stage** to access an index value, **players**, and a specific **Secret** from **secret_code** to access an index value.
2.2	IF the game’s finished state from **game_stage** is False **OR** current_stage being played is 2, display the current board and **end_game.**
2.3	ELSE call **start_gameplay** passing the **game_mode**, **game_board**, **players**, **secret_code**, the game finished state from **game_stage** as the **campaign_flag**, and **current_stage**.

```
def start_gameplay(game_mode: Main_Menu_Option, game_board: Union[Board, tuple[Board,
Board, Board]], players: tuple[Player, Player], secret_code: Union[Secret, tuple[Secret,
Secret, Secret]], campaign_flag: bool = True, current_stage: int = 0) -> None:
    print(f"""
                                                _____________
    ----------------------------------------| {game_mode.name.upper()} |----------------
------------------------
    ----------------------------------------|   GAME MODE   |---------------------------
-------------
    ----------------------------------------|_____________|---------------------------
-------------
            """)
    if game_mode == Main_Menu_Option.Single_Player or game_mode ==
Main_Menu_Option.Multiplayer:
        game_session: tuple[bool, Board] = play_game(game_mode, game_board, players,
secret_code)
        display_board(game_session[1])
        end_game(game_mode, current_stage, game_session[0], players, secret_code)

    elif game_mode == Main_Menu_Option.Campaign and campaign_flag == False:
        end_game(game_mode, current_stage, campaign_flag, players,
secret_code[current_stage])

    elif game_mode == Main_Menu_Option.Campaign and campaign_flag == True:
        print(f"------------------------------------------------------------------ STAGE
{current_stage+1} ------------------------------------------------------------------")
        game_stage: tuple[bool, Board] = play_game(game_mode, game_board[current_stage],
players, secret_code[current_stage], 1, current_stage)

        if game_stage[0] == False:
            display_board(game_stage[1])
```

```
            end_game(game_mode, current_stage, game_stage[0], players,
secret_code[current_stage])

        else:
            if current_stage == 2:
                end_game(game_mode, current_stage, campaign_flag, players,
secret_code[current_stage])
            else:
                end_game(game_mode, current_stage, game_stage[0], players,
secret_code[current_stage])
                return start_gameplay(game_mode, game_board, players, secret_code,
game_stage[0], current_stage+1)
```

playGame

Explanation: This function decides considering the **game_mode** selected how many attempts are available within a specific game. Singleplayer and Multiplayer limit to 6 guess attempts available, whereas Campaign, depending on the current stage being played will decide the whether to provide 6 or 4 guess attempts available.	
Steps	Description
1	**IF game_mode** is **Single_Player OR Multiplayer OR Campaign AND** current_stage is 0 **OR** 1:
1.1	**IF** the **turn_count** is above the set number of 7 or **game_finished** is **True**: A tuple containing the **game_finished** Boolean and the current **game_board** state is returned.
1.2	**ELSE** if both conditions are not the case, then **playRound** is called passing the current **game_board**, **secret_code**, and **turn_count**.
1.3	**play_game** is recursively called passing the parameters, however, the only different parameters passed through are the returned updated **Board** from **round** for the **Board** argument, the **turn_count** incremented by 1, and the **bool** returned into round as the **game_finished** argument. Steps 1.2 and 1.3 continue to be the case if the amount of guesses has not reached 7 or Game's Finished State is not True.
2	**IF game_mode** is **Campaign AND current_stage** is 2:
2.1	**IF** the **turn_count** is above the set number of 5 or **game_finished** is **True**: A tuple containing the **game_finished** Boolean and the current **game_board** state is returned.
2.2	**ELSE** if both conditions are not the case, then **playRound** is called passing the current **game_board**, **secret_code**, and **turn_count**.
2.3	**play_game** is recursively called passing the parameters, however, the only different parameters passed through are the returned updated **Board** from **round** for the **Board** argument, the **turn_count** incremented by 1, and the **bool** returned into round as the **game_finished** argument.

	Steps 2.2 and 2.3 continue to be the case if the amount of guesses has not reached 5 or Game's Finished State is not True.

```
def play_game(game_mode: Main_Menu_Option, game_board: Board, players: tuple[Player, Player], secret_code: Secret, turn_count: int = 1, current_stage: int = 0, game_finished: bool = False) -> tuple[bool, Board]:
    if game_mode == Main_Menu_Option.Single_Player or game_mode == Main_Menu_Option.Multiplayer or (game_mode == Main_Menu_Option.Campaign and current_stage == 0 or current_stage == 1):
        if turn_count == 7 or game_finished == True:
            return (game_finished, game_board)
        else:
            print(f"\n---------- GUESS ATTEMPT NO.#{turn_count} ----------")
            round: tuple[Board, bool] = play_round(game_board, secret_code, turn_count)
            return play_game(game_mode, round[0], players, secret_code, turn_count+1, current_stage, round[1])
    elif game_mode == Main_Menu_Option.Campaign and current_stage == 2:
        if turn_count == 5 or game_finished == True:
            return (game_finished, game_board)
        else:
            print(f"\n---------- GUESS ATTEMPT NO.#{turn_count} ----------")
            round: tuple[Board, bool] = play_round(game_board, secret_code, turn_count)
            return play_game(game_mode, round[0], players, secret_code, turn_count+1, current_stage, round[1])
```

playRound

Explanation:
This function runs the process of completing a single Guess to be made with the following steps being subsequent to it:

- Confirming the **Guess**
- Generating appropriate **Feedback**
- Updating the existing **Board**
- Returning the updated version of the game board with game's finished state

Steps	Description
1	Game Board is displayed through calling the **display_board** passing the **Board** type value.
2	The current Game Board appearance is displayed on Terminal.
3	A running guess prompt is initiated through **get_guess** within a variable to store the soon to be returned **Guess** value.
4	Confirmation choice is then prompted through calling **receive_confirmation_input**. If **confirmation_choice** == True, then steps 5 onwards occur. Alternatively, if **confirmation_choice** == False, then the process goes back to step 1 until **confirmation_choice** == True. This provides the option in the case that the Code Breaker changes their mind.

5	Feedback will then be generated through **get_feedback** being called within another variable to store a **Feedback** value.
6	Game Board value is updated by calling **update_board** within a variable to store the soon to be returned updated version of the **Board** value. Values to be passed are the existing **Board**, new **Guess**, and generated **Feedback** as arguments.
7	Return statement is made to return the updated version of the **Board** value and the Game's Finished State as a Boolean value produced from calling **getFeedback**.

```
def play_round(game_board: Board, secret_code: Secret, turn_count: int) -> tuple[Board,
bool]:
    while True:
        display_board(game_board)
        new_guess: Guess = get_guess()
        confirmation_choice: Confirmation_Option = receive_confirmation_input(new_guess)
        if confirmation_choice == True:
            new_feedback: Feedback = get_feedback(new_guess, secret_code)
            updated_board: Board = update_board(game_board, new_guess, new_feedback[1],
turn_count)
            display_board(updated_board)
            return (updated_board, new_feedback[0])
```

endGame

Explanation:

This function displays the Secret Code alongside the appropriate display of who won a game overall.

If the game_finished is True, the **CodeBreaker** is displayed to have won the game, whilst alternatively the **CPU**.

In the case of the **game_mode** being **Campaign** an additional result message is displayed during stage completions or fails or completing the whole campaign mode.

If **game_mode** is **Campaign**, **current_stage** is 2, and **game_finished** is True, the campaign mode is displayed to be successfully completed.

If the same condition is met for **game_mode** and **game_finished**, however the **current_stage** is less than 2, its displayed that you are to advance to the next level of the campaign mode.

If the same condition is met for **game_mode**, however, **game_finished** is False, its displayed that the player has failed campaign mode.

Code Snippet:

```
def end_game(game_mode: Main_Menu_Option, current_stage: int, game_finished: bool,
players: Players, secret: Secret) -> None:
    print("\n---------- SECRET CODE ----------")
    print(*(print_in_colour(peg) for peg in secret))

    print("\n---------- FINAL RESULTS ----------")
    match game_finished:
        case True:
            print(f"\n{str(players[0])} has won the game!")
        case False:
            print(f"\n{str(players[1])} has won the game!")
```

```
    # if game_mode is Campaign
    if game_mode == Main_Menu_Option.Campaign and current_stage == 2 and game_finished
== True:
        print("You have successfully completed your Campaign game!")
    elif game_mode == Main_Menu_Option.Campaign and current_stage < 2 and
game_finished == True:
        print("Well done! You have advanced to the next stage in your Campaign game!")
    elif game_mode == Main_Menu_Option.Campaign and game_finished == False:
        print("You have failed your Campaign game!")
```

getFeedback

Explanation:

This function takes in the **CodeBreaker**'s **Guess** and the **Secret** code to return whether the game has ended, and the **Feedback** regarding whether the **CodeBreaker** has guessed the **Secret** code correctly or not.

It first checks for the Red **Hint** pegs for **Feedback**, then the White **Hint** pegs for **Feedback**, and joins these **Feedback** lists, prioritising Red **Hint** pegs over White **Hint** pegs, filling in the rest of the pegs with Empty **Hint** pegs.

First, it checks whether **guess** is equal to **secret**, returning a **Feedback** of True (indicating the game has been won by the **CodeBreaker**) and a **Feedback** full of Red **Hint** pegs.

Else, **getFeedback** calls **getRedHints**, **getWhiteHints**, **joinHints**, and **sortHints** to form **Feedback**, returning the value of False along with it.

Code Snippet:

```
def get_feedback(guess: Guess, secret: Secret) -> tuple[bool, Feedback]:
    if guess == secret:
        return [True, tuple([Hint.Red] * 4)]
    else:
        red : list = get_red_hints(guess, secret)
        white : list = get_white_hints(guess, secret, red)
        feedback : list[Hint] = join_hints(red, white)
        final_feedback : Feedback = sort_hints(feedback)
        return [False, final_feedback]
```

getRedHints

Explanation:

This function goes through the **Guess** and **SecretCode** to check which **Code** pegs of a certain index are matched; these are returned to be True to indicate they must be in the **Feedback** as Red **Hint** pegs.

It uses list comprehension to build the returned list, consisting of tuples.

Each tuple contains a boolean value indicating if a Red **Hint** peg has been assigned, along with the current **Code** peg for that index.

Code Snippet:

```
def get_red_hints(guess: Guess, secret: Secret) -> list:
    return [(True, guess[i]) if guess[i] == secret[i] else (False, guess[i]) for i in
range(len(guess))]
```

getWhiteHints

Explanation:
This function recursively checks through the **CodeBreaker**'s **Guess**, returning the **guessLeft** to check and the **runningFeedback** every time until there are no more elements in the **guessLeft** tuple to check.

If the **currentPeg** is a duplicate **Code** peg within the **Secret** code, **occursTwice** is the function called.
Else, if the **currentPeg** is a singular **Code** peg within the **Secret** code, **occursOnce** is the function called.
In the case the base case is reached, this function returns the feedback list of tuples containing a boolean and **Code** peg value. This will later be known as the list **whitePegs**.

Code Snippet:

```
def get_white_hints(guess_left: tuple, running_feedback : list, secret: Secret,
red_pegs: list) -> list:
    if not guess_left:
        return running_feedback

    current_peg : Code = guess_left[0]

    if current_peg in secret:
        match check_if_dupe(current_peg, secret):
            case True:
                new_hint : tuple[bool, Code] = occurs_twice(current_peg, red_pegs,
running_feedback)

            case False:
                new_hint : tuple[bool, Code] = occurs_once(current_peg, red_pegs,
running_feedback)

    else:
        new_hint : tuple[bool, Code] = (False, current_peg)

    return get_white_hints(guess_left[1:], running_feedback + [new_hint], secret,
red_pegs)
```

joinHints

Explanation:
This function joins the Red **Hint** pegs list with the White **Hint** pegs list.
This function uses a mapping function, using lambda to iterate through both lists of **Hint** pegs.
If in **redPegs** the current index's first value within the tuple is True, a Red **Hint** peg is assigned to this index of the list to be returned.
Else, if in **whitePegs** the current index's first value within the tuple is True, a White **Hint** peg is assigned to this index of the list to be returned.
Else, an Empty **Hint** peg is assigned to this index of the list to be returned.

Code Snippet:

```
def join_hints(red_pegs : list, white_pegs : list) -> list[Hint]:
    return list(map(lambda i: Hint.Red if red_pegs[i][0] else Hint.White if
white_pegs[i][0] else Hint.Empty, range(4)))
```

sortHints

Explanation:
This function sorts the **Hints** for **Feedback** to ensure they are ordered from Red **Hints** to White **Hints** to Empty **Hints**, just as **Feedback** would be given in an actual game.

The function creates a dictionary called **feedbackOrder**, storing the ideal order of **Hint** pegs in **Feedback**.
It then uses the built-in sorted function to sort the list **feedback** using **feedbackOrder** as a key.
This is then turned into a tuple and returned as **Feedback**.

Code Snippet:

```
def sort_hints(feedback: list[Hint]) -> Feedback:
    feedback_order = {Hint.Red: 0, Hint.White: 1, Hint.Empty: 2}
    return tuple(sorted(feedback, key=lambda x: feedback_order[x]))
```

checkIfDupe

Explanation:
This pure function was made to check whether the current **Code** peg being checked is duplicated within the **SecretCode**. This would indicate that the **SecretCode** is a hard **SecretCode** and that there is a single pair of the same **Code** pegs within the **SecretCode**.
This function also helps determine which function out of **occursOnce** and **occursTwice** will be called in **checkThroughGuess**.
It returns a boolean value regarding whether it is a duplicate **Code** peg in the **Secret** code or not.

Code Snippet:

```
def check_if_dupe(peg : Code, secret : Secret) -> bool:
    return secret.count(peg) == 2
```

checkGuessedCorrectly

Explanation:
This function was created to check whether the **Code** peg being checked for White **Hint** peg assignment has already been assigned a Red **Hint** peg or not.

It uses the built in count function to count how many times the specified tuple appears in the list **redPegs** and returns this integer.

Code Snippet:

```
def check_guessed_correctly(peg : Code, red_pegs : list) -> int:
    return red_pegs.count((True, peg))
```

checkAlmostGuessed

Explanation:
This function was created to check whether the **Code** peg being checked for White **Hint** peg assignment has already been assigned a White **Hint** peg or not.

It uses the built in count function to count how many times the specified tuple appears in the list **runningFeedback** and returns this integer.

Code Snippet:

```
def check_almost_guessed(peg : Code, running_feedback : list) -> int:
    return running_feedback.count((True, peg))
```

occursOnce

Explanation:
This function checks whether the **Code** peg given meets the criteria to be assigned a White **Hint** peg or not, when it occurs only once in the **SecretCode**.

It returns the Boolean value regarding whether a Red or White **Hint** peg has already been assigned to the current **Code** peg.
If none have been assigned, return True, else, return False, along with the current **Code** peg.

Code Snippet:

```
def occurs_once(peg : Code, red_pegs : list, running_feedback : list) -> tuple[bool, Code]:
    return [(check_guessed_correctly(peg, red_pegs) == 0 and check_almost_guessed(peg, running_feedback) == 0), peg]
```

occursTwice

Explanation:
This function checks whether the **Code** peg given meets the criteria to be assigned a White **Hint** peg or not, when it occurs twice in the **SecretCode**.

It returns the Boolean value regarding whether a two **Hint** pegs have already been assigned to the current **Code** peg.
If less than two have been assigned, return True, else, return False, along with the current **Code** peg.

Code Snippet:

```
def occurs_twice(peg : Code, red_pegs : list, running_feedback : list) -> tuple[bool, Code]:
    return [((check_guessed_correctly(peg, red_pegs) + check_almost_guessed(peg, running_feedback)) < 2), peg]
```

displayBoard

Explanation:
As a whole, displayBoard takes in a **Board**, and prints out the header for the section, as well as the actual board, **Row** by **Row**.

The join method is used to join the lists of string (**header** and **board**) together, separating each element in the new joint list with “\n”.

The two new functions, formatRow and formatPeg were made to ensure a declarative and functional programming style, instead of using a for loop for the entire function.

Code Snippet:

```
def display_board(game_board: Board) -> None:
    header : list[str] = [
```

```
        "DISPLAYING BOARD --------------------------",
        "",
        " GUESS                                 FEEDBACK",
        " -------- -------- -------- --------   -------- -------- -------- -------- "
        ]

    board : list[str] = [format_row(row) for row in game_board]

    print("\n".join(header + board))
```

formatRow

Explanation:
This was created to format each **Row** of the board variable into a string that can be displayed.

It takes in the **Row** to be formatted, and returns a list of strings containing the board cell formats, and the **Code** and **Hint** pegs in a string format.
The join built-in function was used to join the **header**, **rowPegs**, and **footer** lists together, separating each element in the new joint list with "\n".

Code Snippet:

```
def format_row(row: Row) -> list[str]:
    header : list[str] = ["|        |        |        |        |
|        |        |        |        |"]

    guess: Guess = row[0]
    feedback: Feedback = row[1]

    row_pegs: list[str] = [format_peg(peg) for peg in guess] + ["| "] +
[format_peg(peg) for peg in feedback] + ["|"]

    footer : list[str] = ["|        |        |        |        |
|        |        |        |        |",
                  " -------- -------- -------- --------   -------- -------- --------
-------- "]

    return "\n".join(header) + "\n" + "".join(row_pegs) + "\n" + "\n".join(footer)
```

formatPeg

Explanation:
This was created to format each **Code** or **Hint** peg, and to add padding to allow for a consistent size of **Board** slots.
If the string of a peg is less than 6 characters in length (the maximum length of a peg's string), padding is added and the size of padding depends on how many more characters are needed to make the string of the peg's length 6.

Code Snippet:

```
def format_peg(peg: Union[Code, Hint]) -> str:
    match len(str(peg)):
        case 6:
            return f"| {print_in_colour(peg)} "
        case _:
```

```
            padding: str = " " * (6-len(str(peg)))
            return f"| {print_in_colour(peg)}{padding} "
```

printInColour

Explanation:
This function allows the printing of the Code and Hint pegs in their respective colours. This is done by using ANSI escape codes.

The function uses match cases to match the return string to its corresponding **Code** or **Hint** peg.

Code Snippet:

```
def print_in_colour(peg: Union[Code, Hint]) -> str:
    match peg:
        case Code_Peg_Option.Orange:
            return "\033[38;5;208morange\033[0m"
        case Code_Peg_Option.Green:
            return "\033[38;5;82mgreen\033[0m"
        case Code_Peg_Option.Blue:
            return "\033[38;5;12mblue\033[0m"
        case Code_Peg_Option.Yellow:
            return "\033[38;5;184myellow\033[0m"
        case Code_Peg_Option.Purple:
            return "\033[38;5;134mpurple\033[0m"
        case Code_Peg_Option.Brown:
            return "\033[38;5;94mbrown\033[0m"
        case Code_Peg_Option.Empty:
            return "      "
        case Hint_Peg.White:
            return "\033[38;5;15mwhite\033[0m"
        case Hint_Peg.Red:
            return "\033[38;5;124mred\033[0m"
        case Hint_Peg.Empty:
            return "      "
```

updateBoard

Explanation:
This function starts with creating a new **Row** containing **new_guess** and **new_feedback**. Applying **len()** to **game_board**, if the **len()** is 6, using the **turn_count** as the index to which the **new_row** is to updated into, the function will return the **new_row** joined alongside the **game_board** in index sliced form to ultimately return a **Board** including the **new_row**.

The same process applies to if the **len()** of **game_board** is 4, however, match cases are limited to only four cases assuming that the **game_board** passed in as the parameter contains four **Rows.**

Code Snippet:

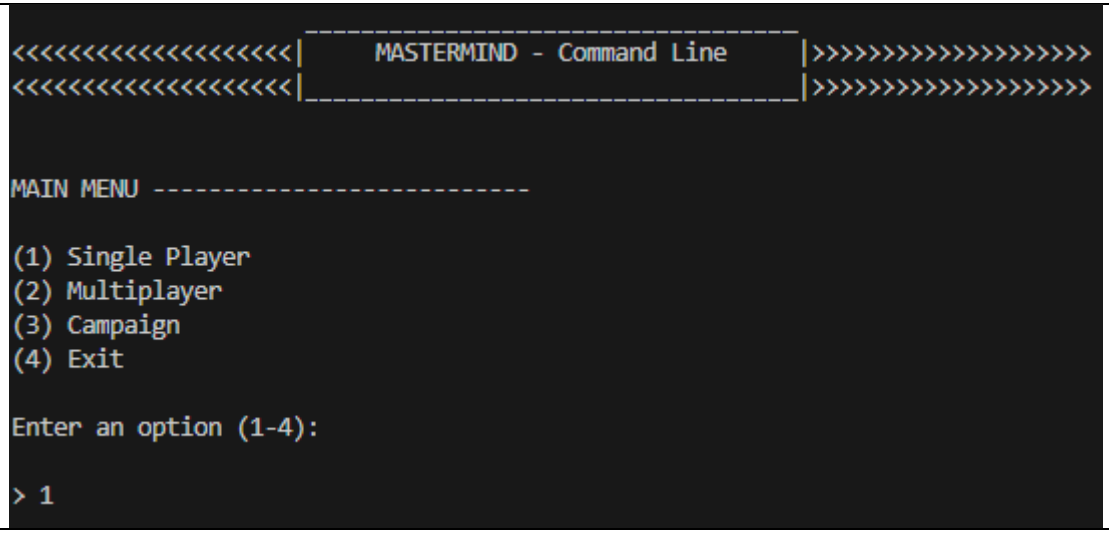
```
def update_board(game_board: Board, new_guess: Guess, new_feedback: Feedback,
turn_count: int) -> Board:
    new_row: Row = (new_guess, new_feedback)
    match len(game_board):
        case 6:
            match turn_count:
```

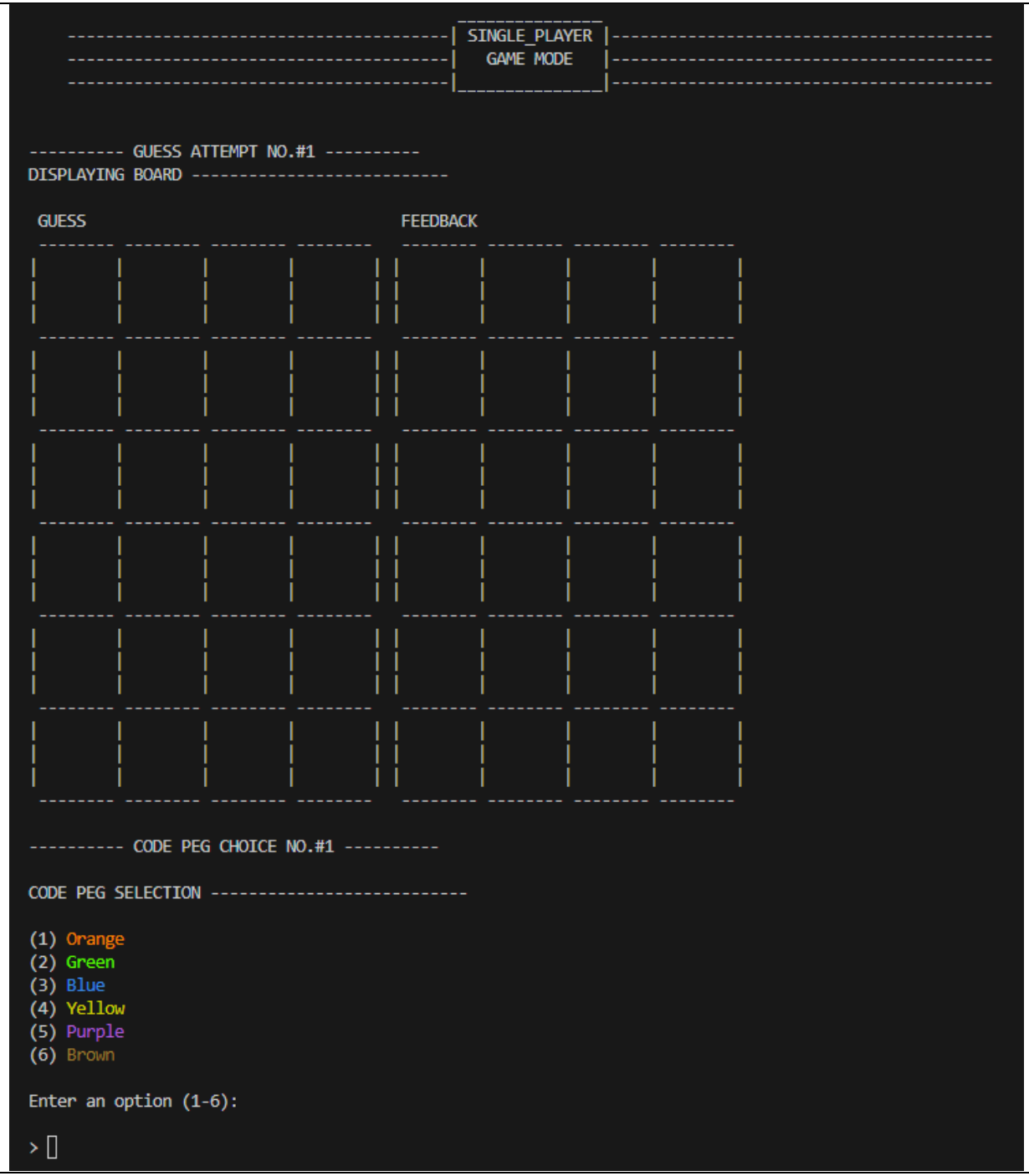
```
            case 1:
                return (new_row, ) + game_board[:-1]
            case 2:
                return game_board[:-5] + (new_row, ) + game_board[2:]
            case 3:
                return game_board[:-4] + (new_row, ) + game_board[3:]
            case 4:
                return game_board[:-3] + (new_row, ) + game_board[4:]
            case 5:
                return game_board[:-2] + (new_row, ) + game_board[5:]
            case 6:
                return game_board[0:5] + (new_row, )
    case 4:
        match turn_count:
            case 1:
                return (new_row, ) + game_board[:-1]
            case 2:
                return game_board[:-3] + (new_row, ) + game_board[2:]
            case 3:
                return game_board[:-2] + (new_row, ) + game_board[3:]
            case 4:
                return game_board[0:3] + (new_row, )
```

T3 – Testing and Verification

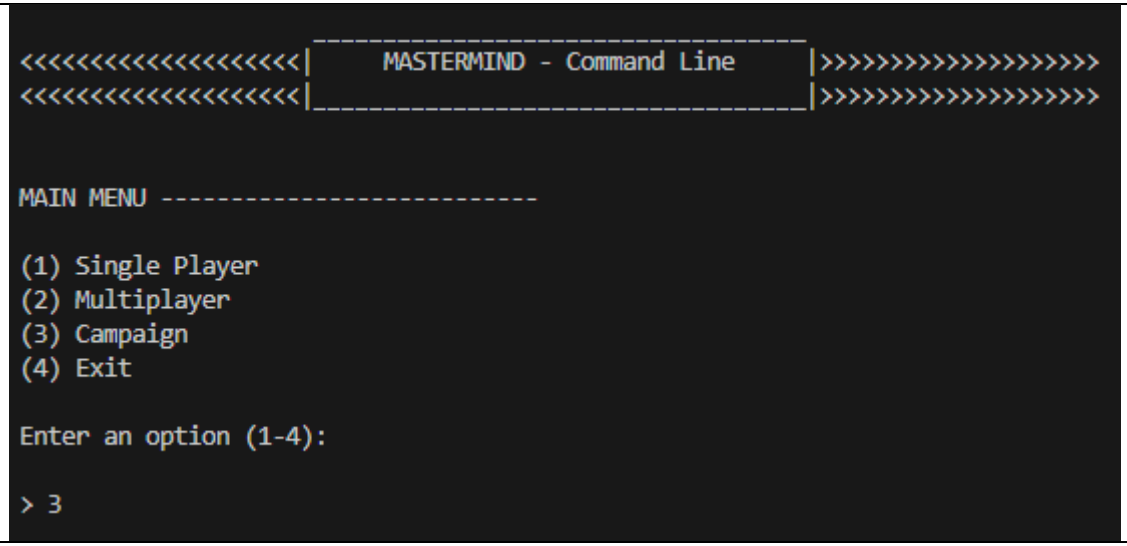
Manual Testing

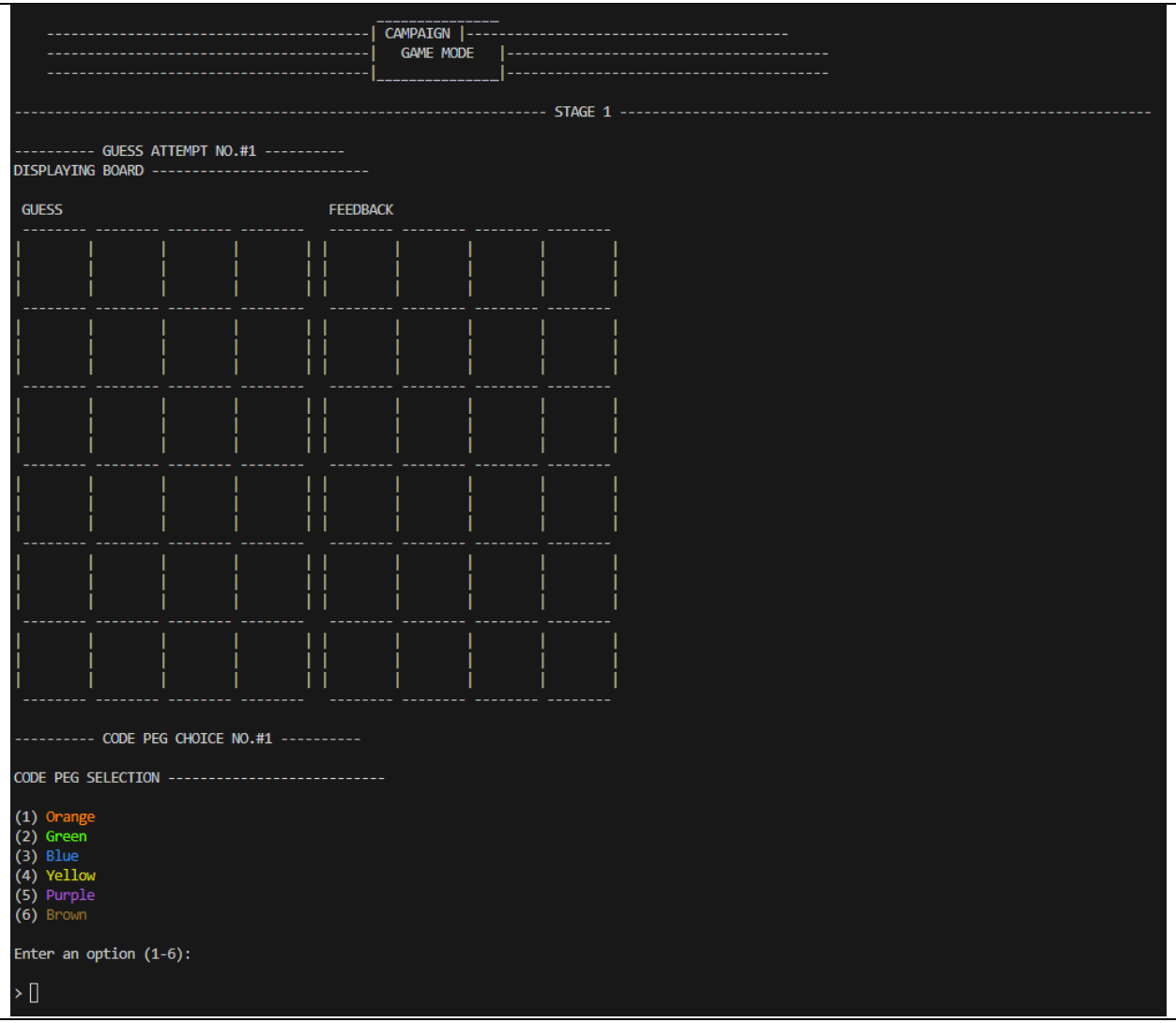
These tests are based off of our Gherkin specifications.

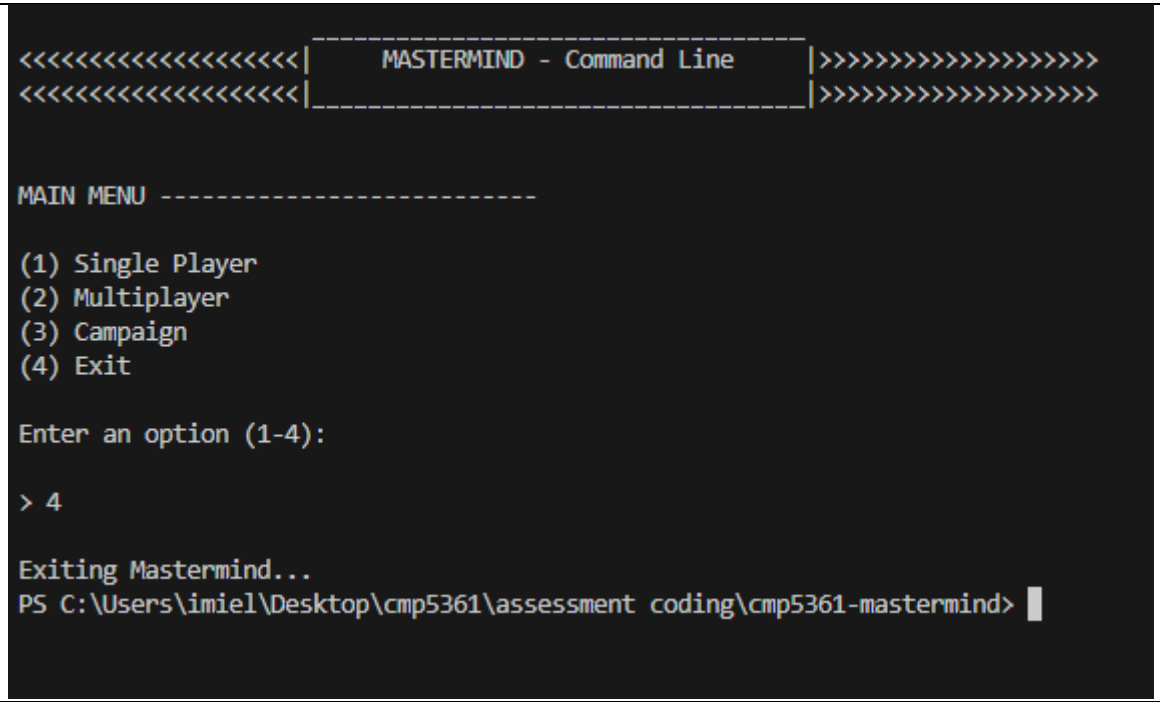
ID	Feature	Steps	Expected	Actual Result
1.1	Select SinglePlayer mode	1. Load program 2. Enter 1 3. Press Enter	1. Main menu displayed 2. Output of “SINGLE_PLAYER GAME MODE.” displayed 3. Game board displayed 4. Prompted for guess	

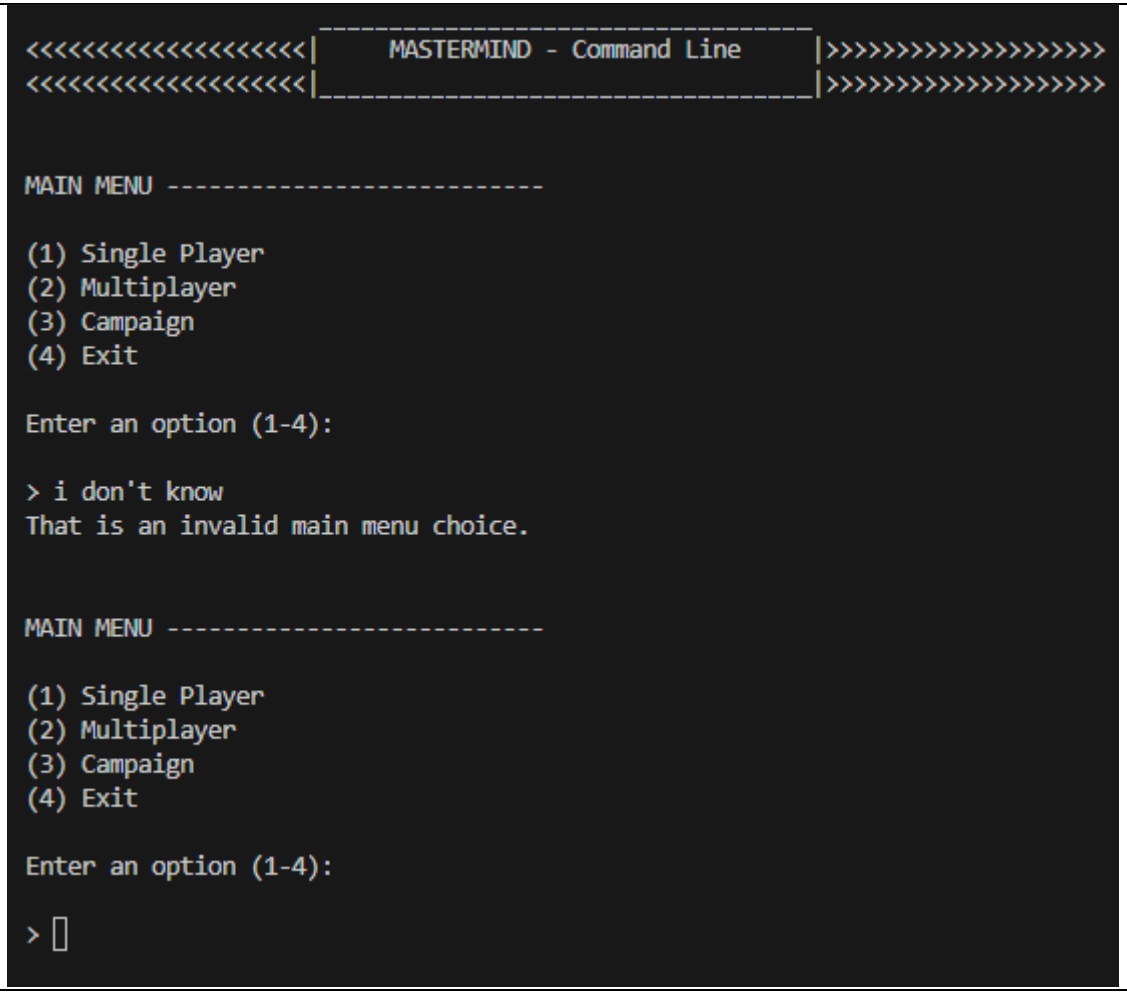

SINGLE_PLAYER
GAME MODE
---------- GUESS ATTEMPT NO.#1 ----------
DISPLAYING BOARD ----------
GUESS
FEEDBACK
---------- CODE PEG CHOICE NO.#1 ----------
CODE PEG SELECTION ----------
(1) Orange
(2) Green
(3) Blue
(4) Yellow
(5) Purple
(6) Brown
Enter an option (1-6):
>

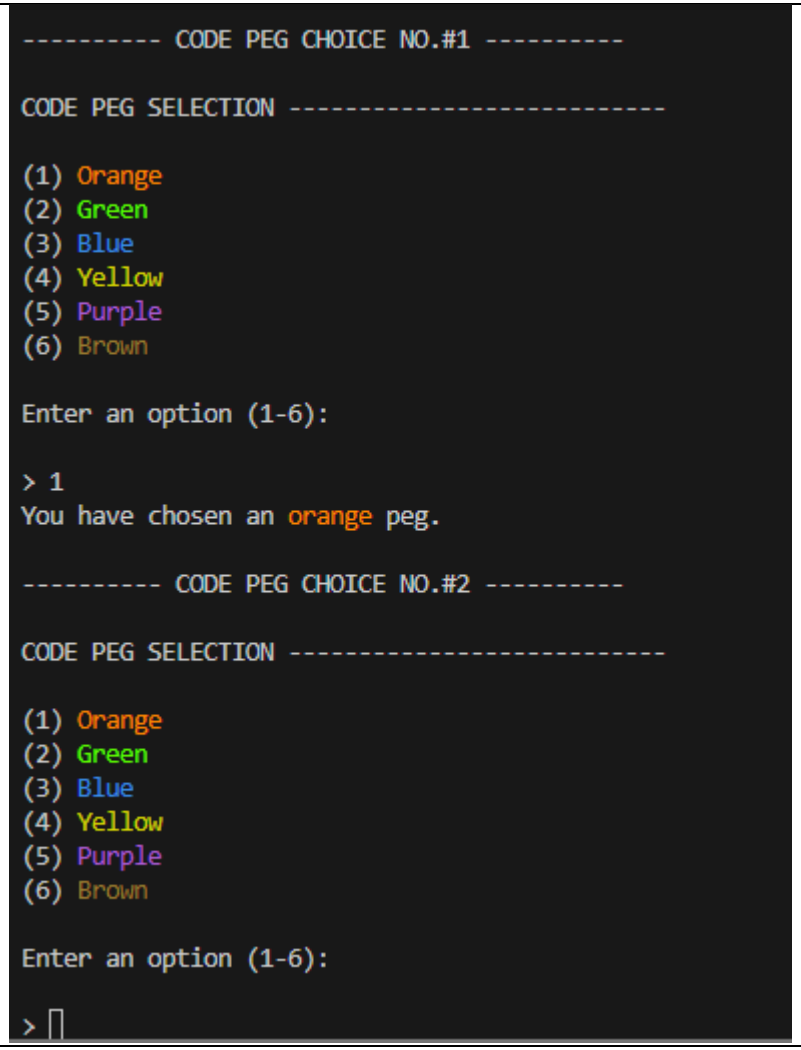
1.2	Select Multiplayer mode	1. Load program 2. Enter 2 3. Press Enter	1. Main menu displayed 2. Prompted for secret code	

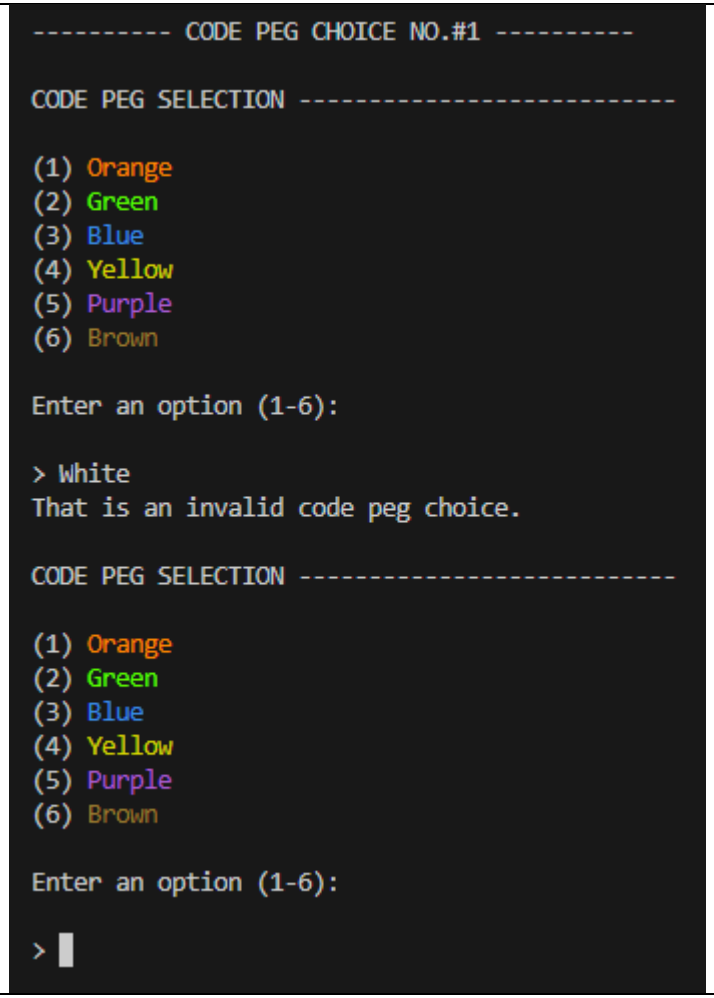
1.3	Select Campaign mode	1. Load program 2. Enter 3 3. Press Enter	1. Main menu displayed 2. Output of “CAMPAIGN GAME MODE STAGE 1.” displayed 3. Game board displayed 4. Prompted for guess		

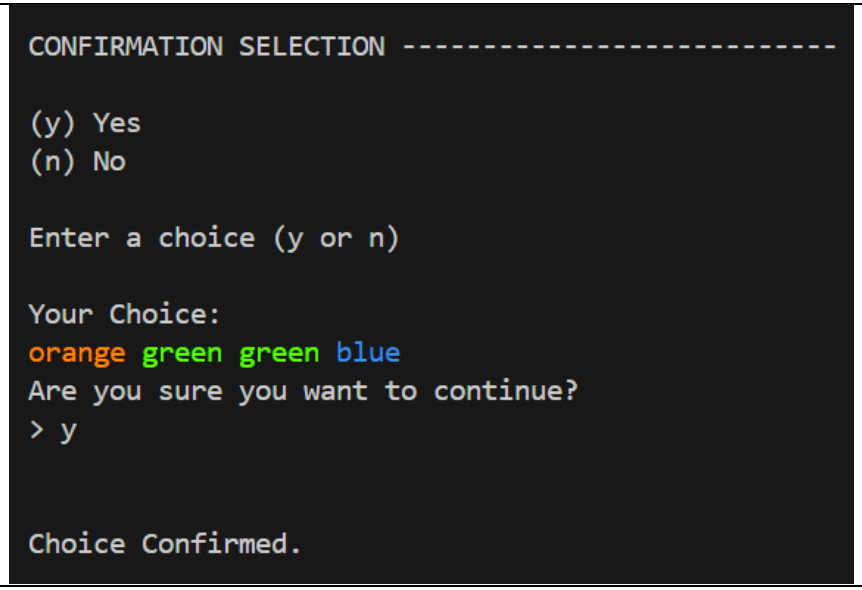

CAMPAIGN
GAME MODE
STAGE 1
---------- GUESS ATTEMPT NO.#1 ----------
DISPLAYING BOARD ----------
GUESS
FEEDBACK
---------- CODE PEG CHOICE NO.#1 ----------
CODE PEG SELECTION ----------
(1) Orange
(2) Green
(3) Blue
(4) Yellow
(5) Purple
(6) Brown
Enter an option (1-6):
>

1.4	Select Exit	1. Load program 2. Enter 3 3. Press Enter	1. Main menu displayed 2. Output of “Exiting Mastermind...” displayed 3. Program is exited	

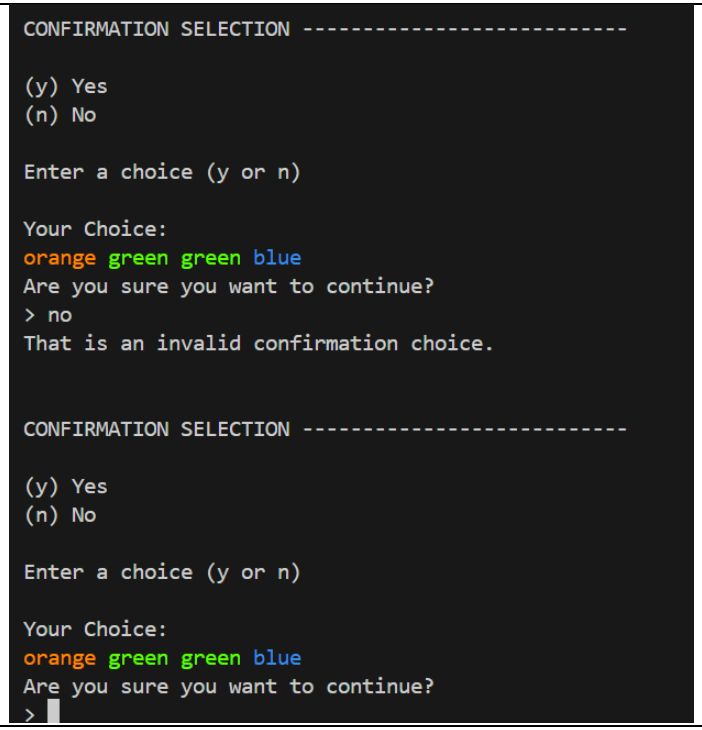
1.5	Select invalid game mode	1. Load program 2. Enter any input that is not 1, 2, 3, or 4 (e.g. i don't know) 3. Press Enter	1. Main menu displayed 2. Output of 'That is an invalid main menu choice.' 3. Prompted for game mode option again	
2.1	Select a code peg	1. Load program 2. Enter 1 3. Enter 1	1. Main menu displayed 2. Output of "SINGLE_PLAYER GAME MODE." displayed 3. Game board displayed 4. Prompted for guess	

			5. Output of “You have chosen an orange peg.” displayed in colour 6. Prompted for the next peg choice	

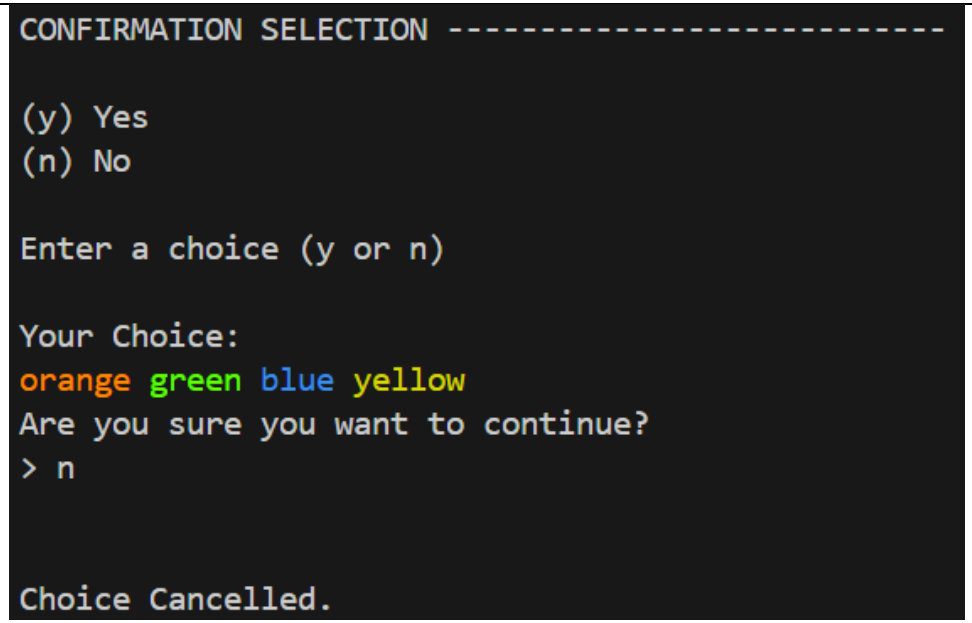
2.2	Select an invalid code peg	1. Load program 2. Enter 1 3. Enter any input that isn't 1, 2, 3, 4, 5, or 6 (e.g. White)	1. Main menu displayed 2. Output of "SINGLE_PLAYER GAME MODE." displayed 3. Game board displayed 4. Prompted for guess 5. Output of "That is an invalid code peg choice." 6. Prompted for guess again	
3.1	Select a confirmation option after setting the secret code: Yes	1. Load program 2. Enter 2 3. Enter 1, 2, 2, 3 4. Enter y	1. Main menu displayed 2. Prompted for secret code 3. Prompted for confirmation option 4. Output of 'Choice	

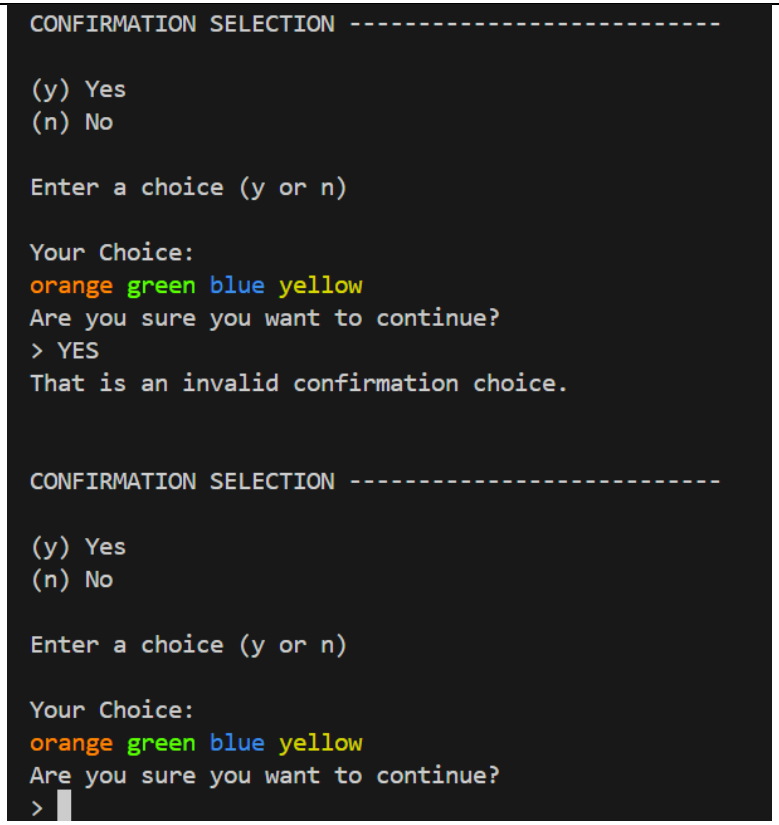
			Confirmed.' Is output		

3.2	Select a confirmation option after setting the secret code: No	1. Load program 2. Enter 2 3. Enter 1, 2, 2, 3 4. Enter n	1. Main menu displayed 2. Prompted for secret code 3. Prompted for confirmation option 4. Output of ‘Choice Cancelled.’ Is displayed 5. Prompted for secret code again	CONFIRMATION SELECTION --------------------------- (y) Yes (n) No Enter a choice (y or n) Your Choice: orange green green blue Are you sure you want to continue? > n Choice Cancelled. CODEMAKER: ENTER SECRET CODE -------------------- ---------- CODE PEG CHOICE NO.#1 ---------- CODE PEG SELECTION --------------------------- (1) Orange (2) Green (3) Blue (4) Yellow (5) Purple (6) Brown Enter an option (1-6): >

3.3	Select a confirmation option after setting the secret code: invalid input	1. Load program 2. Enter 2 3. Enter 1, 2, 2, 3 4. Enter no	1. Main menu displayed 2. Prompted for secret code 3. Prompted for confirmation option 4. Output of 'That is an invalid confirmation choice.' ss displayed 5. Prompted for confirmation choice again		

4.1	Select a confirmation option after making a guess: Yes	1. Load program 2. Enter 1 3. Enter 1, 2, 3, 4 4. Enter y	1. Main menu displayed 2. Output of “SINGLE_PLAYER GAME MODE.” displayed 3. Game board displayed 4. Prompted for guess 5. Prompted for confirmation option 6. Output of ‘Choice Confirmed.’ is displayed 7. The updated board should be displayed	

4.2	Select a confirmation option after making a guess: No	1. Load program 2. Enter 1 3. Enter 1, 2, 3, 4 4. Enter n	1. Main menu displayed 2. Output of “SINGLE_PLAYER GAME MODE.” displayed 3. Game board displayed 4. Prompted for guess 5. Prompted for confirmation option 6. Output ‘Choice Cancelled.’ should be displayed	

4.3	Select a confirmation option after making a guess: invalid input	1. Load program 2. Enter 1 3. Enter 1, 2, 3, 4 4. Enter YES	1. Main menu displayed 2. Output of “SINGLE_PLAYER GAME MODE.” displayed 3. Game board displayed 4. Prompted for guess 5. Prompted for confirmation option 6. Output ‘That is an invalid confirmation choice.’ should be displayed 7. Prompted for a confirmation choice again	

Automated Testing

Pytest

Imports Used

```
import pytest
from main import *
```

parseMainMenuOption

These tests were made to ensure that the Menu type parser function was working. If this were not to work, we would not be able to start a game of Mastermind.

```
def test_parse_main_menu_option():
    assert Main_Menu_Option.parse_main_menu_option("1") ==
Main_Menu_Option.Single_Player
    assert Main_Menu_Option.parse_main_menu_option("2") == Main_Menu_Option.Multiplayer
    assert Main_Menu_Option.parse_main_menu_option("3") == Main_Menu_Option.Campaign
    assert Main_Menu_Option.parse_main_menu_option("4") == Main_Menu_Option.Exit
    assert Main_Menu_Option.parse_main_menu_option("") is None
    assert Main_Menu_Option.parse_main_menu_option("single player") is None
```

parseConfirmationOption

These tests were made to ensure that the Confirm type parser function was working. If this were not to work, we would not be able to confirm a Guess or Secret code made.

```
def test_parse_confirmation_option():
    assert Confirmation_Option.parse_confirmation_option("Y") == Confirmation_Option.Yes
    assert Confirmation_Option.parse_confirmation_option("y ") ==
Confirmation_Option.Yes
    assert Confirmation_Option.parse_confirmation_option("YeS") is None
    assert Confirmation_Option.parse_confirmation_option("    N") ==
Confirmation_Option.No
    assert Confirmation_Option.parse_confirmation_option("n") == Confirmation_Option.No
    assert Confirmation_Option.parse_confirmation_option("no") is None
    assert Confirmation_Option.parse_confirmation_option("hell naur") is None
    assert Confirmation_Option.parse_confirmation_option("") is None
```

parseCodePegOption

These tests were made to check that for every possible kind of Player input for a chosen peg, the parser method would return the correct output.

```
def test_parse_code_peg_option():
    assert Code_Peg_Option.parse_code_peg_option("0") is None
    assert Code_Peg_Option.parse_code_peg_option("1") == Code_Peg_Option.Orange
    assert Code_Peg_Option.parse_code_peg_option("2") == Code_Peg_Option.Green
    assert Code_Peg_Option.parse_code_peg_option("3") == Code_Peg_Option.Blue
    assert Code_Peg_Option.parse_code_peg_option("4") == Code_Peg_Option.Yellow
    assert Code_Peg_Option.parse_code_peg_option("5") == Code_Peg_Option.Purple
```

```
    assert Code_Peg_Option.parse_code_peg_option("6") == Code_Peg_Option.Brown
    assert Code_Peg_Option.parse_code_peg_option("Orange") is None
    assert Code_Peg_Option.parse_code_peg_option("") is None
```

str(codePegOption)

These tests were made to check whether the string methods of each coloured Code peg output the correct string.

```
def test_code_peg_option_str():
    assert str(Code_Peg_Option.Empty) == "empty"
    assert str(Code_Peg_Option.Orange) == "orange"
    assert str(Code_Peg_Option.Green) == "green"
    assert str(Code_Peg_Option.Blue) == "blue"
    assert str(Code_Peg_Option.Yellow) == "yellow"
    assert str(Code_Peg_Option.Purple) == "purple"
    assert str(Code_Peg_Option.Brown) == "brown"
```

str(hintPeg)

These tests were made to check whether the string methods of each coloured Hint peg output the correct string.

```
def test_hint_peg_str():
    assert str(Hint_Peg.Empty) == "empty"
    assert str(Hint_Peg.White) == "white"
    assert str(Hint_Peg.Red) == "red"
```

getFeedback

This function is a crucial aspect to our program. However, in the 3rd test, it failed, indicating something was wrong with the Feedback generated – after inspecting the code, we assume it has something to do with the getWhiteHints function's logic.

```
def test_get_feedback():
    guess = (Code_Peg_Option.Green, Code_Peg_Option.Brown, Code_Peg_Option.Orange,
Code_Peg_Option.Orange)
    secret = (Code_Peg_Option.Green, Code_Peg_Option.Brown, Code_Peg_Option.Orange,
Code_Peg_Option.Orange)
    assert get_feedback(guess, secret) == [True, (Hint_Peg.Red, Hint_Peg.Red,
Hint_Peg.Red, Hint_Peg.Red)]

    guess = (Code_Peg_Option.Green, Code_Peg_Option.Brown, Code_Peg_Option.Orange,
Code_Peg_Option.Orange)
    secret = (Code_Peg_Option.Orange, Code_Peg_Option.Orange, Code_Peg_Option.Green,
Code_Peg_Option.Brown)
    assert get_feedback(guess, secret) == [False, (Hint_Peg.White, Hint_Peg.White,
Hint_Peg.White, Hint_Peg.White)]

    guess = (Code_Peg_Option.Green, Code_Peg_Option.Brown, Code_Peg_Option.Orange,
Code_Peg_Option.Orange)
    secret = (Code_Peg_Option.Green, Code_Peg_Option.Orange, Code_Peg_Option.Green,
Code_Peg_Option.Brown)
    assert get_feedback(guess, secret) == [False, (Hint_Peg.Red, Hint_Peg.White,
Hint_Peg.White, Hint_Peg.Empty)] # logic is working incorrectly
```

joinHints

```
def test_join_hints():
    red_pegs = [(False, Code_Peg_Option.Green), (True, Code_Peg_Option.Brown), (False,
Code_Peg_Option.Orange), (False, Code_Peg_Option.Orange)]
    white_pegs = [(True, Code_Peg_Option.Green), (False, Code_Peg_Option.Brown), (True,
Code_Peg_Option.Orange), (False, Code_Peg_Option.Orange)]
    assert join_hints(red_pegs, white_pegs) == [Hint_Peg.White, Hint_Peg.Red,
Hint_Peg.White, Hint_Peg.Empty]
```

sortHints

```
def test_sort_hints():
    hints = [Hint_Peg.Empty, Hint_Peg.White, Hint_Peg.Red, Hint_Peg.Red]
    assert sort_hints(hints) == (Hint_Peg.Red, Hint_Peg.Red, Hint_Peg.White,
Hint_Peg.Empty)
```

getRedHints

These tests were made to ensure that the correct list of Red Hint peg indicators was output by the getRedHints function depending on the Player's Guess and the Secret code.

```
def test_get_red_hints():
    guess = (Code_Peg_Option.Green, Code_Peg_Option.Brown, Code_Peg_Option.Orange,
Code_Peg_Option.Orange)
    secret = (Code_Peg_Option.Green, Code_Peg_Option.Brown, Code_Peg_Option.Orange,
Code_Peg_Option.Orange)
    assert get_red_hints(guess, secret) == [(True, Code_Peg_Option.Green), (True,
Code_Peg_Option.Brown), (True, Code_Peg_Option.Orange), (True, Code_Peg_Option.Orange)]

    guess = (Code_Peg_Option.Purple, Code_Peg_Option.Yellow, Code_Peg_Option.Orange,
Code_Peg_Option.Orange)
    assert get_red_hints(guess, secret) == [(False, Code_Peg_Option.Purple), (False,
Code_Peg_Option.Yellow), (True, Code_Peg_Option.Orange), (True, Code_Peg_Option.Orange)]
```

checkIfDupe

This test checks for both outputs of Boolean values from checkIfDupe. It ensures that if a Code peg is in the Secret code, True will be output, and if not. False will be output.

```
def test_check_if_dupe():
    secret = (Code_Peg_Option.Green, Code_Peg_Option.Brown, Code_Peg_Option.Orange,
Code_Peg_Option.Orange)
    assert check_if_dupe(Code_Peg_Option.Orange, secret) == True
    assert check_if_dupe(Code_Peg_Option.Green, secret) == False
```

checkGuessedCorrectly

These tests were made to ensure checkGuessedCorrectly correctly counts the number of times a tuple appears in the list of Red Hint peg indicators.

```
def test_check_almost_guessed_0():
    red_pegs = [(True, Code_Peg_Option.Orange)]
    assert check_almost_guessed(Code_Peg_Option.Green, red_pegs) == 0

def test_check_almost_guessed_1():
```

```
    red_pegs = [(True, Code_Peg_Option.Green), (True, Code_Peg_Option.Orange)]
    assert check_almost_guessed(Code_Peg_Option.Orange, red_pegs) == 1

def test_check_almost_guessed_2():
    red_pegs = [(True, Code_Peg_Option.Green), (True, Code_Peg_Option.Orange), (True,
Code_Peg_Option.Blue), (True, Code_Peg_Option.Orange)]
    assert check_almost_guessed(Code_Peg_Option.Orange, red_pegs) == 2
```

checkAlmostGuessed

These tests were made to ensure checkAlmostGuessed correctly counts the number of times a tuple appears in the list of White Hint peg indicators.

```
def test_check_almost_guessed_0():
    running_feedback = [(True, Code_Peg_Option.Orange)]
    assert check_almost_guessed(Code_Peg_Option.Green, running_feedback) == 0

def test_check_almost_guessed_1():
    running_feedback = [(True, Code_Peg_Option.Green), (True, Code_Peg_Option.Orange)]
    assert check_almost_guessed(Code_Peg_Option.Orange, running_feedback) == 1

def test_check_almost_guessed_2():
    running_feedback = [(True, Code_Peg_Option.Green), (True, Code_Peg_Option.Orange),
(True, Code_Peg_Option.Blue), (True, Code_Peg_Option.Orange)]
    assert check_almost_guessed(Code_Peg_Option.Orange, running_feedback) == 2
```

str(Player)

These tests were made to check whether the string methods of each Player output the correct string.

```
def test_code_maker_str():
    assert str(CodeMaker()) == "Code Maker"

def test_code_breaker_str():
    assert str(CodeBreaker()) == "Code Breaker"

def test_cpu_str():
    assert str(CPU()) == "CPU"
```

normalSecretCode

This test was made to ensure that a random Secret code generated by normalSecretCode had 4 different Code peg colours. Asserting using the code converted into a set ensured no duplicate elements were counted.

```
def test_normal_secret_code():
    code = normal_secret_code()
    assert len(code) == len(set(code))
```

hardSecretCode

This test was made to ensure that a random Secret code generated by hardSecretCode had 3 different Code peg colours, with one duplicate. The assertion converts the tuple into a set to check that the number of unique elements was equal to 3.

```
def test_hard_secret_code():
    code = hard_secret_code()
```

```
    assert len(set(code)) == 3
```

formatPeg

These tests were made to ensure that the formatPeg function returned the correct string. If this were to be incorrect, the Board displayed in displayBoard would be deformed.

```
def test_format_peg():
    assert format_peg(Code_Peg_Option.Empty) == "|        "
    assert format_peg(Code_Peg_Option.Orange) == "| \x1b[38;5;208morange\x1b[0m "
    assert format_peg(Code_Peg_Option.Green) == "| \033[38;5;82mgreen\033[0m  "
    assert format_peg(Code_Peg_Option.Blue) == "| \033[38;5;12mblue\033[0m   "
    assert format_peg(Code_Peg_Option.Yellow) == "| \033[38;5;184myellow\033[0m "
    assert format_peg(Code_Peg_Option.Purple) == "| \033[38;5;134mpurple\033[0m "
    assert format_peg(Code_Peg_Option.Brown) == "| \033[38;5;94mbrown\033[0m  "
    assert format_peg(Hint_Peg.Empty) == "|         "
    assert format_peg(Hint_Peg.White) == "| \033[38;5;15mwhite\033[0m  "
    assert format_peg(Hint_Peg.Red) == "| \033[38;5;124mred\033[0m    "
```

endgame

These tests were made to ensure that the endGame function output the correct messages depending on the game mode, the stage of the Campaign game mode, how the game ended, the game's Players, and the Secret code.

The parameter capsys was used to capture and check what the function had output when called within the test functions (Kutaj, 2023).

```
def test_end_single_player_game(capsys):
    end_game(Main_Menu_Option.Single_Player, None, False, (CodeBreaker(), CPU()),
(Code_Peg_Option.Orange, Code_Peg_Option.Green, Code_Peg_Option.Blue,
Code_Peg_Option.Purple))
    capture = capsys.readouterr()

    assert "\x1b[38;5;208morange\x1b[0m \x1b[38;5;82mgreen\x1b[0m
\x1b[38;5;12mblue\x1b[0m \x1b[38;5;134mpurple\x1b[0m\n" in capture.out
    assert "\nCPU has won the game!" in capture.out

def test_end_multiplayer_game(capsys):
    end_game(Main_Menu_Option.Multiplayer, None, False, (CodeBreaker(), CodeMaker()),
(Code_Peg_Option.Orange, Code_Peg_Option.Green, Code_Peg_Option.Blue,
Code_Peg_Option.Purple))
    capture = capsys.readouterr()

    assert "\x1b[38;5;208morange\x1b[0m \x1b[38;5;82mgreen\x1b[0m
\x1b[38;5;12mblue\x1b[0m \x1b[38;5;134mpurple\x1b[0m\n" in capture.out
    assert "\nCode Maker has won the game!" in capture.out


# for campaign message 1
def test_end_campaign_game(capsys):
    end_game(Main_Menu_Option.Campaign, 1, True, (CodeBreaker(), CPU()),
(Code_Peg_Option.Orange, Code_Peg_Option.Green, Code_Peg_Option.Blue,
Code_Peg_Option.Purple))
    capture = capsys.readouterr()
```

```
    assert "\x1b[38;5;208morange\x1b[0m \x1b[38;5;82mgreen\x1b[0m
\x1b[38;5;12mblue\x1b[0m \x1b[38;5;134mpurple\x1b[0m\n" in capture.out
    assert "\nCode Breaker has won the game!" in capture.out
    assert "\nYou have successfully completed your Campaign game!" in capture.out

# for campaign message 2
def test_end_campaign_game(capsys):
    end_game(Main_Menu_Option.Campaign, 2, True, (CodeBreaker(), CPU()),
(Code_Peg_Option.Orange, Code_Peg_Option.Green, Code_Peg_Option.Blue,
Code_Peg_Option.Purple))
    capture = capsys.readouterr()

    assert "\x1b[38;5;208morange\x1b[0m \x1b[38;5;82mgreen\x1b[0m
\x1b[38;5;12mblue\x1b[0m \x1b[38;5;134mpurple\x1b[0m\n" in capture.out
    assert "\nCode Breaker has won the game!" in capture.out
    assert "\nWell done! You have advanced to the next stage in your Campaign game!" in
capture.out

# for campaign message 3
def test_end_campaign_game(capsys):
    end_game(Main_Menu_Option.Campaign, 1, False, (CodeBreaker(), CPU()),
(Code_Peg_Option.Orange, Code_Peg_Option.Green, Code_Peg_Option.Blue,
Code_Peg_Option.Purple))
    capture = capsys.readouterr()

    assert "\x1b[38;5;208morange\x1b[0m \x1b[38;5;82mgreen\x1b[0m
\x1b[38;5;12mblue\x1b[0m \x1b[38;5;134mpurple\x1b[0m\n" in capture.out
    assert "\nCPU has won the game!" in capture.out
    assert "\nYou have failed your Campaign game!" in capture.out
```

```
PS C:\Users\imiel\Desktop\cmp5361\assessment coding\cmp5361-mastermind> pytest automated_testing.
py
==================================== test session starts ====================================
platform win32 -- Python 3.13.2, pytest-8.3.5, pluggy-1.6.0
rootdir: C:\Users\imiel\Desktop\cmp5361\assessment coding\cmp5361-mastermind
collected 22 items

automated_testing.py .....F.................                                          [100%]

========================================= FAILURES ==========================================
_____________________________________ test_get_feedback _____________________________________

    def test_get_feedback():
        guess = (Code_Peg_Option.Green, Code_Peg_Option.Brown, Code_Peg_Option.Orange, Code_Peg_O
ption.Orange)
        secret = (Code_Peg_Option.Green, Code_Peg_Option.Brown, Code_Peg_Option.Orange, Code_Peg_
Option.Orange)
        assert get_feedback(guess, secret) == [True, (Hint_Peg.Red, Hint_Peg.Red, Hint_Peg.Red, H
int_Peg.Red)]

        guess = (Code_Peg_Option.Green, Code_Peg_Option.Brown, Code_Peg_Option.Orange, Code_Peg_O
ption.Orange)
        secret = (Code_Peg_Option.Orange, Code_Peg_Option.Orange, Code_Peg_Option.Green, Code_Peg
_Option.Brown)
        assert get_feedback(guess, secret) == [False, (Hint_Peg.White, Hint_Peg.White, Hint_Peg.W
hite, Hint_Peg.White)]

        guess = (Code_Peg_Option.Green, Code_Peg_Option.Brown, Code_Peg_Option.Orange, Code_Peg_O
ption.Orange)
        secret = (Code_Peg_Option.Green, Code_Peg_Option.Orange, Code_Peg_Option.Green, Code_Peg_
Option.Brown)
>       assert get_feedback(guess, secret) == [False, (Hint_Peg.Red, Hint_Peg.White, Hint_Peg.Whi
te, Hint_Peg.Empty)] # logic is working incorrectly
E       assert [False, (<Hin...eg.White: 1>)] == [False, (<Hin...eg.Empty: 0>)]
E
E         At index 1 diff: (<Hint_Peg.Red: 2>, <Hint_Peg.White: 1>, <Hint_Peg.White: 1>, <Hint_Pe
g.White: 1>) != (<Hint_Peg.Red: 2>, <Hint_Peg.White: 1>, <Hint_Peg.White: 1>, <Hint_Peg.Empty: 0>
)
E         Use -v to get more diff

automated_testing.py:69: AssertionError
================================== short test summary info ==================================
FAILED automated_testing.py::test_get_feedback - assert [False, (<Hin...eg.White: 1>)] == [False,
 (<Hin...eg.Empty: 0>)]
================================ 1 failed, 21 passed in 0.32s ================================
```

Figure 9 Everything but the 3rd getFeedback test is working correctly.

T4 – Understanding and Exploring Team-Based Software Development

OpenRA

OpenRA is an "Open Source real-time strategy game engine for early Westwood games such as Command & Conquer: Red Alert written in C# using SDL and OpenGL." (OpenRA, 2025).

As a game engine, it is a software framework equipped with tools that allow users to develop video games or digital twins for visualisation (Ltd, n.d.). A game engine may include a 2D or 3D graphics rendering engine, physics and collision engines, audio engines, artificial intelligence, animation engines, and more (www.perforce.com, n.d.). Specifically, OpenRA is a game engine that aids users to build 2D and 2.5D real-time strategy games (www.openra.net, n.d.).

The product's target users would be those interested in playing, "rebuilding" and "reimagining" classic 2D and 3D RTS games. With support to integrated online multiplayer, those who enjoy playing games with others online would be attracted to OpenRA.

Documentation for Project Contributors

Documentation for programming projects is a way of providing a basis on describing the nature, how it works, and how to appropriately use it.

Code understandability and maintainability are improved to develop a better notion of how certain parts of the code should be treated as you consider the inputs, outputs and overall purpose. Addiitionally, it helps to generate insights and develop new practices in teams in order to stay consistent within the development workspace (8 Important Points, 2023).

Referring to OpenRA's GitHub repository, it conveniently provides links to areas in relation to FAQs, their Wiki, and guides for their Contribution, Mapping and Modding to help build that basis understanding to how the project as a whole should be treated before contributing.

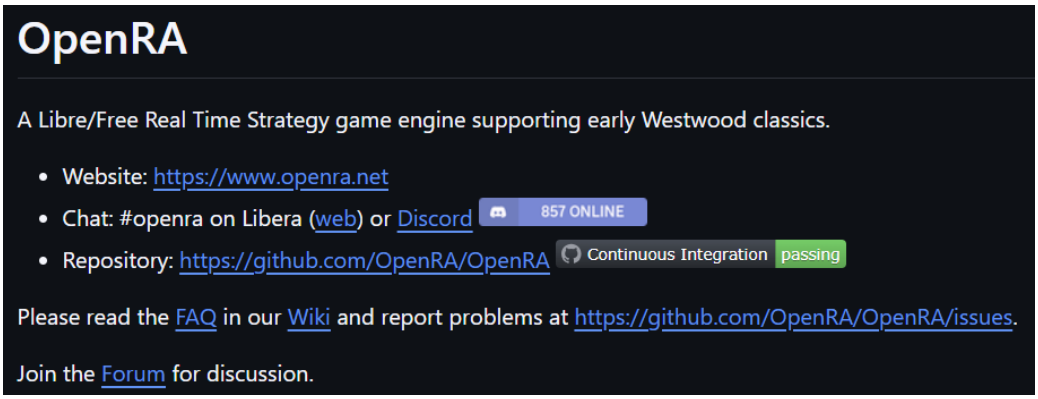


Figure Relevant Links to certain Documentation

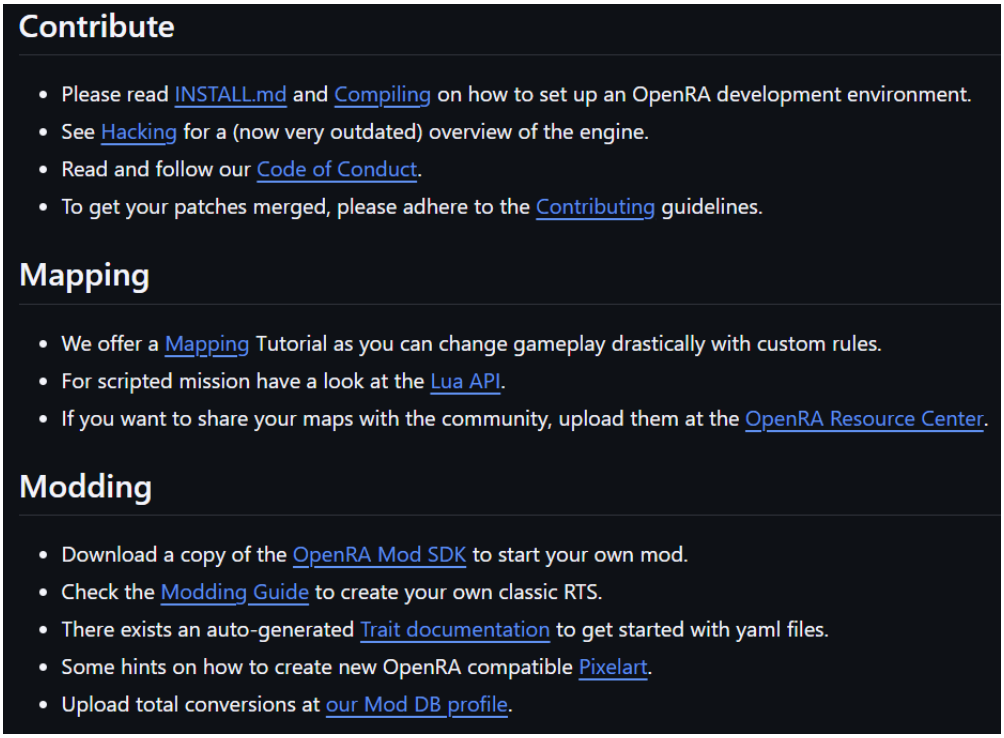


Figure Documentation Links for Contribution, Mapping and Modding

Version Control

Version control systems are tools that allow users to track and manage changes to a project's file system over time (Soumya, 2019). This is especially helpful for team projects, allowing team members to work on different parts of the project simultaneously without overwriting one another's changes and providing a clear commit history, resulting in increase of efficiency, reduction of errors, and a more organised workflow.

In the table below, the main Git commands for version control are covered.

Git Command	What does this command do?
Clone	Creates a copy of an already existing repository.
Commit	Saves the state of a project at time of commit to help keep track of changes.
Pull	Fetches changes from a remote repository and merges it with the local repository.
Push	Updates a remote repository with changes from the local repository.
Merge	Combines the changes from two branches into one.
Branch	Creates a new branch that one can work on individually, without interfering with the main project.
Pull request	A request to merge changes from one branch into another.

Version Control vs. Cloud-based Storage Systems

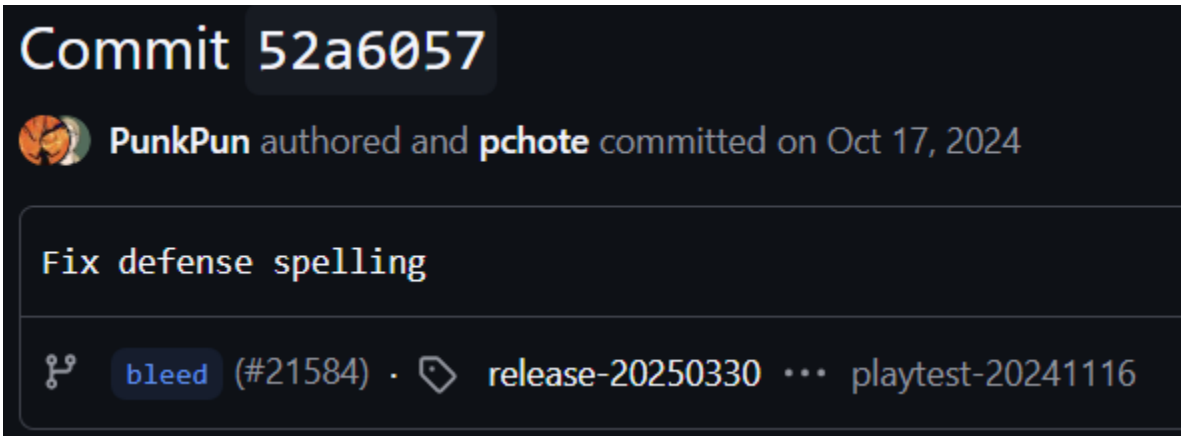
Version Control Systems are software tools that assist the process of software developing teams managing the changes being made to their program code to enhance the workload efficiency. (Atlassian, n.d.).

Cloud-Based Storage Systems are data deposit models with the purpose of storing digital documents, photos, videos and many different types of media within offsite cloud-based servers to be accessed whenever (Chiradeep BasuMallick, 2025). Cloud-Based Storage System do not keep track of any changes compared to VCS as they're solely providing storage and access services.

OpenRA's and Our Commit History

The first major commit of program code was made on June 19, 2007 by a user called chrisf involves several folders with the majority containing C# files added into the repository, which potentially involve setting up the initial framework of the similar Westwood games (OpenRA, 2025).

Commit 52a6057 "Fix defense spelling" at https://github.com/OpenRA/OpenRA/commit/52a605787b9ea95ec1513f7a17290ed5ada5bb3a.



Committed on October 17, 2024, which involved replacing every instance of "defence" and "defences" being replaced with American English spelling of "defense" and "defenses".

This is similarly related to the logical error of displaying the game mode header of a specific game mode as a typo when initiated from Main Menu state. The commit was committed directly into the "bleed" branch.

OpenRA's Issue Tracker

The issue selected is #2021 "Hard to read some colors" at https://github.com/OpenRA/OpenRA/issues/2021.

Figure 10 Screenshot of the issue being discussed.

It is a closed issue that was made by the user kyrreso on April 18, 2012, talking about how certain text colours were impossible to read on their background colours. It is an issue related to user experience and user interface design.

This is related to one of the issues we had when choosing the colours for the printInColour function during implementation.

OpenRA's Pull Request Tracker

The pull request selected is #21817 "Implement sonic blast rendering effect." at https://github.com/OpenRA/OpenRA/pull/21817/files.

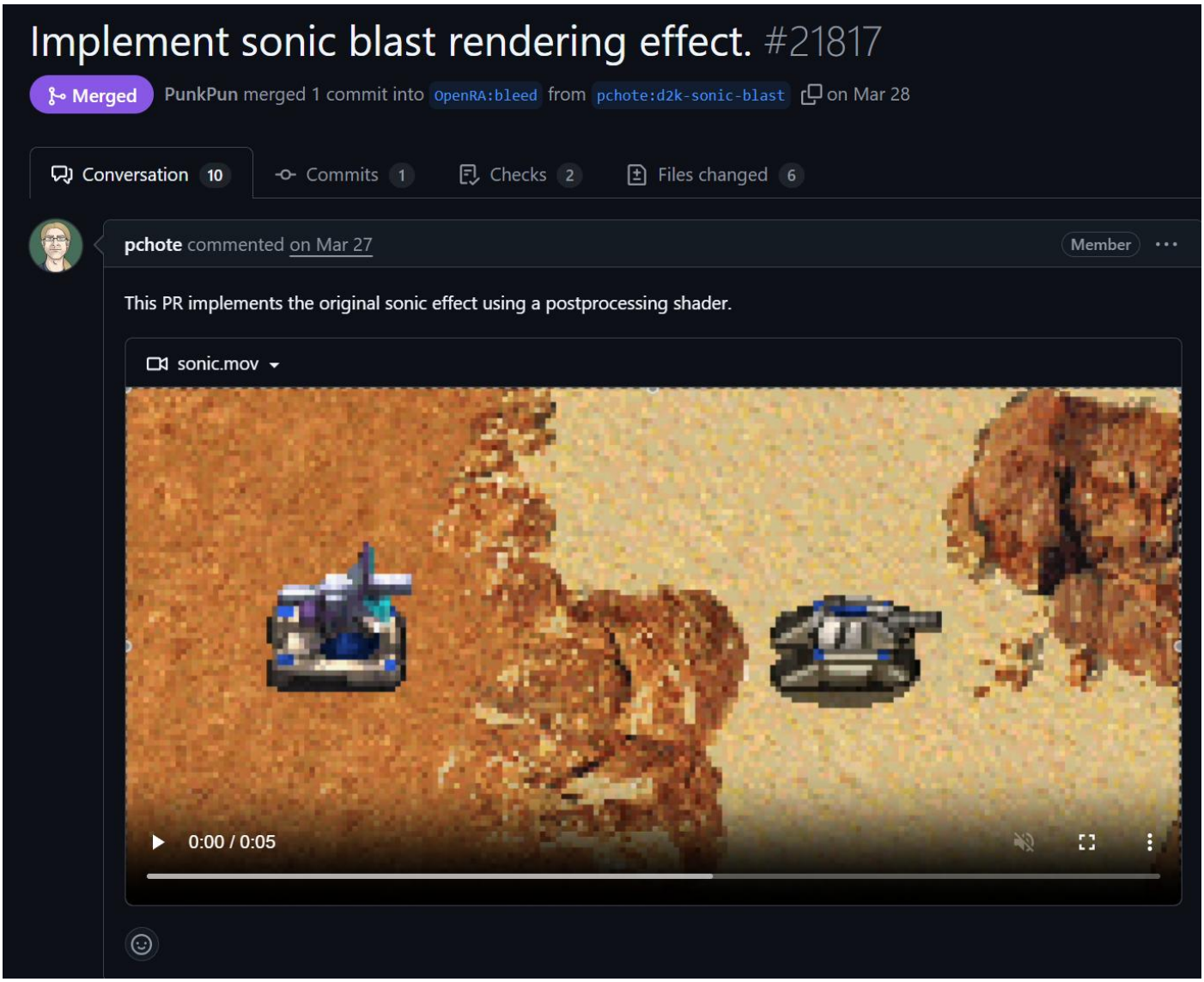


Figure 11 Screenshot of the pull request being discussed.

This pull request was made by the user pchote and within their pull request, changes to three C# code files, one shading file, and two configuration files were made to implement the sonic blast rendering effect.

Pair Project Development Reflection

After trying to implement the first stage of our plans and designs, we received verbal feedback from a Tutor, which led to us to re-iterating and starting a second stage of planning and designing that adopts declarative and functional programming rather than imperative programming.

Implementation workload handling was influenced by task interest, with one developing recursive functions for game play, and another developing smaller functions to be called in game play. Types and user input functions were devised together before starting the rest of the program to ensure that the base functions worked perfectly.

During implementation, we were constantly testing our own functions, and for this document, it was the same process.

The Use of Version Control in Pair Project Development Reflection

Git was utilised for version control involving pull, push and commits being made. However, it would have been even better if we used pull requests to propose amendments, issues to track problems over on the spot messaging, and using branches to work on the code individually.

In future team-based projects, we will do our best to utilise these features of Git version control and ensure that any commit messages, issues or pull requests stated are explained in detail to avoid potential prolonging errors being produced when conducting tests.

References

ONS (2024). *Milestones: journeying through modern life - Office for National Statistics*. [online] www.ons.gov.uk. Available at: https://www.ons.gov.uk/peoplepopulationandcommunity/populationandmigration/populationestimates/articles/milestonesjourneyingthroughmodernlife/2024-04-08.

Wikipedia. (2020). *Mastermind (board game)*. [online] Available at: https://en.wikipedia.org/wiki/Mastermind_(board_game).

Kutaj, P. (2023). *How to Test Printed Output in Python with Pytest and its Capsys Fixture*. [online] Medium. Available at: https://pavolkutaj.medium.com/how-to-test-printed-output-in-python-with-pytest-and-its-capsys-fixture-161010cfc5ad.

OpenRA (2025). *GitHub - OpenRA/OpenRA: Open Source real-time strategy game engine for early Westwood games such as Command & Conquer: Red Alert written in C# using SDL and OpenGL. Runs on Windows, Linux, *BSD and Mac OS X.* [online] GitHub. Available at: https://github.com/OpenRA/OpenRA.

Ltd, A. (n.d.). *What is a Gaming Engine?* [online] Arm | The Architecture for the Digital World. Available at: https://www.arm.com/glossary/gaming-engines.

www.perforce.com. (n.d.). *Complete Game Engine Overview | Perforce*. [online] Available at: https://www.perforce.com/resources/vcs/game-engine-overview.

www.openra.net. (n.d.). *About | OpenRA*. [online] Available at: https://www.openra.net/about/.

8 Important Points (2023) *The importance of documentation in programming*, *Medium*. Available at: https://genuineproductdigital.medium.com/the-importance-of-documentation-in-programming-30c286f86 (Accessed: 23 May 2025).

Soumya (2019). *Version Control Systems*. [online] GeeksforGeeks. Available at: https://www.geeksforgeeks.org/version-control-systems/.

OpenRA (2025). *openra first commit! · OpenRA/OpenRA@b59ba43*. [online] GitHub. Available at: https://github.com/OpenRA/OpenRA/commit/b59ba43934a3a6837410db51cf60157cf854e52d [Accessed 23 May 2025].

Atlassian (n.d.) *What is version control: Atlassian Git Tutorial*, *Atlassian*. Available at: https://www.atlassian.com/git/tutorials/what-is-version-control (Accessed: 23 May 2025).

Chiradeep BasuMallick (2025) *What is cloud storage? definition, types, benefits, and best practices - spiceworks*, *Spiceworks Inc*. Available at: https://www.spiceworks.com/tech/cloud/articles/what-is-cloud-storage/ (Accessed: 23 May 2025).