ECE 309 Final Project Written Report

Joseph Homenick, Jack Maxwell, Trey Parker, Nick Urch

Table of Contents

Github Repository

Here is the link to the Github Repository: https://github.ncsu.edu/jtuck/ECE-309-Final-Project

The Game

The game we chose to implement is the board game *Sorry!* . We decided to implement most of the standard *Sorry!* game rules. The only modifications that we made was the order in which players take their turns. In the standard *Sorry!* rules, the youngest player goes first. Instead, we decided that the order of the players' turns would be based on the color of their pieces (i.e. the order is always the same).

Project Design

Sorry! Board (`Sorry_Board` Class)

Implemented using a 16 x 16 multidimensional array of pointers (`Sorry_Board` class). Each array element corresponds to a spot on the physical *Sorry!* board. The array of pointers is initialized to NULL. If a spot on the board is occupied by a player's *Sorry!* pawn, the corresponding array element points to the player's pawn (i.e. if a spot is empty the array element will be NULL, otherwise it will point to a player's Sorry! piece).

The `Sorry_Board` class contains members for each player's pawns. There are 4 arrays for each player color which contain 4 pawns each. The constructor sets the start positions for each pawn. There is also a function which shows the 'physical' *Sorry!* board. There is a function to check if a player's pawn is in the 'slide' area of the board. There are functions to move a pawn, send a pawn to start, and swap the positions of two pawns.

Card Deck (`Sorry_Deck` Class)

The card deck contains 44 cards. There are 4 of each of the following types of cards: 1, 2, 3, 4, 5, 7, 8, 10, 11, 12, and Sorry! Below are the specific functionalities of each card.

1: Start a pawn out or move the pawn forward one space
2: Start a pawn out or move the pawn forward two spaces
3: Move a pawn forward three spaces
4: Move a pawn backwards four spaces
5: Move a pawn forward five spaces
7: Either move a pawn forward seven spaces or split the move between two pawns. All

seven moves must be used. Cannot be used to start a pawn.
8: Move a pawn forward eight spaces
10: Move a pawn forward ten spaces or move a pawn backwards one space
11: Move a pawn forward eleven spaces or switch any one of your pawns with any one of of your opponent's pawns. You may forfeit your turn if you do not wish to switch with one of your opponent's and it is not possible to move eleven spaces forward.
12: Move a pawn forward twelve spaces
Sorry!: Move one of your pawns from your start and replace it with any opponent's pawn and bump the opponent's pawn back to their start. If there is no opponent's pawn on any space or no pawn on your start, forfeit you turn.

* If a player cannot move the number of spaces drawn, then the player forfeits the turn.

The `Sorry_Deck` class contains functions to shuffle the card deck, draw from the card deck, and display text for each card.

Players (`Sorry_Player` Class)

The `Sorry_Player` class is used to implement the players for the game. It contains two members: the player's color and the player's turn number. The Sorry_Player constructor takes an integer (the turn number) 0 - 3, assigning the color to a player based on the following: 0 = Red, 1 = Blue, 2 = Yellow, and 3 = Green. This class has functions to return the color and turn number of the player. It has a virtual function that allows human players to make a choice when they draw certain cards. This function is implemented in derived human player and computer player classes.

Pawns (`Sorry_Pawn` Class)

The `Sorry_Pawn` class is used to implement each player's *Sorry!* pawn in the game. The `Sorry_Pawn` class has 7 member fields and 13 member functions. Below is a brief description of each member. The constructor sets all the pawns to the start positions on the board.

Member Fields

- `const std::string color`: color of the *Sorry!* Pawn
- `int xPos and int yPos:` the row and column positions of the pawn in the board array.
- `bool isHome, bool isSafe, bool atStart`: boolean members indicating whether the pawn is in the player's home, safe, or start spaces, respectively
- `int pieceNum:` integer (0 - 3) indicating the player's four pawns

Member Functions

- `int getXPos and int getYPos:` returns the row and column positions of the pawns, respectively
- `void setXPos and void setYPos:` sets the row and column positions of the pawns, respectively
- `bool getSafe, bool getHome, bool getStart:` returns whether or not the pawn is in a safe space, home space, or start space, respectively
- `void setSafe, void setHome, void setStart:` sets the member fields `isSafe`, `isHome`, and `atStart` to true or false based on the pawn's position on the board, respectively.
- `void update_pawn_status:` calls the `setSafe`, `setHome`, and `setStart` functions to update the status of the *Sorry!* Pawn
- `std::string getColor and int getPieceNum:` returns the pawn's color and piece number, respectively.

Game Rules (`Card_Rules` Class)

The `Card_Rules` class is used to implement the rule action of each of the cards used in the *Sorry!* game. It contains functions responsible for checking for valid movements, starts and swaps, and the implementation of those movements, starts, and swaps of pawns. It also contains the functions responsible for receiving and implementing player choice action of a card and automating the computer's choice.

Member Functions

- `Check_For_Valid_Move, Check_For_Valid_Swap, Check_For_Valid_Start, Check_For_Valid_Sorry:` boolean functions that return whether or not the requested action (move, swap, or start) is a valid movement for the pawn selected
- `Ask_Player_For_Move, Ask_Computer_For_Move:` void functions that asks the player which option (A or B) the player would like to choose and implements that option, and automating the computer choice and implementing that choice
- `Start_Pawn, Move_Pawn, Swap_Pawn, Sorry_Move, Pawn_to_Start:` void functions to implement the actions of the cards in the deck
- `Check_Win:` boolean function to check if there is a winner

Game State (`int main()`)

The `main` function implements the game state using a state machine. Below is a brief description of the states:

- State 0: Initialize objects and choose players; go to state 1.

- State 1: Start game and draw card; go to state 2.
- State 2: Let player decide move or let computer choose move; go to state 3.
- State 3: Check for a winner; if there is a winner, go to state 4. Otherwise, update the next player and go to state 1.
- State 4: There is a winner; ask if another game is to be played. If yes, go to state 0. Otherwise, stop the program and exit.

R1 - R3 Requirements

- R1: Game logic and game state is implemented using the classes and functions listed above in the Project Design section and the `main` function of our program.
- R2: We implemented players for the board game by designing a `Sorry_Player` class. There is a `Console_Player` class, allowing a human user to play against a computer player (`Computer_Player` class). The `Computer_Player` class will draw card's and move its pawns.
- R3: The game is 'watched' in the terminal. The `main` function displays each player's turn/moves, and it displays the name of the winner of the game once the end of the game is reached.

C1 - C4 Requirements

C1

- Encapsulation: Each class has members and member functions specific to the class object. For example, the `Sorry_Pawn` class has the x and y array positions as members, and member functions that return and manipulate each individual pawn's x and y coordinates. The data and functions that manipulate each class shows encapsulation.
- Abstraction : Using header files and source code files are an example of abstraction in our program. When in `main.c`, the user only sees the functions from the header files called and variables passed into the functions. The underlying code in the source code files are not seen by the user. Another form of abstraction is how each class uses access specifiers. The `private` specifier used in each class protects certain members of each class from being manipulated, such as the color in the `Sorry_Pawn` class.
- Inheritance: The `Computer_Player` and `Console_Player` class inherit from the public `Sorry_Player` class.
- Polymorphism: Appears in the `Computer_Player` and `Console_Player` allowing the classes to behave differently to ask a human player to make a move or allow the program to select the move for the computer player. Also, in the `Sorry_Board` class, operator overloading is used for the assignment operator, copy constructor, and deconstructor.

C2

- The `Sorry_Board` class implements the destructor, copy constructor, and assignment operator, which demonstrates the Rule of Three.

C3

- The `Computer_Player` and `Console_Player` class inherit from the public `Sorry_Player` class. There is a virtual function `bool getIsHuman`.

C4

- The *Sorry!* board is implemented using a two-dimensional array data structure. The board is an array of pointers to *Sorry!* pawns. This choice allows for O(1) element access.

Team Member Contributions

Joseph Homenick

- Created initial design of *Sorry!* board data structure implementation (multidimensional array of pointers)
- Responsible for implementation of `Sorry_Pawn`, `Sorry_Player`, and `Sorry_Board` classes
- Written Report

Jack Maxwell

- Responsible for the creation and implementation of `Card_Rules` class member pawn movement manipulation functions
- Responsible for debugging movement functions and updating them to display the resulting array
- Written Report

Trey Parker

- Responsible for the creation and implementation of `Card_Rules` class member player movement function
- Responsible for initial implementation of card functionality. Specifically in the player decision function(`Ask_Player_For_Move`).
- Responsible for aiding debugging and implementing functions related to the player decision function.

Nick Urch

- Responsible for the `main.cpp` Finite State Machine and Board Display function.
- Implemented the computer movement function.
- Implemented the `Sorry_Deck` class.
- Implemented Sorry_Player and the derived classes of `Console_Player` and `Computer_Player`.
- Written Report.