دانشگاه بوعلی سینا

Advanced programming course
Final project phase 1

Rakab game project report

Prof. Mehdi Sakhaei-nia

Pouya Omidi 40212358002
Pouya Tavakoli 40212358011

TABLE OF CONTENTS

Introduction

This report details the development of a game using C++ and Object-Oriented Programming (OOP) principles.
The game is a strategic card game where players compete by playing different types of cards and acquiring provinces.
The objective is to utilize OOP concepts such as inheritance, polymorphism, and modularity to create a scalable and maintainable codebase.

The main project has multiple phases. In each phase we focus on some aspect of the project.
In phase 1 the focus is on implementing game base logic and making a fully functional text based game in terminal.
This code base was developed by a team of two on Linux Ubuntu and Microsoft Windows and the code runs perfectly on both operating systems.
To effectively manage the codebase and collaborate efficiently, we utilized Git for version control and GitHub as our remote repository.
These terms are used in this report that have these specific meanings :

Battle : one round of game played for capturing one province
Table : a players table is the list of cards the player has played in this battle so far
Hand : players hand is cards handed to that player in the beginning and not played yet

Game Overview

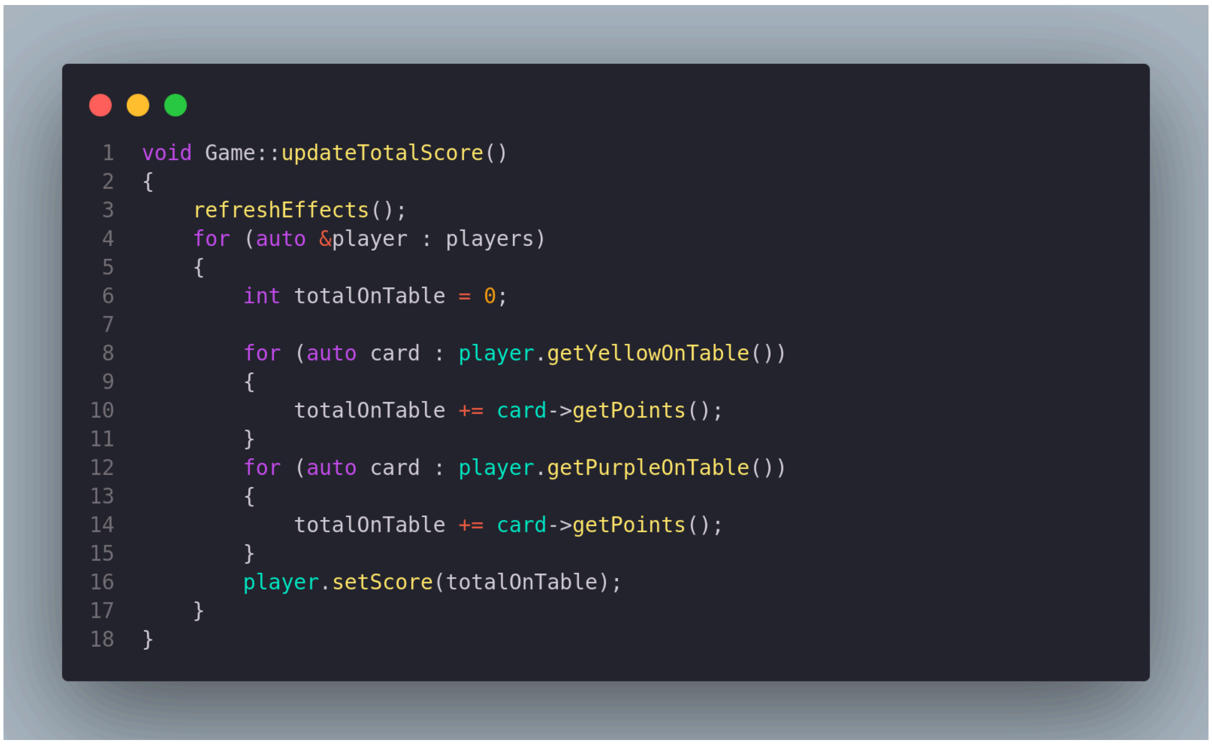
The game involves multiple players who take turns playing cards from their hands. There are two types of cards: Yellow Cards and Purple Cards. Yellow Cards only contribute points, while Purple Cards have special abilities that can influence the game. The ultimate goal for each player is to acquire five provinces.or three neighboring provinces. The player who first acquires five provinces in total or 3 neighboring provinces , wins the game.

Game Class Relations

Our program is a strategic card game designed with a clear and organized class structure. This structure separates different parts of the game, making it easy to manage and expand. The main classes—Game, Player, Card types, Map, and Interface—work together to create an engaging gameplay experience with diverse mechanics and player interactions.
This design makes it simple to add new cards, rules, and features in the future. As a result, the game can be adapted to various scenarios and strategies, ensuring that it remains interesting and fun for players.

Scoring system

The game calculates players scores based on cards on table vectors. Interesting thing in this part is that each card has two numbers. Number written on the card and points . The number on the card is constant and keeps the original value of the card. The point is initialized to number on the card but changes during the game depending on game condition. This separation card's original value and points during the game allows us to have live update of players scores during the game in any condition using only one function.



```
void Game::updateTotalScore()
{
    refreshEffects();
    for (auto &player : players)
    {
        int totalOnTable = 0;

        for (auto card : player.getYellowOnTable())
        {
            totalOnTable += card->getPoints();
        }
        for (auto card : player.getPurpleOnTable())
        {
            totalOnTable += card->getPoints();
        }
        player.setScore(totalOnTable);
    }
}
```

Classes OVERVIEW

Game class

Manages the overall game logic and flow, including initializing the game, handling player interactions, managing game phases (such as battles and seasons), and determining the winner. It also maintains the game state, including the map, players, and card deck.
This class has some essential functions that we talk about here :

`run`() : This function does exactly what it says , it runs the game and you can play.
`startBattle`() : this function initiates a battle phase in the game within a specified province. This function handles player actions, card plays, and determines the winner of the battle. It utilizes the `Interface` class to interact with players and manage the game state.
Note : determining the winner of battles in `startBattle` is by the help of other functions like `checkThisBattleWinner` .

`reorderPurpleOnTable`

`reorderPurpleOnTable()` : there is a predefined order for applying effects of cards on game situations this function ensures that the purple cards on each player's table are sorted according to a predefined order. This function uses an unordered map to define the card order, retrieves the purple cards for each player, sorts them using a custom comparison function, and updates the player's cards with the sorted list. This functionality is essential for maintaining a consistent and logical order of cards during the game, which can affect gameplay and strategy.

+An unordered map is used to specify the order of the card types, where each card type is assigned a priority value.
+A lambda function (`compareCards`) is defined to compare two cards based on their priority in the `orderMap`.
+The `sort` function is used to sort the player's purple cards using the `compareCards` lambda function.

Player class

Represents a player in the game, managing player-specific attributes such as age, name, score, and the cards they hold. It handles player actions such as playing cards, passing turns, and interacting with the game state. The Player class also tracks the player's status and eligibility to participate in the game.

This class has some essential functions that we talk about here :
`playPurpleCard() :` manages the logic for playing a purple card by:
Finding the specified card in the player's hand.
Determining the type of card and applying its effects.
Updating the player's state by moving the card from their hand to the table and marking it as played.
Returning specific values based on the type of card played or an error code if the card cannot be played.
+This function is essential for managing the strategic aspects of playing purple cards and ensuring that the game rules are adhered to when these cards are used.

`applyEffect() :` The `applyEffect` function is responsible for activating the effects of specific purple cards that are currently on the player's table.
Iterates through the purple cards on the table , Ensures each card is evaluated, Checks for specific card types and uses dynamic casting to apply the card's effect, ensuring type safety and proper error handling.
This function is crucial for managing the dynamic aspects of gameplay, ensuring that card effects are correctly activated during a player's turn.
Note : the season effects are activated by the Game class functions and not here.

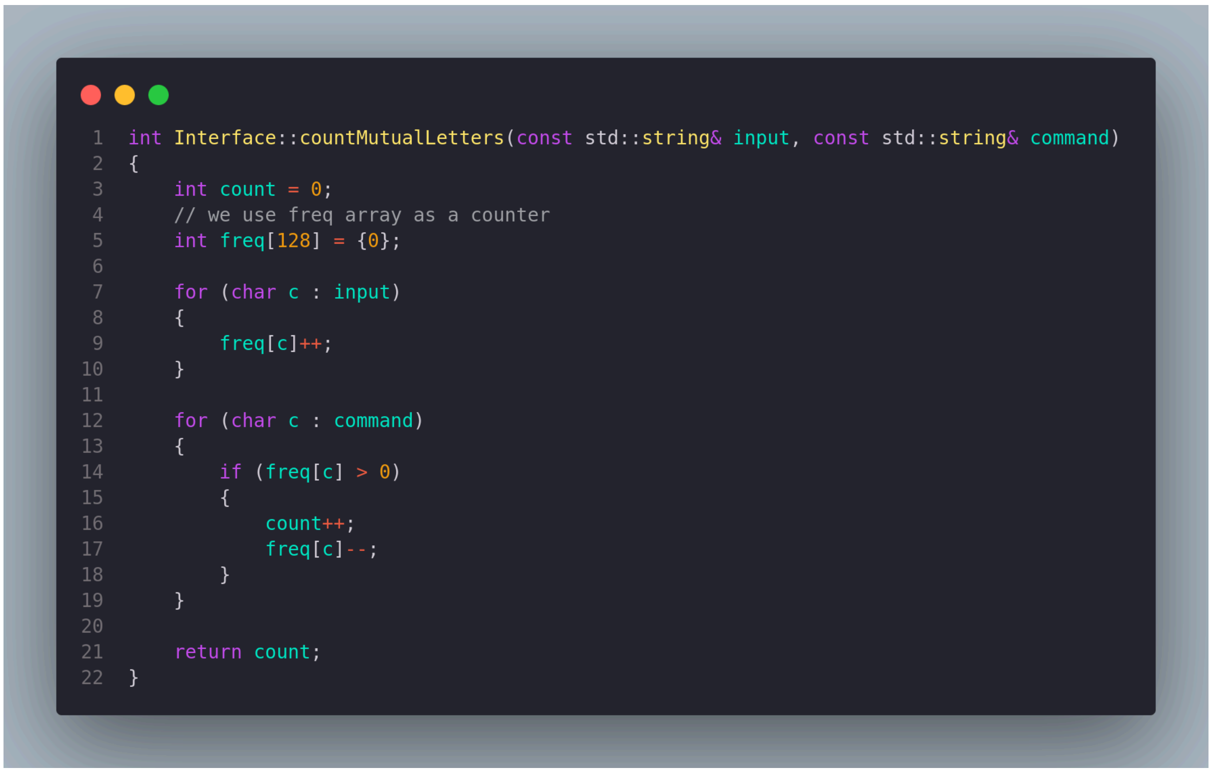
Interface class

The `Interface` class is essential for managing all user interactions within the game. It provides methods for both gathering input from the players and displaying the necessary game information to them. By abstracting these operations, it allows the core game logic to remain clean and focused on the game mechanics.

Did you mean this instead of that?

One of the key functions in the interface class is the typo-checking function. This function compares the input command with a dictionary of commands and returns the closest matching word to the input.

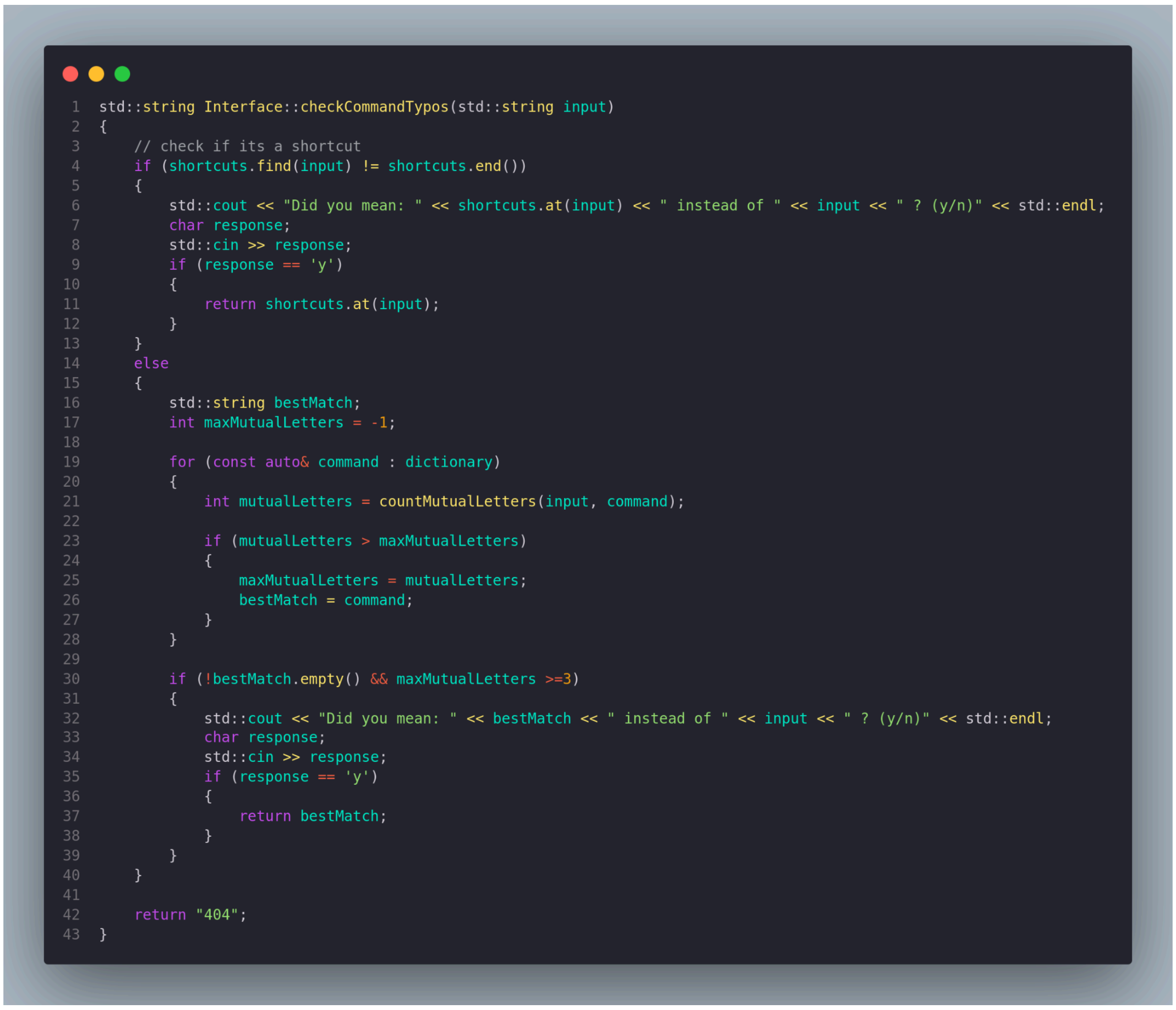
The function in the picture counts mutual letters between two strings



```cpp
int Interface::countMutualLetters(const std::string& input, const std::string& command)
{
    int count = 0;
    // we use freq array as a counter
    int freq[128] = {0};

    for (char c : input)
    {
        freq[c]++;
    }

    for (char c : command)
    {
        if (freq[c] > 0)
        {
            count++;
            freq[c]--;
        }
    }

    return count;
}
```

The function in the following page returns the best match or returns “404”
If the functions returns “404” we know that there is no close match.

The “check command typo” function checks if the input command is a shortcut using a map and then if it's not a shortcut checks for the closest match.



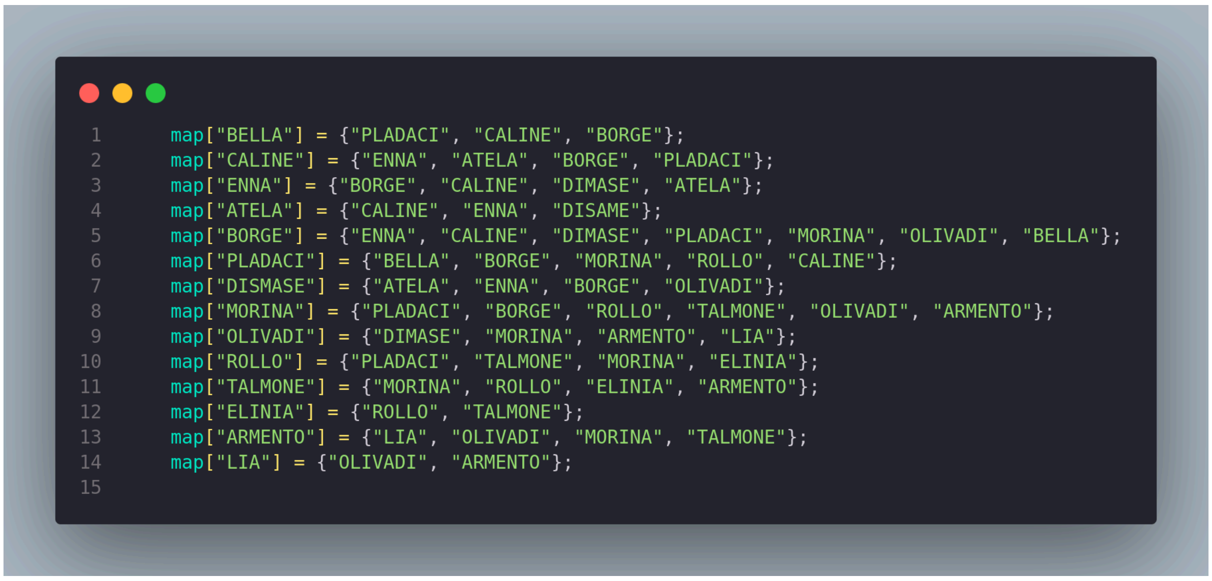
```cpp
std::string Interface::checkCommandTypos(std::string input)
{
    // check if its a shortcut
    if (shortcuts.find(input) != shortcuts.end())
    {
        std::cout << "Did you mean: " << shortcuts.at(input) << " instead of " << input << " ? (y/n)" << std::endl;
        char response;
        std::cin >> response;
        if (response == 'y')
        {
            return shortcuts.at(input);
        }
    }
    else
    {
        std::string bestMatch;
        int maxMutualLetters = -1;

        for (const auto& command : dictionary)
        {
            int mutualLetters = countMutualLetters(input, command);

            if (mutualLetters > maxMutualLetters)
            {
                maxMutualLetters = mutualLetters;
                bestMatch = command;
            }
        }

        if (!bestMatch.empty() && maxMutualLetters >=3)
        {
            std::cout << "Did you mean: " << bestMatch << " instead of " << input << " ? (y/n)" << std::endl;
            char response;
            std::cin >> response;
            if (response == 'y')
            {
                return bestMatch;
            }
        }
    }

    return "404";
}
```

Map class

The Map class is responsible for maintaining the structure and relationships of the game's provinces. It provides methods to access adjacent provinces and

validate the existence of a province, facilitating various game mechanics that depend on spatial relationships. By encapsulating these functionalities, the Map class ensures the game's geographical data is organized and easily accessible.

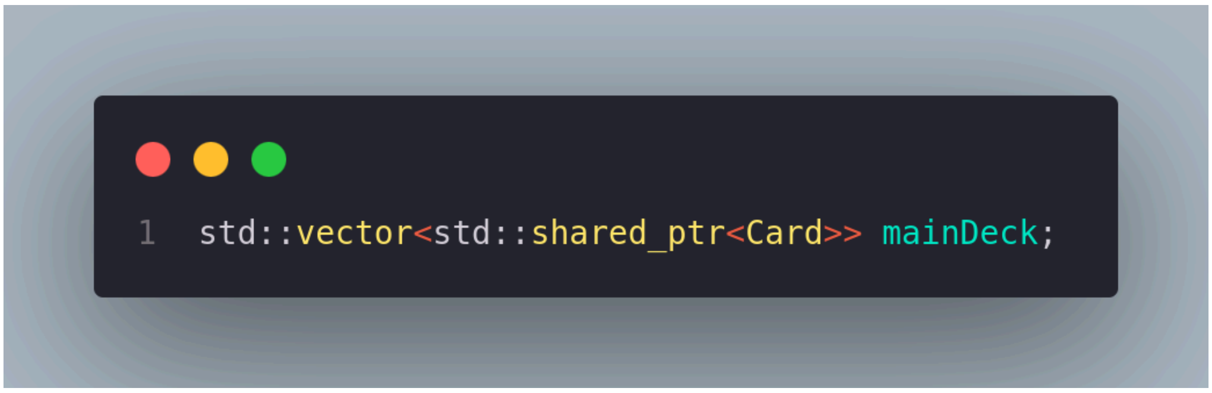
This picture shows how we managed to keep neighboring provinces and find a winner in the situation where the player has 3 provinces but they are neighboring provinces.



```
map["BELLA"] = {"PLADACI", "CALINE", "BORGE"};
map["CALINE"] = {"ENNA", "ATELA", "BORGE", "PLADACI"};
map["ENNA"] = {"BORGE", "CALINE", "DIMASE", "ATELA"};
map["ATELA"] = {"CALINE", "ENNA", "DISAME"};
map["BORGE"] = {"ENNA", "CALINE", "DIMASE", "PLADACI", "MORINA", "OLIVADI", "BELLA"};
map["PLADACI"] = {"BELLA", "BORGE", "MORINA", "ROLLO", "CALINE"};
map["DISMASE"] = {"ATELA", "ENNA", "BORGE", "OLIVADI"};
map["MORINA"] = {"PLADACI", "BORGE", "ROLLO", "TALMONE", "OLIVADI", "ARMENTO"};
map["OLIVADI"] = {"DIMASE", "MORINA", "ARMENTO", "LIA"};
map["ROLLO"] = {"PLADACI", "TALMONE", "MORINA", "ELINIA"};
map["TALMONE"] = {"MORINA", "ROLLO", "ELINIA", "ARMENTO"};
map["ELINIA"] = {"ROLLO", "TALMONE"};
map["ARMENTO"] = {"LIA", "OLIVADI", "MORINA", "TALMONE"};
map["LIA"] = {"OLIVADI", "ARMENTO"};
```

Card class

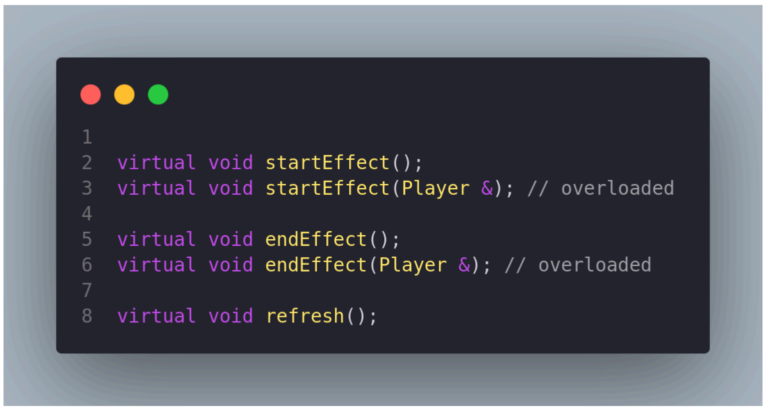
The `Card` class provides a common structure and interface for all card types in the game. By defining pure virtual functions, it ensures that derived classes (such as specific types of purple and yellow cards) implement essential card behaviors. This design allows for flexible and extendable card management within the game.
This class also allowed us to use pointers of type “card” for pointing to all types of cards in the vector of all cards in game.



```
std::vector<std::shared_ptr<Card>> mainDeck;
```

PurpleCard class

The `PurpleCard` class represents a specialized type of card within the game, inheriting from the base `Card` class and adding functionality unique to purple cards. This includes methods for starting, ending, and refreshing the effects of the card, both generally and for specific players. Derived classes must implement the pure virtual getter functions to provide the specific details for each type of purple card.
This class also helps us in using polymorphism in code



```

virtual void startEffect();
virtual void startEffect(Player &); // overloaded

virtual void endEffect();
virtual void endEffect(Player &); // overloaded

virtual void refresh();
```

Matarsak class

The `Matarsak` class extends the `PurpleCard` class, implementing specific behaviors and attributes for the Matarsak card. It includes getters, setters, and effect functions to integrate the Matarsak card into the game's mechanics, allowing meaningful interaction with players and the game's state.

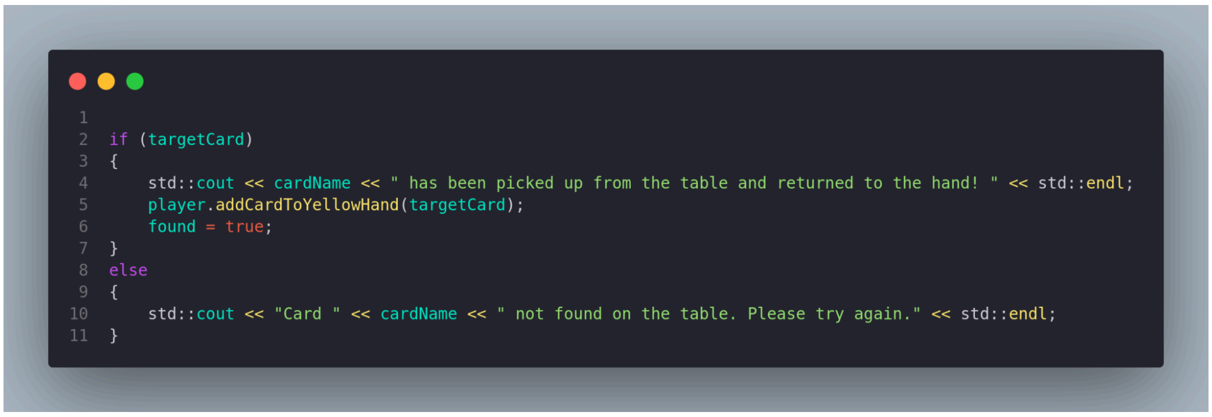
`startEffect()`: This function in this card retrieves a yellow card from the table and returns it to the player's hand, ensuring the effect is applied correctly by prompting and validating user input. This design maintains game mechanics and provides a strategic advantage to the player.

ShirDokht class

The `ShirDokht` class is a concrete implementation of the `PurpleCard` class, representing a specific type of card in the game. It overrides the necessary methods to provide the specific characteristics and behaviors of the `ShirDokht` card. This class can be instantiated and used within the game to perform actions and interact with other game components based on its defined properties.

Note : The card has a fixed point value of 10, which is taken into consideration when updating the total score in the `Game` class.

Here is a simple part of "matarsak" functionality it returns card to player hand



```

if (targetCard)
{
    std::cout << cardName << " has been picked up from the table and returned to the hand! " << std::endl;
    player.addCardToYellowHand(targetCard);
    found = true;
}
else
{
    std::cout << "Card " << cardName << " not found on the table. Please try again." << std::endl;
}
```

TablZan class

The `TablZan` class is a concrete implementation of the `PurpleCard` class, representing a specific type of card (`TablZan`) in the game. It defines the behavior of the card through its overridden methods, including its name, type, point value, and specific effects (`startEffect` and `endEffect`). This class can be instantiated and used within the game to perform actions and interact with other game components based on its defined properties.The important function in this class is :
`startEffect():` The `startEffect` method in the TablZan class is designed to double the points of all yellow cards currently on the table for a specific player when the TablZan card is played.

Season class

The `Season` class represents a type of `PurpleCard` that signifies different seasons within the game.
This structure sets up the card classes as part of a hierarchy that we can expand upon for other types of season cards in our game.

Spring class

The `Spring` class represents a specific type of season card that inherits from the `Season` class.

Winter class

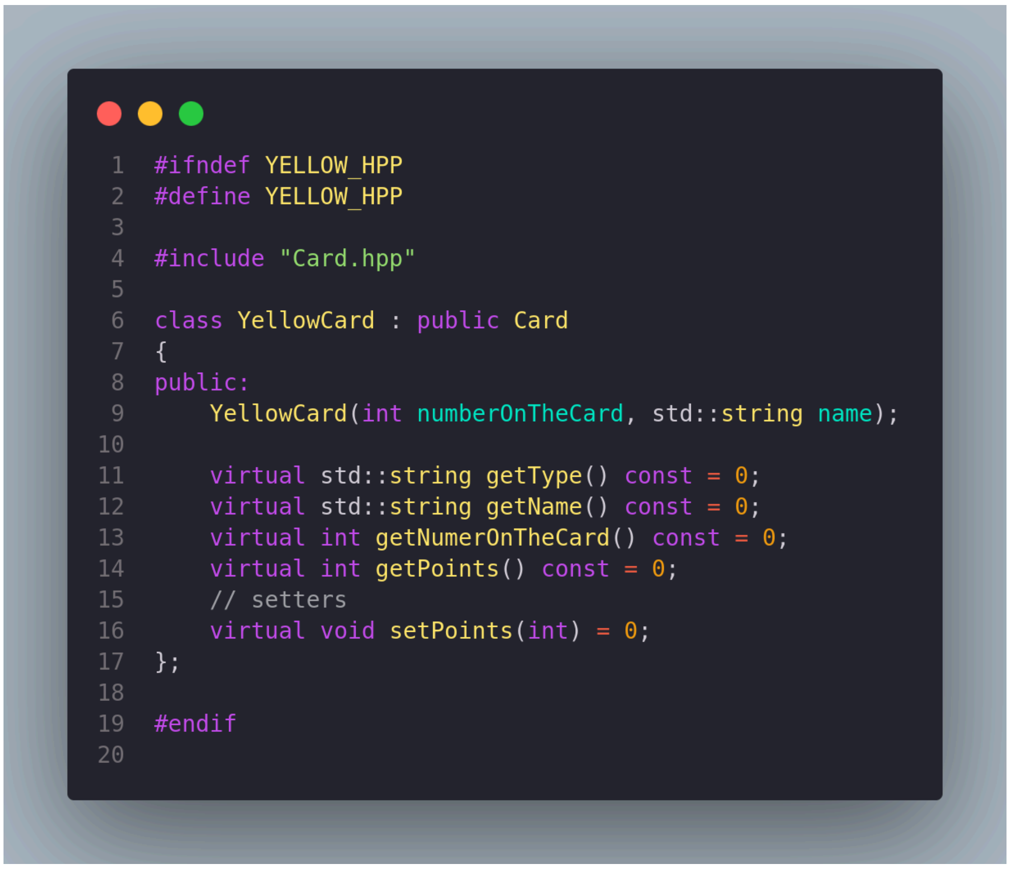
The `Winter` class represents a specific type of season card that inherits from the `Season` class.

YellowCard class

The `YellowCard` class provides a specialized type of card within the game, inheriting from the base `Card` class and adding functionality unique to yellow cards. By defining pure virtual functions for getters and setters, this class ensures that derived classes implement specific behaviors and attributes for each type of yellow card. This design allows for flexible and extendable management of yellow cards within the game.
+By creating a `YellowCard` base class, we avoid duplicating common code across multiple yellow cards (`Yellow1` to `Yellow10`).

Yellow1 -Yellow10 classes

We have multiple yellow cards (Yellow1 to Yellow10) that follow a similar structure.
Note : These cards have a fixed point value(numberOnTheCard) and a changeable one (points), which is taken into consideration when updating the total score in the `Game` class.



```cpp
#ifndef YELLOW_HPP
#define YELLOW_HPP

#include "Card.hpp"

class YellowCard : public Card
{
public:
    YellowCard(int numberOnTheCard, std::string name);

    virtual std::string getType() const = 0;
    virtual std::string getName() const = 0;
    virtual int getNumerOnTheCard() const = 0;
    virtual int getPoints() const = 0;
    // setters
    virtual void setPoints(int) = 0;
};

#endif

```

Challenges Faced

Implementing different card effects using polymorphism

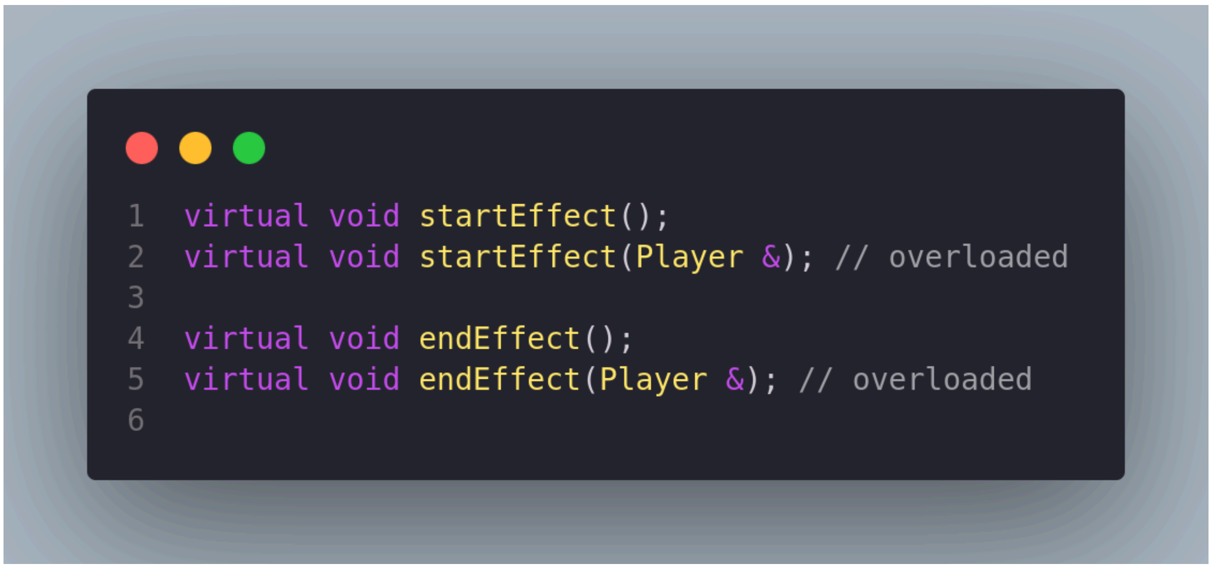
One of the main challenges was implementing polymorphism correctly, ensuring that the correct effect method is called for each card When a player uses a purple card, the effects can vary widely. These cards might impact only the player who used them, alter the entire game's rules, or affect some or all of the other players. For example, the card "TablZan" only affects the player who played it. In contrast, "Winter" impacts all players in the game, while "Spring" cancels the effect of "Winter" and modifies the scores of some, but not all, players.

Implementing these diverse effects using polymorphism to ensure all cards function correctly was one of the main challenges in this project.

Game rules and logic

The game features numerous rules for various conditions, and ensuring that each condition is met accurately was a significant challenge during implementation.

Overloading and overriding functions in the picture below shows how polymorphism was used to write effect functions.



```cpp
virtual void startEffect();
virtual void startEffect(Player &); // overloaded

virtual void endEffect();
virtual void endEffect(Player &); // overloaded

```

Links and resources

https://en.cppreference.com/w/
https://stackoverflow.com/
Deitel, P., & Deitel, H . *C++ How to Program* . Pearson
https://www.geeksforgeeks.org/
https://chatgpt.com/

We used GitHub for version control and teamwork, allowing us to collaborate effectively and track our progress. You can view the full project, including all commits and development milestones, on our GitHub repository.

(Note that GitHub repository is currently private Teaching Assistants can request access, which will be granted immediately upon asking)