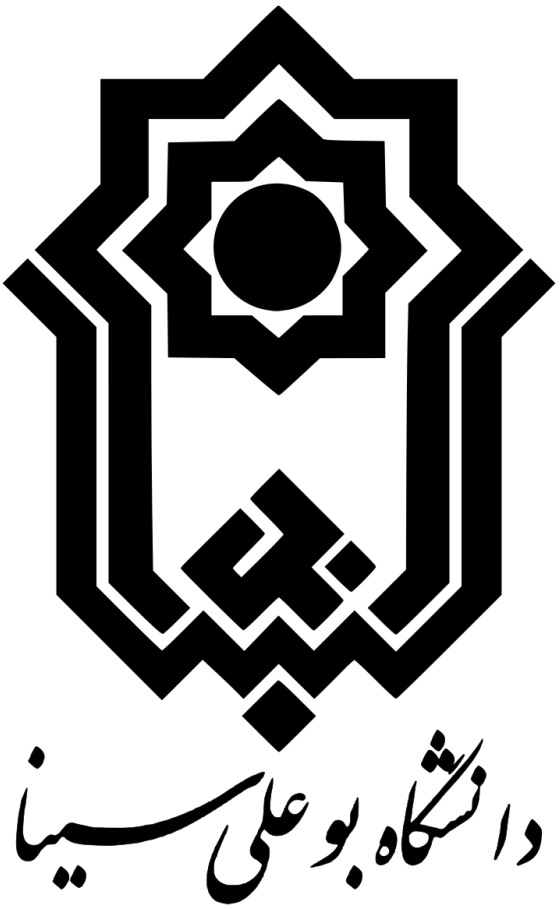


Advanced programming course

Final project phase 2

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 graphical game using C++ , Qt framework 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 was on implementing game base logic and making a fully functional text based game in terminal.
In this phase we created a graphical interface for the game.

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. For the graphical part we used Qt framework which is cross platform and the once written code can be built on multiple 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.

```cpp
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`() : 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.

```qobject.h
class Player : public QObject {
    Q_OBJECT

public:

    Player(QObject *parent = nullptr);
    Player(int age, const std::string &name, QObject *parent = nullptr);


```

The player class inherits from QObject.

Why Inherit from QObject?

In Qt, QObject is the foundation for object-oriented programming and is crucial for creating:

- **Signal and Slot Mechanism:** QObject enables the powerful signal-slot mechanism, a central feature of Qt. This mechanism lets objects communicate with each other in a decoupled, type-safe, and efficient manner.

- **Memory Management:** QObject provides features for automatic memory management using the parent-child relationship. When a parent QObject is deleted, its child QObjects are deleted automatically, reducing the risk of memory leaks.

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 represents a foundational component of the card game, encapsulating common properties and behaviors for all card types. By inheriting from `QLabel`, it integrates seamlessly with Qt's widget system, allowing each card to be displayed as a label in the user interface.

The `setImage()` method sets the card's image using a given path, resizing it appropriately for display. This method ensures that card images are consistently presented in the user interface. The class overrides `mouseReleaseEvent()` to handle card click events. When a card is clicked, it emits a `clicked` signal, which allows other parts of the application (such as game logic or UI components) to respond to the click.

The `getFaceURL()` and `getBackURL()` methods provide access to the URLs for the card's face and back images, respectively. This separation allows cards to be displayed with different images based on their state (e.g., face-up or face-down).

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 the game.

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

```
1
2  virtual void startEffect();
3  virtual void startEffect(Player &); // overloaded
4
5  virtual void endEffect();
6  virtual void endEffect(Player &); // overloaded
7
8  virtual void refresh();
```

```cpp
class Card : public QLabel
{
    Q_OBJECT

public:
    explicit Card(int numberOnTheCardVal, const std::string &typeVal, const std::string &nameVal, QWidget *parent = nullptr);

    virtual std::string getName() const = 0;
    virtual std::string getType() const = 0;
    virtual int getNumberOnTheCard() const = 0;
    virtual int getPoints() const = 0;
    virtual void setName(std::string);
    virtual void setType(std::string);
    virtual void setNumberOnTheCard(int);
    virtual void setPoints(int pointsVal) = 0;
    virtual void setindexOfOwner(int) ;
    virtual int getindexOfOwner() const;
    virtual void setImage(const QString &);

    void mouseReleaseEvent(QMouseEvent *event) override;

    virtual void setFaceURL(std::string);
    virtual void setBackURL(std::string);

    QString getFaceURL();
    QString getBackURL();

    std::string toString() const;
```

The Card game has 4 special functions for getting and setting the image url of face and back of cards.

We use these functions for setting and showing the face and back of cards. We use “back” to hide cards faces when it’s not one player's turn.

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

```
1
2  if (targetCard)
3  {
4      std::cout << cardName << " has been picked up from the table and returned to the hand! " << std::endl;
5      player.addCardToYellowHand(targetCard);
6      found = true;
7  }
8  else
9  {
10     std::cout << "Card " << cardName << " not found on the table. Please try again." << std::endl;
11 }
```

ShirZan class

The `ShirZan` 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 `ShirZan` 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 1, which is taken into consideration when updating the total score in the `Game` class.
Note : In the game logic we also count the times a player has used this card and who has used the most is gonna own the Neshane Jang.

RishSefid class

The `RishSefid` 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 `RishSefid` 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 RishSefid card makes the highest cards unusable and this functionality is implemented in `Game` class.

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.

```cpp
#ifndef SEASON_HPP
#define SEASON_HPP

#include "purplecard.h"

class Season : public PurpleCard
{
    Q_OBJECT

public:
    Season(int numberOnTheCardVal, const std::string &nameVal, QWidget *parent = nullptr);

    // Override virtual functions from PurpleCard
    virtual std::string getName() const override = 0;
    virtual std::string getType() const override = 0;
    virtual int getNumberOnTheCard() const override = 0;
    virtual int getPoints() const override = 0;
};

#endif // SEASON_HPP

```

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 YELLOWCARD_HPP
#define YELLOWCARD_HPP

#include "card.h"

class YellowCard : public Card
{
    Q_OBJECT

public:
    YellowCard(int numberOnTheCard, const std::string &nameVal, QWidget *parent = nullptr);

    // Pure virtual functions to be implemented by derived classes
    virtual std::string getType() const override = 0;
    virtual std::string getName() const override = 0;
    virtual int getNumberOnTheCard() const override = 0;
    virtual int getPoints() const override = 0;
    virtual void setPoints(int pointsVal) override = 0;
};

#endif // YELLOWCARD_HPP

```

GUI

Graphical Interface - Main Menu class

The main menu of the application provides a user-friendly interface to navigate through the game's options, including starting a new game, accessing saved games, and exiting the application. Below is an explanation of the graphical components and functionality implemented in the `mainmenu` class. The `mainmenu` class inherits from `QMainWindow` and is responsible for setting up the main menu interface. It uses a Qt Designer-generated UI file (`ui_mainmenu.h`) to define the layout and elements.

Button Functionality :

Exit Button (`on_exit_btn_clicked`):

- Closes the main menu window, effectively exiting the application.

Saved Games Button (`on_saved_btn_clicked`):

- Creates and shows a new `savedGamesMenu` window to display saved games.
- Closes the current main menu window.

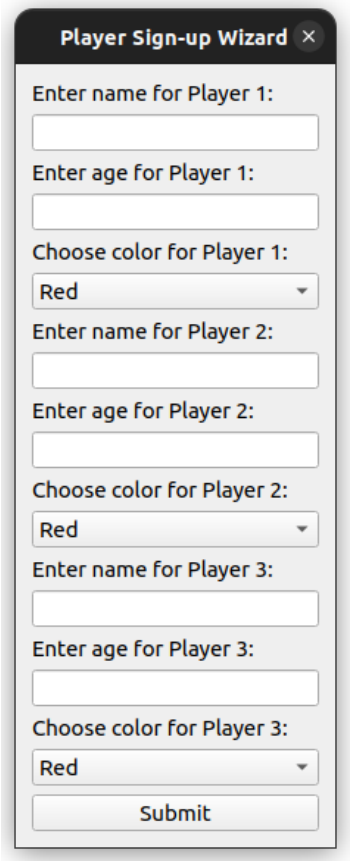
New Game Button (`on_newgame_btn_clicked`):

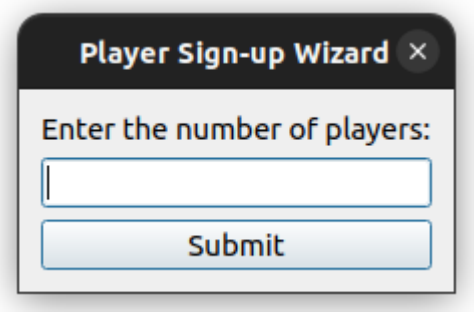
- Creates a new instance of the `Game` class.
- Creates and shows a `getPlayersInfoWindow` to gather player information for the new game.
- Closes the current main menu window.

Graphical Interface - Player Information Window

The getPlayersInfoWindow class is responsible for gathering player information before starting a new game. It provides an interface for users to input the number of players and their details, including names and ages. This window ensures that all necessary information is collected before proceeding to the game map. The getPlayersInfoWindow class inherits from QDialog and interacts with an instance of the Game class to set up the players. It uses a vertical box layout (QVBoxLayout) to organize the input fields and buttons.
In simple words this class does :

- Adds a label to prompt the user for the number of players.
- Adds a line edit for inputting the number of players.
- Adds a submit button to confirm the number of players.





Graphical Interface - Map Window

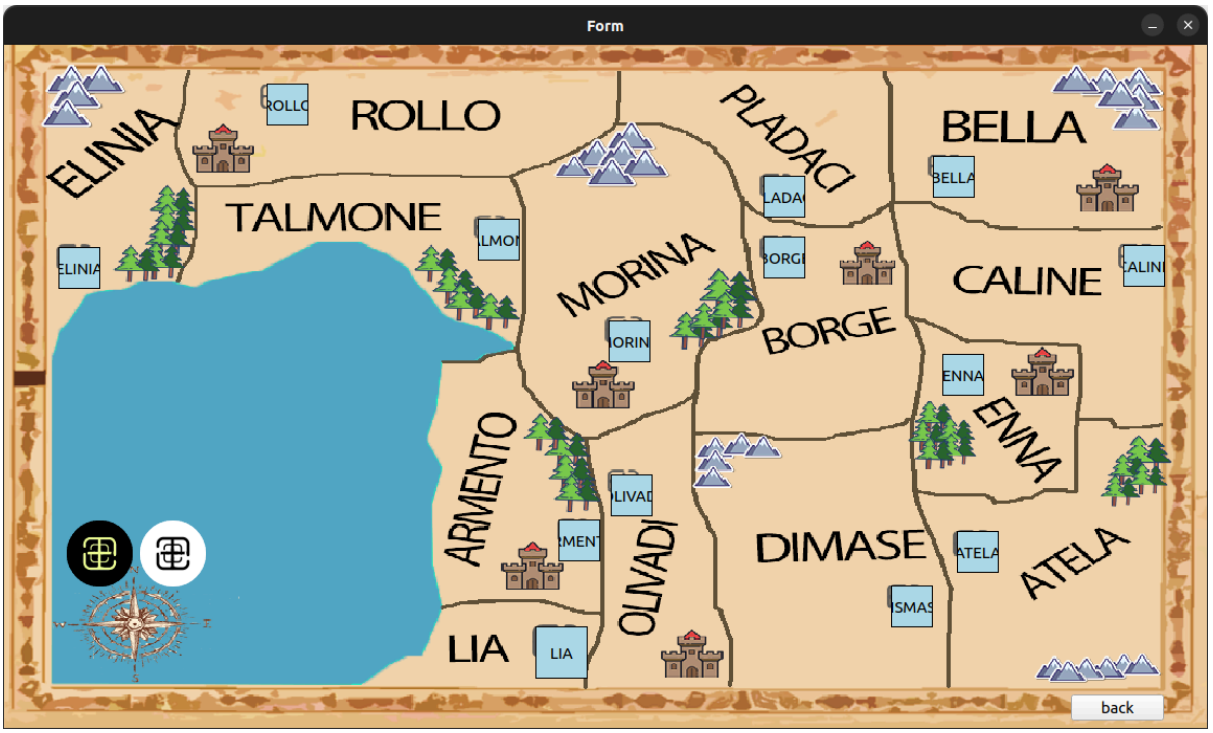
The `mapwindow` class manages a window displaying a map with draggable labels representing different regions or provinces. It supports drag-and-drop functionality to allow users to place these labels onto designated drop areas on the map. This class also handles the initiation of battles based on user interactions with these labels.
This class has some helper functions that we talk about here :

`calculateDistance` : Computes the Euclidean distance between two points to find the nearest label to a drop position.

`findNearestLabel`: Finds the label closest to the drop position based on calculated distances.

`checkAvailable` : Checks if a province is available or already taken by another player.

`askToStartBattle` : Prompts the user to confirm if they want to start a battle on the selected province.



Graphical Interface - Playground Window

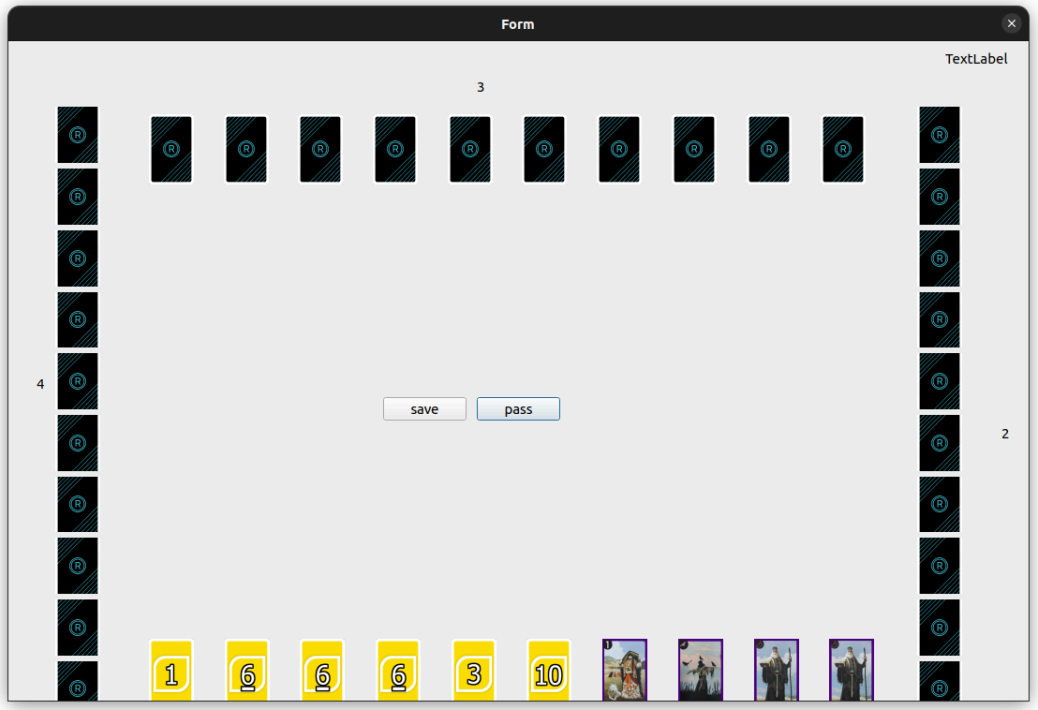
The `Playground` class represents a game playground where players interact with their hands and the table, performing various actions such as playing cards and managing game state. It integrates with the `Game` class to handle game logic and updates the user interface based on player actions and game events.
This class has some essential functions that we talk about here :

`displayCards` :

- **Player Hands**: Iterates through each player to display both yellow and purple cards. Cards are displayed with either their face or back image, depending on whether it's the current player's turn.
- **Player Tables**: Displays cards that are on the table for each player.

`UI Update`:

- `updateUi` : Refreshes the display of cards and other relevant UI elements.
- `handleCardClick` : Processes user interactions with cards, handling different card effects and updating the game state accordingly. Includes special handling for the `Matarsak` card effect and updates scores and game status.



Save and load

In this game we used text files to save game information in files and read and resume a game from file.
Save to file : to ensure similar formatting for each class saved info we used “tostring()” methods for Game , Player and Card classes.
These methods write info we need from a class into a string and return the string so we can save it into a .txt file.
Players can save up to 5 games and they can decide which game to delete when saving the 6th game or even whenever they feel like deleting a saved game .
The save graphical part consists of 5 save buttons each having their own clear button.
On top of each save button we can see its availability status which updates as soon as it changes.
We also have a “clear all” button for when you want to clear all evidence of your gameplay .

The load window has 5 buttons for loading which automatically get disabled when there os no game saved in their corresponding file.

Each part of game data has its own loading function so we can handle the information with least privilege to each part of data.
These functions are :

- bool **readMetaData**(const std::string& playerLine);
- bool **readPlayerDetails**(Player &player, const std::string &playerLine);
- bool **readProvinceDetails**(Player &player, std::ifstream &inputFile);
- bool **readCardDetails**(Player &player, std::ifstream &inputFile, const std::string &handType);
- std::shared_ptr<Card> **createCard**(const std::string &cardType);
- void **loadGame**(const std::string &fileName, std::vector<Player *> &players);

The **readMetaData** function loads flags and other info needed to load the exact previous game and other functions do as their name suggests.
The boolean returned from each function determines whether each function's job was done correctly or not for better debug and trace.

The **loadGame** function is a collection of other functions in a specific order inside and outside loops to read all the game info form a file

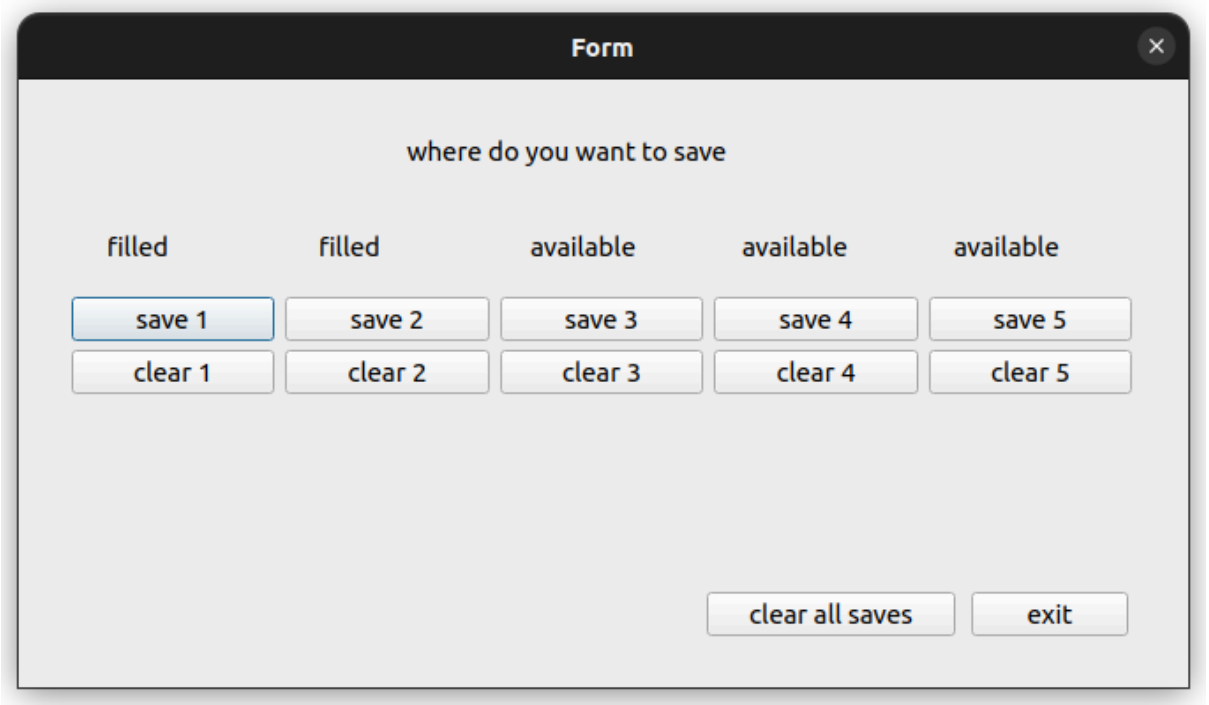
Throughout the whole game we handle cards by shared pointer The **createCard** functions responsibility is to read the file name as string and make a shared pointer for it so we are able to handle all instances cards in the whole game.

The following image is load game window which as you can see has 2 saved games and 3 empty files.

This is the save menu of the same exact game so as we can see in the picture save 1 and 2 are filled and others are available to use .

All availability checks are made possible by checking the word “empty” in a file.

We handled potential misbehavior of this check by **checkGameResources()** function call in main menu constructor.



checkGameResources

One of the most important functions related to save and load part is the check resources function which is part of **Mainmenu** class and its duty is to check all files are present and working. If there is a problem in a file or there is a file missing it creates the file so all other checks in the game can work properly. Before creating the file there is always a check if the files exists to ensure no saved game is lost. This function is called in the main menu constructor before any action is done by player.

```cpp
void mainmenu::checkGameResources() {
    qDebug()<< "checking game resources";
    for (int i = 1; i <= 5; ++i) {
        QString fileName = QString("saved_games_%1.txt").arg(i); // Create the filename

        // Check if the file exists
        std::ifstream file(fileName.toStdString());
        if (!file.good()) {
            // Create the file if it doesn't exist
            std::ofstream newFile(fileName.toStdString());
            if (newFile.is_open()) {
                newFile << "empty";
                newFile.close();
                qDebug()<< "created file : " << fileName ;

            } else {
                // Handle file creation error
                qDebug()<< "Error creating file: " << fileName ;
            }
        }
    }
    qDebug()<< "game resources checked";
}

```

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.

Qt graphics

In this phase learning Qt and its IDE, Qt Creator, quickly was a major challenge. We had to get up to speed on Qt's unique syntax and design while adjusting to a new development environment. This investment in learning paid off, allowing us to use Qt's powerful tools and create a successful application.

Despite dropping items in what appeared to be the correct positions, the system did not recognize them as valid drops. To address this, I had to refine the drop area definitions and ensure that the coordinates used for drop checks were correctly aligned with the visual layout. Adjustments in how the drop positions were calculated and validated helped resolve the issue.

Labels were not being correctly recognized during drag-and-drop operations, leading to failures in identifying where items should be placed. I needed to verify that all labels were correctly instantiated and that their properties were accurately updated. Ensuring consistent label identifiers and thoroughly testing the label recognition logic helped in resolving this problem.

When clicking on cards, they sometimes did not trigger the expected actions or were not recognized as clickable. To resolve this, I reviewed and corrected the signal-slot connections to ensure that card click events were properly routed to the appropriate handlers. I also checked the widget focus and ensured that the game logic was correctly processing the card interactions.

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.

```
1  virtual void startEffect();
2  virtual void startEffect(Player &); // overloaded
3
4  virtual void endEffect();
5  virtual void endEffect(Player &); // overloaded
6
```

Links and resources

https://en.cppreference.com/w/
https://doc.qt.io
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. The graphical development is on branch gui.

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