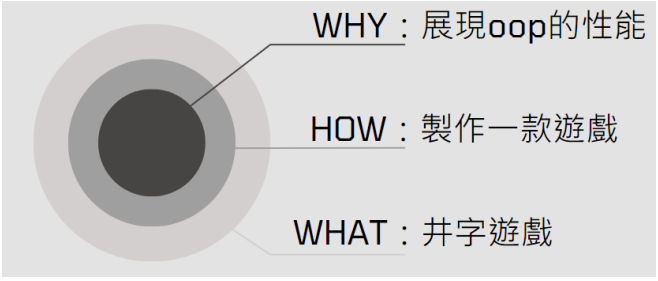
一、 動機、想法

（一）動機與目的



（圖一）

本次最大動機在於該如何完美呈現物件導向的設計，能完美的展現物件導向的必要性。

這次我們選擇了製作一款遊戲當作這次專題的目標，因為遊戲本身需要靠 user 本身的點擊來觸發事件，所以有許多事件能以封裝的方式表現出來。許多的出發機制也有相同的特性與條件，可以完美發會出繼承與多形。

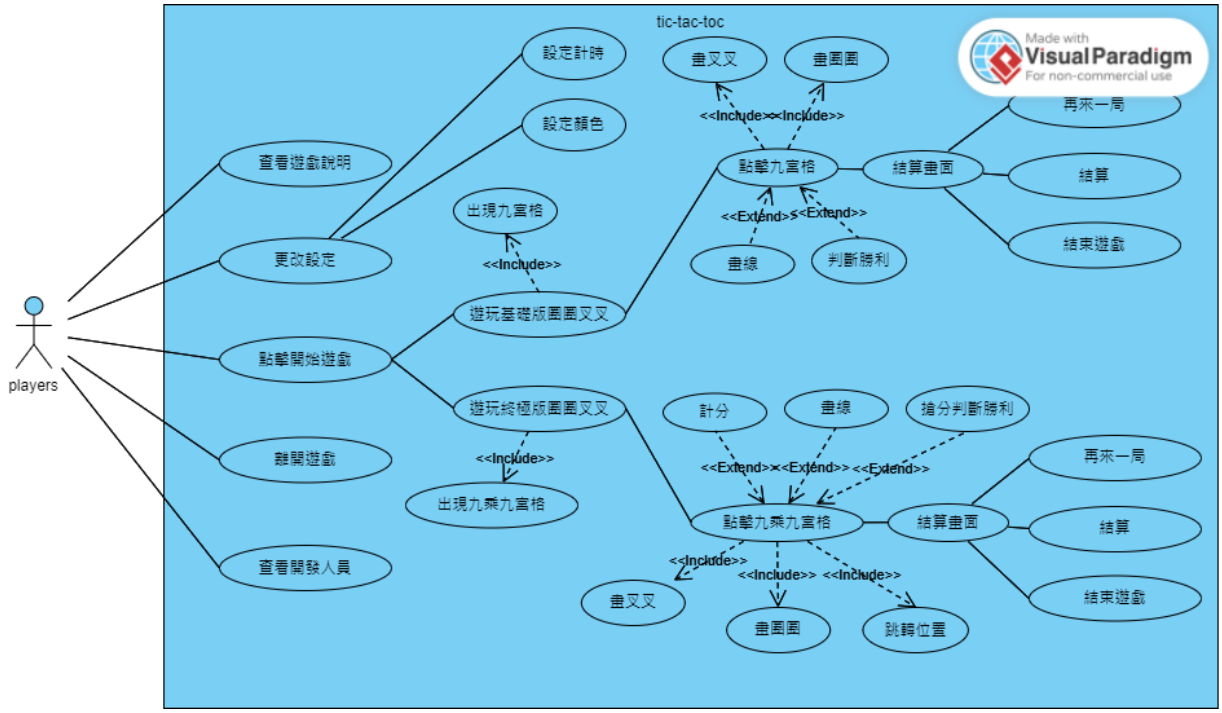
在許多遊戲中，我們選擇開發井字遊戲，雖然他的遊戲規則簡單，很好設計，但正因為他的簡單，更適合作為我們以物件導向思維的方式來訓練撰寫的對象。除了一般所認知的井字遊戲外，我們也另有設計一款進階的井字遊戲，在後面的報告會提到它的玩法。

（二）目標

- **學會物件導向的思維設計**：在撰寫前先規劃好封裝類別，類別間的關係是什麼，事件的觸發該怎麼規劃等。以 UML 的方式先設計出藍圖。最後將封裝、繼承、多行實現出來。
- **實現 UI 畫面**：為了將遊戲實現出來，我們需要 UI 來乘載我們的遊戲，因此在撰寫遊戲規則前，我們需要知道怎麼實現將 UI 實現出來。除此之外，UI 畫面也需要以物件導向的形式去設計，每個 button 將成一個封裝，在不同場景有不同的應用。
- **完成井字遊戲**：主要希望以物件導向的形式寫出井字遊戲，並將其與 UI 畫面做整合，最後將其渲染。

二、 UML

（一） user model

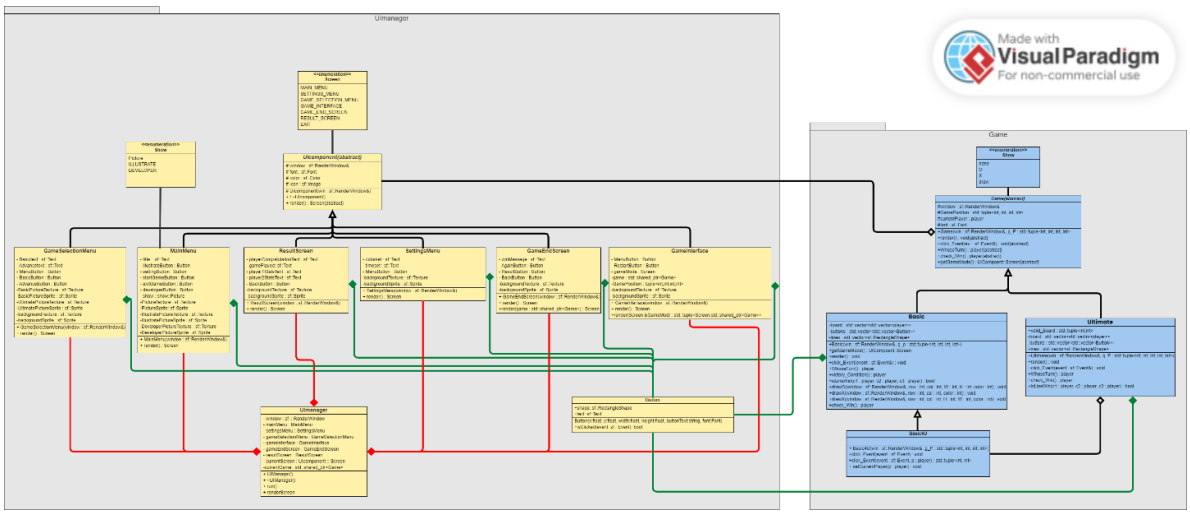


（圖二）

↑上圖為站在使用者的角度去設計的 UML，稱為 user model。<<Include>>代表當原本事件發生後必觸發的另一個事件；<<Extend>>代表原本事件發生後未必觸發的事件，可能會因為時間的關係不同未被觸發。

本專題主要功能是要讓玩家能遊玩一般的井字遊戲與進階版的終極遊戲，所以主要分支有兩個，而這兩種的功能性質相似，差別在於規則上的不同而已。

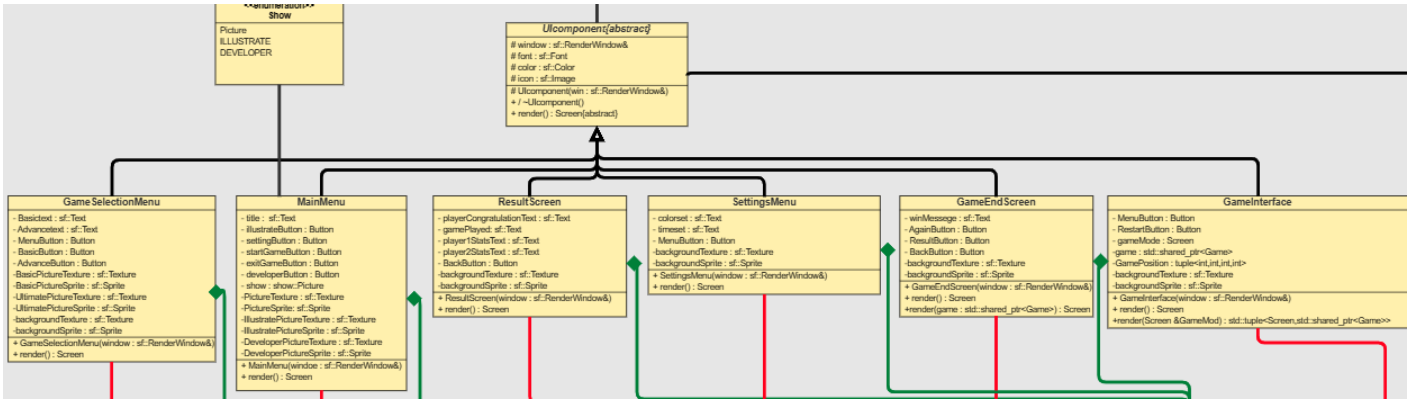
（二）class model



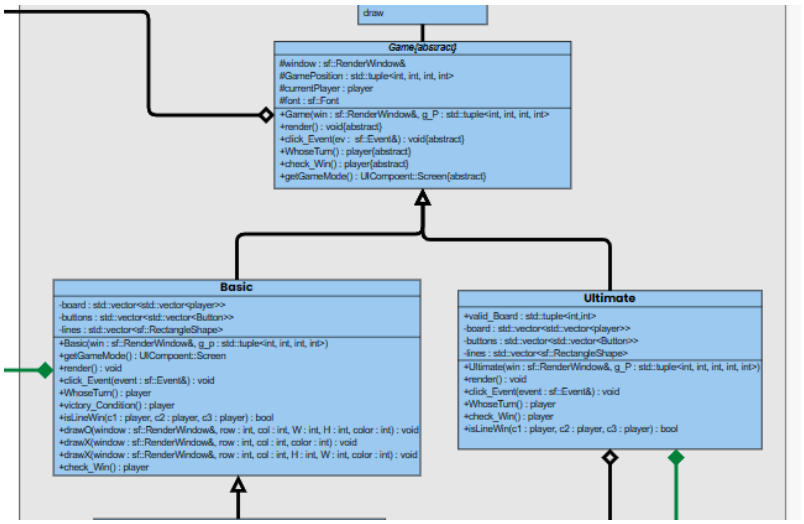
（圖三）

↑上圖是以程式設計的角度所設計的 UML，主要分成兩個區塊，分別為 UI 設計與遊戲設計，其 UML 除了表達了每個封裝間繼承與多型的關係外，也表示了類別之間的關係是強還是弱。

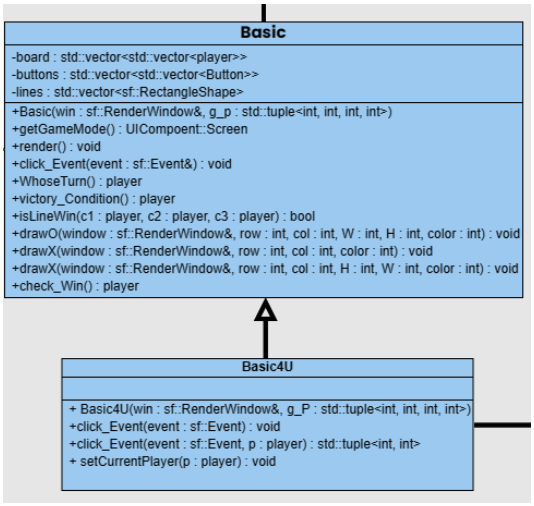
1、封裝、繼承、多型



（圖四）



（圖五）



（圖六）

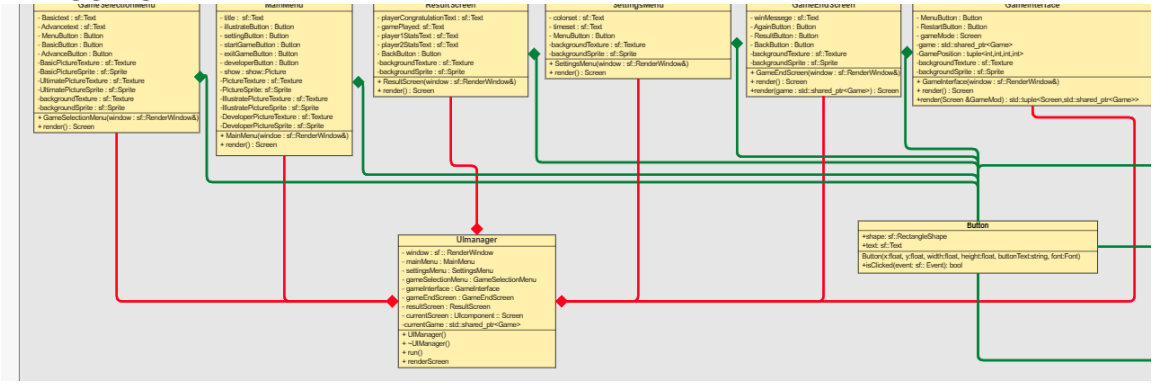
繼承：圖四、五、六都包含了繼承，圖四的部分每個頁面都有相同的共通點，需要一個畫面去做負載，因此他們都有共同的父類別叫做 UIcomponent，每個網頁都能用 UIcomponent 定義自己的功能。圖五是對基礎版與進階版的井字遊戲取共通點作父類別，稱為 Game，兩者模式共有的功能。圖六的 Basic4U 繼承了 Basic 是為了 include 到進階的 Ultimate，其功用在重點程式碼上說明。

多型：圖四、五的部分包含了多形，利用抽象類別來呈現多形，圖四為每個動作都需要頁面，但頁面的內容都不一樣，所以創個 UIcomponent 作抽象類別，每個子類別再去個別實現頁面的運作。

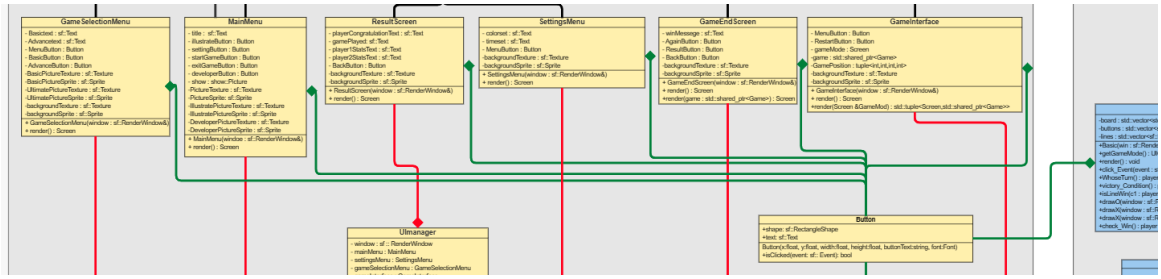
圖五是將 Game 作為抽象類別，因為基礎版與進階版的同個功能都差不多，但判斷條件有所不同，所以寫個 Game 做抽象類別建立個功能，其功能怎麼寫由基礎版與進階版自己決定。

2、關係

Aggregation



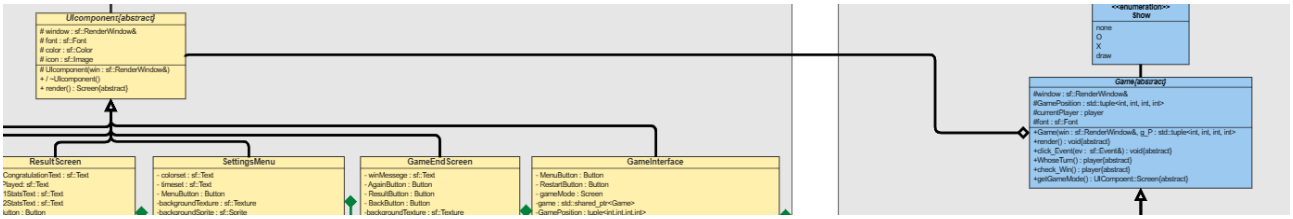
（圖七）



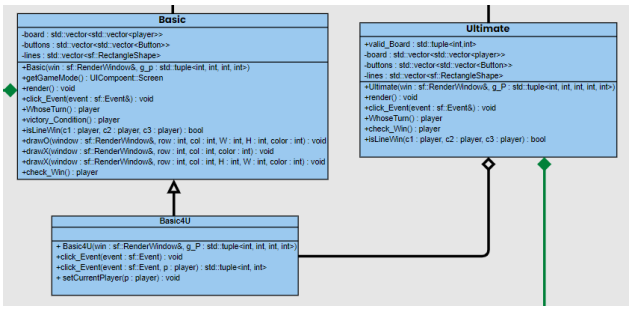
（圖八）

↑Aggregation 代表單向的關係，一個類別需要建立在其他類別上才有意義。如圖七每個頁面都需要透過 UImanager 才能顯示出完整的一套 UI 出來，每個 UI 獨立並沒意義。在圖八的部分，button 需要透過頁面的實現才有意義，不然單獨的 button 並沒辦法實現任何事情。

Composition



（圖九）



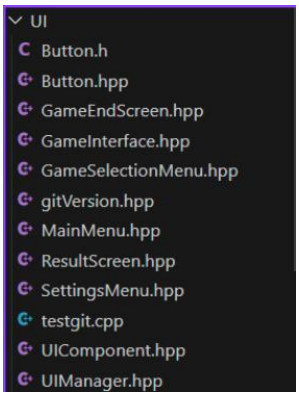
（圖十）

↑Compostion 代表雙向的關係，彼此間為互相依賴的關係，如圖九 UIcomponent 與 Game 的關係，雖然各自能獨立運作，但 UI 沒有 game 就沒有存在 UI 的必要；有 game ´有 UI，game 無法給 user 進行遊玩，因此兩者是雙向關係。

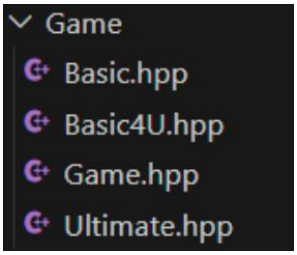
三、 重點程式碼說明

（一）OOP

1、封裝



（圖十一）



（圖十二）

↑上圖是本專題的每個封裝，每個 hpp 都是一個封裝，裡面都有一個 class，主要分為 UI 跟 game 兩塊。

2、繼承

```
class GameEndScreen : public UIComponent {
class GameInterface : public UIComponent {
class GameSelectionMenu : public UIComponent {
class MainMenu : public UIComponent {
class ResultScreen : public UIComponent {
class SettingsMenu : public UIComponent {
```

（圖十三）

```
class Basic : public Game {
class Ultimate : public Game{
```

（圖十四）

```
Basic4U : public Basic{
```

（圖十五）

上圖為每個封裝間的繼承關係，分別有 UIcomponent、Game、Basic 這三個父類別，照著 UML 的規劃分別給各自的子類別作繼承。

3、多形

```
class UIComponent {
public:
    enum class Screen {
        MAIN_MENU,
        SETTINGS_MENU,
        GAME_SELECTION_MENU,
        GAME_BASIC_INTERFACE,
        GAME_ULTIMATE_INTERFACE,
        GAME_END_SCREEN,
        RESULT_SCREEN,
        EXIT
    };
    virtual Screen render() = 0;
    virtual ~UIComponent() {}
```

（圖十六）

```
class Game{
    public:
        enum class player {
            none,
            O,
            X,
            draw
        };
        Game(sf::RenderWindow& win,std::tuple<int,int,in
            currentPlayer = player::O;
            font.loadFromFile("data/ttf/TaipeiSansTCBeta
        };// constructor
        virtual ~Game(){} // destructor
        virtual void render() = 0;
        virtual void click_Event(sf::Event &ev) = 0;
        virtual player WhoseTurn() = 0;
        virtual player check_Win() = 0;
        virtual UIComponent::Screen getGameMode() = 0;
```

（圖十七）

上面圖十六、十七為多型，以抽象類別的方式呈現，父類別給了 virtual function 後，子類別在自己去實現各自的 virtual function。

（二）關係表現

1、 Aggregation

UIManager.hpp

UI > UIManager.hpp > ...

```
#include <iostream>
#include <tuple>
#include <vector>
#include <string>
#include <fstream>
#include <sstream>
#include <SFML/Graphics.hpp>
#include "Button.hpp"
#include "UIComponent.hpp"
#include "MainMenu.hpp"
#include "SettingsMenu.hpp"
#include "GameSelectionMenu.hpp"
#include "GameInterface.hpp"
#include "GameEndScreen.hpp"
#include "ResultScreen.hpp"
#include "gitVersion.hpp"
```

（圖十八）

←左圖為 UIManager 去實現每個頁面，結合起來去做控制。除了 UIManager 外，每個頁面都包含了 button，因此 button 也結合了各個頁面，每個頁面都能使用 button 去實作功能。

2、 Composition

Ultimate.hpp

Game > Ultimate.hpp > Ultimate

```
#ifndef ULTIMATE_H
#define ULTIMATE_H
#include "Game.hpp"
#include <SFML/Graphics.hpp>
#include "../UI/Button.hpp"
#include "Basic4U.hpp"
```

（圖十九）

Game.hpp

Game > Game.hpp > ...

```
#include<bits/stdc++.h>
#include <SFML/Graphics.hpp>
#include "../UI/UIComponent.hpp"
```

（圖二十）

圖十九是 Ultimate 去引用 Basic4U 去實現功能，雖然沒有繼承間的關係，但沒有 Basic4U，Ultimate 無法被實做出來。圖二實同理，Game 沒 UIComponent 的話 Game 無法給使用者遊玩。

（三）重點程式碼說明

```
class UIComponent {
public:
    enum class Screen {//定義畫面的所有狀態
        MAIN_MENU,
        SETTINGS_MENU,
        GAME_SELECTION_MENU,
        GAME_BASIC_INTERFACE,
        GAME_ULTIMATE_INTERFACE,
        GAME_END_SCREEN,
        RESULT_SCREEN,
        EXIT
    };
    virtual Screen render() = 0;//這裡我讓所有渲染完的時候要回傳
下一幀要渲染哪個螢幕狀態。
    //.....
};
```

（圖二十一）

在 UImanerger，存了很多 UI 介面，要怎麼通知換畫面呢?

```
59  // // UI 管理器
60  class UIManager {
61  public:
62      //....
63      UIComponent::Screen currentScreen;
64      std::shared_ptr<Game> currentGame;
65      //...
66      void renderScreen(){
67      std::tuple<UIComponent::Screen, std::shared_ptr<Game>>
        renderResult;
68      switch (currentScreen) {
69          case UIComponent::Screen::MAIN_MENU:
70              currentScreen = mainMenu.render();//回傳時改現在的
                畫面就可以改下一幀選染啥囉
71              break;
72          case UIComponent::Screen::SETTINGS_MENU:
73              currentScreen = settingsMenu.render();
74              break;
75          case UIComponent::Screen::GAME_SELECTION_MENU:
76              currentScreen = gameSelectionMenu.render();
77              break;
78          case UIComponent::Screen::GAME_BASIC_INTERFACE:
79              renderResult = gameInterface.render
                (currentScreen);
80              std::cout << "Game Basic Interface: [Gameplay
                Elements]" << std::endl;
81              currentScreen = std::get<0>(renderResult);
82              currentGame = std::get<1>(renderResult);
83              break;
84          case UIComponent::Screen::GAME_ULTIMATE_INTERFACE:
85              renderResult = gameInterface.render
                (currentScreen);
86              currentScreen = std::get<0>(renderResult);
87              currentGame= std::get<1>(renderResult);
88              break;
89          case UIComponent::Screen::GAME_END_SCREEN:
```

```
89          case UIComponent::Screen::GAME_END_SCREEN:
90              currentScreen = gameEndScreen.render(currentGame);
91              break;
92          case UIComponent::Screen::RESULT_SCREEN:
93              currentScreen = resultScreen.render();
94              break;
95          case UIComponent::Screen::EXIT:
96              window.close();
97              break;
98          default:
99              break;
00      }
01  }
```

（圖二十二）

在 UIManager currentScreen 存現在的螢幕狀態。

而進階版跟基本版都用同一個 GameInterface，這樣才能凸顯動態連結的多型多好玩。他們的 render 要傳入 currentScreen，確保要選染哪種遊戲，而且回傳整個遊戲的動態指標，讓 GameEndScreen 渲染遊戲最後的結果。

```cpp
class GameInterface : public UIComponent {
//...
    Screen render() override {
        // Render the game interface including buttons and
        game elements
        std::cout << "Game Invidual Interface: [Gameplay
        Elements]" << std::endl;//用原本的方式進來是不合法的喔
        return Screen::EXIT;
    }
    std::tuple<Screen,std::shared_ptr<Game>> render(Screen &
    GameMod) {
        gameMode = GameMod;

        if(gameMode == Screen::GAME_BASIC_INTERFACE){
            std::cout << "Game Basic new [Gameplay Elements]"
            << std::endl;
            game = std::make_shared<Basic>(window,
            GamePosition);  // 使用智能指標，動態連結game
        }
        else if(gameMode == Screen::GAME_ULTIMATE_INTERFACE){
            //std::cout << "Game Ultimate new [Gameplay
            Elements]" << std::endl;
                game = std::make_shared<Ultimate>(window,
                GamePosition);
                // return {Screen::GAME_SELECTION_MENU,
                nullptr};
            }
        else{
            std::cout << "Game Ultimate Interface: [Gameplay
            Elements]" << std::endl;
        }
```

```cpp
        }
//...
        backgroundSprite.setTexture(backgroundTexture);
                // 更新游戏状态
                // 渲染游戏和界面
                window.clear(color);
                window.draw(backgroundSprite);
                game->render(); // 渲染游戏
                window.draw(MenuButton.shape);
                window.draw(MenuButton.text);
                window.draw(RestartButton.shape);
                window.draw(RestartButton.text); // 渲染界面元
                素，例如菜单和重新开始按钮
                window.display(); // 更新窗口显示

                // 检查游戏是否结束
                Game::player win_Player=game->check_Win();
                if(win_Player != Game::player::none){
                    return {Screen::GAME_END_SCREEN,game};
                }
            }
        return {Screen::EXIT,nullptr};
    }
    // 其他遊戲功能
```

（圖二十三）

進階版以及基本版都用一樣的渲染框架，很好的表示多型的玩法。

```cpp
class Basic4U : public Basic{
    std::tuple<int,int> click_Event(sf::Event &event, player
p) {

            if (event.type == sf::Event::Closed)
                window.close();
            setCurrentPlayer(p);
            // if(check_Win() != player::none){
            //                 //      return {-1,-1};
            // }
            for(int i = 0; i < 3; i++){
                for(int j = 0; j < 3; j++){
                    if (buttons[i]
[j].isClicked(window,event)) {
                        if (board[i][j] == player::none) {
                            board[i][j] = currentPlayer;

currentPlayer = currentPlayer == player::O ? player::X
:player::O;

                            return {i,j};
                        }
                    }
                }
            }
        return {-2,-2};
    }
    void setCurrentPlayer(player p){
        currentPlayer = p;
    }
};
```

（圖二十四）

Basic4U 唸法是 Basic for Ultimate，繼承 basic 增加回傳被按九宮格的哪個位子，使用 tuple 傳回 Ultimate，以實現在大棋盤的對應位子下 OX。還有要增加現在換誰，現在換誰是 basic 本來就會控制的，會影響渲染的 OX 順序，但是主導權要在 Ultimate，所以有一個 setcurrentplayer

四、 組員工作分配

	錢昱名	劉沛辰	劉柏均	黃羿禎	孫健淳
遊戲主程式	70%	10%	10%	5%	5%
UI designer	20%	50%	15%	10%	5%
UML	10%	20%	5%	35%	30%
PPT	10%	10%	20%	30%	30%
專題發想	20%	20%	20%	20%	20%

五、 組員心得

錢昱名：

我來補充一點技術細節，我在 game 的多型使用了智慧指標，因為如果要維持可讀性，在哪裡 new 一個指標就要在哪裡 delete，但是我們會把 game 指標傳入 endscreem，直接使用 game 的 render 顯示最終結果，使用智慧指標就不用管了，哈哈哈哈。

劉沛辰：

這次做專題，從發想主題，討論了很久該如何實現封裝繼承多型，確定主題，畫 usercase，站在使用者角度設計按鍵，設計 UI、顏色、區塊，然後再畫 uml，設計封裝多形繼承，再到實現 uml，程式碼。

過程遇到許多困難，SFML 的環境建構，git 共同協作，github 的 branch 推動，再來到 SFML 實作 UI，多形繼承 UI Component，PlayerO X 實作，check 的繼承，把上課學到的理論變成實作才讓我更熟悉 OOP

黃羿禎：

在這次的專題中，我從同儕之間學到了更多關於程式的知識，像是如何使用 git 等，也第一次從頭與別人協同合作撰寫程式碼，除了更加熟悉 OOP 三大特性外，也學會了該如何與他人合作、溝通。

在過程中我覺得最為困難的點是學會如何以物件導向程式語言的思維去撰寫程式碼，以往都是撰寫程序性程式語言為主，所以一開始要開始理解、撰寫程式碼非常困難，在這部分我非常感謝並佩服我們組長--企鵝，程式碼幾乎都是由他所完成的，透過觀摩他的程式碼才讓我更加對物件導向語言有更加清晰的了解，漸漸熟悉撰寫物件導向程式語言。

孫健淳：

在這次專題中，除了懂得該如何應用物件導向思維的方式（利用封裝，類別關係的方式去思考如何組織程式碼）去設計程式專案外，也從其他隊友中學習到之前從未踏過的領域，以下是我這次從專題所學到的其他領域。

（一）github 整合程式碼

首先是為了整合大家的程式，避免大家在資訊上的不對稱，造成程式碼重複修改，不同步的情形，因此我們使用 git 來解決這個問題。這也是我第一次對 git 有了具體的概念。這將成為我日後撰寫程式的一大利器。

（二）SMLF 與 chatgpt 的應用

SMFL 是專門提供圖形化介面的函示庫，這是大家第一次接觸到 SMFL 的函示庫，所以對裡面擁有的 function 並不熟悉。但在與 chatgpt 的配合下，算是邊用邊學，大致知道裡面的函示庫該如何使用。這也讓我意識到之後 AI 對於程式設計的重要性，該如何有效的對 AI 下指令是我必須學會的課題。

劉柏均：

這次的期末專題，我們選擇了井字遊戲，但因為不是只有簡單的井字遊戲，我們還加入了一些特殊的規則，我們找到了 SFML 來開發我們的期末專題，我主要是做 UI 設計，除了物件導向以外，我還學到了如何讓整個 UI 看起來比較順眼，還有學到到了如何使用繪圖軟體，在未來跨領域結合受益良多。