Object Oriented Programing

Project #4 - Report

Course	**Object Oriented Programming**
Department	**Computer Science & Engineering**
Professor	**Bong-Soo Sohn**
Team	**#8**
Term	**2023 - 2**
Members	**김진호 김혁진 박수빈 박주연 송민혁 신성섭**

[Table of Contents]

I. Summary

1) Project title

Stock game

2) Team members

20201XX4 김진호
20205XX3 김혁진
20226XX9 박수빈
20200XX6 박주연
20203XX0 송민혁
20226XX2 신성섭

3) Presentation Speaker

20203XX0 송민혁

4) Brief Project Description

종목	전년	올해	등락
A 엔터	50,000	40,000	▼-20%
B 엔터	4,000	2,000	▼-50%
C IT	170,000	170,000	0%
D IT	420,000	350,000	▼-17%
E 바이오	350,000	400,000	▲14%
F 바이오	40,000	100,000	▲150%
G 식품	80,000	100,000	▲25%
H 뷰티	300,000	250,000	▼-17%
I 화학	35,000	30,000	▼-14%
J 조선	20,000	25,000	▲25%

We drew inspiration from the virtual stock game featured on 'Running Man' and created our own stock game. The game spans a total of 10 turns, where each turn sees stock prices fluctuate based on provided theme information. Players utilize three theme hints and their stock information to buy and sell stocks. The goal is to maximize earnings by the end of the game. Players can borrow a limited amount of money from the bank, with the option to repay at their discretion, but forced repayment occurs after a set number of turns.

To implement this program, we chose the C++ language and utilized the SFML library for the user interface. To avoid the difficulties, we experienced with merging in the previous project, we started by dividing classes and headers in advance.

To adhere to Object-Oriented Programming principles, we divided the program into classes: Information class, Information Manager class, Stock class, Stock Manager class, Player class, Portfolio class, Game Manager class, and Loan Manager class.

Become the ultimate stock investor!

II. How to execute

- **Compiler** : visual studio 2022 recently version (using x64)

- **OS** : Window

- **Source Code**
  1) gameManager.cpp
  2) informationManager.cpp
  3) information.cpp
  4) Loan_Manager.cpp
  5) player.cpp
  6) portfolio.cpp
  7) stockManager.cpp
  8) stock.cpp
  9) Window.cpp
  10) Print.cpp
  11) Button.cpp
  12) Graph.cpp
  12) main.cpp

- **Header file**
  1) gameManager.h
  2) informationManager.h
  3) information.h
  4) Loan_Manager.h
  5) player.h
  6) portfolio.h
  7) stockManager.h
  8) stock.h
  9) Window.h
  10) Print.h
  11) Button.h
  12) Graph.h

- **TXT files**
  1) A_bio.txt
  2) B_bio.txt
  3) A_food.txt
  4) B_food.txt
  5) A_tech.txt
  6) B_tech.txt
  7) thema1.txt
  8) thema2.txt
  9) thema3.txt

- **etc**

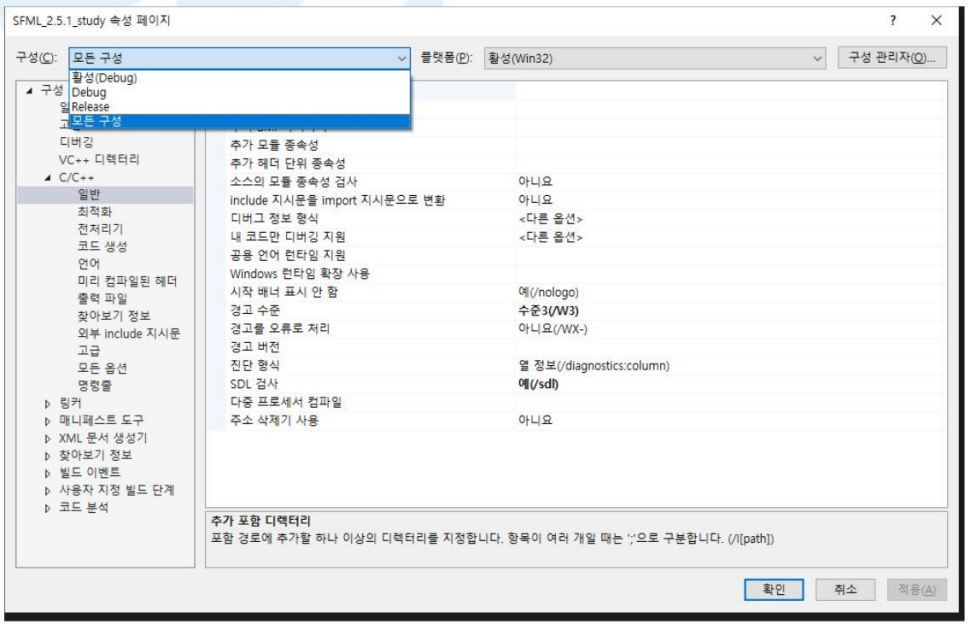
tuffy.ttf

- # Download and set SFML library

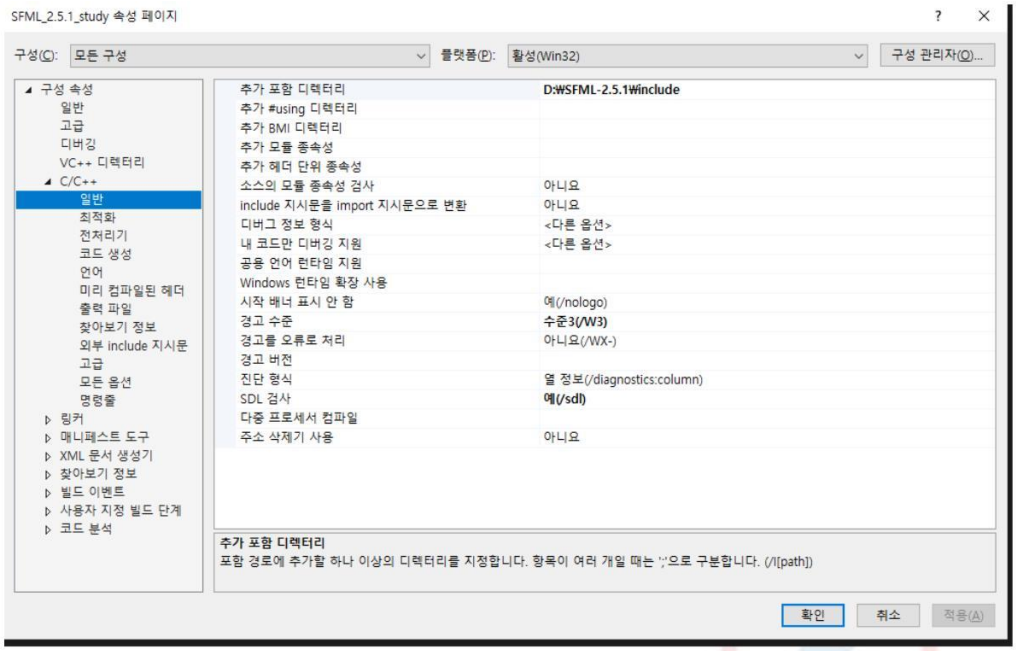
1) Download SFML at https://www.sfml-dev.org/download/sfml/2.6.1/ (Visual C++ 17 (2022) - 64-bit version).
2) Unzip the folder at certain directory.
3) Create a project on Visual studio, and make a c++ file(ex) main.cpp).
4) Open '프로젝트 -> (프로젝트 이름) 속성'

---

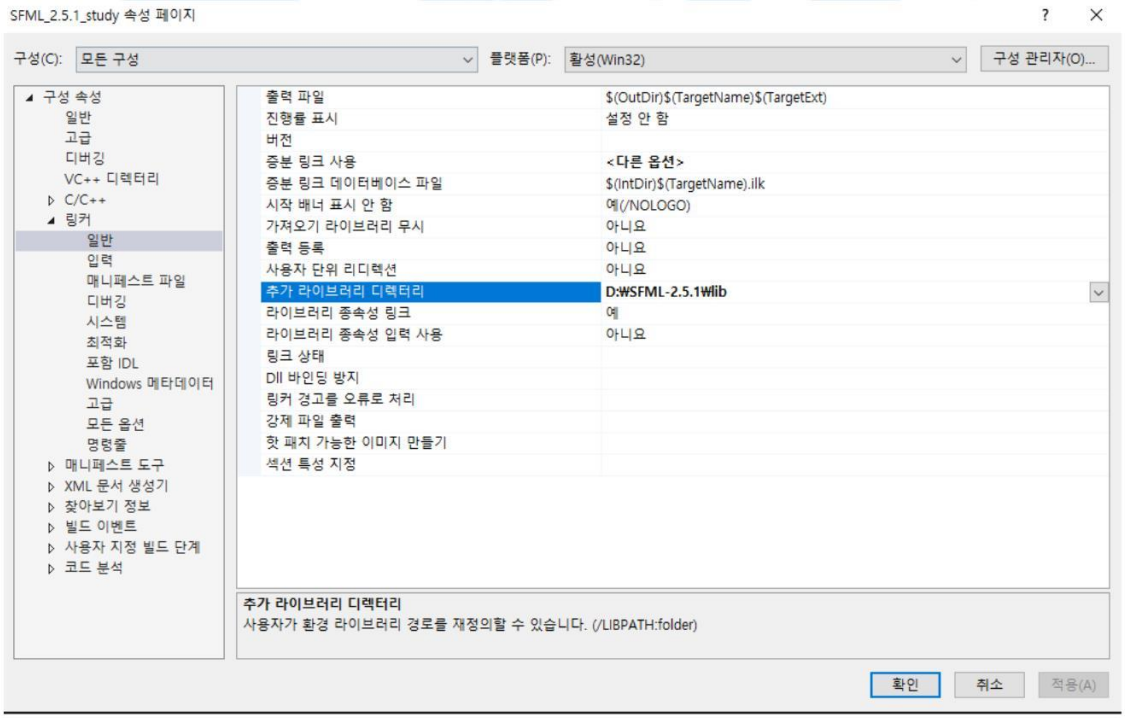
(1) choose '모든 구성'



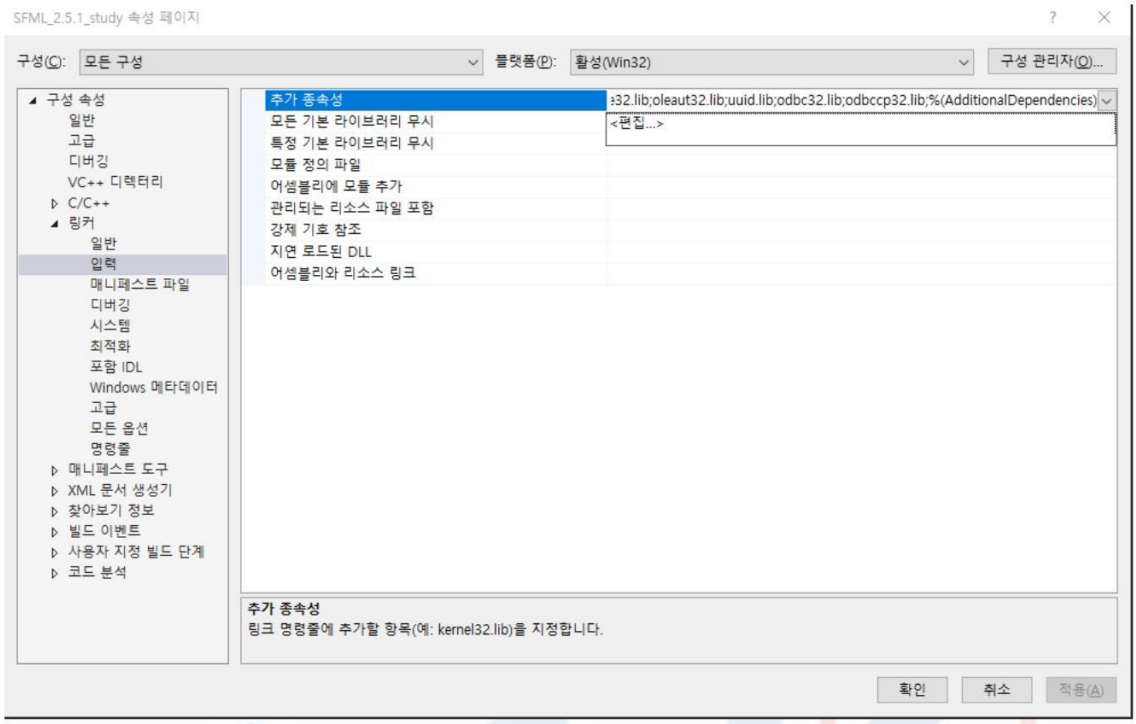
(2) Put the directory address of 'include' folder of SFML at '구성 속성 -> C/C++ -> 일반 -> 추가 포함 디렉터리'.



(3) Put the directory address of 'lib' folder of SFML at '구성 속성 -> 링커 -> 일반 -> 추가 라이브러리 디렉터리'.

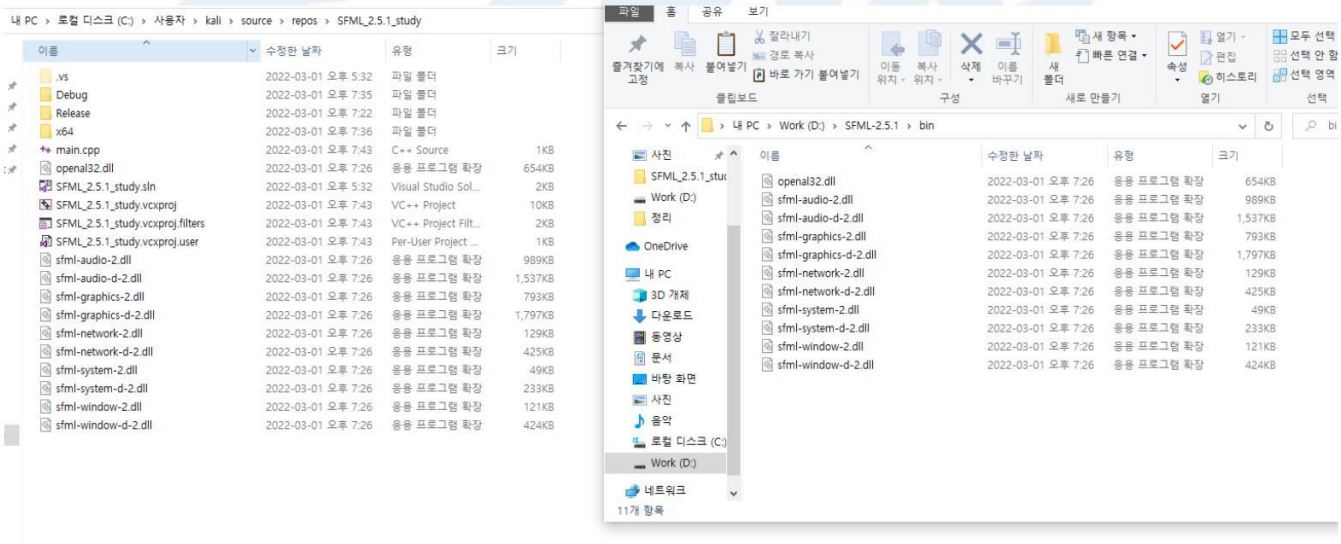


(4) Paste “sfml-main-d.lib;sfml-graphics-d.lib;sfml-window-d.lib;sfml-system-d.lib;sfml-audio-d.lib;sfml-network-d.lib” at ‘구성 속성 -> 링커 -> 입력 -> 추가 종속성’.

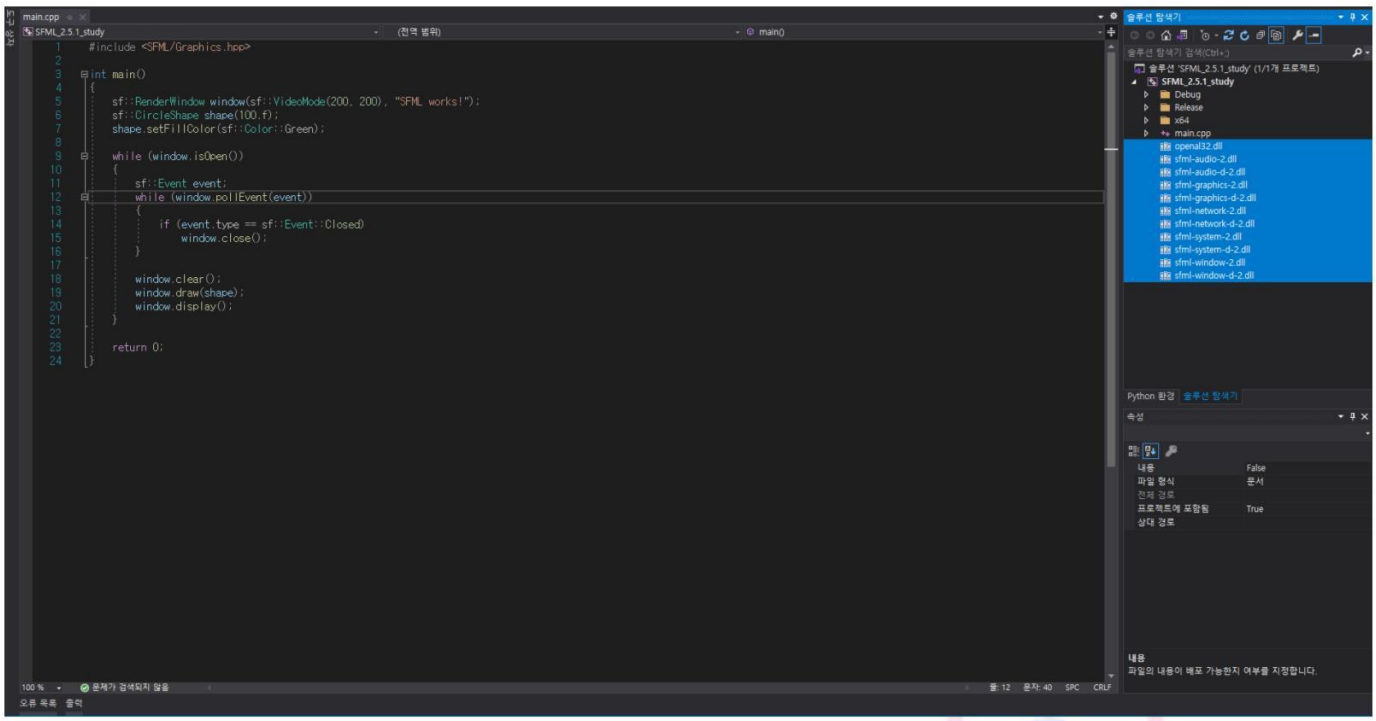


---

5) Move ‘.dll’ files which is in the ‘bin’ folder of SFML into the Visual Studio project folder.



6) Include all the '.dll' files in the project. Then, you're ready to run the game.



After completing the above steps, place all the text files in the folder where the code is executed.

After placing the files into source files and header files, simply click 'run' to execute.

III. Description on functionality

- Stock price determination mechanism

Every turn, it holds information on three randomly selected themes. Additionally, each stock has one randomly selected information. The extent to which this information affects the price is predetermined. The stock price is determined through the formula: stock Price * Theme Information-induced Price Fluctuation * Self-stock Information Influence * Other Companies' Stock Information Influence.

- information txt file

In each line of a text file, we saved information that influences stocks, company names, themes, influence levels, and influence on other companies.

- Information class

```
class information { // 정보 클래스

private: //필드
    string infoTxt; //정보 txt
    string enterpriseName; // 기업명
    string thema; // 테마
    int grade =0; // 정보 등급
    string externalInfluence;// 다른 기업에 미치는 영향
    float ownPriveVariance=1; // 이 정보가 본인 기업의 가격에 미치는 영향 (price* privceVariance 로 계산) , 테마 정보의 경우 이 값 이용
    float externalPriceVariance = 1; // 이 정보가 관련 타 기업의 가격에 미치는 영향  (price* privceVariance 로 계산)
```

This is an information class with attributes such as company information, company name, theme, influence level, and influence on other companies. It also has attributes for the price volatility of the self-company and other companies due to this information. It has methods for calculating the price volatility of the self-company and other companies based on the stored information.

- Information Manager class

```
class informationManager {
private: // 필드
    vector<information> informationDB; // 게임에서 나올 수 있는 모든 정보가 담겨 있음.
    information thisTurnInformation;  //이번 턴에 제공될 정보
    string enterpriseName; //기업명
    string txtFileName; // 정보가 저장되어있는 txt 파일명
```

This is a class that manages information. It has an information database (informationDB) and attributes for the information provided in the current turn. It also has an attribute for storing the company name for which the information is about. It has methods for reading information from Notepad and storing all information in informationDB, as well as selecting information for the current turn.

- stock class

```
class stock {
private: // 필드
    string stockName; // 주식명
    string stockThema; // 테마
    long int stockPrice=0; // 현재 주가
    int fluctuationRate=0; // 이전 라운드에 대한 최근 등락률
    informationManager infoManager; // 정보 매니저
```

It is a stock class created to contain information about stock prices. It has attributes such as stock name, theme, current stock price, recent fluctuation rate for the previous round, and an information manager object. The stock class has functions to call information manager's selectThisTurnInformation() for getting new information about the stock and a method to calculate price changes based on its own stock information.

- stock Manager class

```
class stockManager {
private:
    vector<stock> stockDB; // 모든 주식의 vector
    vector<informationManager> themaManger; //테마 매니저 vector
```

The StockManager class is created to manage stocks comprehensively. It has a stock database (stockDB) as an attribute and multiple theme managers as attributes. The methods include functions to update the prices of all stocks, update all information managers with new information, add stocks to the stock database, and add new theme managers to the theme managers.

- Portfolio class

```
class Portfolio
{

private:
    vector<InvestStruct> v;
    stockManager* stocks; //주식 시장 manager를 가리키는 pointer
```

The object manages information about the stocks owned by the player. It is dependent on the individual player. All methods and data in the portfolio are designed with the assumption that they are called by the player object.

The player manages the stocks they own through a vector of InvestStruct objects. The vector includes all stocks managed by the stockManager. It encompasses stocks not yet purchased by the player, including those with a quantity of 0. To retrieve stock information, it holds a pointer to the address of the stockManager. Both the vector and the stockManager pointer are initialized separately by calling their respective initialization functions.

- InvestStruct structure

```
struct InvestStruct {
    string item;        // 주식 종목 명
    int NumberOfItem;   // 주식개수
    int PurchasingPrice; // 매입가
    int CurrentPrice;   // 현재가
    int profit;         //평가손익
    int EarningsRate;   //수익률
    int amount;         // 평가금액
    bool infoUnlock;    //해당 주식의 정보 언락 여부
```

The structure within the portfolio that stores information about a stock. It includes the stock price, the quantity held, and other numerical values. Noteworthy is the infoUnlock variable, indicating whether to unlock information about the stock.

There are two conditions for unlocking information:

1. Owning the stock above a certain threshold
2. Paying a currency in each round to make the purchase possible.

- LoanManager class

```
class Loan_Manager {
private:
    int loanAmount = 0; //대출액(원금)
    int interestAmount = 0; //이자액
    int interestRate = 0; //이자율, 이자를 한번 받을 때마다 증가
    int deadline; //기한, 4턴
```

The object manages loans and repayments for the player. Similar to the Portfolio class, it is dependent on the individual player. All methods and data are designed with the assumption that they are called by the player object.

The loan amount is managed separately as principal and interest. Additionally, it keeps track of data such as the interest rate and repayment deadline. The interest rate increases by a certain percentage each time a loan is taken, serving as a constraint to prevent the player from abusing loans. The repayment deadline is set at 4 turns, calculated based on the date the loan was taken, and does not change even if additional loans are taken during the counting period. All loan amounts, including principal and interest, must be settled to reset the system. If the deadline is exceeded, a mandatory collection is executed for all outstanding loan amounts. Additionally, if the loan is not settled by the last round, a mandatory collection is also enforced.

-Player class

```
private:
    string      name;                   //player명
    int         asset;                  //현금 자산
    bool        checkLoan = false;      //대출 여부
    Portfolio playerPortfolio;
    Loan_Manager playerLoan_Manager;
```

The object that stores information about the player. It is dependent on the gameManager object and operates by updating data in the loanManager and Portfolio objects.

It has one member variable each for the Portfolio and loanManager objects. Additionally, it manages data such as the player's name, cash assets, and loan status.

- gameManager class

```
class gameManager {
private:
    int turn = 0;
    Player player;
    stockManager gameStockManager;
    bool infoPurchase = false; // 정보 구매 여부, 턴마다 초기화 된다.
```

This is the StockGameManager, responsible for overall management of the stock game. It has attributes indicating the current turn, player object, stock manager object, and infoPurchase to determine whether information was purchased in the current turn. The methods include a function for initializing the game, a function to increment the turn, a function to update stock values and new information, a function to update player information, and a function to check if bankruptcy has occurred.

IV. Implementation

- information txt file

We created a total of 3 theme information txt files and 6 company information txt files, including A_BIO, B_BIO, A_food, B_food, A_IT, and B_IT.

- Information class

1) Constructor

```
information::information(string infoTxt, string enterpriseName, string thema, int grade, string externalInfluence) { //생성하면서 변동비율도 한번에 계산해줌
    this->infoTxt = infoTxt;
    this->enterpriseName = enterpriseName;
    this->thema = thema;
    this->grade = grade;
    this->externalInfluence = externalInfluence;
    calculateChangeRate(); // 변동비율 계산
}
```

It receives information txt, company name, theme, influence level, and influence on other companies as parameters. After it initializes the attributes and calls a function to calculate the price volatility based on the information.

2) calculateOwnChangeRate()

It calculates ownPriceVariance using the formula 1 + influence * constant. Additionally, a random real number within a certain range is obtained and added to ownPriceVariance, introducing a slight randomness to the stock price fluctuations.

3) calculateExternalChangeRate()

It calculates externalPriceVariance using the same formula as ownPriceVariance. In this case, if externalInfluence is positive, it adds influence * constant, and if it is negative, it subtracts influence * constant. If externalInfluence is 0, no calculation is performed. To balance the game, externalPriceVariance has a constant and a range of random real numbers smaller than the constant used for calculating ownPriceVariance.

- Informaion Manager class

1) Constructor

```
InformationManager::InformationManager(string enterpriseName, string txtFileName) { //기업이름, Txt파일 이름을 제공하는 생성자
    this->enterpriseName = enterpriseName;
    this->txtFileName = txtFileName;
    saveAllInformation(); // 메모장에 있는 모든 정보 저장
    selectThisTurnInformation(); // 이번턴에 사용할 정보 한개 랜덤 선택
}
```

It takes the company name and the text file name as parameters to create an informationManager object. It initializes the enterpriseName and txtFileName attributes. Then, it reads all information from the txt file corresponding to txtFileName and stores it in the informationDB. Finally, it selects one piece of information for the current turn.

2) saveAllInformation()

It is a function to store all the information saved in Notepad in the informationDB. It uses the stored txtFileName in the object to open the corresponding txt file and reads it line by line. The read string is then tokenized based on commas (,). The information object is created by passing these pieces of information as parameters. The created information object is stored in the informationDB. This process is repeated until the entire file is read.

3) selectThisTurnInformation()

It selects one piece of information randomly from the informationDB to use for this turn and stores it in thisTurnInformation. To prevent duplicate information from appearing in later turns, the selected information is removed from the informationDB.

- stock class

1) Constructor

```
stock::stock(string stockName, string stockThema, float stockPrice, string txtFileName) {
    this->stockName = stockName;
    this->stockThema = stockThema;
    this->stockPrice = stockPrice;
    this->fluctuationRate = 0;
    this->infoManager = informationManager(stockName,txtFileName);
}
```

The constructor of the Stock class takes parameters such as stock name, stock theme, stock price, and the name of the text file. It initializes the attributes and creates a stock object.

2) selectNewInfo()

The method selectThisTurnInformation() is called on the infoManager object, allowing the information manager to select a new piece of information for the stock.

3) calculateOwninfluence()

The getOwnPriceVariance() method is called on the infoManager object's thisTurnInfo attribute, which allows the information manager to calculate the price variance based on its own information for the stock.

- stock Manager class

1) addStock(string stockName, string stockThema, float stockPrice, string txtFileName)

Receive the stock's name, theme, stock price, and the text file name as parameters using these create a stock object and add it to the stock database (stockDB).

2) addthemaManger(string themaName, string filename)

Receive the theme name and text file name as parameters, create a theme information manager object, and add it to the theme manager (themaManager).

3) updateStockPrice()

Update all stocks with a new price and update the price volatility of each stock. Adjust stock prices considering the impact on prices caused by ownership information, external information, and theme information.

4) updateNewInfo()

Call selectThisTurnInfomation() for all information managers to update the new information influencing stock prices.

5) cacluateRiverinfluence(int stockNum)

If information from other companies affects one's own stock, multiply the own stock price by the extrnalPriceVariance value.

6) calulateThemainfluence()

If the theme of the information matches the theme of the stock, multiply the stock price by the ownPriceVariance value.

-Portfolio class

1) initialize method

```
void initialize(stockManager* stocks);
```

This is the function to initialize the stockManager pointer and the vector of stock items.

2) update method

```
void update();
```

This function updates the data of each item in the vector. It is executed at the end of each turn. Since the current price of each stock fluctuates with each turn, it updates the currentPrice values of each stock and the data affected by it. Depending on the quantity of each stock owned by the player, it also updates the unlock status of the corresponding stock information for the current round.

3) purchase method

```
bool Portfolio::purchase(string item, int NumberOfItem)
{
    // 매입 성공
    for (int i = 0; i < v.size(); i++) {
        if (v[i].item == item) {
            v[i].PurchasingPrice = ((v[i].PurchasingPrice * v[i].NumberOfItem)
                + (stocks->getStock(i).getStockPrice() * NumberOfItem))
                / (v[i].NumberOfItem + NumberOfItem);                    //구매가 갱신
            v[i].NumberOfItem += NumberOfItem;                           //보유개수 갱신

            v[i].amount = v[i].CurrentPrice * v[i].NumberOfItem;         //평가금액 갱신
            v[i].profit = v[i].amount - v[i].PurchasingPrice * v[i].NumberOfItem;   //평가손익 갱신


            return true;
        }
    }

    //매입 실패
    return false;
}
```

This function executes the purchase of stocks. It takes parameters specifying which stock to buy and how much to buy. It searches for the input stock in the vector, and if it exists, it executes the purchase, updating data including the quantity held for that item. If the stock with the given name does not exist in the vector, it considers the purchase unsuccessful and returns false.

4) sell method

```
int Portfolio::sell(string item, int NumberOfItem) {
    int plusAsset = -1;

    for (int i = 0; i < v.size(); i++) {
        if (v[i].item == item) {
            if (v[i].NumberOfItem < NumberOfItem)
                return -1;

            v[i].profit += NumberOfItem * (v[i].CurrentPrice - v[i].PurchasingPrice);
            plusAsset = NumberOfItem * v[i].CurrentPrice;
            v[i].NumberOfItem -= NumberOfItem;

            v[i].amount = v[i].CurrentPrice * v[i].NumberOfItem;                    //평가금액 갱신
            v[i].profit = v[i].amount - v[i].PurchasingPrice * v[i].NumberOfItem;    //평가손익 갱신


            if (v[i].NumberOfItem == 0)
                v[i].PurchasingPrice = 0;

            break;
        }
    }
    return plusAsset;
}
```

This function executes the sale of stocks. Similarly, it takes parameters specifying which stock to sell and how much to sell. It searches for the stock in the vector and, if found, executes the sale, updating data affected by the sale and returning the settlement amount. If the quantity to be sold is greater than the quantity held or if the stock is not found in the vector, indicating a failed sale, it returns -1.

- LoanManager class

1) Max_LoanAmout

```
#define MAX_LOANAMOUNT 3000000;
```

We have declared the maximum loan amount as a constant. In the design, players cannot borrow more than the maximum loan amount. However, it is possible for the total loan amount, including accumulated interest, to exceed the maximum amount. In such cases, no separate collection is enforced.

2) update method

```
bool Loan_Manager::update() {

        if (this->loanAmount > 0) {
            //check deadline
            if (this->deadline > 0)
                this->deadline--;
            if (this->deadline == 0)
                return true;


            //update interestAmount
            this->interestAmount += (int)(this->loanAmount * this->interestRate / 100 );

        }
        return false;
    }
```

This function, executed every turn, updates data such as interest and deadlines. If the deadline has passed, it returns true to notify the player object to execute a forced collection. The deadline is updated only when there is an outstanding loan; otherwise, it remains fixed at 4.

3) collection method

```
int Loan_Manager::collection(int player_cash) {
        return repayment(player_cash);
}
```

This function performs a forced collection, taking the player's entire cash as a parameter. It executes repayments for the entire amount and returns the remaining balance.

4) repayment method

```
int Loan_Manager::repayment(int repaymentAmount) {
        int buf = repaymentAmount;

        //이자액 상환
        if (this->interestAmount > buf) {//잔액이 이자액보다 부족한 경우
            this->interestAmount -= buf;
            return 0;
        }
        else {
            buf -= this->interestAmount;
            this->interestAmount = 0;
        }

        //원금 상환
        if (this->loanAmount > buf) {//상환액 잔액이 원금보다 부족한 경우
            this->loanAmount -= buf;
            return 0;
        }
        else { //원금까지 다 상환하고 금액이 남은 경우 (전액 상환)
            buf -= this->loanAmount;
            this->loanAmount = 0;
            this->deadline = 4; //deadline 초기화
        }

        return buf;
    }
```

This function performs repayments, taking the repayment amount as a parameter. Repayments are made in the order of interest and then principal. If there is a remaining balance after repaying both interest and principal, the function returns that amount. If there is no remaining balance, it returns 0.

-Player class

1) Porfolio Update

```
void Player::PortfolioUpdate() {
    playerPortfolio.update();
}
```

The function that is executed every turn to update the portfolio.

2) Portfolio_initialize

```
void Player::Portfolioinitialize(stockManager* stocks) {
    playerPortfolio.initialize(stocks);
}
```

When initializing the portfolio, the address of the stockManager is passed. As the gameManager manages the stockManager, it is received as a parameter.

3) Loan_Manager Update

```
void Player::Loan_ManagerUpdate(int RoundCounter) {
    if (checkLoan == true) {
        int updateValue = playerLoan_Manager.update();

        //강제 징수 실행
        if (updateValue || RoundCounter == 10)
            asset = playerLoan_Manager.collection(asset);
    }
}
```

The update of the loanManager is only executed when there is an outstanding loan. After the update, if the deadline has passed or the current round has reached the final round, a forced collection is executed. The current round is passed as a parameter to check the round.

4) stock purchase

```
bool Player::purchase(string item, int NumberOfItem, int CurrentPrice) {

    //현금이 부족한 경우
    if (this->asset < (NumberOfItem * CurrentPrice))
        return false;

    //구입이 가능한 경우
    if (playerPortfolio.purchase(item, NumberOfItem)) {
        this->asset -= NumberOfItem * CurrentPrice;
        return true;
    }

    return false;
}
```

The function receives the stock name, desired quantity for purchase, and the current price as parameters. If the purchase cost exceeds the current cash holdings, the purchase fails, and the function returns false. If the purchase is successful, it updates the cash value and returns true.

5) stock sale

```
bool Player::sell(string item, int NumberOfItem) {
    int sellAmount = playerPortfolio.sell(item, NumberOfItem);

    if (sellAmount <= 0) //매도 실패
        return false;

    else {
        asset += sellAmount;
        return true;
    }
}
```

The function takes the stock name and quantity as parameters for selling. If the sale is successful, it updates the asset value and returns true. If the sale fails, it returns false.

6) loan

```
bool Player::loan(int requested_loanAmount) {
    int loanAmount = playerLoan_Manager.loan(requested_loanAmount);

    //대출 실패
    if (loanAmount == 0)
        return false;

    //대출 성공
    else {
        asset += loanAmount;
        checkLoan = true;
        return true;
    }
}
```

The function takes the desired loan amount as a parameter and executes the loan. If the loan is successful, it adds the amount and returns true. If the loan fails, it returns false.

7) repayment

```
bool Player::repayment(int repaymentAmount) {
    //상환 실패 (현금 부족)
    if (repaymentAmount > this->asset)
        return false;

    else {
        asset -= repaymentAmount;
        asset += playerLoan_Manager.repayment(repaymentAmount);
        return true;
    }
}
```

The function takes the repayment amount as a parameter. If there is insufficient cash, it returns false. If the repayment is successful, it updates the asset value and returns true.

-gameManager class

1) initializeGame()

```
void gameManager::initializeGame() {  // 게임의 초기 정보 설정
    for (int i = 0; i < 3; i++) { // 테마 매니저 추가
        gameStockManager.addthemaManger(themaName[i], themaTxtFile[i]);
    }

    for (int i = 0; i < 6; i++) { //주식 추가
        gameStockManager.addStock(stockName[i],stockThema[i],startStockPrice, enterpriseTxtFile[i]);
    }

    //게임 플레이어 초기 설정
    player.setAsset(1000000);
    player.setName("NewBie");
    player.Portfolioinitialize(&gameStockManager);
}
```

This is a function that sets up the initial information for the game. It adds a theme manager, includes stocks, and simultaneously configures the initial settings for the game player.

2) turnUpdate()

```
void gameManager::turnUpdate() {


    if (checkPlayerBankruptcy() || turn > 9)    //파산확인
        gameRun = false;
    increaseTurn();                 //턴 증가
    UpdateStockInfo();              //정보 및 주식값 업데이트
    UpdatePlayer();                 //player 정보 업데이트
    infoPurchase = false;           //정보 구매 여부 초기화
}
```

It checks if it's the last turn or if the user has gone bankrupt, and in such cases, it sets gameRun to false. After that, it increases the turn, updates information and stock values, calls the player information update function, and set the infoPurchase to false

3) increaseTurn()

```
void gameManager::increaseTurn() {
    this->turn += 1;
}
```

It increments the turn by 1.

4) UpdateStockInfo()

```
void gameManager::UpdateStockInfo() { // 정보를 업데이트하고 주식값들을 업데이트 한다.
    gameStockManager.updateStockPrice(); // 정보 반영해서 가격 변동
    gameStockManager.updateNewInfo(); //새로운 정보 취득
}
```

It calls the method to adjust prices based on information and the method to acquire new information. Update the stock information and fluctuate the stock values accordingly.

5) UpdatePlayer()

```
void gameManager::UpdatePlayer() {
    player.Loan_ManagerUpdate(turn); //LoanManager 갱신
    player.PortfolioUpdate(); //포트폴리오 갱신
}
```

To update the player's information, It calls the player's LoanManagerUpdate method and portfolio update method

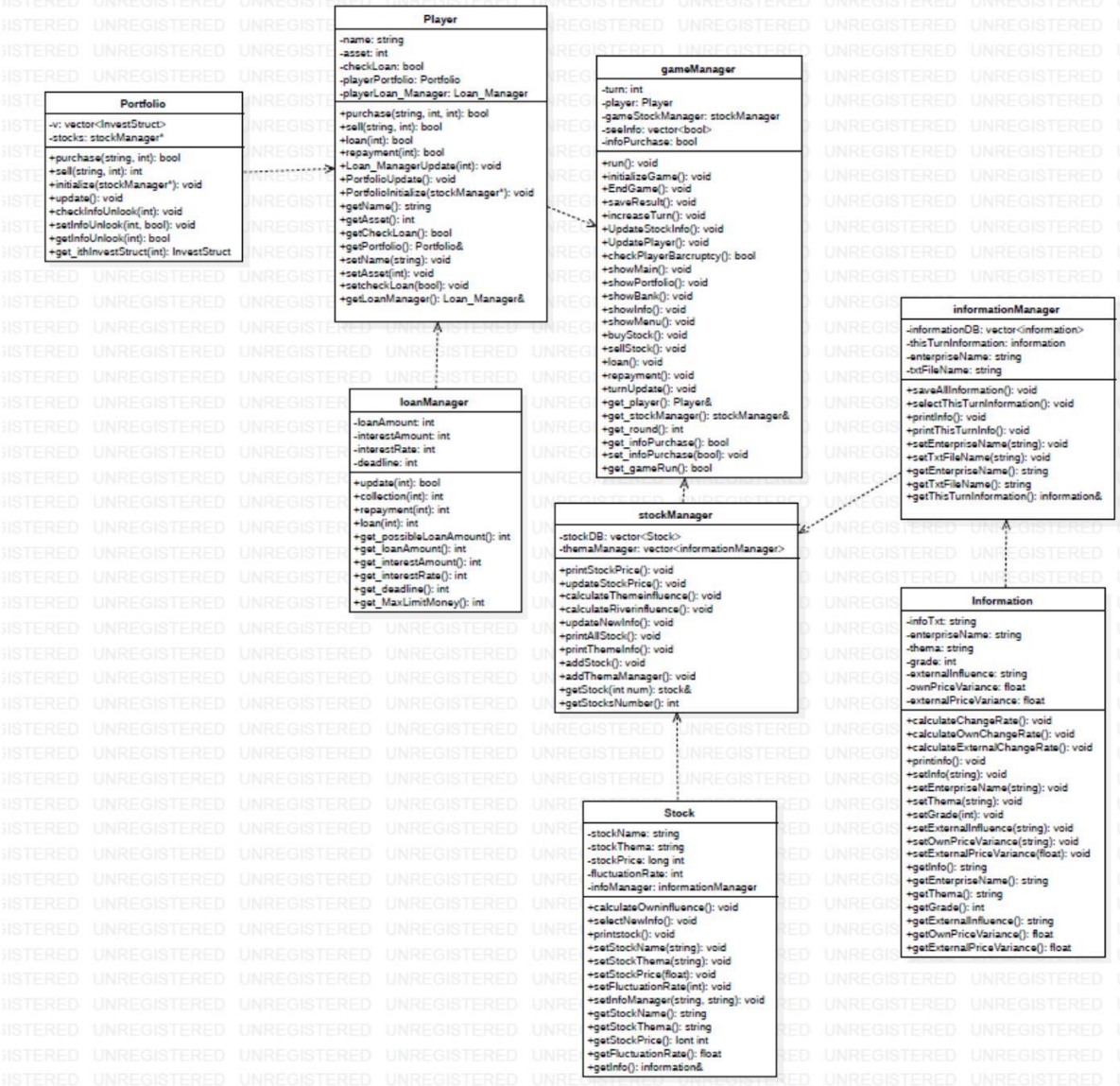
6) checkPlayerBankruptcy()

```
bool gameManager::checkPlayerBankruptcy() {
    /*if (player.getAsset() == 0)
        return true;*/

    int sum_amount = 0;
    for (int i = 0; i < gameStockManager.getStocksNumber(); i++)
        sum_amount += player.getPortfoilo().get_ithInvestStruct(i).amount;
    sum_amount += player.getAsset();

    if (sum_amount == 0)
        return true;


    return false;
}
```

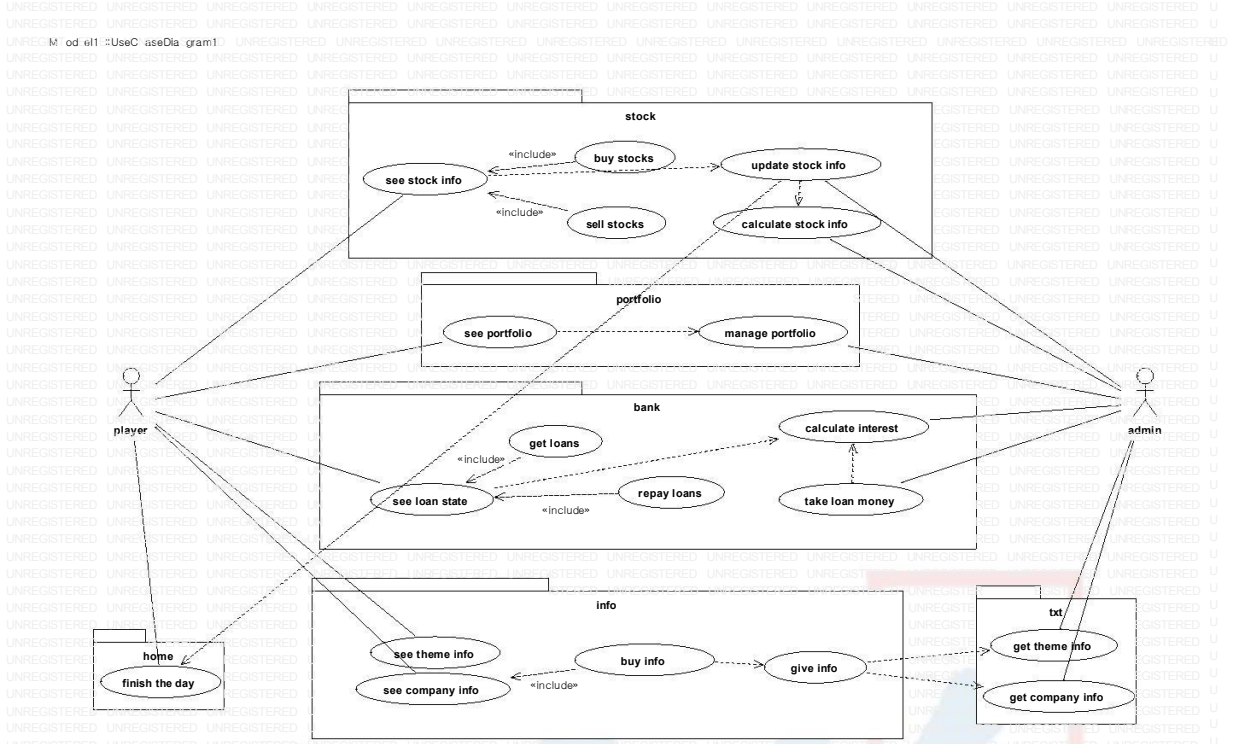
If the player's assets, considering the stocks, become 0 won, it processes the bankruptcy and returns true.

V. SW design result(diagram)

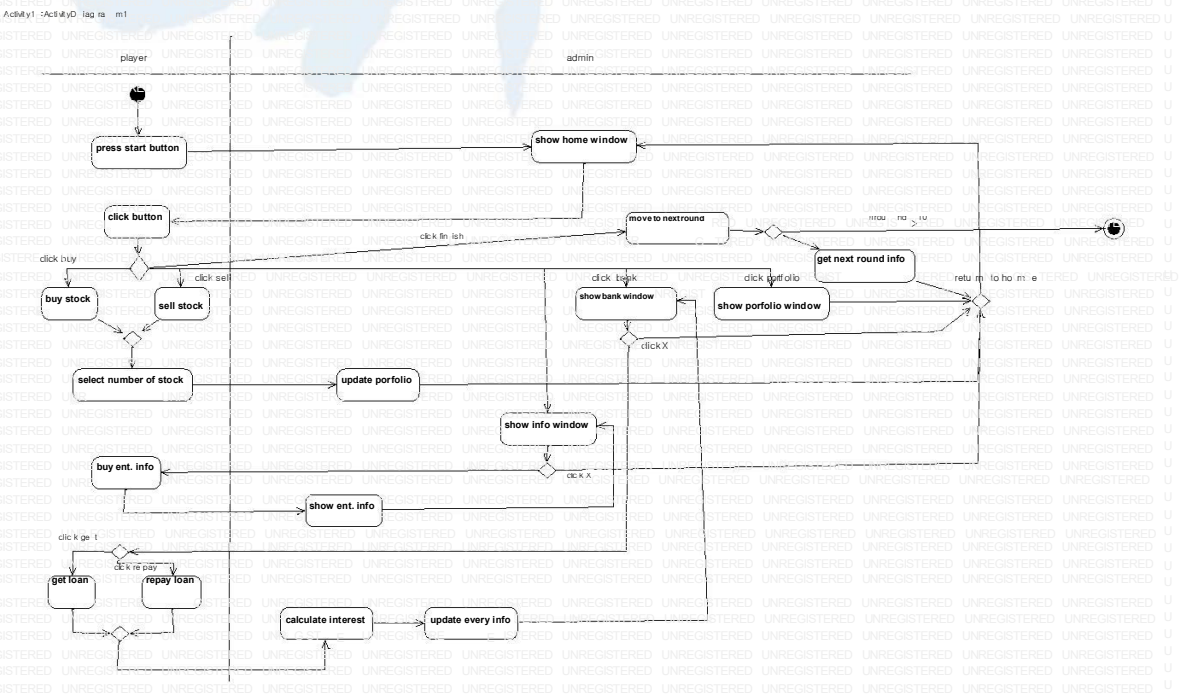
1) class diagram



2) usecase diagram

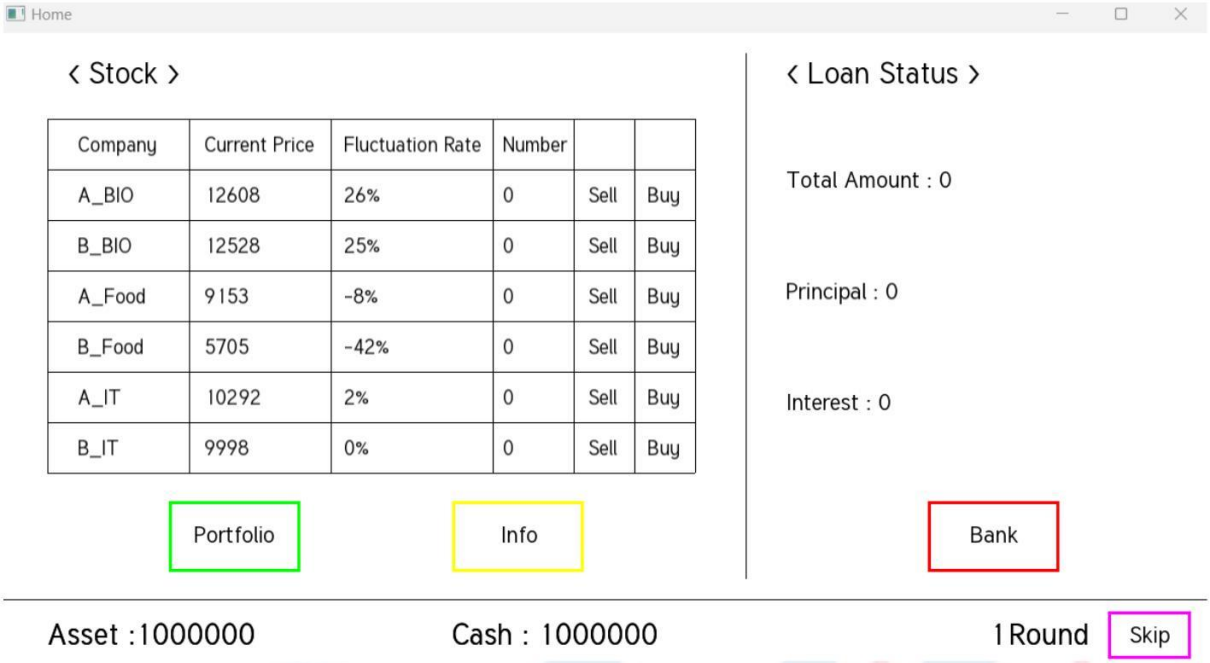


3) activity diagram



VI. Execution Result

1) Home

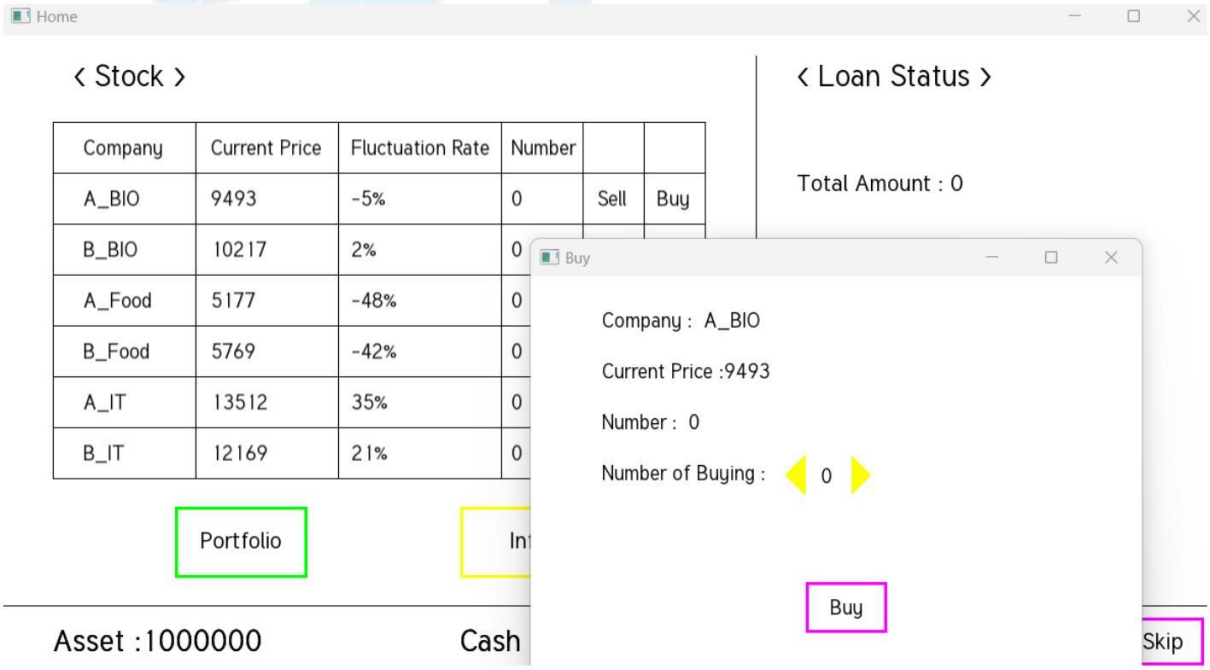


Player can check fundamental information about investment.

"Asset" represent the overall value of player's assets which include total valuation of invested stocks and cashes.

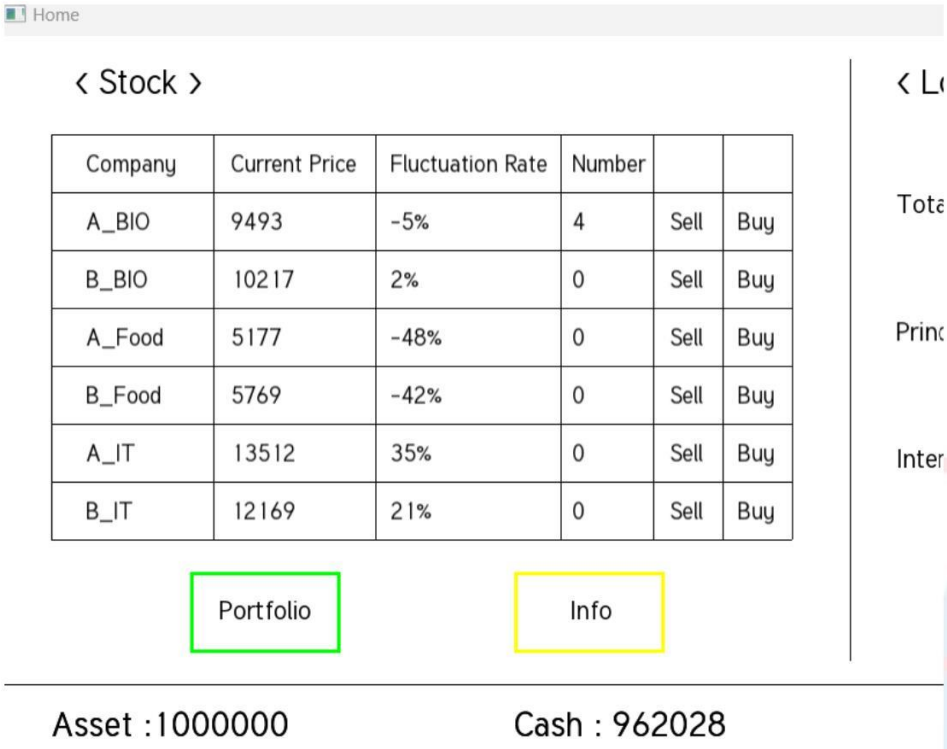
Player can move to other states by clicking on buttons.

1-1) Buying stock



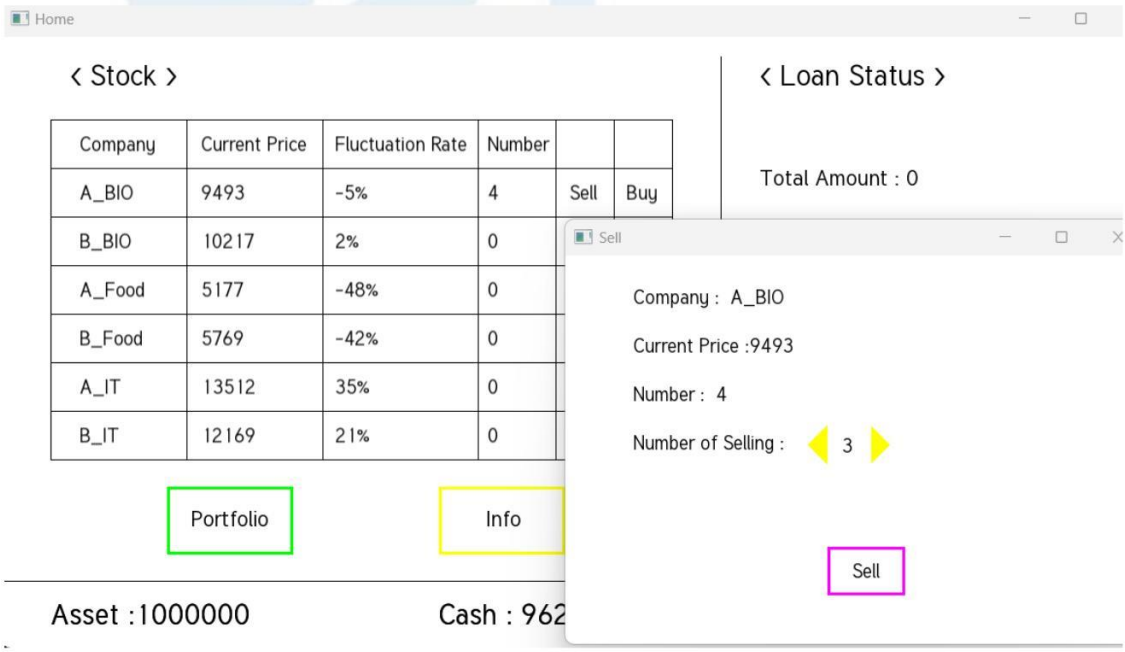
When you choose the stock you want to buy, a new window for buying process is

opened. You can check number of stocks to buy when arrow button. Trade is completed when you select amount and push the Buy button.

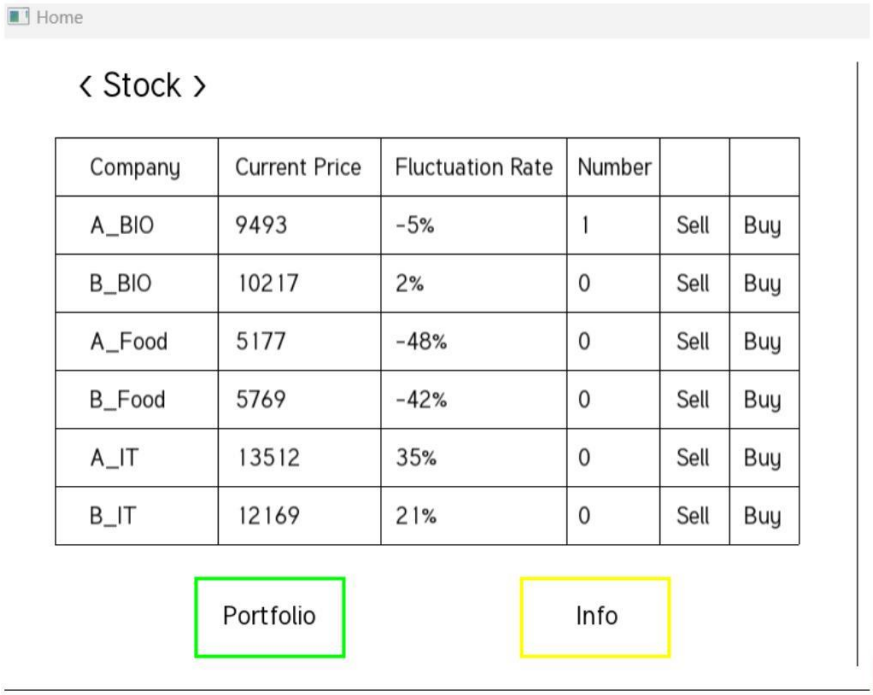


(Buying result)

1-2) Selling stock

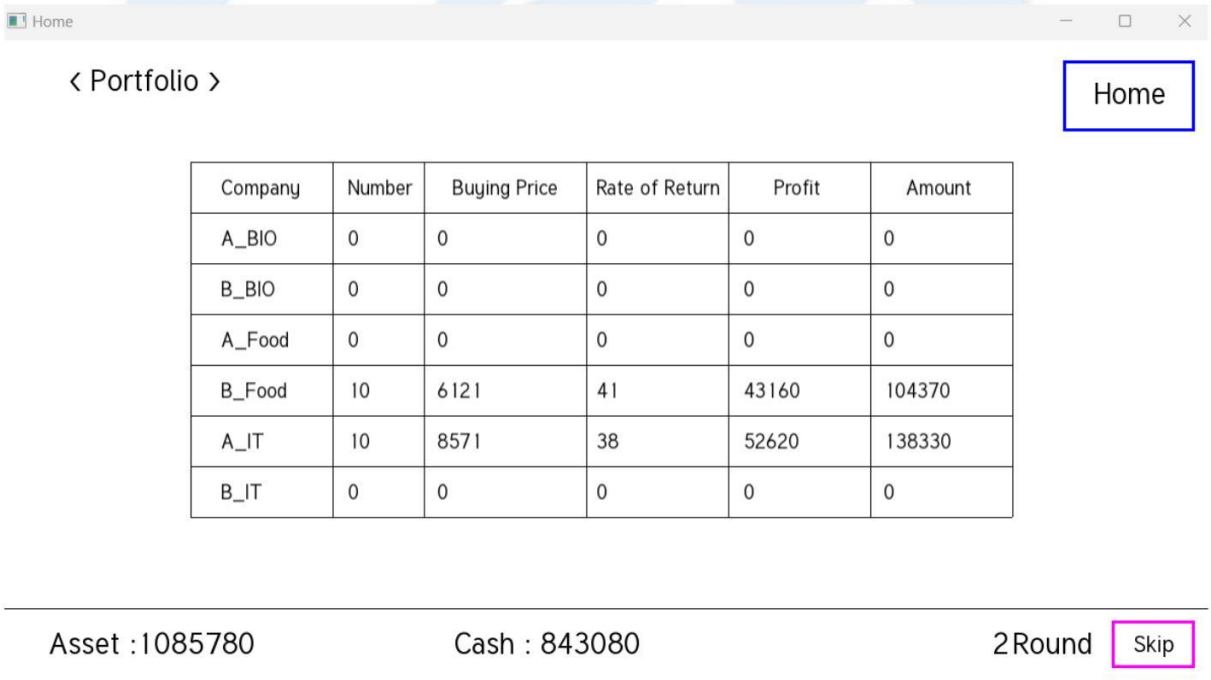


Stock selling process is proceeded equally. Player choose amount of stocks to sell and push the sell button.



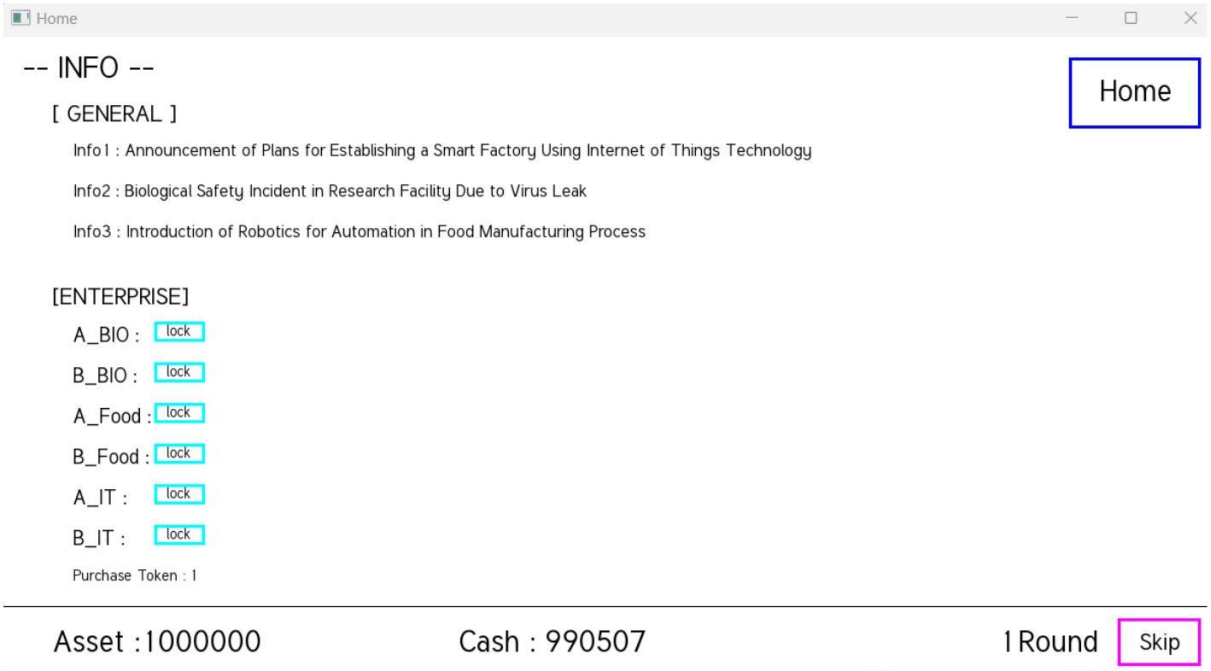
(Selling result)

2) Portfolio



Player can check detailed infomation of stocks in portfolio page.

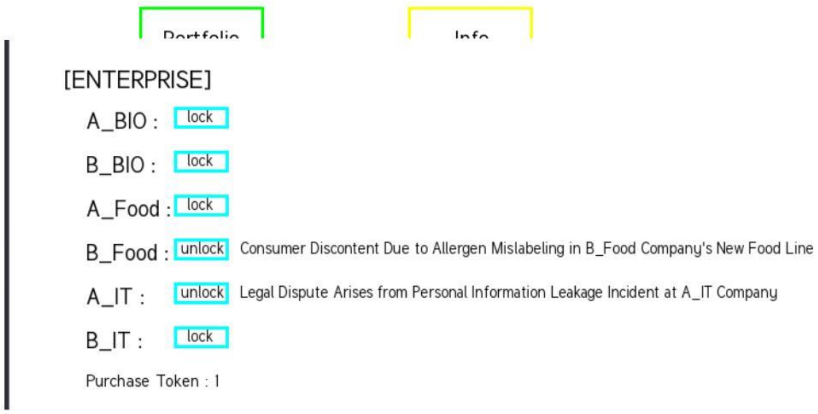
3) Information



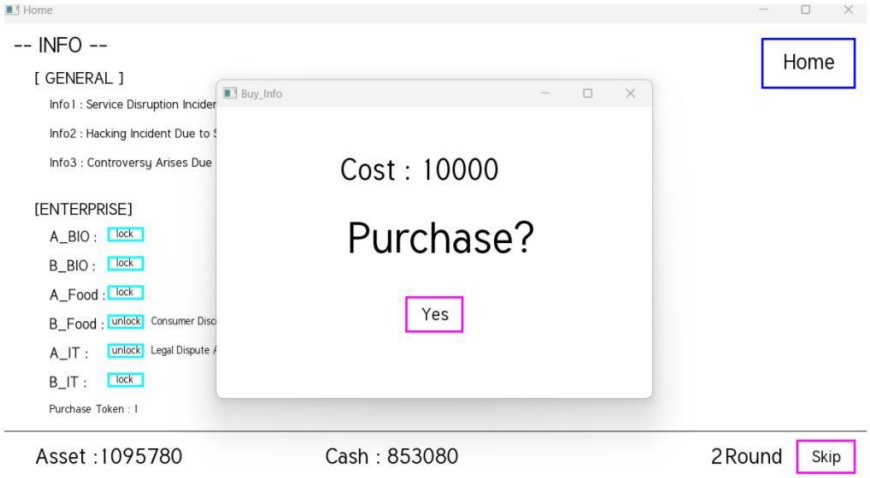
Player can check investment information here. There are three general information which is opened anybody. and there are other secret information of each enterprises which is revealed to only several people. Player can unlock enterprise information by owning that company's stocks more than ten or paying for it. But player can buy info only one time at one round.

< Stock >

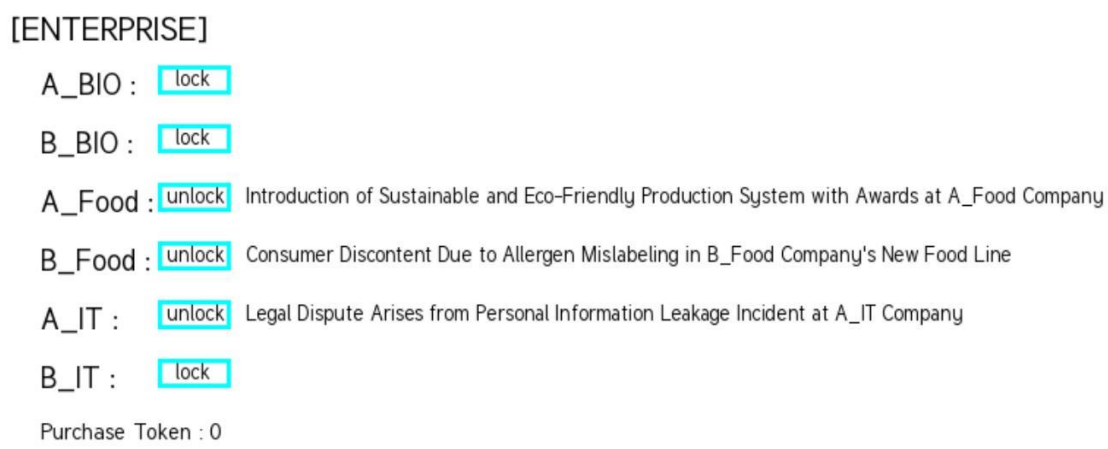
Company	Current Price	Fluctuation Rate	Number		
A_BIO	10249	2%	0	Sell	Buy
B_BIO	10912	9%	0	Sell	Buy
A_Food	11178	11%	0	Sell	Buy
B_Food	6121	-38%	10	Sell	Buy
A_IT	8571	-14%	10	Sell	Buy
B_IT	9590	-4%	0	Sell	Buy



When player own more than 10 stocks of the company, informations are opened at next round.

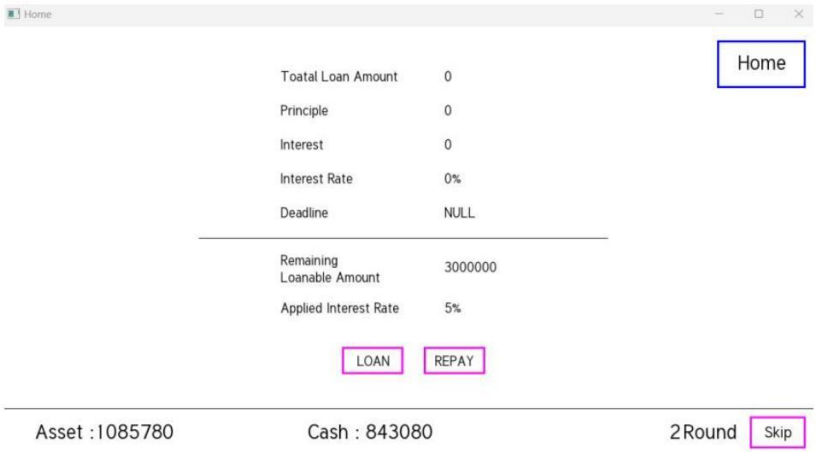


When player click 'lock' button, new window for puchasing process is opend.



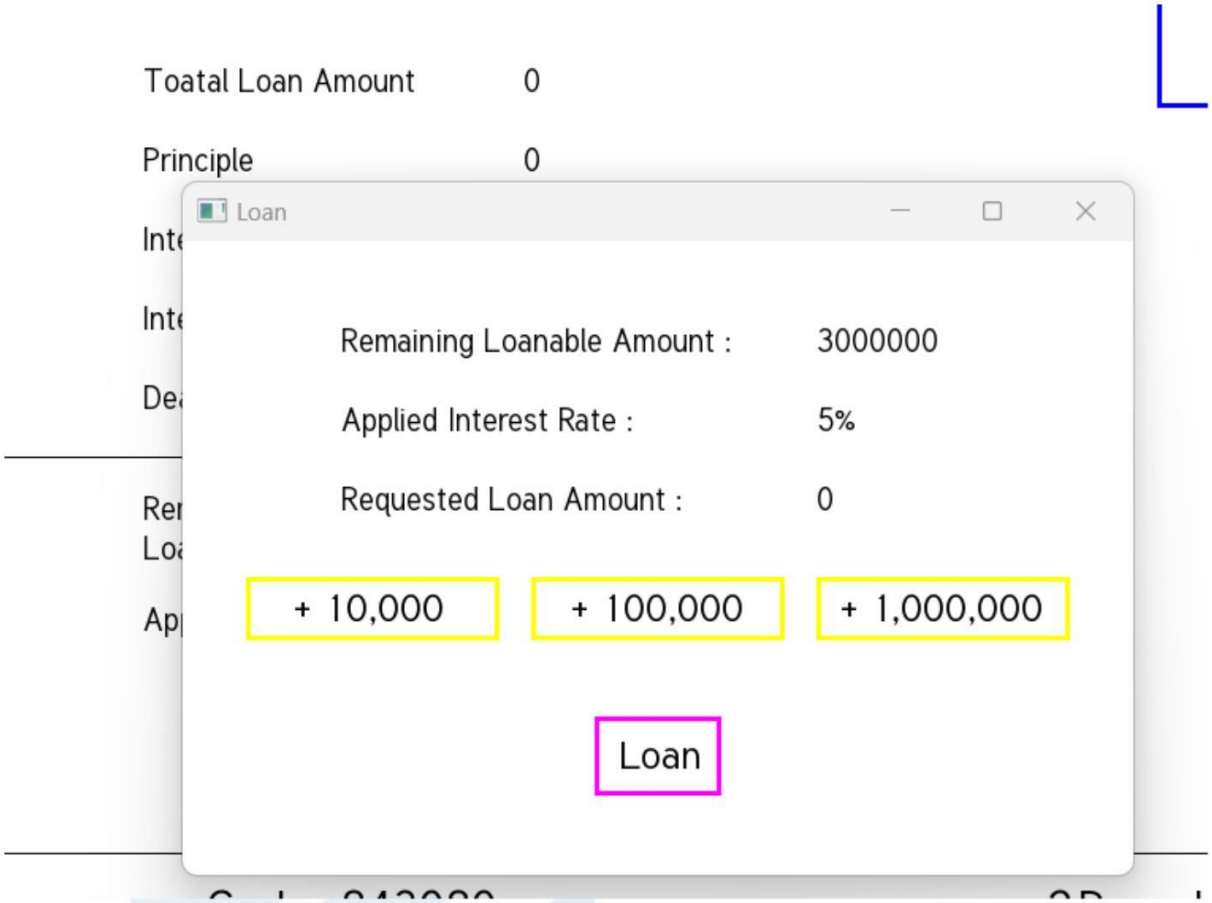
Player buy information about A_Food enterprise. purchase token which mean chance to buying information in round become 0.

4) Bank

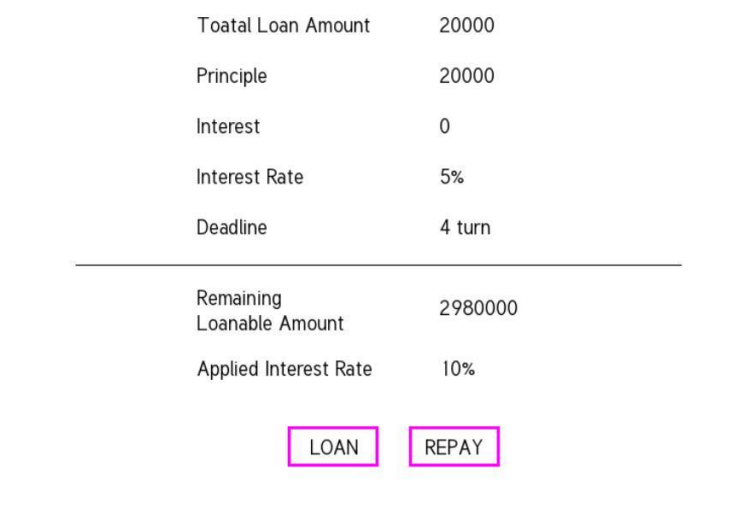


Player can check detailed information about loan/repayment.

4-1) Loan



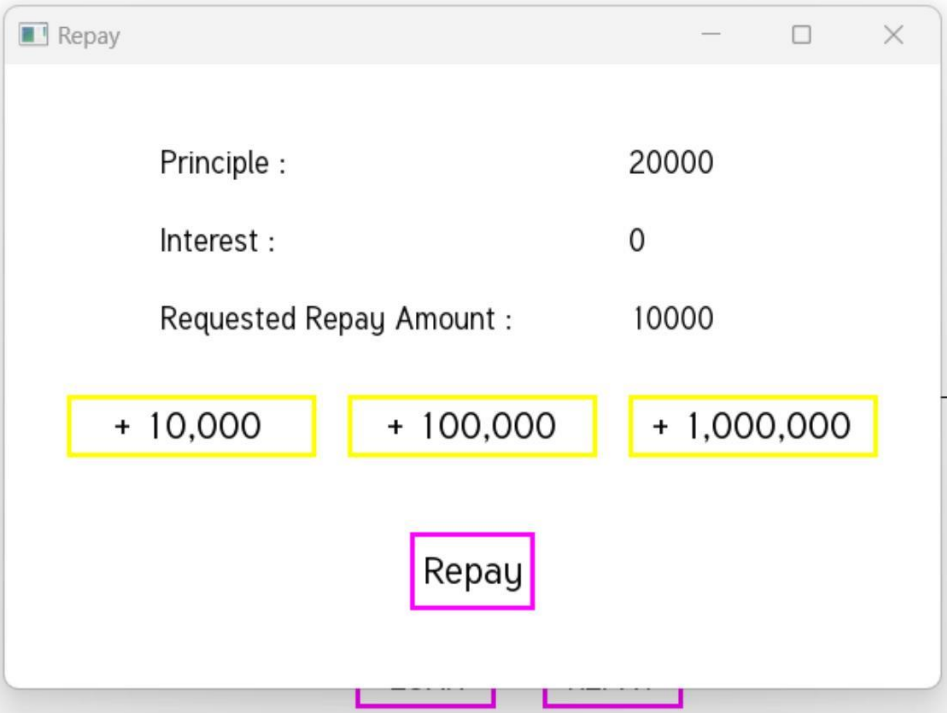
New window is opened. Player can choose amount to loan.



80 Cash : 863080

(Loan result)

4-2) Replayment



Toatal Loan Amount	10000
Principle	10000
Interest	0
Interest Rate	5%
Deadline	4 turn

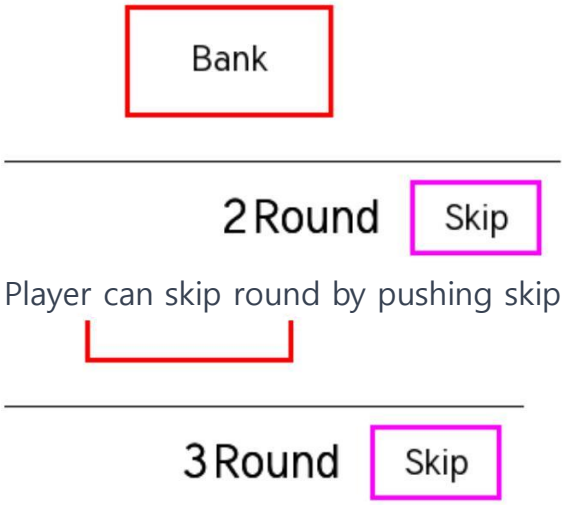
Remaining Loanable Amount	2990000
Applied Interest Rate	10%

LOAN REPAY

Cash : 853080

(Repayment result)

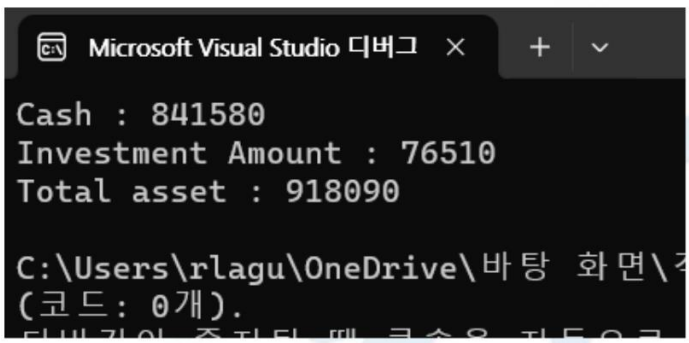
5) Round skip



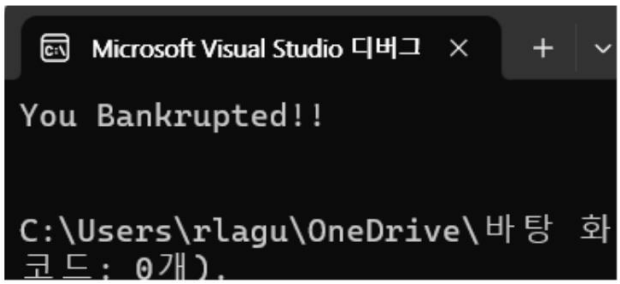
Player can skip round by pushing skip button.

Round passed, and information is updated.

6) Game end



When game ends, result is printed by console.



If player has zero cashes, game failed.

VII. Object Oriented Programming

Before creating the stock game, we first defined the types of classes needed and their respective attributes and methods with the team. Starting with the consideration of interactions between objects, we structured our system into classes such as Information, InformationManager, Stock, StockManager, Portfolio, LoanManager, Player, and GameManager.

For objects with high interdependence, we designed classes to have instances of other classes as attributes. For example, the InformationManager class stores information objects in a vector, and the StockManager class stores stock objects in a vector.

Each object was implemented with the necessary functionality as they interacted with each other.

To separate interfaces from implementations, we created separate header and .cpp files for each class. This approach made it easier for team members to understand the functionality of functions without worrying about the implementation details, resulting in improved code quality. Additionally, it facilitated the resolution of conflicts during the code integration process.

To achieve encapsulation, we declared the main attributes of each class as private. Access to these attributes from other classes was done through setter and getter methods.

VIII. Conclusion

In this project, we made an investing game using the SFML library. We collectively brainstormed and selected the theme and game design. Subsequently, we divided into teams to carry out the project. We created an information text file to determine stock prices, and simultaneously implemented C++ code based on object-oriented programming (OOP) modeling. As mentioned in the previous project report, we learned from the inconvenience and difficulty associated with code merging. Therefore, for this project, we proceeded by dividing classes and headers in advance. We also enhanced consistency in the GUI code using the SFML library by creating separate classes, which contributed to reducing the time spent on code writing.