[Warsaw University of Technology]

Mazen Ibhrahim

Object Oriented Programming Project

Warsaw
11-24-2018

Contents

Object Oriented Programming Project

The Banking System

Goal

The software represents a scenario of a banking simulation system. The purpose of this simulation software is to create a Coded User Interface where the user can create instances of banks, clients and the bank employees as well. The software can simulate the clients to create accounts, close accounts, withdraw and deposit cash. The software also comprises of real life like situations such as approval of loans, disapproval of loans due to insufficient balance. Possibility to transfer money from one account to another.

The employees of the bank can go through records of the clients to approve loans, discuss them with higher authorities and make decisions. The accountants take approvals from the manager and the cashiers take approvals from the accountants when clients want to withdraw cash of a considerably big amount.

Banks can also request other banks for loans, considering the size of the loans and the net-worth of the bank the managers can make decisions.

Such situations can be simulated. The system also offers possibilities to create mistakes with numerous combinations to simulate errors and the system will throw exceptions with custom messages.

Story

A bank is created. The bank can have multiple clients, each client can request for opening a bank account to the manager of the bank. The manager shall make a decision to open an account for the client based on the net worth of the individual.

Client scenarios

- Client can deposit cash to his/her account.
- Client can withdraw cash, if the amount is too large the manager shall approve the request to withdraw the amount to the accountant, the accountant shall permit the cashier to do so.
- Client can request for a loan to the bank. Considering the size of the amount the president of the bank and the manager should decide. The client should submit a certificate of mortgage to back his claims.

- Client can transfer money from one account to another account of another bank as well. The client must fill in details of the transfer to the function's parameters to complete the action.

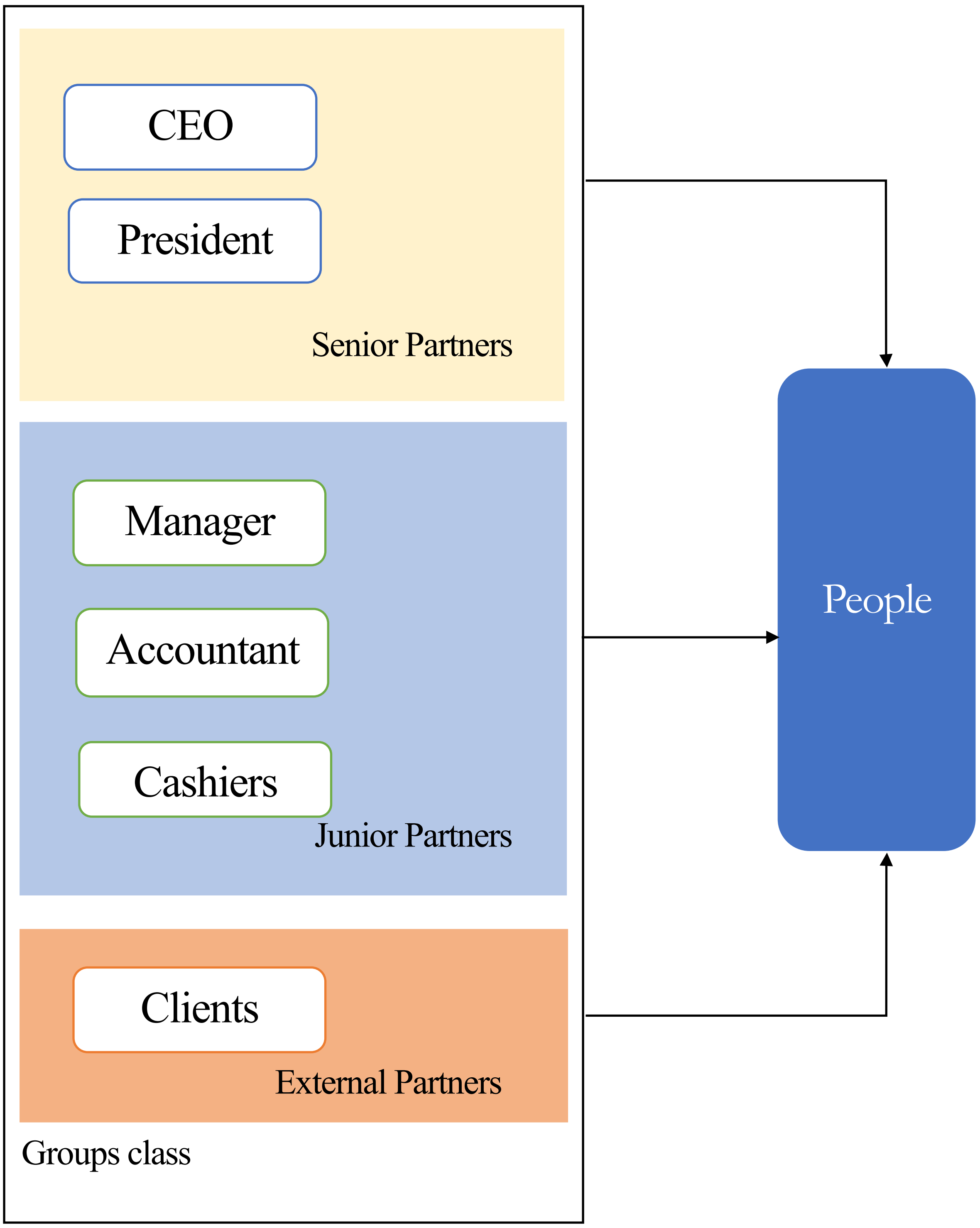
These are some of the scenarios the client can simulate, in case the client cannot complete an action due to insufficient account balance, the console will display the appropriate message with the details of the transaction.

Banking scenarios

- Bank instances can be created. Banks can hire and fire employees. The bank has a hierarchy of employees. Junior Partners and Senior Partners. Banks can contain many employees but only one CEO and president.
- The manager cannot be added to the list of Senior partners and the same with CEO's and presidents. They cannot be added to the list of senior partners.
- The banks can approve or reject loans of clients based on their net worth and the mortgage certificate passed in the parameter.
- The banks can approve disapprove transfers made by clients from within the bank and among other banks.
- The employees of the bank can choose to resign from their positions.
- Banks can request other banks for loans, the managers of other banks can approve of disapprove the loans.

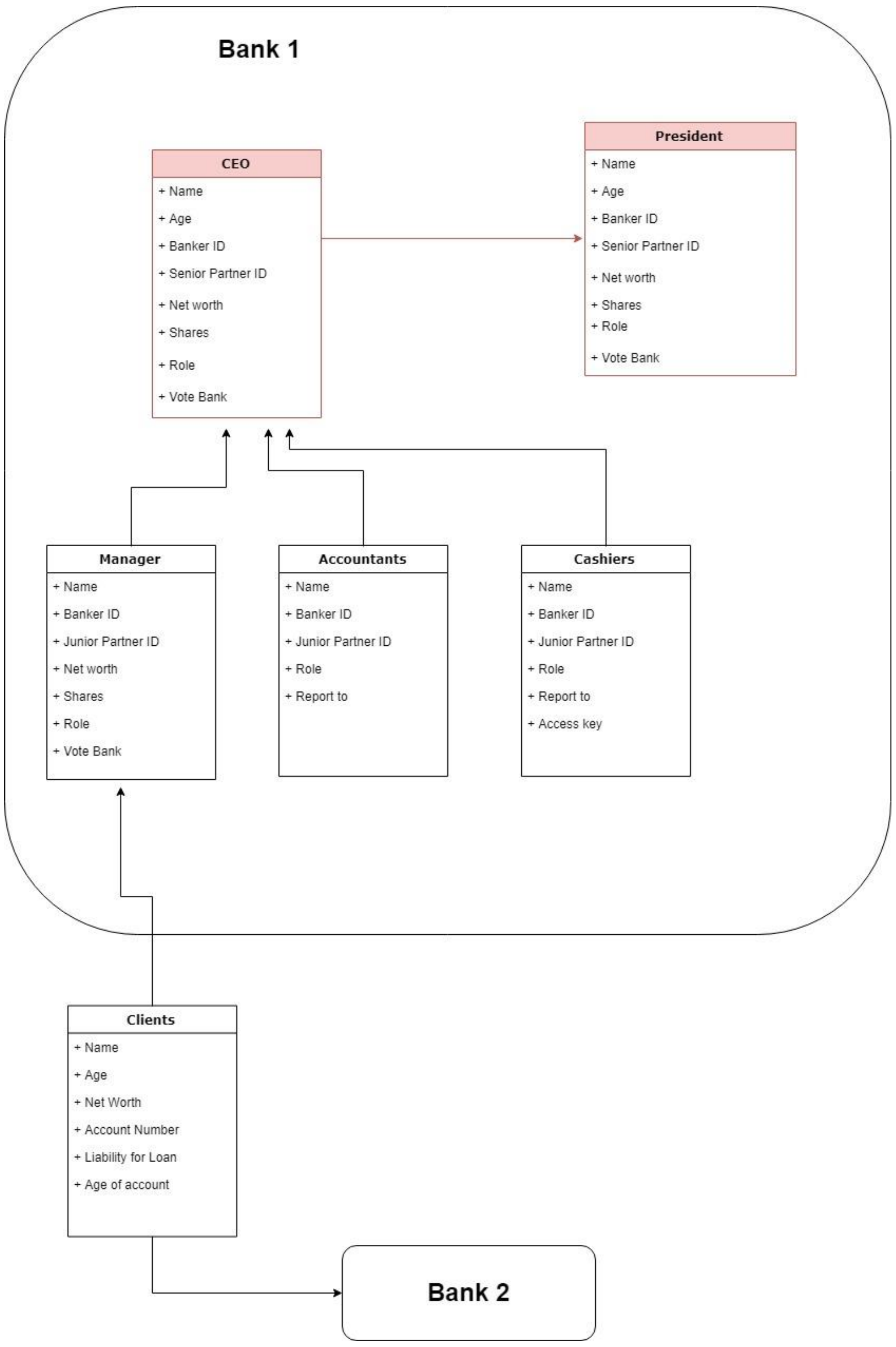
These are some the scenarios the banks can simulate. In case of any of the scenarios fail, custom exceptions will be thrown on the console notifying the user.

Conceptual diagram of the banking system’s hierarchy.



The diagram shows the relation between people classes and groups class. The linked list is made of People nodes. The groups class has 3 derived classes.

Memory mapping



The memory mapping here depicts the possible creation of instances where the client can create multiple bank accounts where as the bank employees can work only in a single bank. A special attribute in the class “*employed*” of type bool will be toggled when hired to a bank and toggled back when fired or resigned. This creates a mapping of which group the employee may belong too.

Class topology

People class – Base class

The people class is a base class of the people of the banking simulator. The people are clients, managers, accountants, cashiers, presidents and CEO's.

Each person has an attribute of ***Role_t***, an enumeration based on their roles. When they are added to the linked list, the list checks for this attribute and is then added. If the attribute doesn't match, the console will throw an exception.

CEO class – Derived class

The CEO class is a derived class of the people class which has the attributes derived from.

The CEO has a couple of functions to serve.

- Hire and fire employees.
- Agree to terminate an employee.
- Approve/disapprove loans if manager can't decide.
- Discuss with president about requesting loan from another bank.
- Discuss with president about approve/disapprove loan for another bank.

The various activities can be passed as a parameter to the *voteForActivity* function.

President class – Derived class

The president class is a derived class of people class which has the attributes derived from its parent class.

The president has a couple of functions to serve.

- Vote for an activity with the CEO. The activity can be granting loans, requesting for loans, hiring and firing.
- The president can request another bank for a loan.

The various activities can be passed as a parameter to the function's parameter.

Manager class – Derived class

The manager is a derived class of its parent class People. The manager belongs to the group of Junior Partners.

The manager has a few functions to serve.

- Respond to CEO for an activity.
- Respond to accountant for loan request, cash withdrawal request and bank transfers.
- Respond to clients for requests to opening/closing bank accounts.

- Fire other Junior Partner members.
- Hire other Junior Partner members.
- Accept resignation letters from other Junior Partner members.

The manager class must handle quite a lot of events that happen under him, most of these events are handled with custom exceptions. In case one of the parameters fail the exceptions with custom messages are thrown to notify the user. Multiple combinations of the events can be created to test the integrity of the manager.

Accountant class – Derived class.

The accountant class is a derived class of the People class, its parent class. Most of the attributes of the class is derived from its parent class.

The accountant has a few functions to serve to the cashier and the manager of the bank.

- Respond to the manager for a client’s request.
- Get account status of the client for withdrawal of money.
- Deposit cash after the cashier sends notification.
- Withdrawal of cash by requesting the manager

The accountant will directly handle the account of the client, when the functions for depositing cash and withdrawal of cash are called the accountant directly adds and deducts cash respectively.

The accountant’s events are triggered by the cashier.

Cashier class – Derived class

The cashier class is a derived class of the people class. Its attributes are inherited from its parent class. The cashier’s range of functionality is situated between the client and the accountant.

Some of the functions that the cashier’s serve.

- Cashier can respond to manager.
- Get account status from the accountant when withdrawing cash for the client, considering he does not need affirmation from the manager and the withdrawal amount is small.
- Deposit cash taken from the client and transfer it to the accountant to handle.
- Withdraw cash when the client requests for one.

All these events are triggered by the actions of the client. These functions are triggered from the Bank controller class.

The bank class has these activities, and the methods in bank class uses the methods of these derived classes wrapped up to perform actions such as loans, hiring, firing, resigning, cash withdrawal and deposits.

Client class – Derived class

The client class is a derived class of People class. It inherits the attributes of its parent class. The client handles events with the other derived classes such as managers, accountants, cashiers and ceo directly and directly.

The actions of the client will trigger the events of the banks class.

Some of the functions served by the client's class are listed below

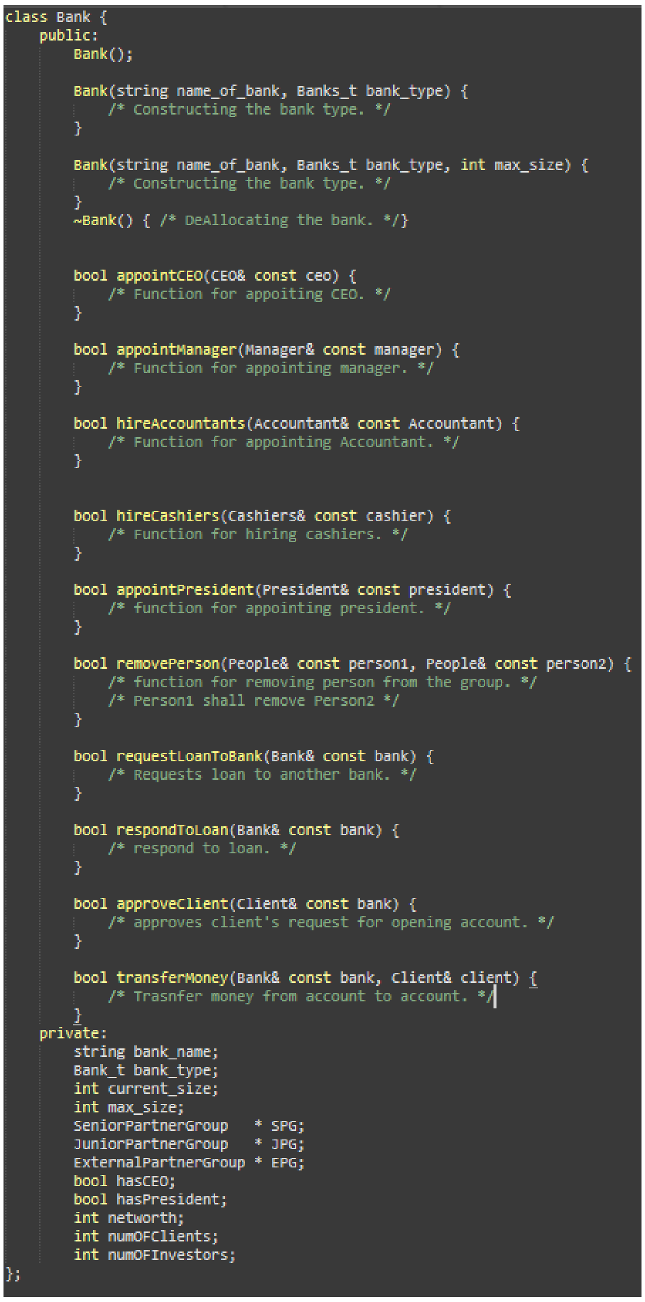
- Getting account status from an accountant before requesting for loan or for any other purpose.
- Depositing cash to the bank account.
- Withdrawing cash from the account.
- Requesting loan from a particular bank.
- Requesting account closures to the manager of the bank.
- Requesting account opening with a recommendation from another client.
- Recommending another client.
- Responding to a manager's claim of recommending another client.
- Transfer money from one bank to another bank or from one account to another account of the same bank.
- Add a mortgage certificate when requesting the bank for a loan.

Bank Class - Controller class

The bank class is a controller class that handles events of the bank and controls the simulation of different activities as mentioned in the story. Multiple banks can be created. Multiple combination of events can take place.

Each bank can hold groups in them as well as trigger activities.

- The banks can hire/fire employees by calling functions from the base-derived classes.



```cpp
class Bank {
    public:
        Bank();

        Bank(string name_of_bank, Banks_t bank_type) {
            /* Constructing the bank type. */
        }

        Bank(string name_of_bank, Banks_t bank_type, int max_size) {
            /* Constructing the bank type. */
        }
        ~Bank() { /* DeAllocating the bank. */}


        bool appointCEO(CEO& const ceo) {
            /* Function for appoiting CEO. */
        }

        bool appointManager(Manager& const manager) {
            /* Function for appointing manager. */
        }

        bool hireAccountants(Accountant& const Accountant) {
            /* Function for appointing Accountant. */
        }


        bool hireCashiers(Cashiers& const cashier) {
            /* Function for hiring cashiers. */
        }

        bool appointPresident(President& const president) {
            /* function for appointing president. */
        }

        bool removePerson(People& const person1, People& const person2) {
            /* function for removing person from the group. */
            /* Person1 shall remove Person2 */
        }

        bool requestLoanToBank(Bank& const bank) {
            /* Requests loan to another bank. */
        }

        bool respondToLoan(Bank& const bank) {
            /* respond to loan. */
        }

        bool approveClient(Client& const bank) {
            /* approves client's request for opening account. */
        }

        bool transferMoney(Bank& const bank, Client& client) {
            /* Trasnfer money from account to account. */
        }
    private:
        string bank_name;
        Bank_t bank_type;
        int current_size;
        int max_size;
        SeniorPartnerGroup   * SPG;
        JuniorPartnerGroup   * JPG;
        ExternalPartnerGroup * EPG;
        bool hasCEO;
        bool hasPresident;
        int networth;
        int numOFClients;
        int numOFInvestors;
};
```

- The banks can approve/disapprove loans by taking decisions of the CEO/president and the manager.
- The banks can help clients to withdraw and deposit money.
- The banks can transfer money from within the bank to other bank accounts as well.
- The banks lend loans based on the mortgage certificates provided by the client when requesting for loans.
- The banks can request loans from other banks as well. The managers can decide based on the size of the loan.

Groups class – container class

The groups class is a container class which has the linked list of people in. The banks class has the list of people in it. The group's class is a base class and there are derived classes that separate the employees of the bank.

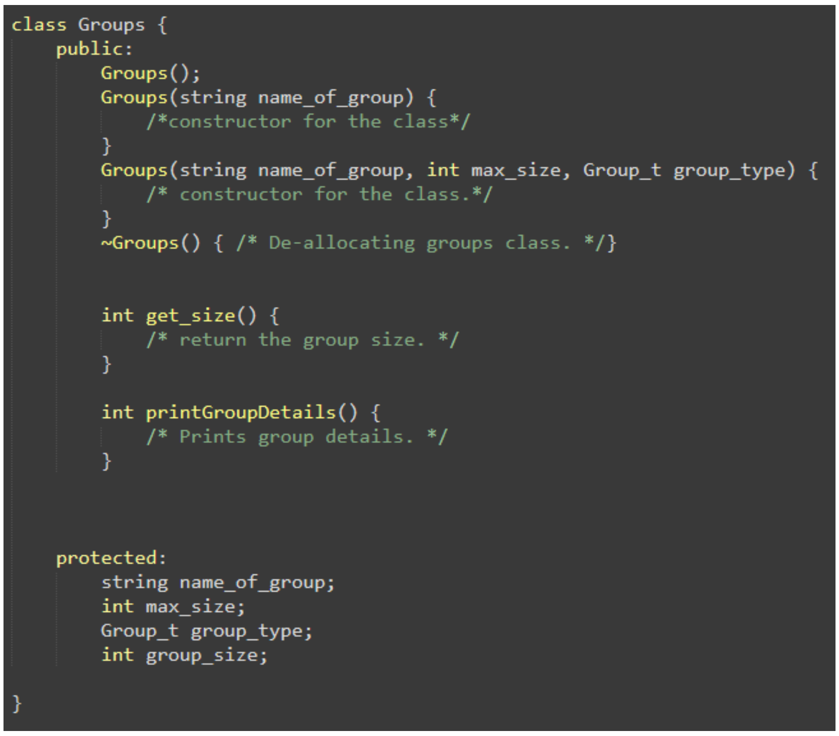
The derived classes.

• Senior Partners group	(CEO & President)
• Junior Partners group	(Manager, Accountant, Cashier)
• External Partners group	(Client)

The derived groups class takes people only from their respective role types. When the people are inserted into the groups the role type attribute is checked for and in case the wrong person is inserted to the group the console will throw an exception on the console for the user to get notified.

The groups class have a few services to offer.

- Insert people to the group.
- Search people on the group.
- Remove people on the group.
- Modify people of the group.
- Print details of the group or specific person.



```
class Groups {
    public:
        Groups();
        Groups(string name_of_group) {
            /*constructor for the class*/
        }
        Groups(string name_of_group, int max_size, Group_t group_type) {
            /* constructor for the class.*/
        }
        ~Groups() { /* De-allocating groups class. */}


        int get_size() {
            /* return the group size. */
        }

        int printGroupDetails() {
            /* Prints group details. */
        }



    protected:
        string name_of_group;
        int max_size;
        Group_t group_type;
        int group_size;

}
```

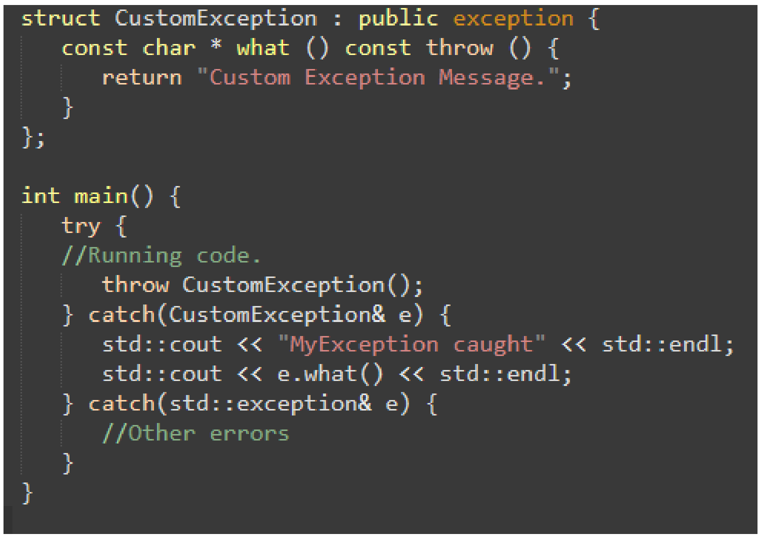
Figure 1: Groups class- Container class.

Exceptions handling.

The exceptions class is a handler class for handling exceptions. The standard exceptions are inherited to throw custom messages. The exceptions class is brought into the banks class when carrying out operations. Once the operations fail the errors are caught, and exceptions are displayed on the console.

The exceptions class serve a few functions.

- Check for the right type of parameters needed for serving an event.
- Check for adequate resource such as bank balance or net worth.
- Check for right activity once its triggered.
- Check for person if he/she exists in the group.
- Check for correct account number when triggering transaction.
- Check for loan requirements based on credentials.
- Check for hiring and firing the employees.



```
struct CustomException : public exception {
   const char * what () const throw () {
      return "Custom Exception Message.";
   }
};

int main() {
   try {
   //Running code.
      throw CustomException();
   } catch(CustomException& e) {
      std::cout << "MyException caught" << std::endl;
      std::cout << e.what() << std::endl;
   } catch(std::exception& e) {
      //Other errors
   }
}
```

Figure 2: Exception handling.

Restrictions and limitations

There a few notable restrictions for events to be triggered. They are exactly the opposite of the functions a class could serve. Restrictions and limitations could a sense of realism to the simulation. Some of the key restrictions and limitations are.

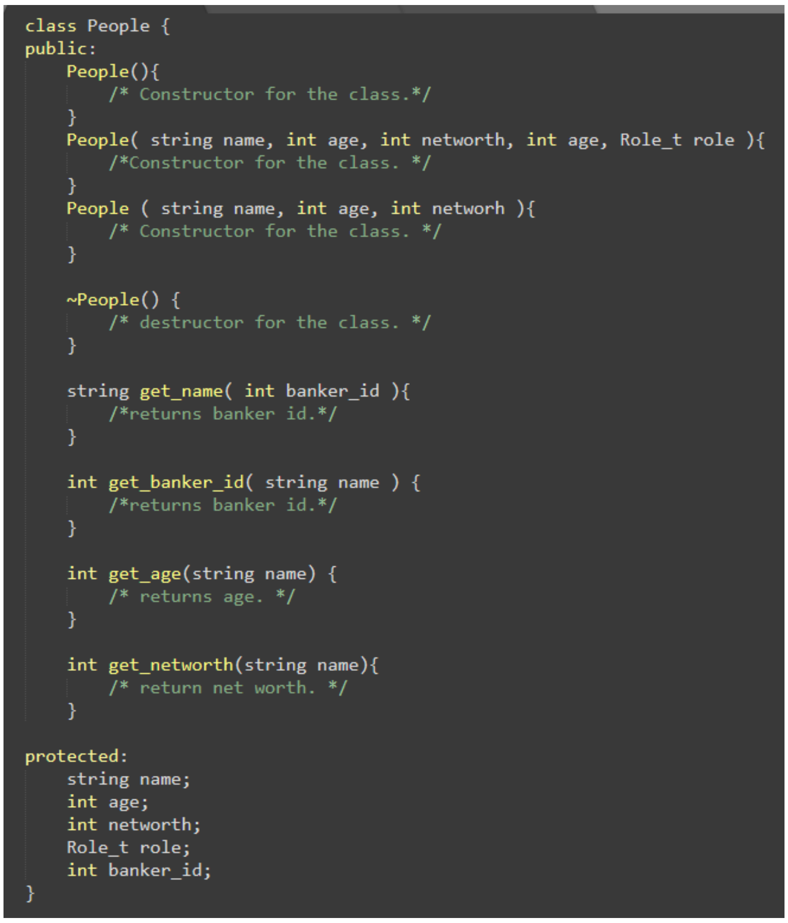
- CEO Class
  1. The ceo cannot pass an approval/rejection without the president.
  2. The ceo cannot request for a loan.
  3. The ceo cannot hire anyone without the manager.
  4. The ceo can fire any employee at will.
  5. The ceo can resign.
  6. Only the ceo can approve/reject the loan request from another bank.
- President class
  1. The president cannot make an approval/rejection.
  2. The president only can request for a loan.
  3. The president cannot hire/fire anyone.
  4. The president can resign.
  5. The president cannot approve/reject the loan request from another bank.
- Manager class
  1. The manager cannot hire/fire people without the consent of the CEO.
  2. The manager cannot request for a loan.
  3. The manager only can decide to open accounts for clients.
  4. Only manager can respond to accountants.
- Accountant class
  1. The accountant only can respond to cashier and manager.
  2. The accountant only can make changes to the bank accounts of the client.
  3. The accountant cannot make decisions, only manager can.
- Cashier class
  1. The cashier can respond only to client and accountant.
  2. The cashier cannot withdraw money more than a certain limit.
  3. The cashier cannot open accounts.
  4. The cashier cannot close accounts.
  5. The cashier cannot approve/reject loans.
- Client class
  1. The client cannot get a loan if the manager doesn't approve.
  2. The client cannot open account if the manager doesn't approve.
  3. The client cannot open an account without the recommendation of another client of the same bank.
  4. The client cannot withdraw cash without adequate balance.
  5. The bank account always needs to have 10% minimum balance.

6. The client can have multiple bank accounts.
7. The client can request to multiple loans.
8. The client needs to have a mortgage certificate if the loan is twice his net worth.
9. The client cannot close his bank account at will, if he has a loan to clear.
10. The client cannot transfer money to his own account.

- Bank class
  1. The bank can have only one CEO.
  2. The bank can have only one President.
  3. The bank can have multiple managers, cashiers and accountants.
  4. The bank can have multiple clients.
  5. The bank can give multiple loans to client, if the conditions are met.
  6. The bank can request for multiple loans to other banks.
  7. The bank can approve/reject loans with CEO's approval.
  8. Multiple bank instances can be created.
  9. The bank can have only 3 derived group classes.
- Derived groups
  1. The Senior partner groups can have only one CEO and president.
  2. The junior partner groups can have multiple managers and other employees.
  3. An employee of one bank can work only in one bank.
  4. External partners groups can have multiple clients, a client can be in any group.

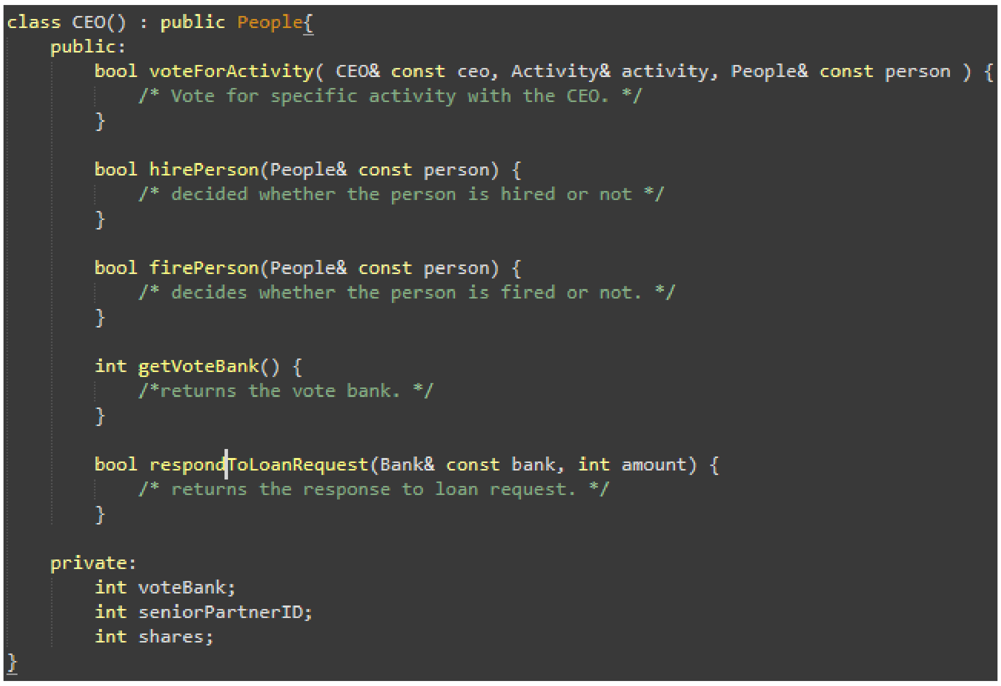
Class declaration.

These are a few class declarations for explanatory purposes, they do not possess any functionality of precision of following limitations and restrictions yet

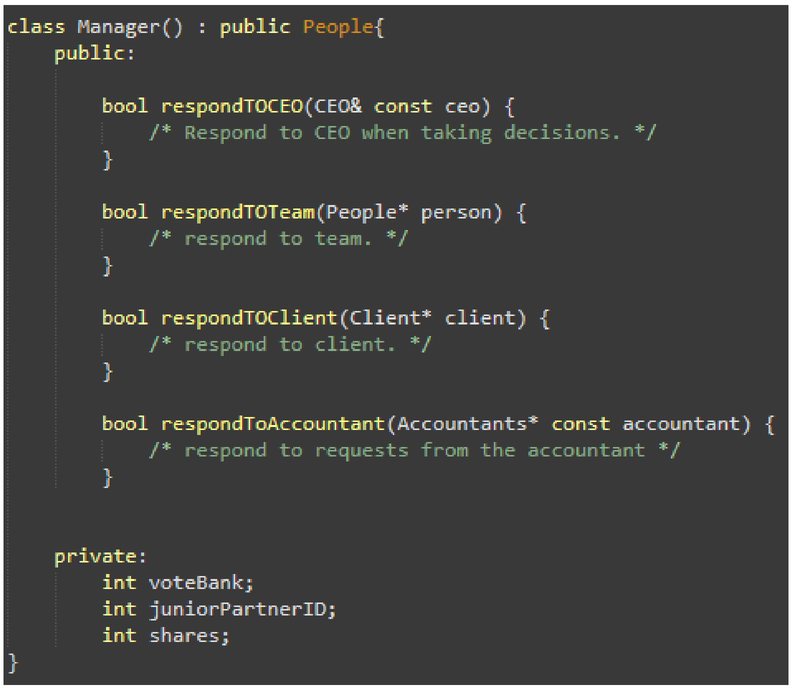


```cpp
class People {
public:
    People(){
        /* Constructor for the class.*/
    }
    People( string name, int age, int networth, int age, Role_t role ){
        /*Constructor for the class. */
    }
    People ( string name, int age, int networh ){
        /* Constructor for the class. */
    }

    ~People() {
        /* destructor for the class. */
    }

    string get_name( int banker_id ){
        /*returns banker id.*/
    }

    int get_banker_id( string name ) {
        /*returns banker id.*/
    }

    int get_age(string name) {
        /* returns age. */
    }

    int get_networth(string name){
        /* return net worth. */
    }

protected:
    string name;
    int age;
    int networth;
    Role_t role;
    int banker_id;
}
```

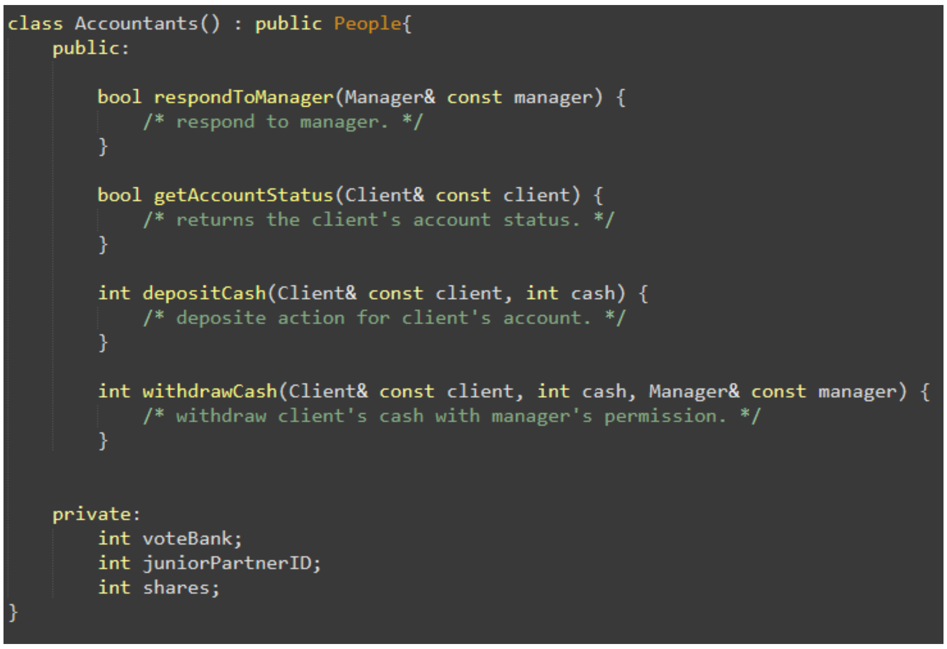
Figure 3: People class.



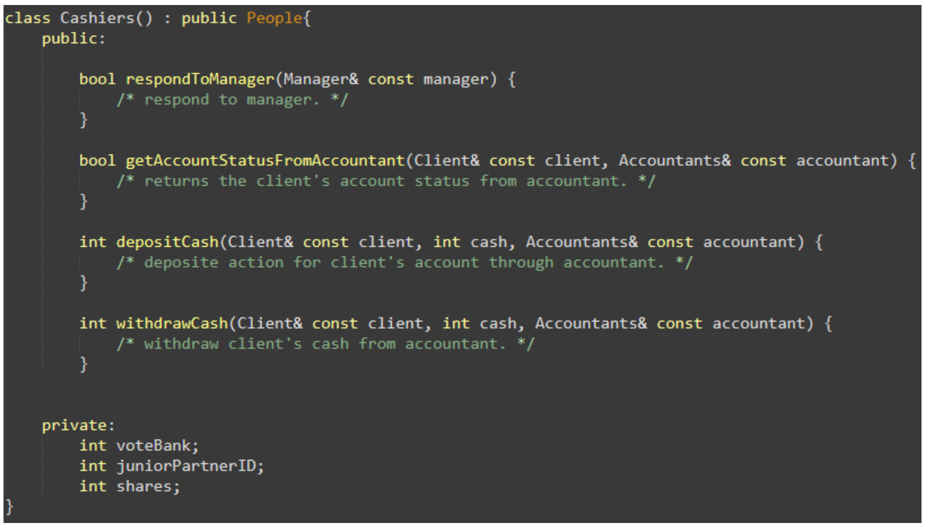
```cpp
class CEO() : public People{
    public:
        bool voteForActivity( CEO& const ceo, Activity& activity, People& const person ) {
            /* Vote for specific activity with the CEO. */
        }

        bool hirePerson(People& const person) {
            /* decided whether the person is hired or not */
        }

        bool firePerson(People& const person) {
            /* decides whether the person is fired or not. */
        }

        int getVoteBank() {
            /*returns the vote bank. */
        }

        bool respondToLoanRequest(Bank& const bank, int amount) {
            /* returns the response to loan request. */
        }

    private:
        int voteBank;
        int seniorPartnerID;
        int shares;
}
```



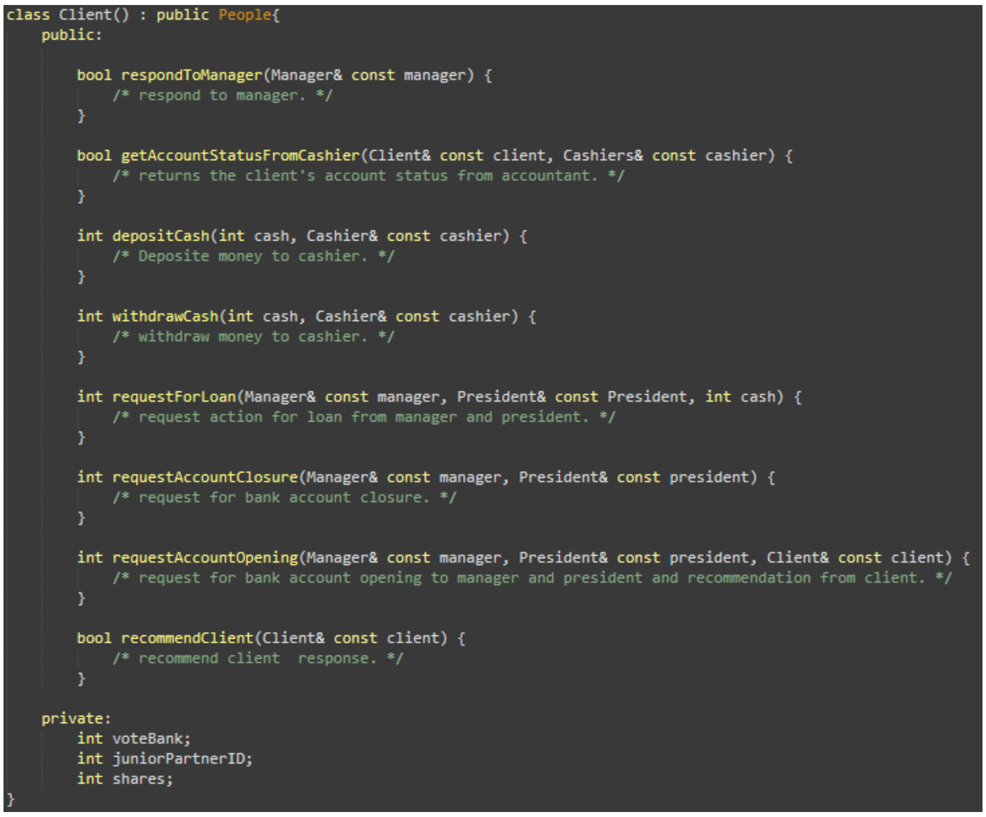
```cpp
class Manager() : public People{
    public:

        bool respondTOCEO(CEO& const ceo) {
            /* Respond to CEO when taking decisions. */
        }

        bool respondTOTeam(People* person) {
            /* respond to team. */
        }

        bool respondTOClient(Client* client) {
            /* respond to client. */
        }

        bool respondToAccountant(Accountants* const accountant) {
            /* respond to requests from the accountant */
        }


    private:
        int voteBank;
        int juniorPartnerID;
        int shares;
}
```



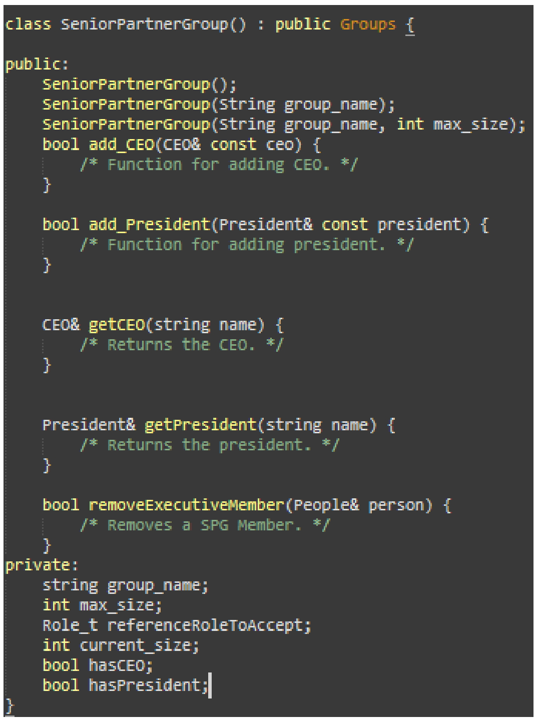
```cpp
class Accountants() : public People{
    public:

        bool respondToManager(Manager& const manager) {
            /* respond to manager. */
        }

        bool getAccountStatus(Client& const client) {
            /* returns the client's account status. */
        }

        int depositCash(Client& const client, int cash) {
            /* deposite action for client's account. */
        }

        int withdrawCash(Client& const client, int cash, Manager& const manager) {
            /* withdraw client's cash with manager's permission. */
        }


    private:
        int voteBank;
        int juniorPartnerID;
        int shares;
}
```



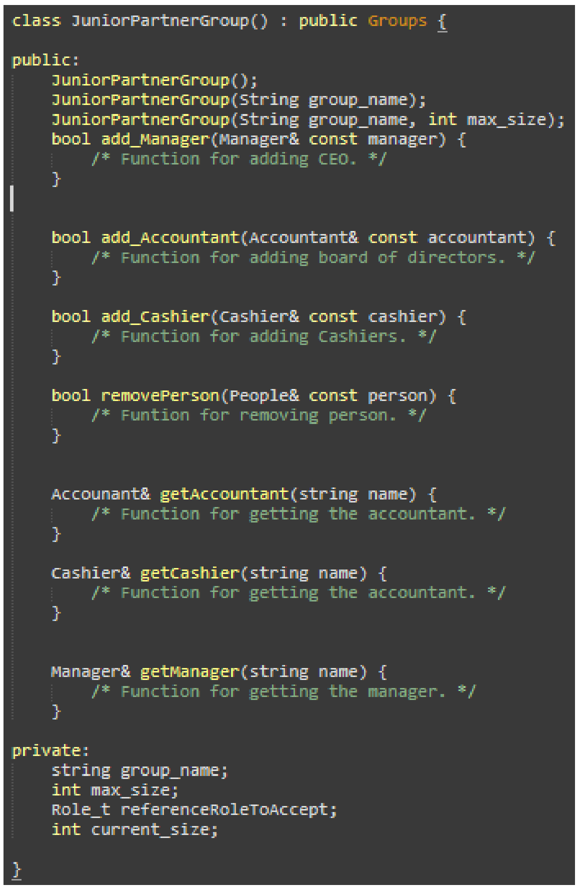
```cpp
class Cashiers() : public People{
    public:

        bool respondToManager(Manager& const manager) {
            /* respond to manager. */
        }

        bool getAccountStatusFromAccountant(Client& const client, Accountants& const accountant) {
            /* returns the client's account status from accountant. */
        }

        int depositCash(Client& const client, int cash, Accountants& const accountant) {
            /* deposite action for client's account through accountant. */
        }

        int withdrawCash(Client& const client, int cash, Accountants& const accountant) {
            /* withdraw client's cash from accountant. */
        }


    private:
        int voteBank;
        int juniorPartnerID;
        int shares;
}
```



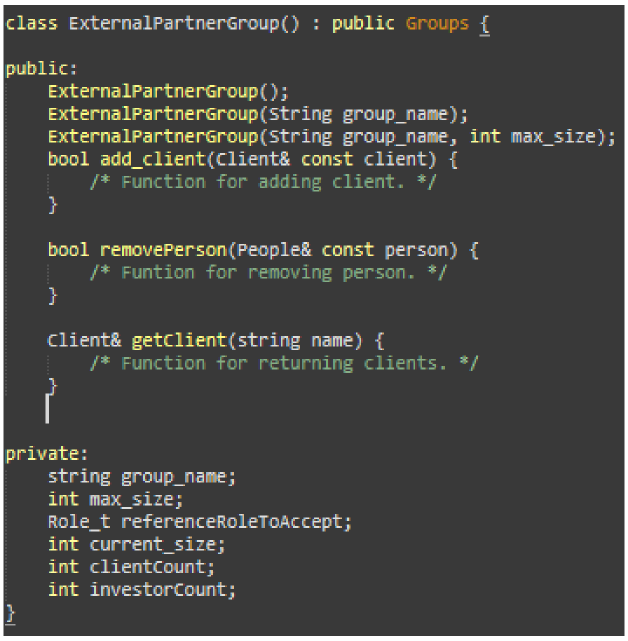
```cpp
class Client() : public People{
    public:

        bool respondToManager(Manager& const manager) {
            /* respond to manager. */
        }

        bool getAccountStatusFromCashier(Client& const client, Cashiers& const cashier) {
            /* returns the client's account status from accountant. */
        }

        int depositCash(int cash, Cashier& const cashier) {
            /* Deposite money to cashier. */
        }

        int withdrawCash(int cash, Cashier& const cashier) {
            /* withdraw money to cashier. */
        }

        int requestForLoan(Manager& const manager, President& const President, int cash) {
            /* request action for loan from manager and president. */
        }

        int requestAccountClosure(Manager& const manager, President& const president) {
            /* request for bank account closure. */
        }

        int requestAccountOpening(Manager& const manager, President& const president, Client& const client) {
            /* request for bank account opening to manager and president and recommendation from client. */
        }

        bool recommendClient(Client& const client) {
            /* recommend client  response. */
        }

    private:
        int voteBank;
        int juniorPartnerID;
        int shares;
}
```



```cpp
class SeniorPartnerGroup() : public Groups {

public:
    SeniorPartnerGroup();
    SeniorPartnerGroup(String group_name);
    SeniorPartnerGroup(String group_name, int max_size);
    bool add_CEO(CEO& const ceo) {
        /* Function for adding CEO. */
    }

    bool add_President(President& const president) {
        /* Function for adding president. */
    }


    CEO& getCEO(string name) {
        /* Returns the CEO. */
    }


    President& getPresident(string name) {
        /* Returns the president. */
    }

    bool removeExecutiveMember(People& person) {
        /* Removes a SPG Member. */
    }
private:
    string group_name;
    int max_size;
    Role_t referenceRoleToAccept;
    int current_size;
    bool hasCEO;
    bool hasPresident;
}
```



```cpp
class JuniorPartnerGroup() : public Groups {

public:
    JuniorPartnerGroup();
    JuniorPartnerGroup(String group_name);
    JuniorPartnerGroup(String group_name, int max_size);
    bool add_Manager(Manager& const manager) {
        /* Function for adding CEO. */
    }


    bool add_Accountant(Accountant& const accountant) {
        /* Function for adding board of directors. */
    }

    bool add_Cashier(Cashier& const cashier) {
        /* Function for adding Cashiers. */
    }

    bool removePerson(People& const person) {
        /* Funtion for removing person. */
    }

    Accounant& getAccountant(string name) {
        /* Function for getting the accountant. */
    }

    Cashier& getCashier(string name) {
        /* Function for getting the accountant. */
    }


    Manager& getManager(string name) {
        /* Function for getting the manager. */
    }

private:
    string group_name;
    int max_size;
    Role_t referenceRoleToAccept;
    int current_size;

}
```



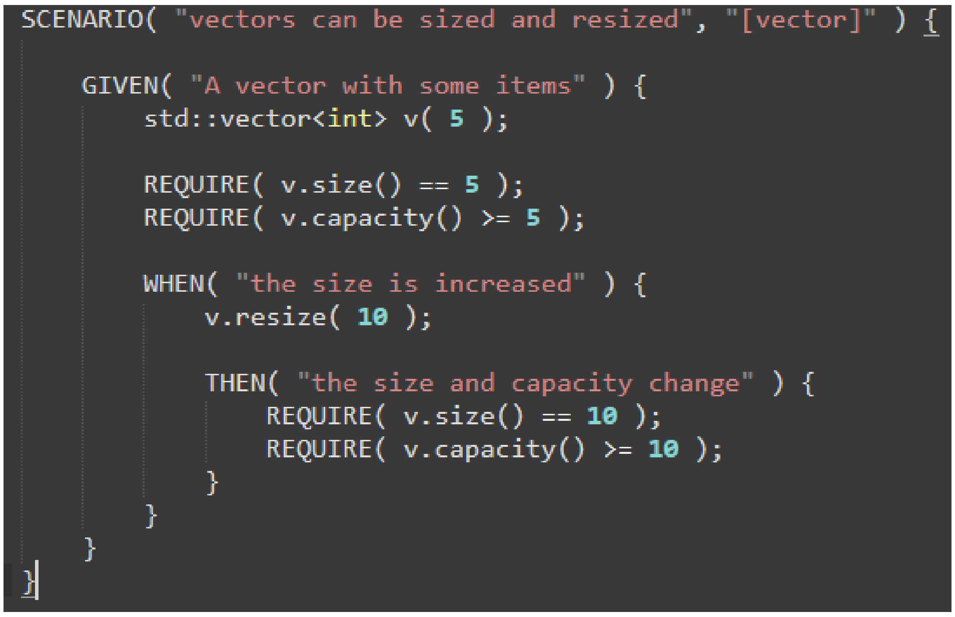
```
class ExternalPartnerGroup() : public Groups {

public:
    ExternalPartnerGroup();
    ExternalPartnerGroup(String group_name);
    ExternalPartnerGroup(String group_name, int max_size);
    bool add_client(Client& const client) {
        /* Function for adding client. */
    }

    bool removePerson(People& const person) {
        /* Funtion for removing person. */
    }

    Client& getClient(string name) {
        /* Function for returning clients. */
    }


private:
    string group_name;
    int max_size;
    Role_t referenceRoleToAccept;
    int current_size;
    int clientCount;
    int investorCount;
}
```

Testing

The testing phase of the project is to test all the cases, activities and events of the class and see if the exceptions are thrown correctly and the code does not break.

The testing phase will use catch2 for some automated testing. The services offered.

- Make 1000's of clients and have them to open accounts and check for exceptions.
  Catching of REQUIRE.
- Fast insertions and delete and to make sure the numbers match up.
  SECTION checking.
- Performance testing and timing events.
- SCENARIO based testing. Since this is a simulation, the highlight of testing cases would be with scenarios and GIVEN conditions.
- Make wrong insertions to groups.
- Make multiple combinations of instances. Loan request with and without required parameters.
- Hiring/firing without the right employees.
- Withdrawing and depositing cash with the wrong employee and without adequate balance.
- Hiring of employees of different banks.
- Adding multiple CEO's and presidents to an account.



```
SCENARIO( "vectors can be sized and resized", "[vector]" ) {

    GIVEN( "A vector with some items" ) {
        std::vector<int> v( 5 );

        REQUIRE( v.size() == 5 );
        REQUIRE( v.capacity() >= 5 );

        WHEN( "the size is increased" ) {
            v.resize( 10 );

            THEN( "the size and capacity change" ) {
                REQUIRE( v.size() == 10 );
                REQUIRE( v.capacity() >= 10 );
            }
        }
    }
}
```

Figure 4: Sample Test case.