FINAL PROJECT REPORT

The Banking System

Keshav Dandeva
302333

Table of Contents

The Banking System

EOOP Project

Goal

The system that simulates a banking system in C++ by realizing the concepts of object-oriented programming. The end to end solution is a system that handles the employees and the clients of the bank, the bank being able to negotiate loans based on the net worth of individuals, calculating the return amounts and time given to the clients based on the size of the loan taken. Not only does this have simple algorithms, but will also have a database to store all these systematically.

The fundamentals of object-oriented programming is employed to re-create this system such as polymorphism, hierarchy, and encapsulation. An error-free and non-redundant software is presented at the end of this report.

Story

The agenda of the story is to make an application with the theme of the Banking system, the system is closely bounded which follows a strict hierarchy of managers, associates, accountants, cashiers, bankers, customers and investors.

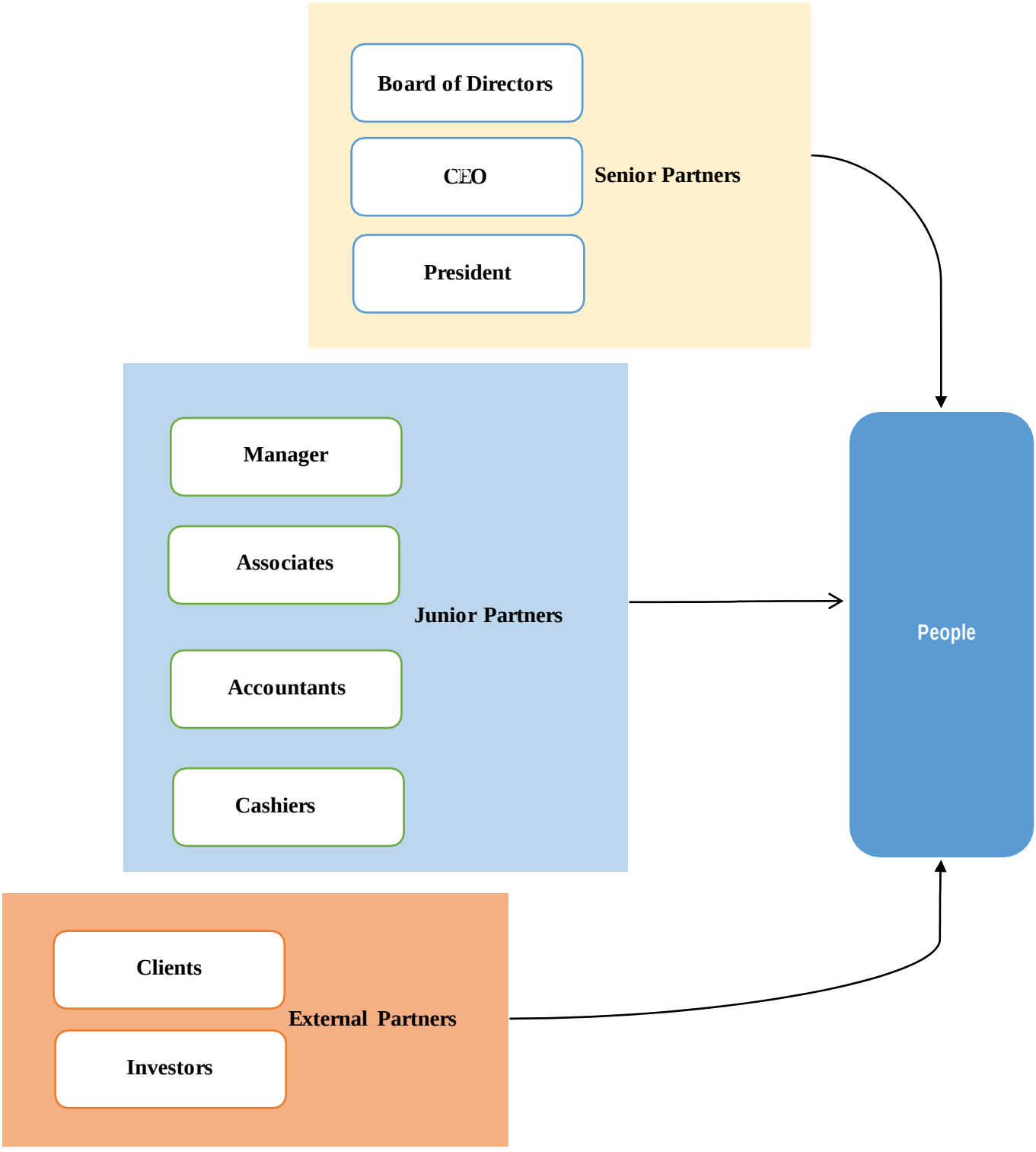
The people of the bank are all derived from a base class of people. The hierarchy is based on enumerations given to each Person, the enumerations are compared and the people are placed on that hierarchy.

A bank will contain all of these classes of people. Multiple bank instances can be made to have a bigger picture.

A database is used to keep a record of all the banks, and the people of the banks. Here, relational database is used and relations are made to the instances. The instances are appended to the linked list and certain methods are employed to retrieve the relations from those linked lists.

The system will be able to appoint associates to customers and clients, a manager will lead a group of associates, accountants and cashiers. Bankers will fall in the same group as managers, financial officers, president and CEO. All of these people fall under the class of Board of Directors who are the pinnacle of the hierarchy.

Conceptual diagram of the banking system's hierarchy.



Memory map and class specification

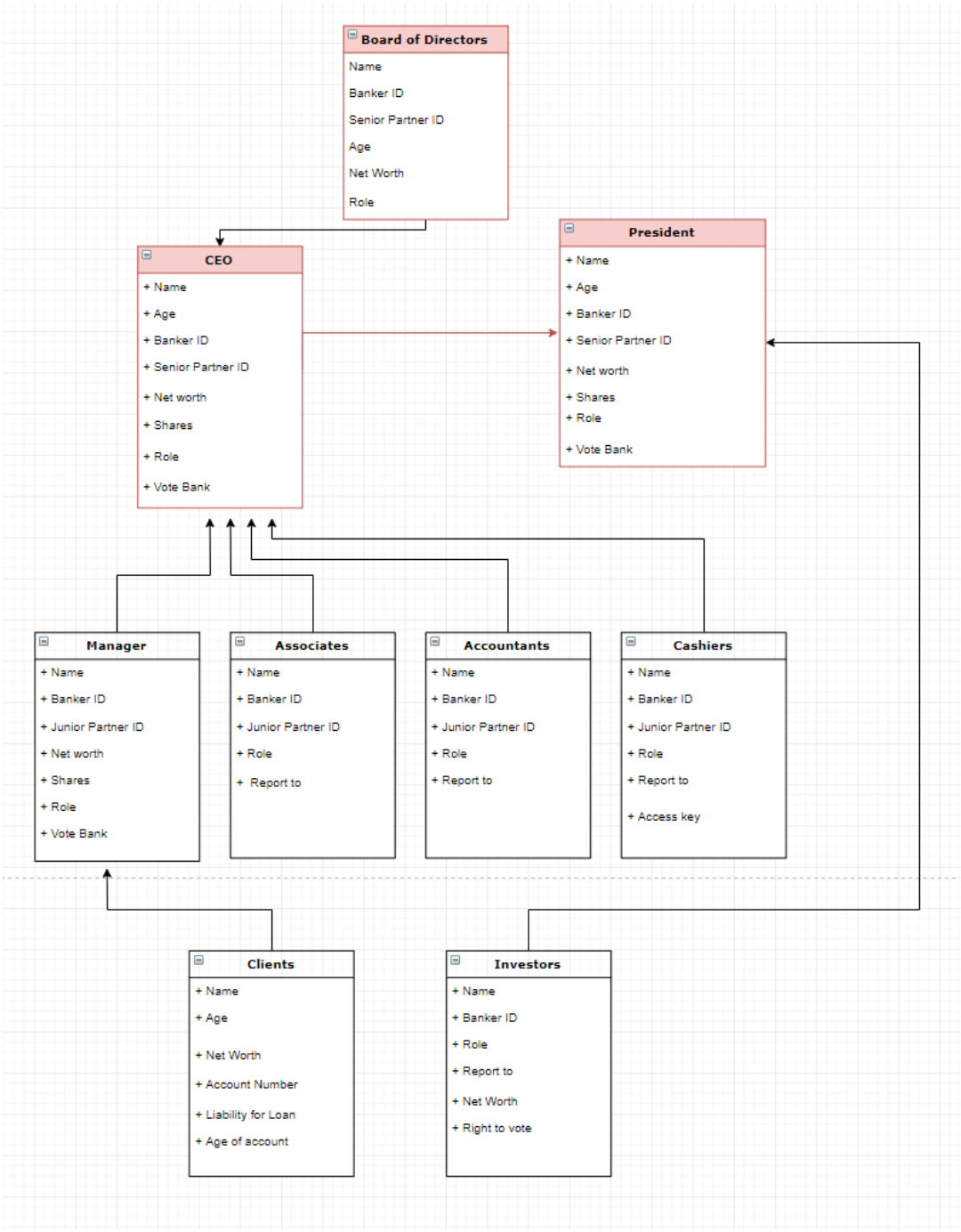


Figure 1: Memory map of the banking system.

Class Topology

People.h – Base class

The People class is a Generic class that stands as a background for all the other classes of people. The members of the bank are derived from this class.

There are certain attributes of a Person which is common for all. Such attributes are defined here. The Getter and Setter function are included for accessing private variables of this class.

Groups.h – Base class

The Groups.h is a base class which has a database of the people class instances. Three different Groups classes are made here for the different hierarchies.

- **Senior Partner Database**
- **Junior Partner Database**
- **External Partner Database**

The Groups.h class makes it easy to segregate people among different groups, this makes it easier for users to retrieve the information from different hierarchy types. The instances are put into linked lists which stores all the instances linearly, this can help the user to find the person based on **Names** or **Banker ID.**

Relationships between Groups can be made to segregate bankers into groups made by the **Managers**, each group is assigned a manager and some support stuff under him. Support stuff from other groups can also participate in different groups, If the manager agrees to do so.

The president of the club has to sign off the petition for group assignments.

Banks.h – Base class

Banks.h class is a Superior class that contains all the classes underneath it. This class can create multiple banks. A bank instances can have multiple groups of People and different activities; each activity has to be signed off by a person of higher authority.

The Bank class can create possibility to have a more complex banking system where banks can fund other banks and so on.

Activities.h – Controller Class

The activities class is a controller class that takes in Bank instance parameters and a supported activity can be chosen from a list of Enumerations. An algorithm is used to check for basic requirements to conduct and activity.

If the basic requirements are surpassed the class makes sure the activity is passed. If not, the class will make decisions based on the necessity of the requirement and the activity can be passed or failed.
A list of activities is set by default.

- Getting a loan.
- Voting of new Manager/President/CEO
- Investments
- Lottery
- Bank-Bank funding

Most activities take in Bank as an instance and the parameters are overloaded for easier and more flexible usage.

Enumerations.h – Static class

An enumerations class is a static class that holds a list of enumerations for different classes of the project. The People class is a Base class for people, since the preceding classes needs a hierarchy, enumerations can be made for different types of people. This makes the tasks easier when appointing the role of specific people.

There is enumeration for different activities, so when an activity is conducted, the parameter can take in an enumeration of an activity, this avoids the use of strings.

Enumerations are also available for different types of banks such as corporate banks, government banks, private banks and so. This helps us create a diverse set of Banking systems.

The types of enumerations are.

- People

    1. Board of directors.
    2. CEO
    3. President
    4. Manager
    5. Associates
    6. Accountants
    7. Cashiers
    8. Clients
    9. Investors

- Activities

    1. Loans
    2. Voting
    3. Investments
    4. Lottery
    5. Bank-Bank funding.

- Banks

    1. Corporate Banks
    2. Private Banks
    3. Government Banks

Exceptions.h – Handler Class

The exception class is handler class for handling errors and logs in the programme. Certain errors are overloaded to create custom errors that can be raised when situations aren't favourable.

During creation of groups or conducting of bank activities certain conditions might not be favourable such as not enough funds, no voting power, too many loan requests such errors can be raised by the Exceptions class.

This makes it easier to handle errors and keep them well structured in the same file for easier access and uniformity.

```
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

   }

}
```

Class Declarations

People class

```
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. */
    }
private:
    string name;
    int age;
    int networth;
    Role_t role;
    int banker_id;
}
```

Figure 2: Base class of People.

Groups class

```
class Groups {
    public:
        Groups();
        Groups(Group_t group) {
            /*constructor for the class*/
        }

        Groups(Group_t group, int max_size) {
            /* constructor for the class.*/
        }

        bool add_member(Group_t group, Role_t role, int banker_id) {
            /* function for adding members to the group. */
        }

        bool add_manager(Group_t group, Manager& const manager) {
            /* add manager to the group. */
        }

        bool remove_member(Group_t group, Manager& const manager) {
            /* remove memeber for the group. */
        }

        bool remove_manager(Group_t group, People& const Person) {
            /* Remove manager from the group. */
        }

        int get_group_size( Group_t group, string group_name ) {
            /* returns the size of the group. */
        }

        int get_commom_group_association(Group_t, People& const person) {
            /* returns the number of common group associations of a person. */
        }

        String get_common_group_association(Group_t, People& const person) {
            /* returns the first name of a common group association. */
        }

    private:
        string group_name;
        Group_t group;
        int currentSize;
        int maxSize;
        People groupLeader;
        Manager groupManager;

}
```

Figure 3: Skeletal code for the Groups class.

The groups class will also contain the database (linear linked list) for automatically adding members to the group on creating the instance of a People object.

Class Banks

```
class Banks {
    public:
        Banks();
        Banks(string name_of_bank, Bank_t bank_type) {
            /*constructor for the class*/
        }

        Banks( string name_of_bank ) {
            /* constructor for the class.*/
        }

        bool appoint_ceo(Group& const group){
            /* appointing CEO for banks.*/
        }

        bool appoint_leaders(Group&  const group) {
            /* appointing leaders for the bank.*/
        }

        bool permission_for_activity(People& const person, Activity_t activity) {
            /* Returns the permission for conducting the activity. */
        }

        int get_net_worth( ) {
            /* returns the banks networth. */
        }

        bool loan_money_to_other_bank( Banks& const recipientBank, int amount ) {
            /* returns the permission for the bank loan. */
        }

        bool request_load_to_bank(Banks* const requestingBank, int amount) {
            /* returns the result for request for loan. */
        }

    private:
        string name_of_bank;
        int max_size;
        int current_size;
        int networth_of_bank;
        Group* group_of_people;
        int debt;
        int margins;
        int past_loan_requests;
}
```

Figure 4: Skeletal code for banks class

Board of directors

```
class BoardOfDirector(): public People{
    public:
        BoardOfDirector();
        BoardOfDirector( string name, int age, int net_worth, Role_t role  ){
            /* Constructor of the class */
        }
        BoardOfDirector( string name, int age, int net_worth ){
            /* Constructor of the class */
        }
        ~BoardOfDirector(){
            /* Destructor of the class */
        }

        voteWithCEO( Ceo& const ceo, Activity_ activity, People& const people ){
            /* function for voting for an activity with the CEO. */
        }
        voteWithCEO( Ceo& const ceo, Activity_ activity, Investor& const investor){
            /* function for voting with the CEO for an activity with the investor */
        }


        int get_age( string name ) {
            /* returns name */
        }
        int get_networth( string name ) {
            /* returns net worth */
        }
        int get_role( string name ){
            /* return role */
        }
        int get_bankerID(string name) {
            /* return banker ID */
        }
        int get_name( int banker_id ) {
            /* returns name */
        }
        int get_senior_partner_id( string name ) {
            /* return senior partner id */
        }

    private:
        string __name;
        int __age;
        int __net_worth;
        Role_t __role;
        int __banker_id;
        int __senior_partner_id;

};
```

Figure 5: Skeletal code for Board of Directors

Segment Enumerations

```
typedef enum Roles {
    BRD_DIRECTOR  = 0,
    CEO           = 1,
    PRSDNT        = 2,
    MNGR          = 3,
    ASSCATE       = 4,
    ACCNTS        = 5,
    CASHRS        = 6,
    CLNTS         = 7,
    INVSTRS       = 8
} Role_t;


typedef enum Activities {

    LOANS       = 0,
    VOTING      = 1,
    INVSTMNTS   = 2,
    LOTTRY      = 3,
    BBFNDING    = 4,
    CUSTM_ACTY  = 5
} Activity_t;


typedef enum Banks {

    CORPORATE_BANKS  = 0,
    PRIVATE_BANKS    = 1,
    GOVERNMENT_BANKS = 2
} Banks_t;
```

Figure 6: Snippet of code representing the enumerations.

A snippet of code describing the enumerations for different activities, roles and bank types. Further types will be made as the project progresses, an element of flexibility is maintained.

Enumerations can help create a global labelling for certain events and those functions requiring the need to decide based on an event can look up to this snippet for critical information.

Additional Classes

```
class Database {
    DataBase();
    DataBase(string name);
    Database(string name, Group& const group);
    ~Database();

    bool add_member(People& const person, Group& const group);
    bool delete_member(People& const person, Group& const group);
    bool search_memeber(People& const person, Group& const group);
};

class Manager: public Prople {
    Manager();
    Manager(string name);
    Manager(string name, int age, int networth, int votebank);
    ~Manager();


private:
    string name;
    int age;
    int networth;
    int votebank;
    int banker_id;
    int junior_partner_id;
}
```

Figure 7: Additional classes, the database class and the manager classes.

The database class is a linked list class. The node of the linked list is a simple structure with a head pointer.

Testing

Separate classes will be made for testing the Groups and People classes by adding 10k People into lists and making sure the **lists don't break** and to check for memory integrity.

Fast insert and delete and search functions are tested and **asserted**. The exception class has to **catch the required errors without breaking the code during runtime** and the errors are logged into a text file-based **logging system**.

The **equalOperator** and **copyOperators** functions are implemented for shallowCopy and deepCopy testing.

```
Testing!
1. Printing employee information.
2. Firing of employees and re-hiring them.
3. Opening and closing of bank accounts.
4. Withdrawing and depositing cash.
5. Conducting inter and outer bank transactions.
6. Requesting loans, repayment and account closure.
Enter choice:
```

Figure 8: Testing Cases for the software