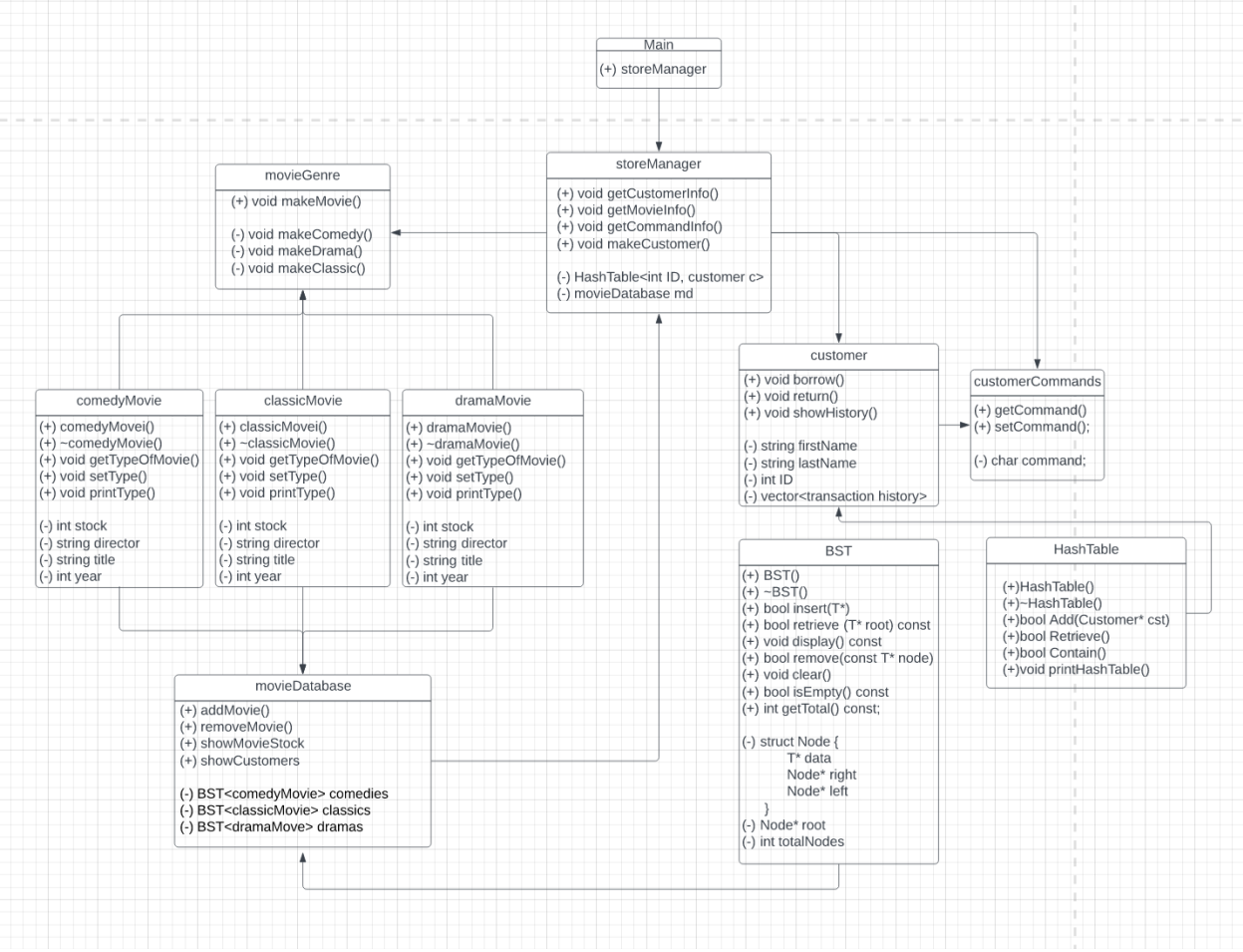
Group 7: Andrzej Dawiec, Noah Nguyen, Bastien Orbigo, Anthony Vu
Dr. Afra Mashhadi
CSS 343A Fall 22
Homework 4: Movies Project Design
15 November 2022

1. UML Class diagram: Only include class names (OK to include more details for classes, but not necessary) You can use a program or neatly draw it by hand. Use simple UML notation for your class diagrams: arrow for inheritance (is-a); line with diamond for composition (has-a or part-of). Composition (has-a or part-of) is a special kind of association. Some things in the UML of the design are best shown as classes. For example, a List class may be implemented as an STL list or even as an array, but in the design, it is critical the reader knows it is a collection.



2. Class Interactions: Explanation of how classes interact (Show the program flow, the methods invoked, for standard Use Cases such as insert or borrow.). For example a Borrow operation would on a high level entail: finding the customer; finding the movie; associating customer with movie, so the movie becomes a part of the customer's history; associating movie with customer so it is known that the movie is borrowed by the customer and that the movie has been borrowed. Give objects and methods called, i.e., pseudocode, so that later you can take the design and implement the functionality. Note that conciseness is as important as clarity, i.e., do not burden the reader with excessive documentation or unnecessary detail.

For our design, the storeManager is the forefront of the design, which will read in from all three text files and call will build the movieDatabase object and create a hashtable of customer objects, which will be created as the text files are read in. The movieDatabase object has an addMovie method which will be invoked per line as the movies and stock are read in. We also have a class called movieGenre which will make a movie, of the specific class based on the type of movie read in. Each movie will hold data such as the stock, director, title, and year it is produced. To organize the movies, we have three separate BSTs in the movieDatabase class, which organize them by genre. The customer class will hold the name, ID, and currently a vector of their transaction history, although we aren't fully set on if that's the best way to do it. The customer class will also know about the customerCommands class which will be created as the text file of commands is read in.

3. Main: Include a description of what would be called from main.cpp, which should be very, very short.

Creates a StoreManager object and calls the methods of storeManager.

4. **.h Files**: Since this is a design, you might not include all, or any, parameters in methods. Or they might change. That's okay, but overall the design is to be sound. You do not need to include .h files of classes you will not implement, of the extensions beyond the assignment specifications, but you must include a description of those classes. Order the files properly, i.e., put the most important classes first, put parent classes before children classes.

Movies > C BST.h > BSTree<T>

```h
#include <iostream>
using namespace std;
template <typename T>
class BSTree
{
public:
    BSTree();
    ~BSTree();

    bool insert(T *rhs);
    bool retrieve(T*& rhs) const;
    void display() const;
    bool remove(const T* node);
    void empty();
    bool isEmpty() const;
    int getTotal() const;
private:
    struct Node
    {
        T *data;
        Node *right;
        Node *left;
    };
    Node *root;
    int totalNodes;
};
```

Movies > C Customer.h > Customer > ~Customer()

```h
#ifndef CUSTOMER_H_
#define CUSTOMER_H_

#include <iostream>
#include <vector>

using namespace std;

class Customer {
    friend ostream &operator<<(ostream &out, const Customer &rhs);

public:
    Customer();
    Customer(int customerID, string firstName, string lastName);
    ~Customer();

    void borrowMovie();
    void returnMovie();
    void showHistory();

private:
    int customerID;
    string firstName;
    string lastName;
    vector<string> transaction_history;
};

#endif
```

Movies > C ComedyMovie.h > ComedyMovie > printType()

```h
#ifndef COMEDY_MOVIE_H_
#define COMEDY_MOVIE_H_
#include <iostream>
using namespace std;

class ComedyMovie {
    public:
        ComedyMovie();
        ~ComedyMovie();
        void getTypeOfMovie();
        void setTypeOfMovie();
        void printType();
    private:
        int stock;
        string director;
        string title;
        int year;
};
#endif
```

Movies > C CustomerCommands.h > CustomerCommands > setCommand()

```h
#ifndef CUSTOMER_COMMANDS_H_
#define CUSTOMER_COMMANDS_H_
#include <iostream>
class CustomerCommands {
    public:
        CustomerCommands(char ins_command);

        char getCommand() const;
        void setCommand();

    private:
        char command;
};
#endif
```

```h
#ifndef CLASSIC_MOVIE_H_
#define CLASSIC_MOVIE_H_
#include <iostream>

class ClassicMovie {
    public:
        ClassicMovie();
        ~ClassicMovie();
        void getTypeOfMovie();
        void setTypeOfMove();
        void printType();
    private:
        int stock;
        string director;
        string title;
        int year;
};
#endif
```

Movies > DramaMovie.h > DramaMovie > printType()

```h
#ifndef DRAMA_MOVIE_H_
#define DRAMA_MOVIE_H_
#include <iostream>
using namespace std;

class DramaMovie {
    public:
        DramaMovie();
        ~DramaMovie();
        void getTypeOfMovie();
        void setType();
        void printType();
    private:
        int stock;
        string director;
        string title;
        int year;
};

#endif
```

Movies > HashTable.h > HashTable > retrieve(int) const

```h
#ifndef HASHTABLE_H_
#define HASHTABLE_H_
#include "Customer.h"
#include <iostream>

using namespace std;

class HashTable {
public:
  HashTable();
  ~HashTable();

  bool insert(Customer *cst);
  bool retrieve(int id) const;
  bool contains(int cstID) const;
  void printHashTable();

private:
  struct HashNode {
    int key;
    Customer *c;
  };
  HashNode *hashNode = nullptr;
  const int NUM_OF_BUCKETS = 99999;
};
#endif
```

Movies > MovieDatabase.h > MovieDatabase > classics

```h
#ifndef MOVIE_DATABASE_H_
#define MOVIE_DATABASE_H_
#include "BST.h"
#include "ComedyMovie.h"
#include "DramaMovie.h"
#include "ClassicMovie.h"

class MovieDatabase {
    public:
        void addMovie();
        void removeMovie();
        void showMovieStock();
        void showCustomers();

    private:
        BST<ComedyMovie> comedies;
        BST<DramaMovie> dramas;
        BST<ClassicMovie> classics;
}
#endif
```

Movies > MovieGenre.h > MovieGenre

```h
#ifndef MOVIE_GENRE_H_
#define MOVIE_GENRE_H_
#include <iostream>
using namespace std;
class  MovieGenre {
    public:
        void makeMovie();
        void makeComedy();
        void makeDrama();
        void makeClassic();

};
#endif
```

Movies > StoreManager.h > StoreManager > getCustomerInfo()

```h
#ifndef STORE_MANAGER_H_
#define STORE_MANAGER_H_
#include <iostream>
#include "MovieDatabase.h"
#include "Customer.h"
using namespace std;
class StoreManager {
    public:
        void getCustomerInfo();
        void getMovieInfo();
        void getCommandInfo();
    private:
        vector<MovieDatabase> movies;
        vector<Customer> customers;
};
#endif
```

Movies > main.cpp > main()

```cpp
#include <iostream>

using namespace std;

int main() {


  return 0;
}
```

The design will take considerable effort. A useful approach is to imagine that you are writing a design that someone else will need to implement and extend. Document your design as you would want someone else to document for you. Design beyond the specifications (what you turn in). Design so that the answer to all these questions is yes (maintaining good design principles).

1. Can your design be extended beyond the specifications given here?
2. Could you easily add new videos or DVDs to your design?
3. Can you easily add other categories of videos or DVDs?
4. Could you easily add new categories of media to your design, for example, music?
5. Could you expand to check out other kinds of items, for example VCRs or DVD players?

6. Could you easily add new operations to your design?
7. Could you incorporate time, for example, a due date for borrowed items?
8. Could you easily add an additional store, or handle a chain of stores?

Your design can go beyond the scope of these specifications (and you won't need to implement extensions). Thinking of possible extensions in advance often improves the design.