Assignment 4 - Design Document

Group Number: 5

Group Members: Srikar Duggempudi, Brandon Hu, Darren Kevin Luk, Mohammad Ali

Customer

The customer class will contain a string for the name, an integer for the ID, and a vector of strings to keep track of the history. The customers will contain accessor methods for its variables and functions to add movies to its borrowed list and return movies from its borrowed list.

CustomerCollection

The CustomerCollection class will contain a map that will find a Customer when given their unique ID. This class also contains accessor and mutator methods for the Customer.

Movie

The Movie class will contain a char which represents a Type for what kind of movie it is, a string for its title, an integer for the year, a string for the director, and an int for the stock of the Movie. The Movie class will be split into Comedy, Drama, or Classic. The Classic subclass will also contain a string for its major actor, and an integer for its month. The Movie class will contain a pure virtual method for returning its information which will be defined by its children classes.

Classic Movie

The classic movie class has additional attributes like major actor and release month. This class implements the virtual method getInfo() to suit its output.

Comedy Movie

The comedy movie class implements the virtual method getInfo() to suit its output.

Drama Movie

The drama movie class implements the virtual method getInfo() to suit its output.

MovieFactory

The MovieFactory class will contain a pure virtual function create(), to return Movie pointers. This class will create Movies based on the MovieType given by the Movie class and return them back to the Movie class. We use this approach because of the different formats that are used for the different categories. For example, the classic movie type has a major actor associated with it as well as the month of the year it was released.

MovieCollection

The MovieContainer will contain a vector of Movie pointers. The MovieContainer has a pure virtual sort() function which the subclasses will have to implement to sort according to the type of Movie it contains. The MovieContainer will have the subclasses ClassicContainer (sorted by title then year), DramaContainer (sorted by director, then title), and ComedyContainer(sorted by release date, then major actor). The MovieContainer will have a void display() function to display all movies in its sorted manner, a bool contains(string) to check if it contains a movie, and a bool add(string) to create a Movie* to add a movie to its list.

Classic Movie Collection
The classic movie collection class implements the virtual method sort() to sort its attributes based on its unique sorting attributes. The findyBy() function will retrieve the respective genre Movie based on their unique sorting attributes.

Comedy Movie Collection
The comedy movie collection class implements the virtual method sort() to sort its attributes based on its unique sorting attributes. The findyBy() function will retrieve the respective genre Movie based on their unique sorting attributes.

Drama Movie Collection
The drama movie collection class implements the virtual method sort() to sort its attributes based on its unique sorting attributes. The findyBy() function will retrieve the respective genre Movie based on their unique sorting attributes.

Inventory
The inventory will have a map that will find the Movie in the MovieCollection when given the MovieType character. The Inventory also has a bool borrow(string) that will return false if the movie isn’t in it or can’t be borrowed and true if it can be borrowed, a return(string) bool which will return true if the movie can be returned and false if it doesn’t exist or is not borrowed, and a void display() which will print out all the movies in the inventory. Once a Movie has been borrowed it will be added to the BorrowedMovies vector.

Store
The store class is the class that has access to both the CustomerCollection as well as the Inventory. The store class will contain all the parsing methods such as loadMovieFile, loadCustomerFile, and the loadCommandsFile.

HashTable
The HashTable class is responsible for adding the right genre Movie to the Inventory. The genre will either be a Comedy(‘F’), Drama(‘D’), or a Classic(‘C’).

Class Interactions

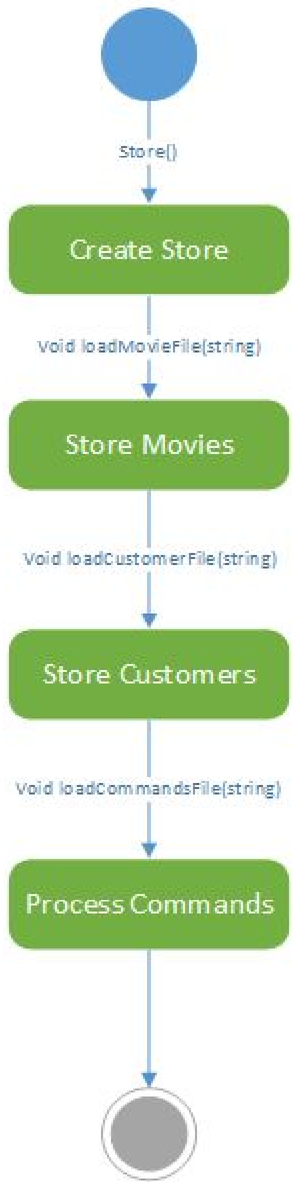

Store()
Create Store
Void loadMovieFile(string)
Store Movies
Void loadCustomerFile(string)
Store Customers
Void loadCommandsFile(string)
Process Commands

Figure 1: Main Data Flow Diagram

Figure 1 covers the data flow of the main.cpp file. It begins by creating a store that will be used to hold all the information about the movies, customers, and commands. After creating a store it will load the movie file and store the information inside of it. Similarly, it will do the same for the customer and command file.

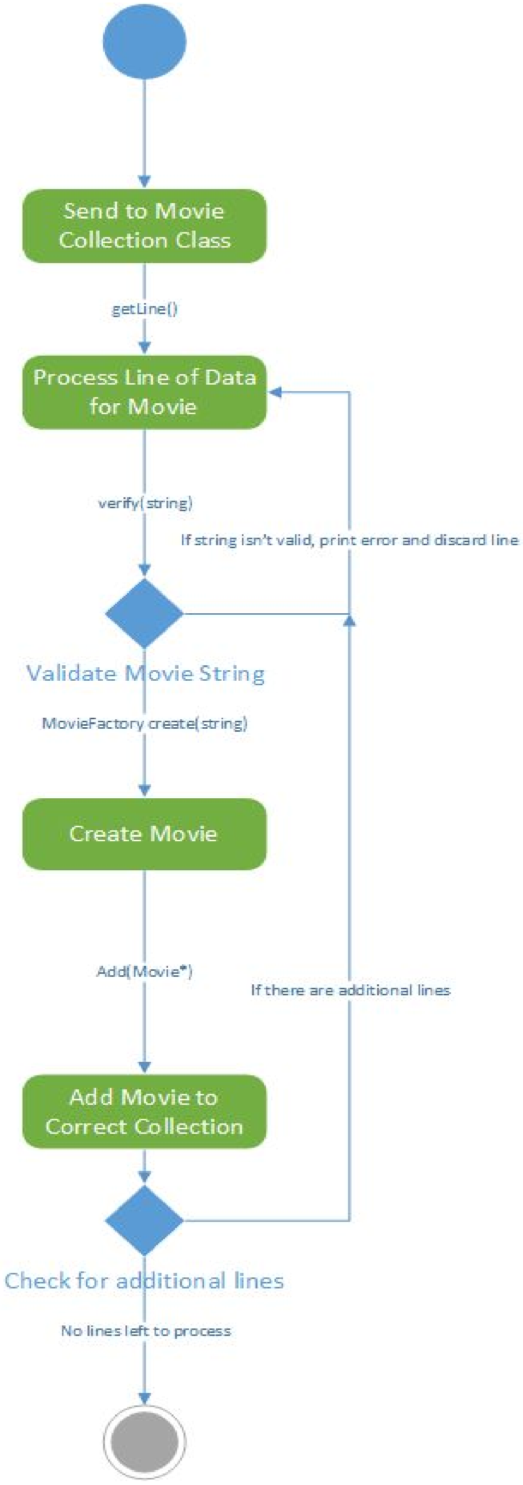

Send to Movie Collection Class
getLine()
Process Line of Data for Movie
verify(string)
If string isn't valid, print error and discard line
Validate Movie String
MovieFactory create(string)
Create Movie
Add(Movie*)
If there are additional lines
Add Movie to Correct Collection
Check for additional lines
No lines left to process

Figure 2: Movie File Processing

Figure 2 shows the data flow when processing the information inside the movie.txt file. It will go through each line and check if each line is valid and discard the line if it is not valid. With each line that is valid it will be added into the movie collection. It will repeat this until there are no lines left to process.

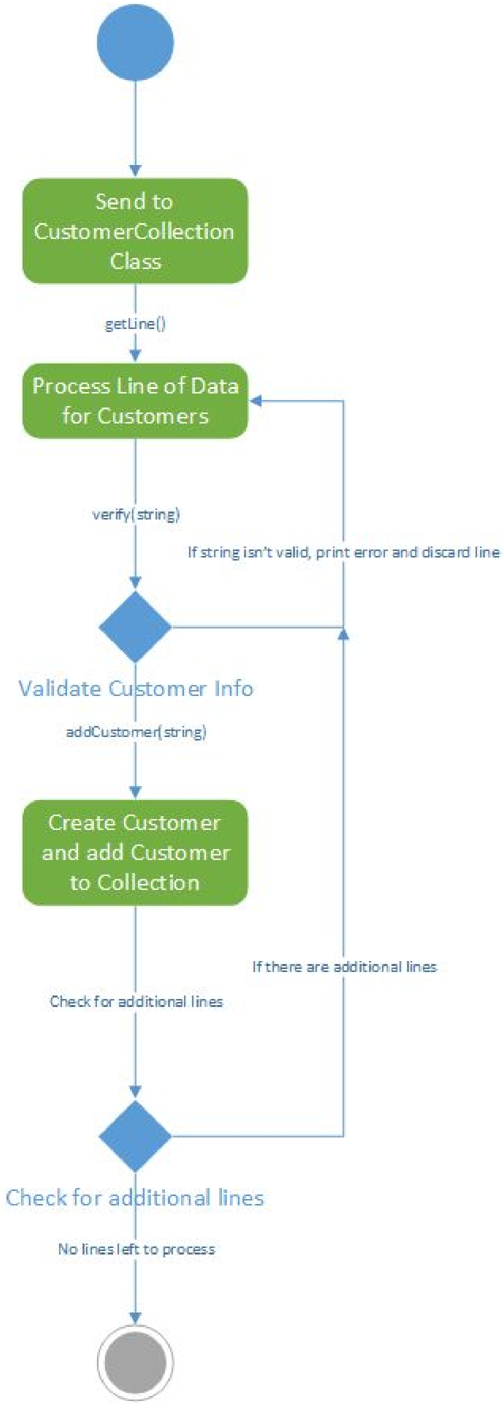

Send to CustomerCollection Class
getLine()
Process Line of Data for Customers
verify(string)
If string isn't valid, print error and discard line
Validate Customer Info
addCustomer(string)
Create Customer and add Customer to Collection
If there are additional lines
Check for additional lines
Check for additional lines
No lines left to process

Figure 3: Customer File Processing

Figure 3 shows the data flow when processing the information inside of the customer.txt file. It is quite similar to processing information inside the movie.txt file. It will begin validating each line and discard each incorrect line. It will add customers into customer collection and repeat this until there are no lines left to process.

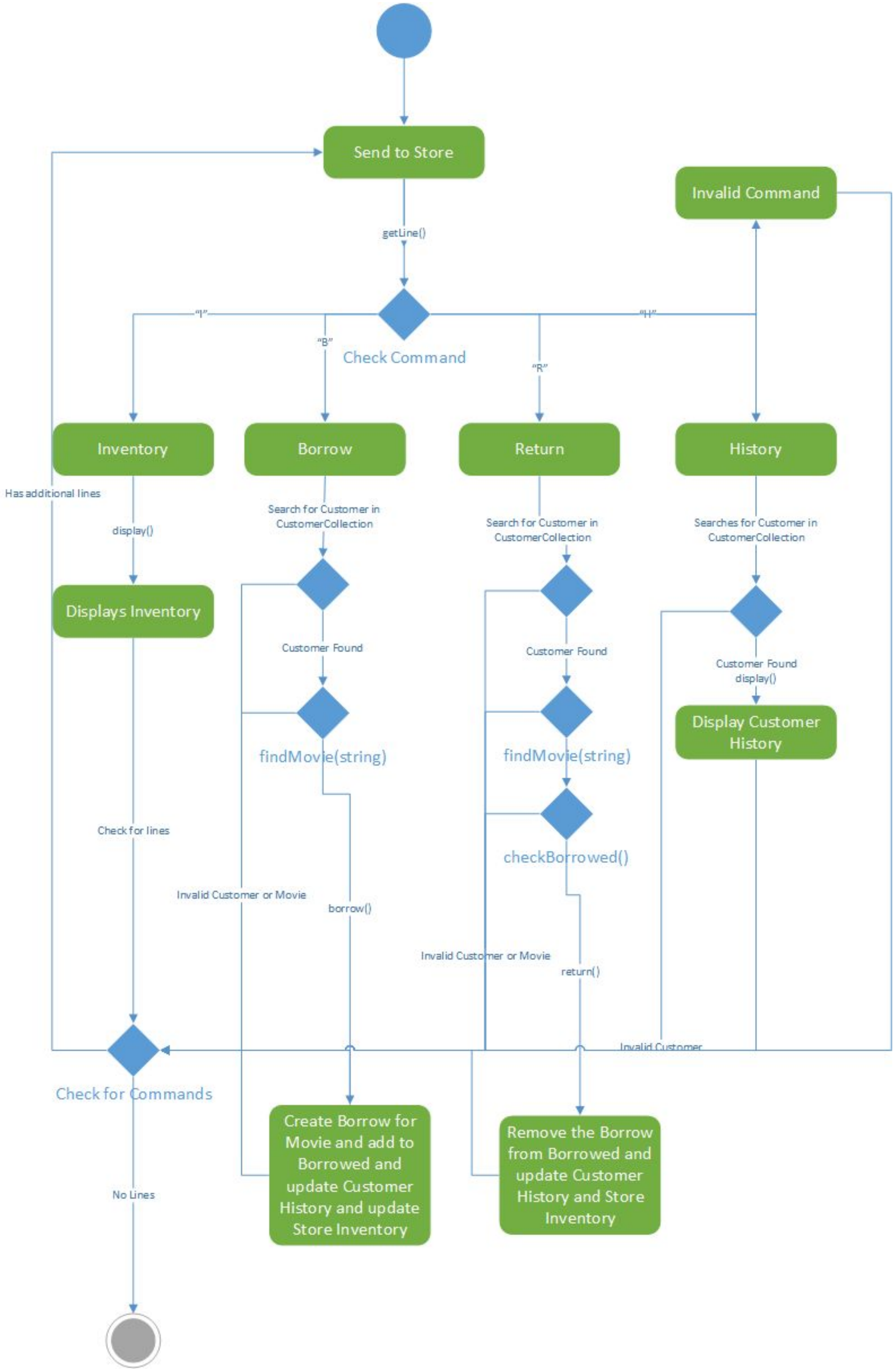

Send to Store
getLine()
Invalid Command
“I”
“B”
Check Command
“H”
“R”
Inventory
Borrow
Return
History
Has additional lines
display()
Search for Customer in CustomerCollection
Search for Customer in CustomerCollection
Searches for Customer in CustomerCollection
Displays Inventory
Customer Found
Customer Found
Customer Found
display()
Display Customer History
findMovie(string)
findMovie(string)
checkBorrowed()
Check for lines
Invalid Customer or Movie
borrow()
Invalid Customer or Movie
return()
Invalid Customer
Check for Commands
Create Borrow for Movie and add to Borrowed and update Customer History and update Store Inventory
Remove the Borrow from Borrowed and update Customer History and Store Inventory
No Lines

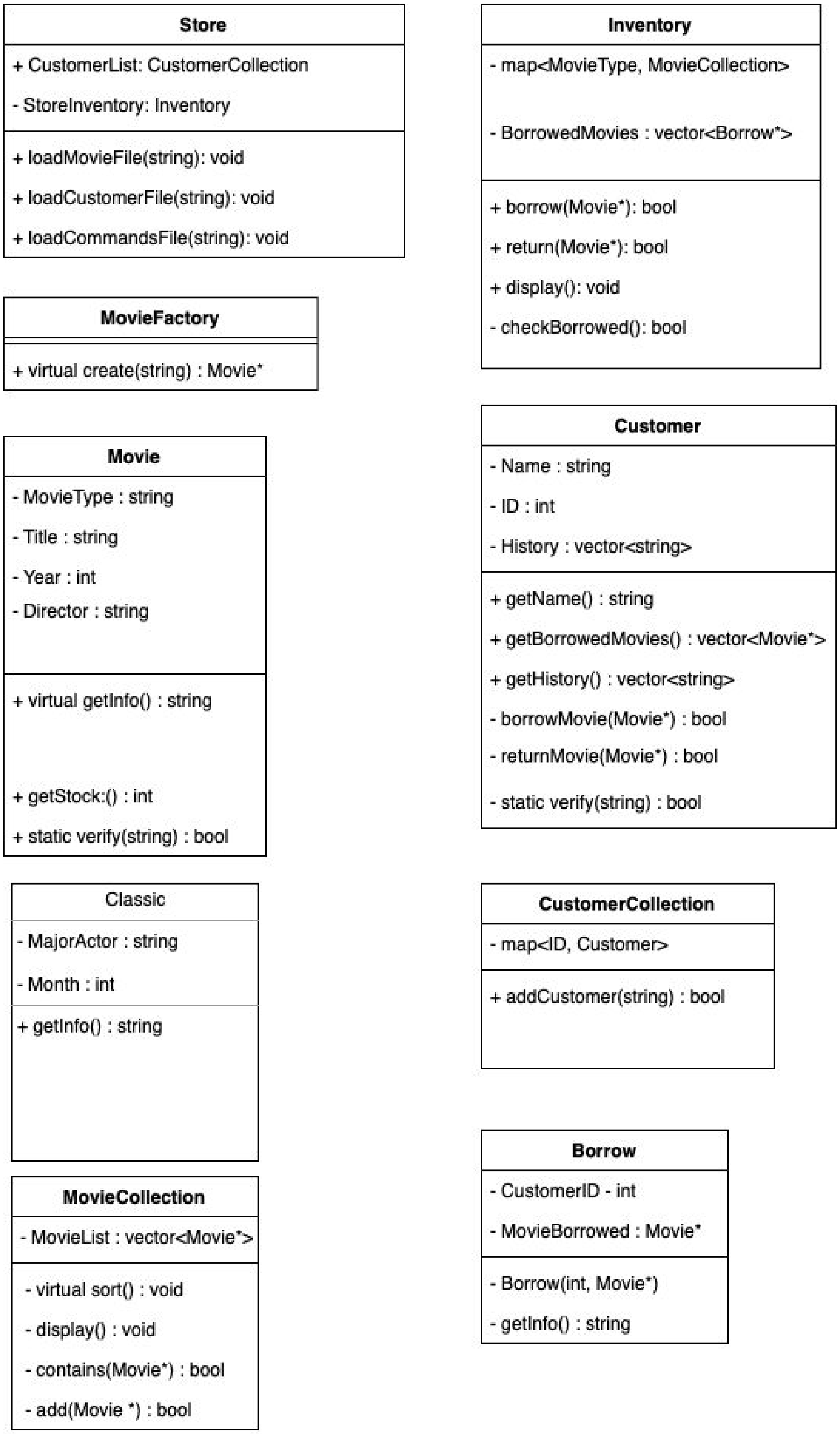
Figure 4: Commands File Processing
The commands file processing diagram illustrates the actions which occur when the store receives a text file of commands to process. The program will read through each line of the text file to check for valid commands and will end when there are no longer any commands to process. If the command is invalid, then it checks for new lines to process.

1. If the command found is inventory, the store calls the display() function for the inventory and checks for more commands.
2. If the command found is to borrow, the store searches the CustomerCollection for the customer ID provided and checks for whether it has the movie available to borrow in the Inventory. If these checks fail, then it will discard the line and check for more commands. If the customer and movie exists in the inventory, then the movie will be added to the customer's history, the borrow will be created and the store inventory and borrowed list will be updated. Then it will check for additional commands.
3. If the command found is to return, the store searches the CustomerCollection for the customer ID provided and checks whether that movie has been borrowed from the inventory. If these checks fail, then it will discard the line and check for additional commands. If the customer exists and the movie is borrowed, then the customer history will be updated, and the store inventory and borrowed list will be updated. Then it will check for additional commands.
4. If the command found is for the history, the store will look for the customer and display the history of the customer, if found. If the customer doesn't exist or after their history has been displayed, the program will check for additional commands.

Main.cpp Description

Create a store to hold customers and movie inventory. Load movies.txt file to fill inventory with movies. Load customers.txt file to fill customer collection with customers. Load commands.txt file to process commands.

.h Files



Store

+ CustomerList: CustomerCollection
- StoreInventory: Inventory

+ loadMovieFile(string): void
+ loadCustomerFile(string): void
+ loadCommandsFile(string): void

MovieFactory

+ virtual create(string) : Movie*

Movie

- MovieType : string
- Title : string
- Year : int
- Director : string

+ virtual getInfo() : string
+ getStock:() : int
+ static verify(string) : bool

Classic

- MajorActor : string
- Month : int

+ getInfo() : string

MovieCollection

- MovieList : vector<Movie*>

- virtual sort() : void
- display() : void
- contains(Movie*) : bool
- add(Movie *) : bool

Inventory

- map<MovieType, MovieCollection>
- BorrowedMovies : vector<Borrow*>

+ borrow(Movie*): bool
+ return(Movie*): bool
+ display(): void
- checkBorrowed(): bool

Customer

- Name : string
- ID : int
- History : vector<string>

+ getName() : string
+ getBorrowedMovies() : vector<Movie*>
+ getHistory() : vector<string>
- borrowMovie(Movie*) : bool
- returnMovie(Movie*) : bool
- static verify(string) : bool

CustomerCollection

- map<ID, Customer>

+ addCustomer(string) : bool

Borrow

- CustomerID - int
- MovieBorrowed : Movie*

- Borrow(int, Movie*)
- getInfo() : string

```
#include <string>

using namespace std;

class Movie {
public:
   /***
    * Gets the information for the movie
    *
    * @return string containing
    * information about the movie
    */
   virtual string getInfo() const = 0;

   /***
    * Sets the stock of the movie
    *
    * @param In
    * The integer to set the stock to.
    * Has to be a positive number.
    */
   void setStock(int In);

   /***
    * Gets the number of movies available
    *
    * @return integer that is the stock of the movie
    */
   int getStock() const;
private:
   string MovieType;
   string Title;
   int Year;
   string Director;
};
```

```
#include "movie.h"

class Classic : public Movie {
public:
   string getInfo() const;

private:
   string MajorActor;
   int ReleaseMonth;
};
```

```
#include <vector>
#include "movie.h"

class MovieCollection {
public:
   virtual void sort() = 0;
   void display() const;
   bool contains(Movie *In) const;
   bool add(Movie *In);

private:
   vector<Movie*> MovieList;
};
```

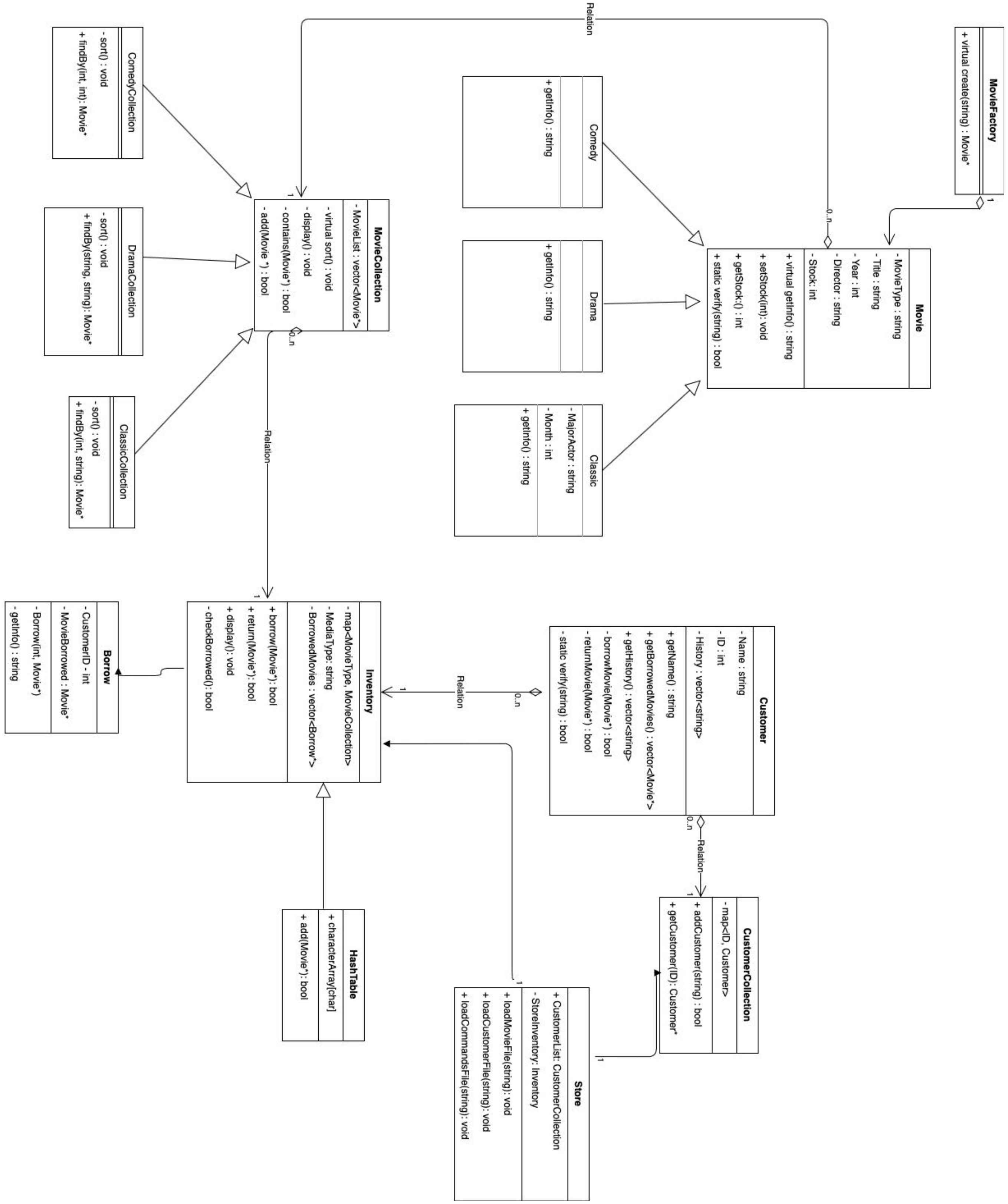
```
#import <map>
#import "movieCollection.h"
#import "movie.h"

class Inventory {
public:
   bool borrowMovie(Movie *In);
   bool returnMovie(Movie *In);
   void display() const;
   bool checkBorrowed();
private:
   map<string, MovieCollection> StoreInventory;
};
```

```
#include "movie.h"
#include <vector>

class Customer {
public:
   string getName() const;
   vector<Movie*> getBorrowedMovies() const;
   vector<string> getHistory() const;
   bool borrowMovie(Movie *ToBorrow);
   bool returnMovie(Movie *ToReturn);

private:
   string Name;
   int ID;
   vector<Movie*> BorrowedMovies;
   vector<string> History;
};
```

```
#include "customer.h"
#include <map>

class CustomerCollection {
public:
   bool addCustomer(string);
   Customer* getCustomer(int ID) const;
private:
   map<int, Customer> Customers;
};
```

```
#include "customerCollection.h"
#include "inventory.h"

class Store {
public:
   bool loadMovieFile(string FileName);
   bool loadCustomerFile(string FileName);
   bool loadCommandsFile(string FileName);

private:
   CustomerCollection CustomerList;
   Inventory StoreInventory;
};
```

UML Diagram


MovieFactory
+ virtual create(string) : Movie*
1
Movie
- MovieType : string
- Title : string
- Year : int
- Director : string
- Stock: int
+ virtual getInfo() : string
+ setStock(int): void
+ getStock:() : int
+ static verify(string) : bool
0...n
Relation
Comedy
+ getInfo() : string
Drama
+ getInfo() : string
Classic
- MajorActor : string
- Month : int
+ getInfo() : string
ComedyCollection
- sort() : void
+ findBy(int, int): Movie*
DramaCollection
- sort() : void
+ findBy(string, string): Movie*
ClassicCollection
- sort() : void
+ findBy(int, string): Movie*
1
MovieCollection
- MovieList : vector<Movie*>
- virtual sort() : void
- display() : void
- contains(Movie*) : bool
- add(Movie *) : bool
0...n
Relation
1
Inventory
- map<MovieType, MovieCollection>
- MediaType: string
- BorrowedMovies : vector<Borrow*>
+ borrow(Movie*): bool
+ return(Movie*): bool
+ display(): void
- checkBorrowed(): bool
Borrow
- CustomerID - int
- MovieBorrowed : Movie*
- Borrow(int, Movie*)
- getInfo() : string
1
Relation
0...n
Customer
- Name : string
- ID : int
- History : vector<string>
+ getName() : string
+ getBorrowedMovies() : vector<Movie*>
+ getHistory() : vector<string>
- borrowMovie(Movie*) : bool
- returnMovie(Movie*) : bool
- static verify(string) : bool
0...n
Relation
1
CustomerCollection
- map<ID, Customer>
+ addCustomer(string) : bool
+ getCustomer(ID): Customer*
HashTable
+ characterArray[char]
+ add(Movie*): bool
1
1
Store
+ CustomerList: CustomerCollection
- StoreInventory: Inventory
+ loadMovieFile(string): void
+ loadCustomerFile(string): void
+ loadCommandsFile(string): void