Assignment 4 Design

Jeremy Tandjung, Krystle Levin, Ethan Thomas

TABLE OF CONTENTS

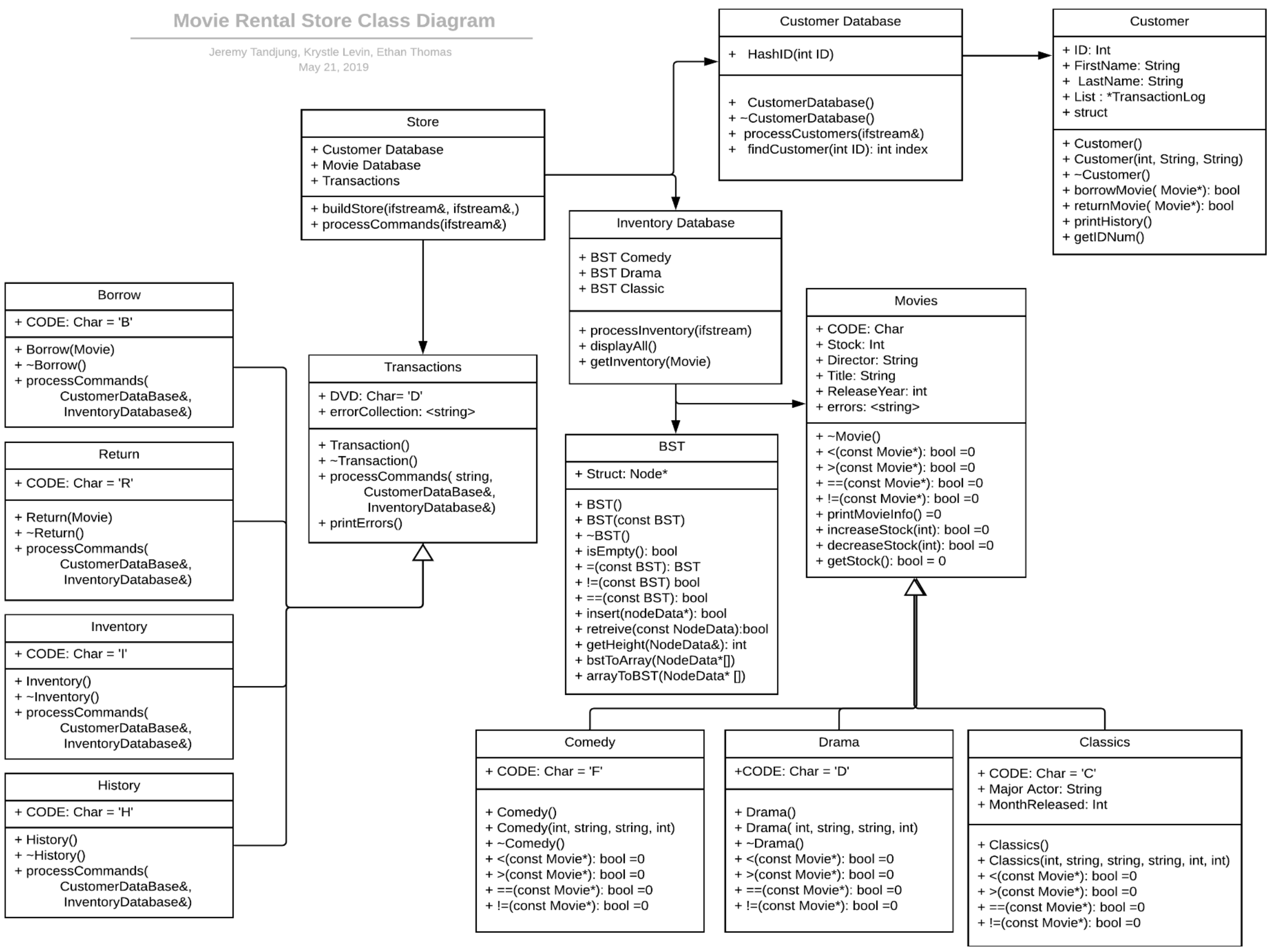
1 PROJECT OVERVIEW

- This is a design for a program that will automate an inventory tracking system. This design was built around the idea of it being used for a movie rental company that only rents out DVDS, but the code can be modified for other items and transactions as well. This program will allow the user to track every item that is borrowed and returned. It will also allow the user to see customer history information.
- The design has a Store class which includes a InventoryDatabase, Transactions, and a CustomerDatabase class
- InventoryDatabase is a class that contains a collection of items (in this case DVDs) and stores them in binary search trees for quick look up. For this particular case, each genre will have its own binary search tree. They will be sorted by their respected sorting criteria (e.g. Comedy will be sorted by title and then year released). InventoryDatabase allows the user to add new items,

delete items, and update stock when an item is checked out or returned. InventoryDatabase can also process a formatted text file of commands.

- Movie is a pure virtual base class for Comedy, Drama, and Classics. It will use a class called MovieFactory that will create a new movie using the factory method.
- Sorting for each sub class:
    - Comedy is sorted by Title, then Year it released
    - Drama is sorted by Director, then Title
    - Classics is sorted by Release date, then Major actor

- Transactions is a class that will process one and/or many transactions. It will use a TransactionsFactory that will create a new transaction using the factory method. Transactions will call on InventoryDatabase to update stock and CustomerDatabase to update individual customer accounts.
    - Transactions is a base class for Borrow, Return, Inventory, and History
- CustomerDatabase is a collection of Customers using a hash table. Customers are identified by their unique 4-digit number. CustomerDatabase is responsible for maintaining customer accounts. It can add or delete customers.


Movie Rental Store Class Diagram
Jeremy Tandjung, Krystle Levin, Ethan Thomas
May 21, 2019
Customer Database
+ HashID(int ID)
+ CustomerDatabase()
+ ~CustomerDatabase()
+ processCustomers(ifstream&)
+ findCustomer(int ID): int index
Customer
+ ID: Int
+ FirstName: String
+ LastName: String
+ List : *TransactionLog
+ struct
+ Customer()
+ Customer(int, String, String)
+ ~Customer()
+ borrowMovie(Movie*): bool
+ returnMovie(Movie*): bool
+ printHistory()
+ getIDNum()
Store
+ Customer Database
+ Movie Database
+ Transactions
+ buildStore(ifstream&, ifstream&,)
+ processCommands(ifstream&)
Inventory Database
+ BST Comedy
+ BST Drama
+ BST Classic
+ processInventory(ifstream)
+ displayAll()
+ getInventory(Movie)
Borrow
+ CODE: Char = 'B'
+ Borrow(Movie)
+ ~Borrow()
+ processCommands(CustomerDataBase&, InventoryDatabase&)
Return
+ CODE: Char = 'R'
+ Return(Movie)
+ ~Return()
+ processCommands(CustomerDataBase&, InventoryDatabase&)
Inventory
+ CODE: Char = 'I'
+ Inventory()
+ ~Inventory()
+ processCommands(CustomerDataBase&, InventoryDatabase&)
History
+ CODE: Char = 'H'
+ History()
+ ~History()
+ processCommands(CustomerDataBase&, InventoryDatabase&)
Transactions
+ DVD: Char= 'D'
+ errorCollection: <string>
+ Transaction()
+ ~Transaction()
+ processCommands(string, CustomerDataBase&, InventoryDatabase&)
+ printErrors()
BST
+ Struct: Node*
+ BST()
+ BST(const BST)
+ ~BST()
+ isEmpty(): bool
+ =(const BST): BST
+ !=(const BST) bool
+ ==(const BST): bool
+ insert(nodeData*): bool
+ retreive(const NodeData):bool
+ getHeight(NodeData&): int
+ bstToArray(NodeData*[])
+ arrayToBST(NodeData* [])
Movies
+ CODE: Char
+ Stock: Int
+ Director: String
+ Title: String
+ ReleaseYear: int
+ errors: <string>
+ ~Movie()
+ <(const Movie*): bool =0
+ >(const Movie*): bool =0
+ ==(const Movie*): bool =0
+ !=(const Movie*): bool =0
+ printMovieInfo() =0
+ increaseStock(int): bool =0
+ decreaseStock(int): bool =0
+ getStock(): bool = 0
Comedy
+ CODE: Char = 'F'
+ Comedy()
+ Comedy(int, string, string, int)
+ ~Comedy()
+ <(const Movie*): bool =0
+ >(const Movie*): bool =0
+ ==(const Movie*): bool =0
+ !=(const Movie*): bool =0
Drama
+CODE: Char = 'D'
+ Drama()
+ Drama(int, string, string, int)
+ ~Drama()
+ <(const Movie*): bool =0
+ >(const Movie*): bool =0
+ ==(const Movie*): bool =0
+ !=(const Movie*): bool =0
Classics
+ CODE: Char = 'C'
+ Major Actor: String
+ MonthReleased: Int
+ Classics()
+ Classics(int, string, string, string, int, int)
+ <(const Movie*): bool =0
+ >(const Movie*): bool =0
+ ==(const Movie*): bool =0
+ !=(const Movie*): bool =0

- int main()
```
{
        Store firstStore();
        ifstream infile1("data4commands.txt");
        ifstream infile2("data4customers.txt");
        ifstream infile3("data4movies.txt");
        firstStore.buildstore(infile2, infile3);
        firstStore.processCommands(infile1);
}
```

2 Class Description Overview

2.1 Store

The Store class represents the entire store. It contains a CustomerDatabase, InventoryDatabase, and Transactions. Allows the user to build inventory list, customer list, and process transaction commands.

```
class Store
{

public:
              // builds store by creating movie database and customer database objects
       void buildStore(ifstream& customerList, ifstream& inventoryList);

       void commandsReader(ifstream& commandsList);            // process command lines

private:

       CustomerDatabase allCustomers;                      // Customer Database object
       InventoryDatabase allInventory;                    // Inventory Database object
       Transactions allTransactions;                      // Transactions object
};
```

`-void buildStore(ifstream& customerList, ifstream& inventoryList)`

Purpose: This function builds the hypothetical "Store" by reading in a input file that contains information about the customers and the inventory.

Parameter(s):

-`ifstream& customerList`: The ifstream object that takes in the text file containing the customer list

-`ifstream& inventoryList`: The ifstream object that takes in the text file containing the inventory

Return: None

```
-void commandsReader(ifstream& commandsList);
```

Purpose: This function is used to read commands from a text file that contains the command list

Parameter(s):

- `ifstream& commandsList`: The ifstream object that contains the list of command the user inputs

Return: None

2.2 INVENTORYDATABASE

The InventoryDatabase class is a collection of items (in this case DVDs) that are stored in binary search trees. They are sorted by their respected sorting criteria.

```
class InventoryDatabase
{
public:
	void processInventory(ifstream& inventoryFile);		// process inventory
	Movie* getMovie(string movieInfo, char code);		// retrieve movie from BST
	void displayAll();	//Displays all genres of movies in their respective order
private:

	vector<string> errors;			// error collector
	BST drama;				// BST for drama movies
	BST classics;				// BST for classics movies
	BST comedy;				// BST for comedy movies
};
```

```
-void processInventory(ifstream& inventoryFile);
```

Purpose: This function builds up the hypothetical ‘Inventory’ by reading an input file containing information for the inventory.

Parameter(s):

-`ifstream& inventoryFile`: The ifstream object that takes in the text file containing the inventory list.

Return: None

`-Movie* getMovie(string movieInfo, char code);`

Purpose: This function is a getter method that gets a Movie object within this InventoryDatabase object

Parameter(s):

`-string movieInfo`: string that is passed in that contains the movie to find

`-char code`: the code that identifies what genre the movie is

Return: The targeted movie.

`-void displayAll();`

Purpose: This function prints out all information of all movies currently in this inventory in a readable format.

Parameter(s): None

Return: None

2.2.1 Movie

Pure virtual function. This class is used as a base class for Comedy, Drama, Classics

```
class Movie
{
public:

	virtual ~Movie() {};                  // destructor

	virtual bool operator==(const Movie* rhs) const =0;   // == operator
	virtual bool operator!=(const Movie* rhs) const =0; // != operator
	virtual bool operator>(const Movie* rhs) const =0; // greater than operator
	virtual bool operator<(const Movie* rhs) const =0;  // less than operator

	virtual void printMovieInfo() const =0;      //  prints movies information

	virtual bool increaseStock(const int& amount) =0;    // increase stock
	virtual bool dicreaseStock(const int& amount) =0;    // decrease stock
	virtual int getStock();                      // return current stock

protected:

	explicit Movie();                            // constructor


	char type;                            // holds a movie type
	int stock;                            // holds movie stock
	string director;                      // holds a movie director
```

```
        string title;                       // holds a movie title
        int year;                           // holds a movie year

};
```

`-virtual ~Movie()`

Purpose: Destructor for base Movie class

Parameter(s): None

Return: None

`-virtual bool operator==(const Movie* rhs) const =0;`

Purpose: This function overwrites the == operator

Parameter(s):

-`const Movie* rhs`: The right hand side Movie object we are comparing with

Return: Returns true if every attribute of this Movie object matches the rhs Movie obj, false otherwise

`-virtual bool operator!=(const Movie* rhs) const =0;`

Purpose: Overwrites the != operator

Parameter(s):

-`const Movie* rhs`: The right hand side Movie object we are comparing with

Return: Returns the negation of the == operator result

`-virtual bool operator>(const Movie* rhs) const =0;`

Purpose: This function overwrites the > operator

Parameter(s):

-`const Movie* rhs`: The right hand side Movie object we are comparing with

Return: Returns true if *this is greater than rhs (based on sorting criteria)

-`virtual bool operator<(const Movie* rhs) const =0;`

Purpose: This function overwrites the < operator

Parameter:

-`const Movie* rhs`: The right hand side Movie object we are comparing with

Return: Returns true if *this is less than rhs (based on sorting criteria)

-`virtual void printMovieInfo() const =0;`

Purpose: This function prints out the string representation of the Movie object in a readable format.

Parameter(s): None

Return: None

-`virtual bool increaseStock(const int& amount) =0;`

Purpose: This function is used to increase the stock of this Movie object

Parameter(s):

-`const int& amount`: The amount which the user wants to increase the stock by

Return: Returns true if the increment was successful, returns false otherwise i.e. bad input

-`virtual bool decreaseStock(const int& amount) =0;`

Purpose: This function is used to decrease the stock of this Movie object

Parameter(s):

-`const int& amount`: The amount which the user wants to decrease the stock by

Return: Returns true if the decrement was successful, returns false otherwise i.e. bad input

-`virtual int getStock();`

Purpose: Getter method to get the current stock of this Movie object

Parameter(s): None

Return: The integer representation of the stock of this Movie

2.2.1.1 Comedy

The Comedy class represents a comedy Movie

```
class Comedy : public Movie
{
public:

       Comedy(int stock, string director, string title, int year);  // constructor
       ~Comedy();                                      // destructor

       bool operator==(const Movie* rhs) const =0;    // == operator
       bool operator!=(const Movie* rhs) const = 0; // != operator
       bool operator>(const Movie* rhs) const =0; // greater than operator
       bool operator<(const Movie* rhs) const =0;  // less than operator

       void printMovieInfo() const;              // return string of movie info

       static const char CODE = 'F';                    // static identifier for the class
};
```

-`Comedy(int stock, string director, string title, int year);`

Purpose: Constructor for Comedy class

Parameter(s):

-`int stock`: The stock of this Comedy object

-`string director`: The director of this Comedy movie

-`string title`: The comedy movie's title

-`int year`: The year this comedy movie was released

Return: Comedy object

-`bool operator==(const Movie* rhs) const =0;`

Purpose: This function overwrites the == operator

Parameter(s):

-`const Movie* rhs`: The right hand side Movie object we are comparing with

Return: Returns true if every attribute of this Movie object matches the rhs Movie obj, false otherwise

`-bool operator!=(const Movie* rhs) const =0;`

Purpose: Overwrites the != operator

Parameter(s):

`-const Movie* rhs`: The right hand side Movie object we are comparing with

Return: Returns the negation of the == operator result

`-bool operator>(const Movie* rhs) const =0;`

Purpose: This function overwrites the > operator

Parameter(s):

`-const Movie* rhs`: The right hand side Movie object we are comparing with

Return: Returns true if *this is greater than rhs (based on sorting criteria)

`-bool operator<(const Movie* rhs) const =0;`

Purpose: This function overwrites the < operator

Parameter(s):

`-const Movie* rhs`: The right hand side Movie object we are comparing with

Return: Returns true if *this is less than rhs (based on sorting criteria)

`-void printMovieInfo() const =0;`

Purpose: This function prints out the string representation of the Movie object in a readable format.

Parameter(s): None

Return: None

2.2.1.2 *Drama*

The Drama class represents a drama Movie

```
 class Drama : public Movie
{
public:

       Drama(int stock, string director, string title, int year);  // constructor
       ~Drama();                                         // destructor

       bool operator==(const Movie* rhs) const =0;    // == operator
       bool operator!=(const Movie* rhs) const = 0; // != operator
       bool operator>(const Movie* rhs) const =0; // greater than operator
       bool operator<(const Movie* rhs) const =0;  // less than operator

       string printMovieInfo() const;              // return string of movie  info

       static const char CODE = 'D';                   // static identifier for the class
};
```

-`Drama(int stock, string director, string title, int year);`

Purpose: Constructor for the Drama class that takes in the

Parameter(s):

-`int stock`: The stock of this Drama object

-`string director`: The director of this Drama object

-`string title`: The drama movie's title

-`int year`: The year this drama movie was released

Return: Drama object

-`bool operator==(const Movie* rhs) const =0;`

Purpose: This function overwrites the == operator

Parameter(s):

-`const Movie* rhs`: The right hand side Movie object we are comparing with

Return: Returns true if every attribute of this Movie object matches the rhs Movie obj, false otherwise

`-bool operator!=(const Movie* rhs) const =0;`

Purpose: Overwrites the != operator

Parameter(s):

`-const Movie* rhs`: The right hand side Movie object we are comparing with

Return: Returns the negation of the == operator result

`-bool operator>(const Movie* rhs) const =0;`

Purpose: This function overwrites the > operator

Parameter(s):

`-const Movie* rhs`: The right hand side Movie object we are comparing with

Return: Returns true if *this is greater than rhs (based on sorting criteria)

`-bool operator<(const Movie* rhs) const =0;`

Purpose: This function overwrites the < operator

Parameter(s):

`-const Movie* rhs`: The right hand side Movie object we are comparing with

Return: Returns true if *this is less than rhs (based on sorting criteria)

`-void printMovieInfo() const =0;`

Purpose: This function prints out the string representation of the Movie object in a readable format.

Parameter(s): None

Return: None

2.2.1.3 Classics

The Classics class represent a classic Movie

```
class Classics : public Movie
{
public:

	Classics(int stock, string director, string title, string majorActor, int month,
	int year);  // constructor
	~Classics();                                // destructor

	bool operator==(const Movie* rhs) const =0;   // equal comparison operator
	bool operator!=(const Movie* rhs) const = 0; // not equal comparison operator
	bool operator>(const Movie* rhs) const =0; // greater than operator
	bool operator<(const Movie* rh) const =0;  // less than operator

	string printMovieInfo() const;              // return string of full movie  info

	static const char CODE = 'C';                   // static identifier for the class
protected:
	int month;
	string majorActor;
};
```

`-Classics(int stock, string director, string title, string majorActor, int month,  int year);`

Purpose: Constructor for the Classic class

Parameter(s):

-`int stock`: The stock of this Classic object

-`string director`: The director of this Classic object

-`string title`: The Classic movie’s title

-`string majorActor`: The main star of this Classic movie

-`int month`: The month this Classic movie was released

-`int year`: The year this Classic movie was released

Return: Classic object

`-bool operator==(const Movie* rhs) const =0;`

Purpose: This function overwrites the == operator

Parameter(s):

-`const Movie* rhs`: The right hand side Movie object we are comparing with

Return: Returns true if every attribute of this Movie object matches the rhs Movie obj, false otherwise

`-bool operator!=(const Movie* rhs) const =0;`

Purpose: Overwrites the != operator

Parameter(s):

-`const Movie* rhs`: The right hand side Movie object we are comparing with

Return: Returns the negation of the == operator result

`-bool operator>(const Movie* rhs) const =0;`

Purpose: This function overwrites the > operator

Parameter(s):

-`const Movie* rhs`: The right hand side Movie object we are comparing with

Return: Returns true if *this is greater than rhs (based on sorting criteria)

`-bool operator<(const Movie* rhs) const =0;`

Purpose: This function overwrites the < operator

Parameter(s):

-`const Movie* rhs`: The right hand side Movie object we are comparing with

Return: Returns true if *this is less than rhs (based on sorting criteria)

```
-void printMovieInfo() const =0;
```

Purpose: This function prints out the string representation of the Movie object in a readable format.

Parameter(s): None

Return: None

2.3 TRANSACTIONS

The Transactions class processes one and/or many transactions. Calls on InventoryDatabase and CustomerDatabase to update their respected data (e.g. InventoryDatabase increases or decreases stock from customer return or borrow) This class is used as a base class for Borrow, Return, Inventory, and History

```
class Transaction
{
public:
       Transaction();                // constructor
       virtual ~Transaction() {};    // destructor

       // processes commands
       virtual void processCommands(CustomerDatabase&, InventoryDatabase&);
       void printErrors();                                 // prints error massages
       static const char DVD = 'D';                    // shared by all transactions
protected:

       vector<string> errorCollection;   //vector that holds all type of errors during
       reading of the commands
};
```

```
-Transaction();
```

Purpose: Constructor for Transaction class

Parameter(s): None

Return: Transaction object

```
-virtual ~Transaction()
```

Purpose: Destructor for Transaction class

Parameter(s): None

Return: None

2.3.1 Borrow

Represents a type of transaction

```
class Borrow : public Transaction
{
public:

	Borrow();			// constructor
	~Borrow();			// destructor
	static const char MyType = 'B';		// static identifier for the class
	virtual void processCommands(CustomerManager&, InventoryDatabase&); // process
Transaction
protected:

	void borrowMovie(Customer*, Movie*);	// borrowed movie for customer

};
```

`-Borrow();`

Purpose: Constructor for Borrow class

Parameter: None

Return: Borrow object

`- ~Borrow()`

Purpose: Destructor for Borrow class

Parameter(s): None

Return: None

`-virtual void processCommands(CustomerManager&, InventoryDatabase&);`

Purpose: This function processes the Inventory and Customer database based on the action code

Parameter(s):

-`CustomerManager`&: Customer database

-`InventoryDatabase`&: The inventory database that stores the movies.

Return: None

2.3.2 Return

Represents a type of transaction

```
class Return : public Transaction
{
public:

	Return();			// constructor
	~Return();			// destructor
	static const char MyType = 'R';		// static identifier for the class
	virtual void processCommands(CustomerManager&, InventoryDatabase&); // process
Transaction
protected:

	void returnMovie(Customer*, Movie*);   // borrowed movie for customer

};
```

`-Return();`

Purpose: Constructor for Return class

Parameter(s): None

Return: Return object

`- ~Return ()`

Purpose: Destructor for Return class

Parameter(s): None

Return: None

`-virtual void processCommands(CustomerManager&, InventoryDatabase&);`

Purpose: This function processes the Inventory and Customer database based on the action code

Parameter(s):

-`CustomerManager`&: Customer database

-`InventoryDatabase`&: The inventory database that stores the movies.

Return: None

2.3.3 Inventory

Represents a type of transaction

```
class Inventory : public Transaction
{
public:

	Inventory();			// constructor
	~Inventory();			// destructor
	static const char MyType = 'I';		// static identifier for the class
	virtual void processCommands(CustomerManager&, InventoryDatabase&); // process
Transaction
};
```

`-Inventory();`

Purpose: Constructor for Inventory class

Parameter(s): None

Return: Inventory object

`- ~Inventory()`

Purpose: Destructor for Inventory class

Parameter: None

Return: None

`-virtual void processCommands(CustomerManager&, InventoryDatabase&);`

Purpose: This function processes the Inventory and Customer database based on the action code

Parameter(s):

-`CustomerManager`&: Customer database

-`InventoryDatabase`&: The inventory database that stores the movies.

Return: None

2.3.4 History

Represents a type of transaction

```
class History : public Transaction
{
public:

	History();			// constructor
	~History();			// destructor
	static const char MyType = 'H';		// static identifier for the class
	virtual void processCommands(CustomerManager&, InventoryDatabase&); // process
Transaction
};
```

`-History();`

Purpose: Constructor for History class

Parameter(s): None

Return: History object

`- ~History()`

Purpose: Destructor for History class

Parameter(s): None

Return: None

`-virtual void processCommands(CustomerManager&, InventoryDatabase&);`

Purpose: This function processes the Inventory and Customer database based on the action code

Parameter(s):

-`CustomerManager&`: Customer database

-`InventoryDatabase&`: The inventory database that stores the movies.

Return: None

2.4 CustomerDatabase

The CustomerDatabase class maintains a collection of customers implemented by using a hash table. The class is responsible for maintaining each customer account and adding or deleting customers

```
int const ROWS = 101;
int const COLUMNS = 199;

class CustomerDatabase
{
public:

       CustomerDatabase();                     // constructor
       ~CustomerDatabase();                    // destructor
       void proccessCustomers(ifstream& customerFile);    // process file with customers

private:
       bool addCustomertoHashTable(Customer newCustomer);
       void retreiveCustomerFromHash( );

       struct HashTable{
              Customer theCustomer;                       // the customer
              History  customerHistory;     // The name of the Customer
       };

       HashTable tableOfCustomers[ROWS][COLUMNS];  // HashTable
};
```

`-CustomerDatabase();`

Purpose: Constructor for CustomerDatabase class

Parameter(s): None

Return: CustomerDatabase object

`- ~CustomerDatabase();`

Purpose: Destructor for CustomerDatabase class

Parameter(s): None

Return: None

`-void proccessCustomers(ifstream& customerFile)`

Purpose: This function builds the hypothetical HastTable that holds the customer database by reading information from the input file.

Parameter(s):

`-ifstream& customerFile`: The input file that contains all customer's information

Return: None

`-void CustomerDatabase::addCustomerToHashTable(Customer newCustomer)`

Purpose: This function adds a Customer object to the HashTable

Parameter(s):

`-Customer newCustomer`: The Customer object we want to add

Return: None

2.4.1 HashTable Pseudo Code

Function : add customer at Hash[i^2 + ID % ROWS][i^2 + ID % COLUMNS]

where i = number of tries

```
Void CustomerDatabase::addCustomerToHashTable(Customer newCustomer){
    If(theCustomer.getID() < 1 || theCustomer.getName()=="" or null) //invalid input
        Return false;
    Int iD=theCustomer.getID();
    Int i=0
    Bool done = false;
    While(!done){
        Int iSquared= i*i;
        Check HashTable[iSquared + ID % Rows][iSquared + ID % COLUMNS]
            If empty{  // doesn't exist yet
                set theCustomer to newCustomer
                customerHistory points to Null
                return true
            }
            Else{  // spot is full
                if (theCustomer == newCustomer)
                    return true; //same customer

            //at this point newCustomer has not been added to hash
                i++          //increment i by +1 and retry
            }
    }
}
```

2.4.2 Customer

Represents a single customer. Each customer has a unique 4-digit ID number. Each customer will have a transaction history that is stored in a linked list.

```
class Customer
{
public:

	Customer();                  // default constructor
	Customer(int id, string firstName, string lastName); // constructor
	~Customer();                    // destructor
	bool movieBorrow(Movie*);       // borrow movie
	bool movieReturn(Movie*);       // return movie

	void printHistory();            // print customer borrow and return history
	int getIDNum() const;           // return customer ID

private:

	bool returnMovie(Movie*);       // returns one or many movie(s) for customer (if
possible)
	bool borrowMovie(Movie*);       // borrows one or many movie(s) for customer (if
possible)
	int ID;                             // customer id
	string lastName;                // customer last name
	string firstName;               // customer first name


	// structure for customer transaction history
	struct  TransactionLog
	{
		TransactionLog* next;
		char transaction;    // transaction type (Borrow or Return)
		Movie* info;        // pointer to the movie
	};
	TransactionLog* head;

};
```

`Customer();`

Purpose: Default Constructor for Customer class

Parameter(s): None

Return: Customer object

`-Customer(int id, string firstName, string lastName)`

Purpose: Constructor for Customer class with parameters

Parameter(s):

- `int id`: the customer's id
- `string firstName`: customer's first name
- `string lastName`: customer's last name

Return: Customer object

`- ~Customer();`

Purpose: Destructor for Customer class

Parameter(s): None

Return: None

`-bool movieBorrow(Movie* m);`

Purpose: Simulates a customer borrowing a movie

Parameter(s):

- `Movie* m`: the movie the customer wants to borrow

Return: Returns true if customer successfully borrowed a movie, false otherwise

`-bool movieReturn(Movie* m)`

Purpose: Simulates a customer returning a movie

Parameter(s):

- `Movie* m`: the movie the customer wants to return

Return: Returns true if customer successfully returned a movie, false otherwise

`-void printHistory();`

Purpose: This function prints out the lending history of this Customer

Parameter(s): None

Return: None

```
-int getIDNum() const;
```

Purpose: This function is a getter method to get this Customer’s ID number

Parameter(s): None

Return: Integer representation of this Customer’s ID number