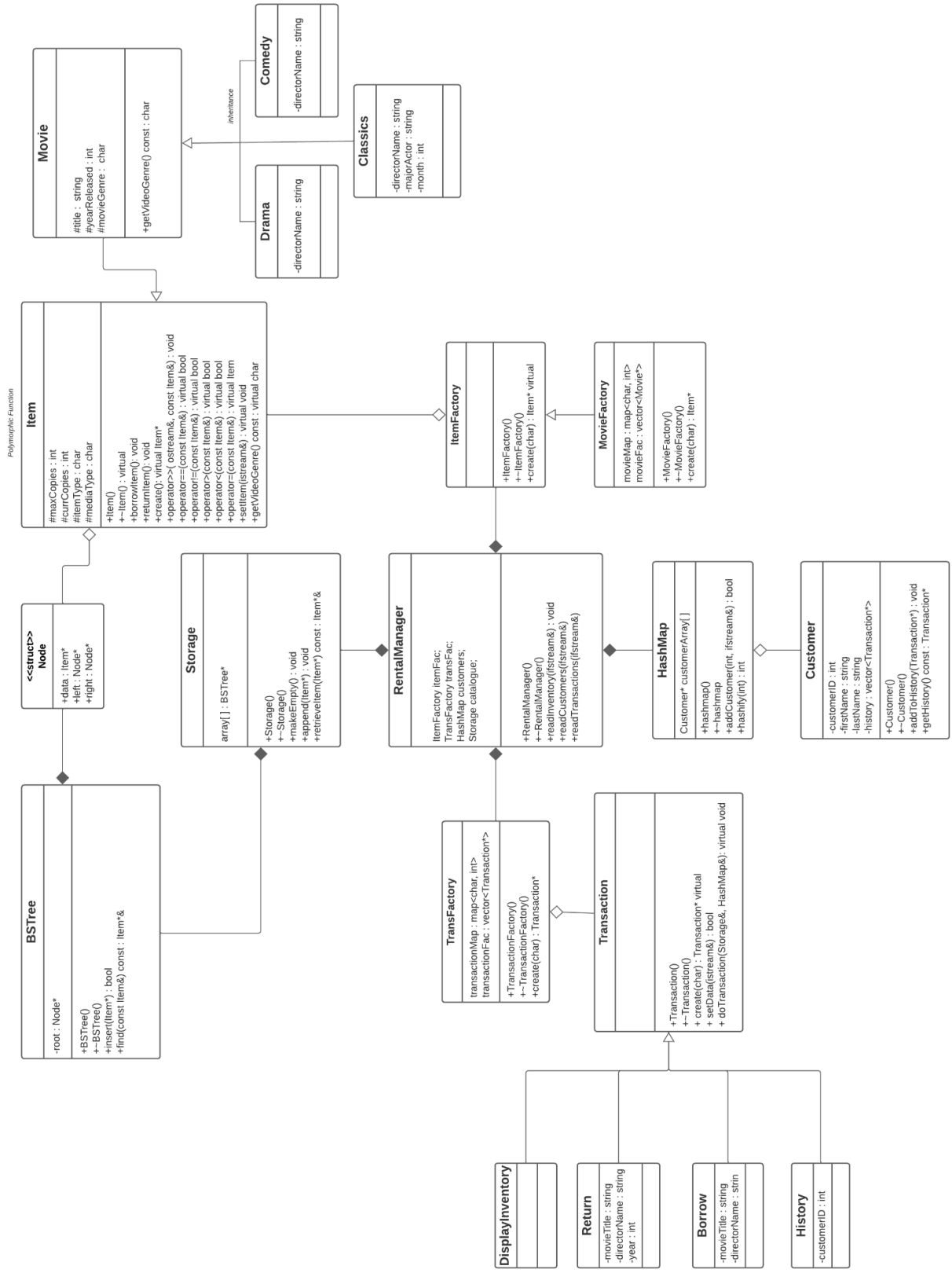
Movie Design Project

Written & Designed by Group 3
Amanda Todakonzie, Chanhee Park, Jessica Mironyuk, Tammy Le

Table of Contents

UML Class diagram:



Classes Overview

Our Movie Management Project design uses 18 classes, 12 of which are hierarchical classes and use a suite of polymorphic functions to carry out instantiating the inventory, instantiating the Customers and reading in commands for the system to carry out.

The * symbol indicates a hierarchical class and its children classes described directly after the parent class in the description of each class below.

- **RentalManager** will be reading in the text files for inventory, customers, and commands.
- **Storage** stores binary search trees of certain Items that are in the inventory.
- **BSTree** is a binary search tree data structure to store the items in the inventory.
- ***Item** Class as parent of **Movie** class, which is parent of children classes. **Classics**, **Drama**, **Comedy.** These classes will store the data for each movie.
- ***ItemFactory** as parent of **MovieFactory** class. ItemFactory will produce Items that can be stored in the BSTree Class. Having the BSTree class store Items allows for the program to be extended and have the Rental Management System store more than just Movies in its inventory. MovieFactory will produce Movie class objects when called upon to create().
- **Customer** class to hold customer data.
- **Hashmap** class to store Customer class objects, uses a hashtable and a hashing function to provide the index to store said Customer objects.
- ***Transaction** Class reads in the commands txt file and called on **TransactionFactory** to create children transaction classes (**Borrow** Class, **Return** Class, **History** Class and **DisplayInventory** Class). Each of these children classes will handle the command based on its class and carry out the transaction specified.
- **TransactionFactory** class to handle creating Transaction object.

Class Interactions/Program Flow

From main.cpp → creates an instance of RentalManager and ifstream objects needed to read in the files needed for the program.

Within RentalManager are the classes of Storage, Hashmap, ItemFactory and TransactionFactory which are used to initialize the inventory and customer data, and process commands to be performed and recorded in the system.

1. **RentalManager** calls on **readInventory**() to initialize a char variable to read in >> first char of data from the file containing data for Movie Rental inventory.

    1. Storage class object is called upon to read in data file, which takes the first char at the start of every new line of data.
    2. Char variable is passed off to a ItemFactory to create an Item corresponding to that char → create(char);
    3. Storage then takes Item object just created and inserts it into the Storage Inventory containing Binary Search Trees
    4. Within storage are multiple Binary Search Trees containing the different genres of Movies : Comedy, Drama and Classics

2. **RentalManager** calls on **readCustomers**() to initialize an integer variable to read in >> first number of data from the file containing data for Customers of the Movie Rental store.

    1. HashMap class object contains an array that is designated to hold Customer pointers.
    2. Hashmap is called upon to read in a data file, which takes in the first integer at the start of every new line of data as customerID.
    3. Integer variable is passed off to the HashMap to create a Customer object to store Customer data and call hashify() it into the Hashmap for storage.

3. **RentalManager** calls on **readCommands**() to initialize an integer variable to read in >> first number of data from the file containing data for Customers of the Movie Rental store.

    1. **readCommand**() method starts with initializing a char variable to read in the first char of every line of data in the file.
    2. This char variable is passed off to a TransactionFactory to create a Transaction object
    3. This Transaction object will perform a virtual doTransaction() function, and carry out the task according to its individual class's function definition.
    4. Transaction class is an abstract parent class for the following objects that will carry out the task designated to it.
    5. Children classes of Transaction are: Borrow, Return, DisplayInventory and History.

Use Cases

Borrow Pseudocode:

1. Error-checking to see Movie data is valid
2. Check to if Movie exists in the Inventory → findItem(title, year)
3. Error-checking to see if Customer ID is valid
4. Check to see if customer exists in the Hashmap → findCustomer(ID)
5. If Movie and Customer are found
    a. Check stock of the movie to see if copies are available
    b. If copies are available (currCopies > 0 == true) → call Borrow() on the movie Item
    c. Borrow() would decrement the currCopies variable within the Movie Object
6. Add Borrow object to customer's history vector → addToHistory(Transaction)

Return Pseudocode:

1. Error-checking to see Movie data is valid
2. Check to if Movie exists in the Inventory → findItem(title, year)
3. Error-checking to see if Customer ID is valid
4. Check to see if customer exists in the Hashmap → findCustomer(ID)
5. If Movie and Customer are found
    a. Check stock of the movie to see if space is available
    b. If space is available (currCopies < 5) → call Return() on the movie Item
    c. Return() would increment the currCopies variable within the Movie Object
6. Add Return object to customer's history vector → addToHistory(Transaction)

Main:

```
/**
 * Driver for starting movie store tests
 */

#include <iostream>
#include "RentalManager.h"

using namespace std;


int main() {

  RentalManager myManager;

  ifstream inFile1("data4movies.txt");
  if (!inFile1) {
    cout << "File could not be opened." << endl;
    return 1;
  }
  ifstream inFile2("data4customers.txt");
  if (!inFile2) {
    cout << "File could not be opened." << endl;
    return 1;
  }
  ifstream inFile3("data4commands.txt");
  if (!inFile3) {
    cout << "File could not be opened." << endl;
    return 1;
  }

  myManager.readInventory(inFile1);
  myManager.readCustomers(inFile2);
  myManager.readCommands(inFile3);

  inFile1.close();
  inFile2.close();
  inFile3.close();

  testAll();

  cout << "Done." << endl;
  return 0;
}
```

RentalManager.h: Reads from file and directs to methods to read further and to create objects

```
#ifndef RENTALMANAGER_H
#define RENTALMANAGER_H
#include <iostream>
#include <fstream>
#include <string>

#include "BSTree.h"
#include "Classic.h"
#include "Comedy.h"
#include "Customer.h"
#include "Drama.h"
#include "HashMap.h"
#include "Movie.h"
#include "MovieFactory.h"
#include "Storage.h"
#include "TransFactory.h"

using namespace std;

class RentalManager {
public:
 //constructor default and destructor
 RentalManager();
 ~RentalManager();
  void readMovies(ifstream& infile); // read movies from data
 void readInventory(ifstream& infile); // read inventory from data
 void readCustomers(ifstream& infile); // read customers from data
 void readCommands(ifstream& infile); //readTransactions or readCommands

private:
 MovieFactory movieFac_; //movie factory
 TransFactory transFac_; //transaction factory
 HashMap customers_; //hashmap of customers
 Storage catalouge_; //inventory
};
#endif //RENTALMANAGER_H
```

ItemFactory.h: Creating Item objects

```
#ifndef ITEMFACTORY_H
#def ITEMFACTORY_H

#include "Item.h"

/*
ItemFactory is the class where we create Item objects
*/
class ItemFactory {
public:

 // constructor
 ItemFactory();

 // destructor
 virtual ~ItemFactory();

 // where we create the item, using a char prefix that's been read and     passed in.
We keep it virtual since it's going to be overridden.
 virtual Item *create(char prefix) const = 0;
};

#endif // ITEMFACTORY_H
```

MovieFactory.h: Child to ItemFactory, creating movie objects.

```
#ifndef MOVIEFACTORY_H
#def MOVIEFACTORY_H

#include "ItemFactory.h"
#include "Movie.h"
#include "Drama.h"
#include "Comedy.h"
#include "Classics.h"
#include <map>
#include <vector>

/*
MovieFactory is the child of ItemFactory. In MovieFactory
we're creating the movie objects
*/
class MovieFactory: public ItemFactory {
public:
 // constructor
 MovieFactory();

 // destructor
 virtual ~MovieFactory();

 // creating the movie object, overrides Item's create function
 Item *create(char prefix) const override; // override is     optional

private:

 // Hashmap with keys equal to char variables of every possible movie genre and values
equal to integers ranging from 0 to the number of move types -1
 unordered_map<char, int> movieMap;

 // vector with elements that are initialized as new movie objects of every possible
movie type
 vector<Movie*> movieFac;
};

#endif // MOVIEFACTORY_H
```

Storage.h: Stores the BST for each of Movie's children.

```
#ifndef STORAGE_H
#define STORAGE_H
#include <iostream>
#include <fstream>
#include <string>
#include "BSTree.h"
#include "RentalManager.h"

using namespace std;

class Storage {
public:
 //constructors
 Storage(); //default
 ~Storage(); //destructor

 void makeEmpty();
 bool insertItem(Item*); //insert means add, changing to true/false (bool)
 Item& retrieveItem(string itemName) const;
 bool retrieve(BSTree*) // add retrieve BST*
 void printStorage(); // add print function

private:
//Key of map are 2-element char arrays. The first element indicates the type of the
items in the corresponding BSTree object and the second element indicates the subtype
of the items in the BSTree objects that all have the same Item type and subtype
 unordered_map<string, BSTree*> BSTreeMap;
}
#endif //STORAGE_H
```

BSTree.h: BST which stores item object

```
// BSTree Class: binary search tree, no duplicates
#ifndef BSTREE_H
#define BSTREE_H

#include <iostream>
#include <string>
#include <vector>

#include "Movie.h"
#include "Storage.h"


using namespace std;

class BSTree {
 struct Node;
 friend ostream &operator<<ostream &out, const BSTree &bst);

public:
 //constructors
 BSTree(); //default
 ~BSTree(); //destructor
  bool isEmpty() const; //returns bool if BSTree is empty or not
 void makeEmpty(Node *&); //makeEmpty - recursive delete helper
 bool insert(Item *item); //inserts item into BSTree
 Item& find(string name) const; //find an item object from the BSTree
 //retrieves an item from the BSTree
 bool retrieve(Item *target, Item *&retrieverItem) const;
 void printMovie(); // prints
private:
 struct Node{
   Item *data; //pointer to data
   Node *left; //left subtree pointer
   Node *right; //right subtree pointer
   }
 Node* root; //root of the tree
 Node *retrieveHelper(Node *&, Item *) const; // retrieve helper function - recursive
 void print(ostream &) const; //print for ostream <<
}
#endif //BSTREE_H
```

Item.h: Item object

```
#ifndef ITEM_H
#define ITEM_H

#include /////

using namespace std;

class Item {
//ostream operators used for printing
friend ostream &operator<<(ostream &out, const Item &item);

public:
 // constructor
 Item();
  // destructor
 virtual ~Item();

 // decrement copies of private variables
 void borrowItem();

 // increment copies of private variables
 void returnItem();

 // won't be created in Item.cpp & Movie.cpp,
 // but appear in chilren classes
 // Item & Movie ==> abstract classes
 // create() function can be used after the types of Movie
 virtual Item* create() const = 0;

 // compare operators to sort items in BSTree
 virtual bool operator==(const Item &other) const = 0;

 virtual bool operator!=(const Item &other) const = 0;

 virtual bool operator<(const Item &other) const = 0;

 virtual bool operator>(const Item &other) const  = 0;
  // assignment operator
 virtual Item& operator=(const Item &other) = 0;

 // setting item
```

```
 virtual void setItem(istream& data) = 0;

protected:
 // gives children class access
 int maxCopies;
 int currCopies;
 char itemType; // indicating it's a movie
 char mediaType; // indicating the format (D for DVD)
};

#endif // ITEM_H
```

Movie.h: Child to Item class

```
#ifndef MOVIE_H
#define MOVIE_H

#include "Item.h"

using namespace std;

class Movie : public Item {
//ostream operators used for printing
friend ostream &operator<<(ostream&, const Item &);

public:
 // constructor
 Movie();
  // destructor
 virtual ~Movie();

 // won't be created in Item.cpp & Movie.cpp,
 // but appear in chilren classes
 // Item & Movie ==> abstract classes
 // create() function can be used after the types of Movie
 virtual Item* create() const = 0;

 // compare operators to sort items in BSTree
 virtual bool operator==(const Item &other) const = 0;

 virtual bool operator!=(const Item &other) const = 0;

 virtual bool operator<(const Item &other) const = 0;

 virtual bool operator>(const Item &other) const = 0;
  // assignment operator
 virtual Item& operator=(const Item &other) = 0;

 // setting item
 virtual void setItem(istream& data) = 0;

 // return genre of movie:
 // F - Comedy, D - Drama, C- Classics
 char getMovieGenre() const;
```

```
protected:
 // gives chilren class access
 char movieGenre;
 string title;
 int yearReleased;
};

#endif // MOVIE_H
```

Comedy.h: Child to movie class.

```
#ifndef COMEDY_H
#define COMEDY_H

#include "Movie.h"

using namespace std;

class Comedy : public Movie {

public:
 // constructor
 Comedy();
  // destructor
 virtual ~Comedy();

 // create() function can be used after the types of Movie
 virtual Item* create() const;

 // compare operators to sort items in BSTree
 virtual bool operator==(const Item &other) const;

 virtual bool operator!=(const Item &other) const;

 virtual bool operator<(const Item &other) const;

 virtual bool operator>(const Item &other) const;
  // assignment operator
 virtual Item& operator=(const Item &other);

 // setting item
 virtual void setItem(istream &data);

private:
 string director;
};

#endif // COMEDY_H
```

Classics.h: Child to movie class

```
#ifndef CLASSICS_H
#define CLASSICS_H

#include "Movie.h"

using namespace std;

class Classics : public Movie {

public:
 // constructor
 Classics();
  // destructor
 virtual ~Classics();

 // create() function can be used after the types of Movie
 virtual Item* create() const;

 // compare operators to sort items in BSTree
 virtual bool operator==(const Item &other) const;

 virtual bool operator!=(const Item &other) const;

 virtual bool operator<(const Item &other) const;

 virtual bool operator>(const Item &other) const;
  // assignment operator
 virtual Item& operator=(const Item &other);

 // setting item
 virtual void setItem(istream &data);

private:
 string director;
 string majorActor;
 int month;
};

#endif // CLASSICS_H
```

Drama.h: Child to movie class.

```
#ifndef DRAMA_H
#define DRAMA_H

#include "Movie.h"

using namespace std;

class Drama : public Movie {

public:
 // constructor
 Drama();
  // destructor
 virtual ~Drama();

 // create() function can be used after the types of Movie
 virtual Item* create() const;

 // compare operators to sort items in BSTree
 virtual bool operator==(const Item &other) const;

 virtual bool operator!=(const Item &other) const;

 virtual bool operator<(const Item &other) const;

 virtual bool operator>(const Item &other) const;
  // assignment operator
 virtual Item& operator=(const Item &other);

 // setting item
 virtual void setItem(istream& data);

private:
 string director;
};

#endif // DRAMA_H
```

HashMap.h: Hash table is implemented using an array and we store customer objects inside the hash table.

```
#ifndef HASHMAP_H
#def HASHMAP_H

#include "Customer.h"

/*
 Hashmap stores customers into an array, that way
 customers can be accessed easily. We are implementing the hash table as an array.
*/
class HashMap {
public:

 // constructor
 HashMap();

 // destructor
 virtual ~HashMap();

 // adding customers into the Hash table
 bool addCustomers(int spot, ifstream& line);

 // deleteCustomer deletes a customer based on the account number
 bool deleteCustomers(const int accountNum);

 // get customer gets the customer account pointer connected to the accountNum
parameter
 Customer* getCustomer(const int accountNum*);

 // hashifying
 int hashify(int spot);

private:
 // array which stores customers
 Customer* customerArray[];
};

#endif // HASHMAP_H
```

Customer.h: Customer object that stores customer information that's been read from the file. Includes ID, first, and last name.

```
#ifndef CUSTOMER_H
#def CUSTOMER_H

#include "Transaction.h"
#include <string>
#include <vector>

/*
Customer object stores the customer info that's been read from the file. This includes
a unique four-digit ID, their first and last name, and also a history vector where we
store the transactions of the customer.
*/
class Customer {

friend class Hashmap;

public:
 // constructor
 customer();

 // destructor
 virtual ~customer();

 // adding the transaction to customer's history
 void addToHistory(Transaction*& );

 // printing out customer's history
 void printHistory const();

 // returning customer information
 int getCustomerID();

private:
 // unique ID for customer
 int customerID;

 // customer firstname
 string firstName;

 // customer lastname
```

```
 string lastName;

 // vector where we store the transactions
 vector<Transaction*> history;
};

#endif // CUSTOMER_H
```

TransFactory.h: Deals with reading in and creating transaction objects.

```
#ifndef TRANSFACTORY_H
#define TRANSFACTORY_H
#include "borrow.h"
#include "displayinventory.h"
#include "history.h"
#include "return.h"
#include "transaction.h"
#include <unordered_map>
#include <vector>
using namespace std;

class TransFactory {

 public:
   TransFactory();                              // default constructor
   virtual ~TransFactory();                     // virtual destrcutor
   Transaction *createTransaction(char) const; // create Treansaction method

 private:
   // HashMap with keys equal to char variables of every possible movie type
   // and values equal to integers ranging from 0  to the number of
   // transaction types - 1.
   unordered_map<char, int> transMap;

   // Vector with elements that are initialized as new
   // Transaction objects of every possible transaction type
   vector<Transaction *> transFac;
};

#endif // TRANSFACTORY_H
```

Transaction.h: Parent class for all transactions which includes: borrow, return, history, and display inventory

```
#ifndef TRANSACTION_H
#define TRANSACTION_H
#include "hashmap.h"
#include "customer.h"
#include "storage.h"
#include <fstream>
#include <string>
using namespace std;

class Transaction {

 public:
   Transaction();                          // default constructor
   virtual ~Transaction();                 // virtual destructor
   virtual Transaction *create() const = 0; // virtual create method
   virtual bool setData(istream &) = 0;     // virtual set data method
   // virtual doTransaction method, defined in derived classes
   virtual void doTransaction(Storage &, HashMap &) = 0;
};

#endif // TRANSACTION_H
```

Borrow.h: Class that handles borrowing of Item objects

```
#ifndef BORROW_H
#define BORROW_H
#include "itemfactory.h"
#include "transaction.h"
#include <string>
using namespace std;

class Borrow : public Transaction {

 private:
   Item *theItem;           // the movie that is being returned
   int custID;            // the Customer that returns the movie
   ItemFactory itemFac;     // factory to create Item objects

 public:
   Borrow();                      // default constructor,
   virtual ~Borrow();             // destructor
   virtual bool setData(istream &); // sets data for item and patron involved
                                    // in the Borrow transaction

   virtual Transaction*
   create() const; // creates and return new Borrow object
   virtual void doTransaction(Storage &,
                              HashMap &); // overridden from Transaction
                                          // performs Borrow on movie and
                                          // adds Borrow details to
                                          // customer's vector of transactions
};

#endif
```

Return.h: Class that deals with returning item objects

```
#ifndef RETURN_H
#define RETURN_H
#include "itemfactory.h"
#include "transaction.h"
#include <string>
using namespace std;

class Return : public Transaction {

 private:
   Item *theItem;           // the movie that is being returned
   int custID;            // the customer that returns the movie
   ItemFactory itemFac; // factory to create Item objects

 public:
   Return();                        // default constructor
   virtual ~Return();               // destructor
   virtual bool setData(istream &); // sets data for item and customer involved
                                    // in the return transaction
   virtual Transaction *
   create() const; // creates and returns new Return object
   virtual void doTransaction(Storage &,
                              HashMap &); // overridden from Transaction
                                          // performs return on movie and
                                          // adds return details to
                                          // customers's vector of transactions
};

#endif
```

History.h: Displays transaction history of customer.

```
#ifndef HISTORY_H
#define HISTORY_H
#include "customer.h"
#include "transaction.h"
using namespace std;

class History : public Transaction {

 private:
   int custID; // the Customer whose history will be displayed

 public:
   History();          // constructor for class History
   virtual ~History(); // destrcutor for History
   bool setData(istream &);  // setData() method
   virtual Transaction* create() const; // create method to create and
                                        //return a new History object pointer
   virtual void doTransaction(Storage &, HashMap &); // displays transaction
                                                     // history of customer with
                                                     // ID equal to value of
                                                     // data member custID
};

#endif
```

DisplayInventory.h: Displays current inventory of Item objects

```
#ifndef DISPLAYINVENTORY_H
#define DISPLAYINVENTORY_H
#include "item.h"
#include "transaction.h"
#include <string>
using namespace std;

class DisplayInventory : public Transaction {

 public:
   DisplayInventory();                                // default constructor
   virtual ~DisplayInventory();                       // destructor
   virtual Transaction *create() const;               // virtual create method
   virtual bool setData(istream &);                   // virtual set data method
   virtual void doTransaction(Storage &, HashMap &); // displays the catalogue
};
#endif
```