CSS 343: Program 4: Design

Prepared by: Teja Dasari, Shreyas Sundar Ganesh

Prepared for: Dr. Wooyoung Kim

Date: 3/1/2025

Table of Contents:

Overview/Specification:

This program automates the inventory tracking system of a local movie rental store. It manages three types of DVDs–Comedy, Drama, and Classics–and tracks customer transactions such as borrowing and returning movies. The system reads movie data, customer records, and transaction requests from input files, processes them accordingly, and outputs relevant information, including inventory status and customer rental history.

The constraints and requirements of Program 4 are as follows:

- The program must use inheritance for Movie classes.
- The program must use inheritance for Transaction classes.
- The program must use a hash table implementation that does not use any STL other than array.
- The program must have a hashing algorithm implemented.
- The program must have a correctly implemented and working history transaction.
    - The history transaction should show Lisa's customer DVD transactions chronologically (latest to earliest) and specify whether the movie was borrowed or returned.
- The program must have a correctly implemented and working inventory transaction.
- The program must have a correct working history transaction.
    - It should output all comedy movies, then all dramas, and then all classics. The film in each category should be ordered according to the sorting criteria discussed in the instructions.

- The program must have a correctly implemented and working borrow transaction.
- The program must have a correctly implemented and working return transaction.
- The program must display an error output for an invalid action, video, or customer ID.
    - The program must display an error, especially for borrow/return actions.

Class diagram:

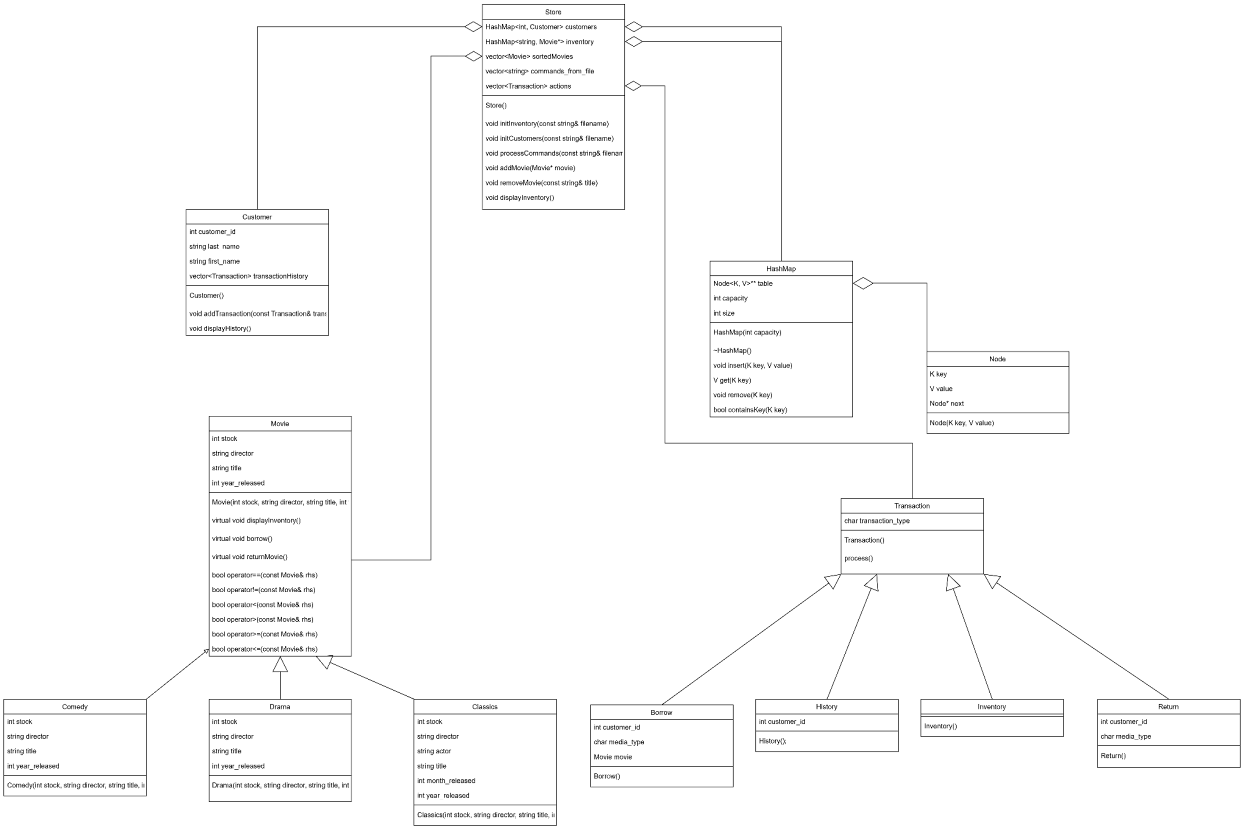
Store
HashMap<int, Customer> customers
HashMap<string, Movie*> inventory
vector<Movie> sortedMovies
vector<string> commands_from_file
vector<Transaction> actions
Store()
void initInventory(const string& filename)
void initCustomers(const string& filename)
void processCommands(const string& filenam
void addMovie(Movie* movie)
void removeMovie(const string& title)
void displayInventory()
Customer
int customer_id
string last name
string first_name
vector<Transaction> transactionHistory
Customer()
void addTransaction(const Transaction& trans
void displayHistory()
HashMap
Node<K, V>** table
int capacity
int size
HashMap(int capacity)
~HashMap()
void insert(K key, V value)
V get(K key)
void remove(K key)
bool containsKey(K key)
Node
K key
V value
Node* next
Node(K key, V value)
Movie
int stock
string director
string title
int year_released
Movie(int stock, string director, string title, int
virtual void displayInventory()
virtual void borrow()
virtual void returnMovie()
bool operator==(const Movie& rhs)
bool operator!=(const Movie& rhs)
bool operator<(const Movie& rhs)
bool operator>(const Movie& rhs)
bool operator>=(const Movie& rhs)
bool operator<=(const Movie& rhs)
Transaction
char transaction_type
Transaction()
process()
Comedy
int stock
string director
string title
int year_released
Comedy(int stock, string director, string title, i
Drama
int stock
string director
string title
int year_released
Drama(int stock, string director, string title, int
Classics
int stock
string director
string actor
string title
int month_released
int year released
Classics(int stock, string director, string title, i
Borrow
int customer_id
char media_type
Movie movie
Borrow()
History
int customer_id
History();
Inventory
Inventory()
Return
int customer_id
char media_type
Return()

Figure 1: UML Object-oriented diagram for the main program.

This diagram could be extended, for example:

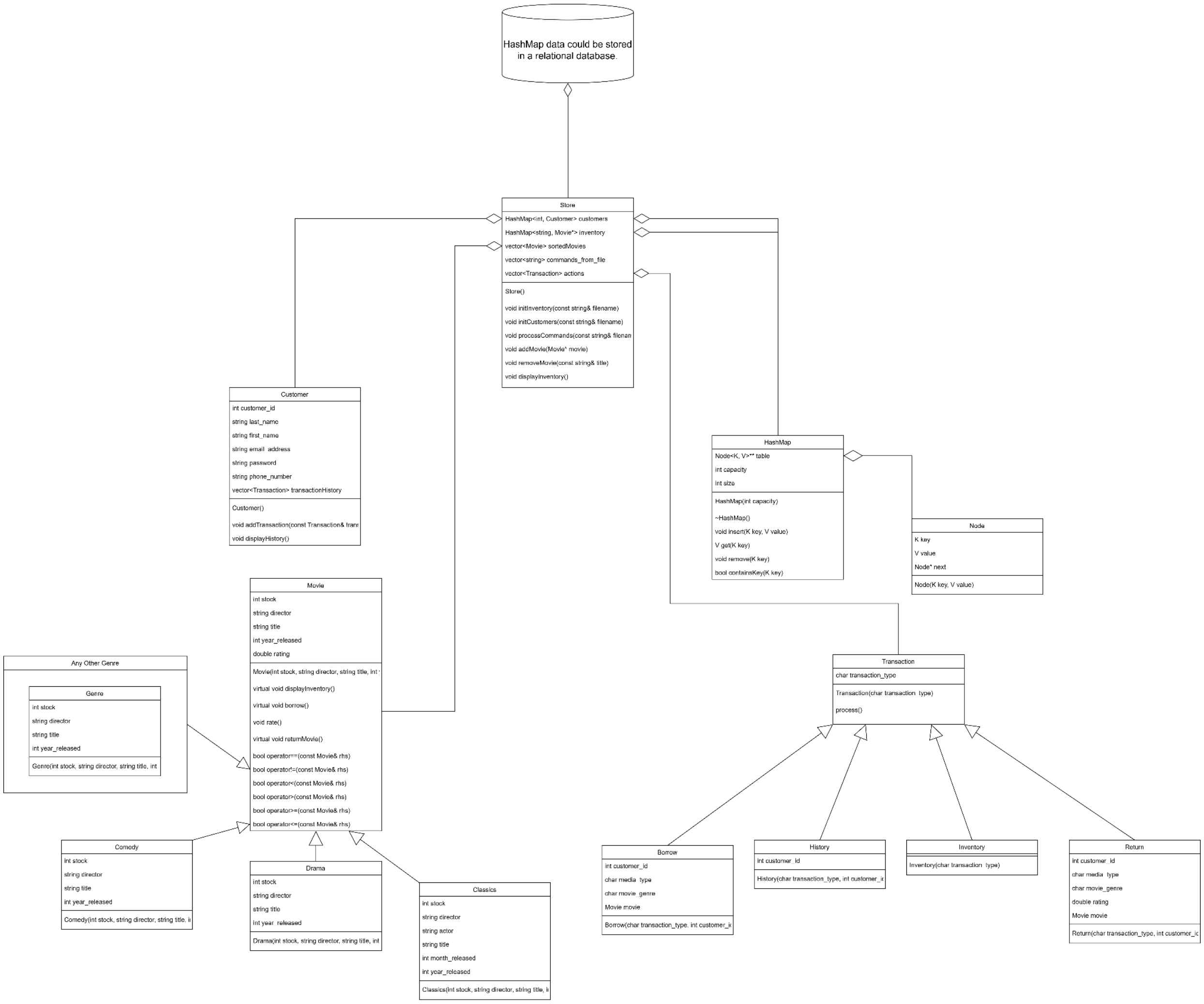
HashMap data could be stored in a relational database.
Store
HashMap<int, Customer> customers
HashMap<string, Movie*> inventory
vector<Movie> sortedMovies
vector<string> commands_from_file
vector<Transaction> actions
Store()
void initInventory(const string& filename)
void initCustomers(const string& filename)
void processCommands(const string& filenan
void addMovie(Movie* movie)
void removeMovie(const string& title)
void displayInventory()
Customer
int customer_id
string last_name
string first_name
string email_address
string password
string phone_number
vector<Transaction> transactionHistory
Customer()
void addTransaction(const Transaction& trans
void displayHistory()
HashMap
Node<K, V>** table
int capacity
int size
HashMap(int capacity)
~HashMap()
void insert(K key, V value)
V get(K key)
void remove(K key)
bool containsKey(K key)
Node
K key
V value
Node* next
Node(K key, V value)
Movie
int stock
string director
string title
int year_released
double rating
Movie(int stock, string director, string title, int
virtual void displayInventory()
virtual void borrow()
void rate()
virtual void returnMovie()
bool operator==(const Movie& rhs)
bool operator!=(const Movie& rhs)
bool operator<(const Movie& rhs)
bool operator>(const Movie& rhs)
bool operator>=(const Movie& rhs)
bool operator<=(const Movie& rhs)
Any Other Genre
Genre
int stock
string director
string title
int year_released
Genre(int stock, string director, string title, int
Comedy
int stock
string director
string title
int year_released
Comedy(int stock, string director, string title, i
Drama
int stock
string director
string title
int year_released
Drama(int stock, string director, string title, int
Classics
int stock
string director
string actor
string title
int month_released
int year_released
Classics(int stock, string director, string title, i
Transaction
char transaction_type
Transaction(char transaction_type)
process()
Borrow
int customer_id
char media_type
char movie_genre
Movie movie
Borrow(char transaction_type, int customer_i
History
int customer_id
History(char transaction_type, int customer_i
Inventory
Inventory(char transaction_type)
Return
int customer_id
char media_type
char movie_genre
double rating
Movie movie
Return(char transaction_type, int customer_i

Figure 2: Extended UML diagram for possible increased functionality of the main program.

There are a couple ways to extend it, including but not limited to:

- HashMap data could be put into and retrieved from a relational database, like Postgress SQL.
    - Because of this, it would be possible to maintain multiple Stores, like chains. This way, you'd also be able to transfer movies from one location to another, like a library holding system.
- Movies could have additional attributes, like ratings, maybe a list of producing staff and/or actors, or maybe some unique ID associated with any movie's production, if a standard exists for it.
- Additional genres, like Adventure, Romance, or Thriller could be added, and if there was another unique way to sort them, the implementation could be added for that, too. More subclass under Movie would have to be made for them.
- Additional transactions could be added, like maybe changing the id of a movie, or extending the rental duration of a movie, and these would be sub-/child classes of the Transaction class.
- A interface and components for a web application could be included or some sort of front-end. It's a bit beyond the system itself, hence why it wasn't included in Figure 2.
- Different video formats, such as CDs and Blu-ray could be added.

- Other media types, such as music, could be added, as well as media-viewing technology.
- Time could be implemented as a way of maintaining rental duration and notify customers of soon due materials, like a Time duration field in the Movie class, to be inherited by genres of Movies.

Class descriptions:

Class	Description
hashmap.h	Implements a hash table for fast lookup of movies and customers
store.h	Manages the entire rental system. Acts as the central controller of the program
customer.h	Stores customer information and tracks rental history
movie.h	Parent class for all movie types. Needed for storing and managing movie details
action.h	Base class for all customer transactions
comedy.h	Sub-/Child class of Movie for comedy movies, sorted by title, then year
drama.h	Sub-/Child class of Movie for drama movies, sorted by director then title
classics.h	Sub-/Child class of Movie for classic movies, sorted by release date then actor
borrow.h	Sub-/Child class of Action, handles borrowing movies.
return.h	Sub-/Child class of Action, handles returning of movies.
inventory.h	Displays store inventory sorted by category
history.h	Displays customer’s rental history

The following table has comments written to describe the purpose and data flow that occurs during a method call.

<table>
<tr><th>Class</th><th>Pseudo code</th></tr>
<tr><td>hashmap.h</td><td>

```
class HashMap{
public:
   HashMap(int capacity)
     //Initialize table with capacity
     //Set size to 0
   ~HashMap()
     //Clear all LinkedLists in array indices for the table
   void insert(K key, V value)
      //To insert values into the HashMap
      //Would calculate hash value through a hash function
      //Would insert into table with key and value
      //Upon collision, would handle using open hashing,
      separate chaining, adding a node linked to nodes
      already at that index
   V get(K key)
      //To get values from the HashMap
      //Would go to the hash value/index in the table
      through traversal
      //Look for the key in the "bucket", the LinkedList
      at that index
      //Retrieve the value associated with that key
   void remove(K key)
      //To remove key-value pairs from the Map
      //Would follow the same logic as the get() method,
      but would delete the node containing the key-value
      pair
   bool containsKey(K key)
      //To check if a key is already within the Map
      //Again, would follow the same value as the get()
      method to find the key, returns true or false if the
      key was found.
private:
   Node<K, V>** table //Pointer to an array of LinkedlList
   Nodes with LinkedLists attached to those.
   int capacity //Fixed size for hash table
   int size //Tracks number of elements
}
```

</td></tr>
</table>

store.h

```
class Store{
public:
      void initInventory(const string& filename)
          //Read movie file using fstream
          //Create movie objects through constructors and if
      conditionals to ascertain movie genre
          //Store in inventory and sorts movies using operator
      overloads
      void initCustomers(const string& filename)
          //Read customer file using fstream
          //Create customer objects through constructors
          //Store in HashMap<int, Customer> customers
      void processCommands(const string& filename)
          //Read command file using fstream
          //create transaction objects using constructors
          //store in commands_from_file
          //translate from string data to Transaction objects
          //execute all transactions
      void addMovie(Movie* movie)
          //If movie exists, increase stock
          //Else, add to inventory and sorted movies
      void removeMovie(const string& title)
          //Remove from inventory and sorted movies
          //May destroy it if it ends up being dynamically
      allocated
      void displayInventory()
          //Print all movies in sorted order, assuming that they
      are already sorted in sortedMovies
private:
      HashMap<int, Customer> customers //Enables fast lookup of
      customers by ID
      HashMap<string, Movie*> inventory //Enables fast lookup of
      movies by title
      vector<Movie> sortedMovies //Stores movies in required
      sorted order
      vector<string> commands_from_file //Stores command inputs
      to translate into Action objects
      vector<Transaction> transactions //Stores actions for
      processing individuals
}
```

customer.h

```
class Customer{
public:
      Customer() //Default constructor
      void addTransaction(const Transaction& transaction) //Add
      a transaction the Customer has made to the
      transactionHistory for that Customer
      void displayHistory() //Prints out history in
      chronological order as the transaction appears in the
      transactions file.
private:
      int customer_id //Stores unique customer ID
      string last_name //Stores last name of the customer
      string first_name //Stores first name of the customer
      vector<Transaction> transactionHistory // Stores
      transaction history of the customer in the order that is
      in the input transaction file
}
```

movie.h

```
class Movie {
public:
      Movie(int stock, string director, string title, int
      yearReleased) //Default constructor, assigns values to
      respective fields
      virtual void display() //To display movies in their
      respective formats, depending on the genre
      virtual void borrow() //To be used by the Borrow
      transaction
            //Would decrement stock and handle any stock
            decrement beyond a stock of 0 through conditionals.
      virtual void returnMovie() //To be used by the Return
      transaction
            //Would decrement stock and handle any stock
            decrement beyond a stock of 0 through conditionals.
      bool operator==(const Movie& rhs) //Equivalancy operator
      overload
      bool operator!=(const Movie& rhs) //Non-equivalancy
      operator overload
      bool operator<(const Movie& rhs) //Less than operator
      overload
      bool operator>(const Movie& rhs) //Greater than operator
      overload
      bool operator<=(const Movie& rhs) //Less than or equal to
      operator overload
      bool operator>=(const Movie& rhs) //Greater than or equal
      to operator overload
      //All comparison operators would be used to sort,
      therefore would follow sorting criteria laid out for each
      genre
private:
      int stock //Stores number of DVDs this particular movie
      has
      string director //Stores name of director
      string title //Stores title of movie
      int year_released //Stores the year of release
}
```

transaction.h	class Transaction{
	public:
	Transaction() //Default Action constructor
	virtual void process() //Inherited method to process the type of transaction where this method would be called, if ever it needed to be. Can be overloaded with additional parameters if necessary for a transaction.
	private:
	char transaction_type //Stores the character representing the transaction type in the input file, which is the only field the transactions have in common; all other fields exist in sub-/child classes.
	}

Implementation and Test Plan:

The implementation of the movie rental system follows an object-oriented approach beginning with the development of core classes including movie.h and transaction.h with its subclasses, as well as hashmap.h. The system reads input files (data4commands.txt, data4movies.txt, data4customers.txt) to load movie and customer data while handling errors gracefully. Testing is conducted by unit testing, where individual components such as movie sorting, stock updates, transaction validation, and customer retrieval are verified. Edge cases are tested, such as borrowing an out-of-stock movie, returning an unborrowed movie, and handling invalid commands. The final validation confirms that the program accurately processes the input and maintains the correct inventory and transaction records.