Program 4

Design

Prepared for CSS 343: Data Structures, Algorithms, & Discrete Math II

Team 3: Kyle & Phohanh

Dr. Kim

November 18, 2024

Table of Context

1. Executive Summary

Managing inventory and tracking transactions for movie rental stores can be complex and error-prone when relying on outdated manual processes. As the demand for streamlined inventory management increases, movie rental stores are required to use automated solutions to improve efficiency, customer service, and minimize manual errors.

Our project, **Movie Inventory and Customer Transaction System**, addresses these challenges by implementing an automated system that tracks inventory and customer actions, such as borrowing, returning, and viewing history. This program supports three movie categories - Comedy, Drama, and Classics, while implementing sorting criteria tailored for each type. The system includes robust error handling for invalid data/commands and then notifies the user. Additionally, our system will be able to display customer transaction history and a complete inventory overview.

By utilizing data structures such as hash tables and object-oriented principles such as inheritance, our system guarantees fast and efficient operations, particularly in larger datasets. Additionally, our system boasts features like chronologically-sorted transaction history, customizable inventory sorting (based on movie type), and robust error reporting. Through this project, we aim to utilize core concepts from data structures & algorithms and software design to deliver a practical tool to meet real-world needs of movie rental businesses.

2. Overview

The Movie Inventory and Customer Transaction System is designed to automate inventory management for a local movie rental store. The system manages three categories of DVDs - Comedy, Drama, and Classics, and tracks customer transactions such as borrowing, returns and viewing history. The program reads from three files: data4movies.txt (inventory data), data4customer.txt (customer information), and data4commands.txt (operations to execute).

The program contains the following features:

- Sorting: Each movie category has specific and different sorting criteria:
    - Comedy: By title, then year.
    - Drama: By director, then title.
    - Classics: By release date, then major actor.
- Error Handling: Invalid data or commands, such as incorrect customer IDs or invalid actions codes, are identified, logged, and discarded without crashing the program.
- Hash Tables: Used for efficient customer search/lookup.
- Inheritance: Parent classes are used for shared functionalities across movie types and transaction processing.

Runtime errors we anticipate include: invalid movie codes, unavailable stock, and unregistered customers, all of which are handled with descriptive error messages.

3. Specific Requirements

3.1 Functionality

This subsection contains the requirements for the movie inventory and transaction management system. These requirements are derived from the project description and are organized to best capture the functional needs of the system.

3.1.1 Inventory Management

1. The system shall initialize the inventory from the movie data file (data4movies.txt).
2. The system shall sort movies into three categories:
    - Comedy: By Title, then Year.
    - Drama: By Director, then Title.
    - Classics: By Release Date, then Major Actor.
3. The system shall sum stock for Classic movies with multiple entries (same title, different actors).
4. The system shall notify the user if invalid movie data is detected (e.g., invalid codes, missing attributes).
5. The system shall allow the display of all items in inventory, sorted by category.

3.1.2 Transaction Management

1. The system shall initialize customers from the customer data file (data4customers.txt).
2. The system shall process commands from the command file (data4commands.txt), including:
    - Borrow (B): Decreases the stock by 1.
    - Return (R): Increases the stock by 1.
    - Inventory (I): Outputs all items in inventory.
    - History (H): Outputs all transactions for a specific customer in reverse chronological order.
3. The system shall prevent invalid transactions, such as:
    - Returning an item that was not borrowed.
    - Borrowing an item with zero stock remaining.
4. The system shall notify users of errors in transaction commands, such as:
    - Invalid action codes.
    - Unregistered customer IDs.
    - Movies not found in inventory.

3.1.3 User Interaction

1. The system shall allow the user to view error notifications for invalid data or commands.
2. The system shall output formatted results for inventory and history commands.

3.1.4 Extensibility

- The system shall allow for adding new movie categories with minimal code changes.
- The system shall be designed to accommodate new media types beyond DVDs.
- The system shall allow new transaction types to be implemented efficiently.

3.1.5 Performance

- The system shall perform inventory lookups and customer searches in O(1) time using a hash table.
- The system shall process commands in under 5 seconds for datasets with up to 1,000 movies and 1,000 customers.

Functional SRS Requirements (MoSCoW Format)

1. The system must allow for the initialization of inventory using a movie data file (data4movies.txt).
2. The system must allow the initialization of customers using a customer data file (data4customers.txt).
3. The system must process transaction commands from a command file (data4commands.txt), including borrow, return, inventory, and history actions.
4. The system must handle invalid data (e.g. invalid action codes, movie codes, or customer IDs) and notify the user about the errors.
5. The system must update inventory counts when movies are borrowed or returned.
6. The system must provide customer transaction history in reverse chronological order (latest to earliest).
    a. The system must sort Comedy movies by title and year.
    b. The system must sort Drama movies by director and title.
    c. The system must sort Classic movies by release date and major actor.
7. The system must store customer transaction histories and allow retrieval by customer ID.
8. The system must provide the current inventory status for all movie categories.
9. The system must handle multiple transactions in chronological order for history queries.

10. The system should notify users of any invalid or missing data in the input files.
11. The system could allow adding new operations (e.g. overdue tracking).
12. The system could include a feature for tracking overdue borrow returns.
13. The system won’t support multimedia files types other than three included text files DVD in its current implementation.
14. The system won’t provide a graphical user interface in this iteration of the project.

4. User Stories

1. As a store manager, I want to initialize the movie inventory and customer data from files so that the system is ready for operations.
2. As a store employee, I want to borrow a movie for a customer, ensuring the inventory is updated correctly.
3. As a store employee, I want to process return commands so that inventory is restocked automatically.
4. As a store employee, I want to view customer transaction history so that I can answer inquiries and resolve disputes.
5. As a customer, I want my transaction history to be accurately recorded so that I can trust the system.
6. As a developer, I want the system to gracefully handle invalid inputs so that it remains stable.
7. As a developer, I want the inventory sorted by specific attributes for each movie category to ensure data consistency.
8. As a store administrator, I want the system to generate error notifications for invalid inputs so that I can address issues promptly.
9. As a developer, I want to extend the system easily to accommodate new media types, like music or Blu-ray.
10. As a store employee, I want to provide alternatives when a specific Classic movie stock runs out by suggesting other actors for the same title.
11. As a developer, I want to use inheritance for shared functionalities among movies and transactions to simplify code maintenance.

System Architecture

Class Diagram

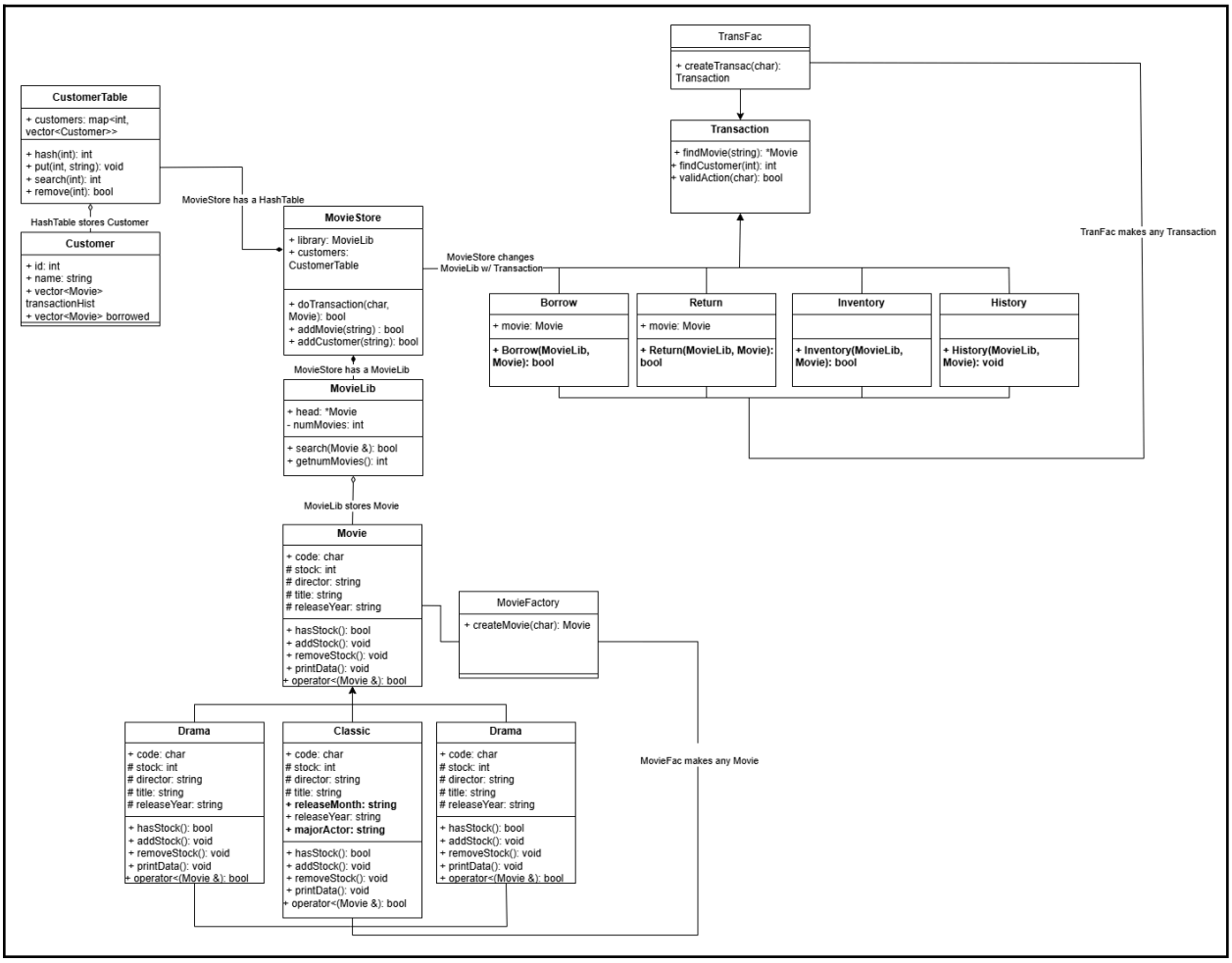

Figure 1.1 Class Diagram Author: Phohanh Tran, Created on Nov 18, 2024

Class Descriptions

MovieStore

public MovieLib library

public CustomerTable customers

Constructor initializes empty MovieLib and CustomerTable.

Destructor calls MovieLib and CustomerTable destructors implicitly to iterate through and deallocate allocated MovieLib Movie nodes.

bool doTransaction(char, Movie) takes an action code (ie ‘B’, ‘R’, etc.) and a Movie object. Uses a different Transaction action class to modify *library*. Returns true if could

do the transaction, false if else (ie, invalid action code, borrowing movie with 0 stock, returning movie that is not currently borrowed).
bool addMovie(string) takes a string of movie data in the form: Stock, Director, Title, Year it released or Stock, Director, Title, Major actor, Release date. Calls MovieFactory to create a new movie that will be stored in *library*. Returns true if could create and store new movie, false if else (ie invalid movie data).
bool addCustomer(string) takes a string of customer data in the form: 4-digit ID number that uniquely identifies them, last name, first name. Creates a customer object and stores in CustomerTable. Returns true if could create and store new customer, false if else (ie invalid customer data).

TransacFactory

Constructor
Destructor deallocates all created Transaction objects, which should be possible implicitly through each Transaction's own destructor.
createTransac(char) creates Transaction items

Transaction

Constructor
Destructor
***Movie findMovie(string)** takes a string of movie data and creates a Movie object based off that, then calls MovieLib::search(Movie &), which traverses MovieLib to find a movie matching the input movie data. If not found, but a movie of the same name is found, return that. If no matches are found, outputs error message for incorrect movie data if movie is not found and returns nullptr.
bool findCustomer(int) takes an int customer ID. Calls CustomerTable::search(int). If customer is found, return true. If a customer is not found, output error message for invalid customer ID and return false.
bool validAction(char) takes a action code and returns true if its valid (ie 'B', 'R', etc.). Returns false if else. Outputs error message for incorrect action codes.

Borrow

protected Movie movie - the movie the transaction was done for
Constructor initializes an empty Movie (all attributes are empty or 0).
Destructor
bool doBorrow(MovieLib, int, Movie) takes the MovieStore's MovieLib, a customer's ID, and a Movie. Calls findMovie() to find it in MovieLib and decrements the Movie's *stock* attribute. Adds the borrowed movie to the customer's transaction history. Returns true if was able to borrow the movie, false if the movie's stock was 0 before this function

is called (decrementing would make the stock become negative). Sets *movie* to be the passed in Movie.

Return

Constructor initializes an empty Movie (all attributes are empty or 0).
Destructor
protected Movie movie - the movie the transaction was done for
bool doReturn(MovieLib, int, Movie) takes the MovieStore's MovieLib, an int of a customer's ID, and a Movie. Calls findMovie() to find it in MovieLib and adds to the input movie's stock. Adds the returned movie to the customer's transaction history. Returns true if the input movie is currently borrowed and can be returned, returns false if else. Sets *movie* to be the passed in Movie.

Inventory

Constructor
Destructor
bool doInventory(MovieLib) returns true if MovieLib contains movies, false if empty. Prints the entire contents of MovieLib.

History

Constructor
Destructor
void doHistory(int) takes an int of a customer's ID. Calls findCustomer() and prints the customer's transaction history.

MovieLib

*public *Movie head* - head of Movie BST
private int numMovies
Constructor initializes head as nullptr and numMovies as empty
Destructor recurses through BST to deallocate all Movie pointers
bool addMovie(Movie &) adds returns true if the movie could be added, false if else.
bool search(const string) const returns true if a movie of the input data is present
int getNumMovies() const returns total number of movies

MovieFactory

*Private vector<*Movie> createdMovies*- vector of all movies created
Constructor empty
Destructor deallocates all created Movie objects

Movie createMovie(char) creates a movie item based on the input video code. If input is not ‘F’, ‘D’, or ‘C’ outputs error message for invalid video code and does not create a movie. Returns the created Movie object, or null if the code was invalid.

Movie

public const char genre- the value denoting movies in the .txt file stores ‘’
public const char dataType- the value denoting the datatype ‘D’
protected int stock - stores the amount of copies of the movie are in the library
public const string director - stores the name of the movie director
public const string title - stores the movie title
public const string releaseYear - stores the release year of the movie
Constructor sets all attributes to 0 or “”
Constructor(char) sets all attributes to 0 or “”, sets genre to input
Constructor(int, string, string, string) sets genre to ‘’, and all attributes to passed in values
virtual Destructor all movie subclasses don’t need to explicitly deallocate memory
bool hasStock() returns true if *stock* is at least 1, false if *stock* is 0
void addStock() adds one to *stock*
void removeStock() removes one from *stock*. If called and stock is already 0, output error message for borrowing a movie with no stock.
virtual void printData () prints Movie data (abstract)
[virtual / abstract] bool operator<(Movie &) overload returns true if the input movie reference has larger data values. Compares the two data values of each movie object used to sort it when stored.
virtual bool operator==(Movie &) overload returns true if the input movie has the same title, director, releaseYear, and char genre, returns false if else, or if the input movie is a different type.

Comedy

public const char genre- the value denoting Comedy movies in the .txt file stores ‘F’
public const char dataType- the value denoting the datatype ‘D’
protected int stock - stores the amount of copies of the movie are in the library
public const string director - stores the name of the movie director
public const string title - stores the movie title
public const string releaseYear - stores the release year of the movie
Constructor sets genre to ‘F’ and all other attributes as 0 or “”
Constructor(int, string, string, string) sets genre to ‘F’, and all attributes to passed in values
Destructor
bool hasStock() returns true if *stock* is at least 1, false if *stock* is 0

void addStock() adds one to *stock*

void removeStock() removes one from *stock*. If called and stock is already 0, output error message for borrowing a movie with no stock.

void printData() prints Movie data in form: genre, stock, Title, Year of release, and director. Used when outputting Customer transaction history.

bool operator<(Movie &) overload returns true if the input Movie has a *title* with a greater alphabetical value. If the *title* is the same, then compare *releaseYear*. Returns false if both the *title* and *releaseYear* are smaller than or equal. Compares *title* and *releaseYear* with the two data values used for sorting if the input Movie is not a Comedy. This function will be used to order when storing Movie objects in MovieLib.

bool operator==(Movie &) overload returns true if the input movie has the same title, director, releaseYear, and char genre, returns false if else, or if the input movie is a different type.

Classic

public const char genre- the value denoting Classic movies in the .txt file stores 'C'

public const char dataType- the value denoting the datatype 'D'

protected int stock - stores the amount of copies of the movie are in the library

public const string director - stores the name of the movie director

public const string title - stores the movie title

public const string releaseMonth - stores the release month of the movie

public const string releaseYear - stores the release year of the movie

public const string majorActor - stores the name of the major actor in the movie

private map<string, int> actors- stores the major actors and the stock for all the different versions of the movie

Constructor sets genre to 'C' and all other attributes as 0 or ""

Constructor(int, string, string, string) sets genre to 'C', and all attributes to passed in values

Destructor

bool hasStock() returns true if *stock* is at least 1, false if *stock* is 0

void addStock() adds one to *stock*

void removeStock() removes one from *stock*

void addActor(string, int) adds a new actor to the actors vector, as well as the stock for the movie version with that actor

void printData() prints Movie data in form: genre, stock, Release date, Major actor, release date, and director. Used when outputting Customer transaction history.

bool operator<(Movie &) overload returns true if the input Movie has a more recent *releaseDate*. If the *releaseDate* is the same, then compare the name of the *majorActor* alphabetically. Returns false if both the *releaseDate* and *majorActor* name are smaller than or equal. Compares with the *releaseDate* and *majorActor* with whatever two values

are used to sort the input Movie if it is not a Classic. This function will be used to order when storing Movie objects in MovieLib.
bool operator==(Movie &) overload returns true if the input movie has the same title, director, releaseMonth, releaseYear, and char genre, returns false if else, or if the input movie is a different type.

Drama

public const char genre- the value denoting Drama movies in the .txt file stores 'D'
public const char dataType- the value denoting the datatype 'D'
protected int stock - stores the amount of copies of the movie are in the library
public const string director - stores the name of the movie director
public const string title - stores the movie title
public const string releaseYear - stores the release year of the movie
Constructor sets genre to 'D' and all other attributes as 0 or ""
Constructor(int, string, string, string) sets genre to 'D', and all attributes to passed in values
Destructor
Destructor
bool hasStock() returns true if *stock* is at least 1, false if *stock* is 0
void addStock() adds one to *stock*
void removeStock() removes one from *stock*. If called and stock is already 0, output error message for borrowing a movie with no stock.
void printData() prints Movie data in form: genre, stock, Director, then Title, release year, and director. Used when outputting Customer transaction history.
bool operator<(Movie &) overload returns true if the input movie has a *director* with a greater alphabetical value. If the *director* is the same, then compare *title*. Returns false if both the *director* and *title* are smaller than or equal. Compares with the *director* and *title* with whatever two values are used to sort the input Movie if it is not a Classic. This function will be used to order when storing Movie objects in MovieLib.
bool operator==(Movie &) overload returns true if the input movie has the same title, director, releaseYear, and char genre, returns false if else, or if the input movie is a different type.

CustomerTable

protected vector<Customer> customers - stores customers
Constructor constructs an empty map
Destructor no need to explicitly deallocate memory
int hash(int) takes the customer id and hashes it to store

void put(int, string) takes the customer id and name. Creates a Customer based on the input information, calls the hash function hash() to create a key for the customer, and stores. Output error message if customer id already exists in CustomerTable.
int search(int) takes a customer id, calls the hash function hash() then searches the hashtable using the hashed customer id. Returns the customer's key, -1.
bool remove(int) takes the customer id and removes from the CustomerTable. If removal was successful returns true, if else, returns false (ie the customer id does not exist in the table).

Customer

public int id - the customer's 4 digit id
public string name - the customer's first and last name
private vector<Movie> transactionHist - stores the transactions the customer has done
private vector<Movie> borrowed - stores currently borrowed movies
private Customer next* - stores pointer to next customer
Constructor sets all attributes as empty