

Movies Project


[New Attempt](#)

- Due Jun 4, 2023 by 11:59pm
- Points 25
- Submitting a file upload

Ass 4: Movies

If you are completing this project as a pair, both people should submit their names as a comment to the assignment **at least 72 hours** before the due date. You can submit an empty file and leave comments ahead of the due date. Only one person needs to submit the final set of files before the due date.

This assignment requires you to have a deep understanding of polymorphism and factory pattern. Make sure you understand these patterns before starting your design.

Start your repository by visiting [this link](https://classroom.github.com/a/OmbzD4Dj)  (<https://classroom.github.com/a/OmbzD4Dj>). BUT the design and the implementation will be group projects, so make sure everybody in your group is working on the correct repository.

If you are working as a pair watch [this video](https://www.youtube.com/watch?v=rG_U12uqRhE)  (https://www.youtube.com/watch?v=rG_U12uqRhE).

Design

- Work in groups of 3-4 students – submit members of your group to Canvas. Email me if you'd like me to assign you to a group
- **"Design Peer Evaluation"**. Turn in soft copy on Canvas and bring hard copy of your design to class. See dues dates on Canvas. Your design will be peer evaluated. (30 points)

Implementation

- Work individually or in pairs. If working in a pair
- **"Implementation"**. Download your GitHub repository as a zip file and submit it. If working as a pair, only one person needs to submit the code.

Description

A local movie rental store wishes to automate their inventory tracking system. Currently there are three types of movies/videos (in DVD media) to be tracked:

- **Comedy** (denoted as 'F' for funny)
- **Drama** (denoted as 'D')
- **Classics** (denoted as 'C')

Borrows and returns of items by customers are also to be tracked. Four types of actions are desired in the system:

- **Borrow** (denoted as 'B'): (stock – 1) for each item borrowed
- **Return** (denoted as 'R'): (stock + 1) for each item returned
- **Inventory** (denoted as 'I'): outputs the inventory of all the items in the store
- **History** (denoted as 'H'): outputs all the transactions of a customer

You will design and implement a program that will initialize the contents of the inventory from a file (**data4movies.txt**), the customer list from another file (**data4customers.txt**), and then process an arbitrary sequence of commands from a third file (**data4commands.txt**).

Details

File formats and file details are as follows:

data4movies.txt:

The information about each movie is listed as follows:

- Comedy: F, Stock, Director, Title, Year it released
- Drama: D, Stock, Director, Title, Year it released
- Classics: C, Stock, Director, Title, Major actor Release date

For example,

F, 10, Nora Ephron, You've Got Mail, 1998

D, 10, Steven Spielberg, Schindler's List, 1993

C, 10, George Cukor, Holiday, Katherine Hepburn 9 1938

C, 10, George Cukor, Holiday, Cary Grant 9 1938

Z, 10, Hal Ashby, Harold and Maude, Ruth Gordon 2 1971

D, 10, Phillippe De Broca, King of Hearts, 1967

1. You may assume correct formatting, but codes may be invalid; e.g., in this data, the 'Z' code is an invalid entry so this line has to be discarded and users should be notified about this issue.
2. While the stock for each line is 10, do not assume that is the case in your design and implementation.
3. The movies should be sorted as follows when stored in the system:
 1. **Comedy** ('F') sorted by Title, then Year it released
 2. **Dramas** ('D') are sorted by Director, then Title
 3. **Classics** ('C') are sorted by Release date, then Major actor
4. Each item is uniquely identified by its complete set of sorting attributes.
5. The classical movie type has a different format than the other two.

1. Major actor is always formatted as two strings, FirstName and LastName, separated by a space.
2. The Release date contains month and year information, and no comma (but a space) between Major actor and Release date.
3. In addition, for classical movies, one movie (e.g., Holiday) may have multiple lines, but since each classical movie is uniquely identified by its sorting attributes (Release date, then Major actor in this case), the two entries are treated as separate movies.

data4customers.txt:

The information about each customer is listed as follows:

- Customer: CustomerID LastName FirstName

For example,

1111 Mouse Mickey

1000 Mouse Minnie

1. You may assume that the data is formatted correctly.
2. CustomerID is 4-digits and uniquely identifies each customer

data4commands.txt:

The information about each command is listed as follows:

- Inventory I
- History H CustomerID
- Borrow B CustomerID MediaType MovieType (movie sorting attributes)
- Return R CustomerID MediaType MovieType (movie sorting attributes)

For example,

B 1234 D C 9 1938 Katherine Hepburn

B 1234 D F Pirates of the Caribbean, 2003

R 1234 D C 9 1938 Katherine Hepburn

B 1234 D D Steven Spielberg, Schindler's List

I

H 1234

X 1234 Z C 9 1938 Katherine Hepburn

B 1234 D **Y** 2 1971 Ruth Gordon

B **9999** D F Pirates of the Caribbean, 2003

B 1234 D **C 2 1975 Blah Blah**

1. You may assume correct formatting, but codes may be invalid. You must handle an **invalid command code** (e.g., 'X' in the above example), **invalid movie type** (e.g., 'Y'), **invalid customer ID**

(i.e., not found. For example, 9999), and **invalid movie** (i.e., not found. For example, classic movie in month 2 of 1975 with a "Blah Blah" title). For bad data, discard the line and notify users.

2. The MediaType is currently on 'D', but may be expanded in the future.
3. The **B** and **R** commands identify the movie using the two sorting attributes, using comma or space to separate them as in the movie data file.
 - Comedy: B 1234 D F Pirates of the Caribbean, 2003
 - Drama: B 1234 D D Steven Spielberg, Schindler's List,
 - Classics: R 1234 D C 9 1938 Katherine Hepburn

Overall Requirements

1. Do not print output for successful '**B**' or '**R**' commands, but print error messages for incorrect data and/or incorrect command.
2. Output for '**H**' and '**I**' commands should be neatly formatted with one line per item/transaction.
3. '**I**' should output all **Comedy** movies, then all **Dramas**, then all **Classics**. Each category of movies should be ordered according to the sorting criteria discussed above. The data should include how many movies are borrowed and how many remain.
4. '**H**' should show a list of DVD transactions of a customer in chronological order (latest to earliest) and specify whether the movie was borrowed or returned.
5. You are required to use **at least one hash table** in this assignment. There are actually multiple places where they could be used. You can use STL map, vector and other data structures, but you must have one hash table that you implement yourself. The hash table you implement does not need to be extensive, minimum features to store key-id pairs and retrieve them in $O(1)$ time is fine.
6. You must use inheritance and have polymorphic functions. If you find you're using templates a lot, run it by me, as this assignment is designed for you to practice using inheritance and polymorphism.
7. This assignment is to be fully object-oriented so when you store multiple pieces of information, it would be stored in an object. Strings are only to be used in a primitive sense, for example, one name, one title, one of anything. Do NOT build long strings that hold complex information.

The structure of your classes and taking advantage of the design patterns is especially important for this assignment. You need to follow the basic design principles, have it extendable, efficient, well-documented, etc.

Design Documentation

Your design should document the work that needs to be done to complete the assignment. It should be a complete and clear description of how the program is organized. The more time you spend on your design, the less you will spend coding, debugging, and modifying.

Your design should include (**at least**) the following components in this order:

1. **UML Class diagram:** Only include class names (OK to include more details for classes, but not necessary) You can use a program or neatly draw it by hand. Use simple UML notation for your class diagrams: arrow for inheritance (is-a); line with diamond for composition (has-a or part-of). Composition (has-a or part-of) is a special kind of association. Some things in the UML of the design are best shown as classes. For example, a List class may be implemented as an STL list or even as an array, but in the design, it is critical the reader knows it is a collection.
2. **Class Interactions:** Explanation of how classes interact (Show the program flow, the methods invoked, for standard Use Cases such as insert or borrow.). For example a Borrow operation would on a high level entail: finding the customer; finding the movie; associating customer with movie, so the movie becomes a part of the customer's history; associating movie with customer so it is known that the movie is borrowed by the customer and that the movie has been borrowed. Give objects and methods called, i.e., pseudocode, so that later you can take the design and implement the functionality. Note that conciseness is as important as clarity, i.e., do not burden the reader with excessive documentation or unnecessary detail.
3. **Main:** Include a description of what would be called from main.cpp, which should be very, very short.
4. **.h Files:** Since this is a design, you might not include all, or any, parameters in methods. Or they might change. That's okay, but overall the design is to be sound. You do not need to include .h files of classes you will not implement, of the extensions beyond the assignment specifications, but you must include a description of those classes. Order the files properly, i.e., put the most important classes first, put parent classes before children classes.

The design will take considerable effort. A useful approach is to imagine that you are writing a design that someone else will need to implement and extend. Document your design as you would want someone else to document for you. Design beyond the specifications (what you turn in). Design so that the answer to all these questions is yes (maintaining good design principles).

1. Can your design be extended beyond the specifications given here?
2. Could you easily add new videos or DVDs to your design?
3. Can you easily add other categories of videos or DVDs?
4. Could you easily add new categories of media to your design, for example, music?
5. Could you expand to check out other kinds of items, for example VCRs or DVD players?
6. Could you easily add new operations to your design?
7. Could you incorporate time, for example, a due date for borrowed items?
8. Could you easily add an additional store, or handle a chain of stores?

Your design can go beyond the scope of these specifications (and you won't need to implement extensions). Thinking of possible extensions in advance often improves the design.

(when class is held face-to-face)

Bring a hardcopy of your design to class. We will have in-class design reviews and then your hard copy will be turned in.

Hints and Suggestions

Come to class, participate in the design discussion, review the design patterns and class code. There are lots of subtleties in this assignment. Expect to revise and refactor your code multiple times.

Good software practice says that each class should get its own .h and .cpp file. For this assignment, you will likely have over 10 classes, so for ease of programming you can put closely related classes into a common .h file (and a common .cpp) file. This is only for very closely related classes.

Submission

There are lots of ways to implement this assignment, so submitting your project in a way that can be easily compiled and run is crucial.

1. While you do not have to explicitly test all your functions using assert, you should still write tests as you develop your program. You should not output any extraneous information or the output for successful 'B' or 'R' commands. Test cases can have a single start/end printout to indicate the test has been performed. When your program is compiled and run, **the last test must use data4commands.txt, data4movies.txt and data4customers.txt** files. Previous tests can use your custom files. Make sure all the necessary files for running your project are in your repository.
2. Your program must compile and run using **"./create-output.sh"** on CSS Linux machines. You can modify **create-output.sh** as needed. You can take advantage of Github Actions output which compiles and runs your program as you develop, but CSS Linux machines are the agreed "common ground" for programs. (maximum mark 50% if project does not compile/run using **create-output.sh**)
3. You must have output.txt produced using **"./create-output.sh > output.txt 2>&1"**. (maximum mark 75% if output.txt is missing)
4. You must have self-evaluation.md file that follows the provided template