PROG 10004 - Midterm Exam - Practical

Practical Section - Rules

During the practical section of the exam all students are expected to follow the rules outlined below:

1. The practical part consists of writing a program in the Visual Studio IDE.
2. Time Allocated: 2 hours
3. All questions must be solved **individually**
4. **Students are not allowed to use multiple monitors**.
5. The exam will be recorded using Teams Meetings. The students **are required to share their Screen throughout the exam**.
6. All means of information and data sharing (except MS-Teams) must be **EXITED**: IM, e-mail, chat programs, ftp, virtual machines, **DropBox, SkyDrive, GoogleDrive** etc. *Do not just sign off*.
7. Students are *not allowed to post any information on the Internet or send any information to anyone for any reason through any means*.
8. **Students are not allowed to access any code generative models such as Chatgpt or homework help sites like Chegg**
9. If you need any clarification, you must ask your professor.
10. Use of laptops and other materials:
    a. In the practical section, you are allowed to use any materials including slides, books, assignments/programs created by you, or the solutions provided by the professor, and Python Documentation.
    b. Code will be submitted in SLATE.
    c. **Students are not allowed to share solution / ideas**
11. **Keep committing the changes to git repository after every 15 minutes.**
12. Besides implementing the required functionality submissions, you are required to use the correct coding conventions used in class, professional organization of the code, alignment, clarity of names, are all going to be part of the evaluation. *Comments are recommended but not required*.

I have read, understood, and agreed to follow these rules as well as Sheridan's policies and procedures on academic honesty as per https://policy.sheridanc.on.ca/dotNet/documents/?docid=917&mode=view

Student Name: _____ Student Number: _____

Detailed Requirements (Practical Section (50% of Total-2 hours)

You are required to create a Python application for managing a movie rental store, as per the instructions given below. The UML Class Diagram corresponding to the requirements is provided at the end of the problem statement. **The names of the attributes and the methods in the class diagram are just placeholders. Pick a name to represent each element, following the best practices and the naming conventions established in the lectures. Add comments in your code to clarify your assumptions.**

Part I (5): Project Setup

Open the folder containing the starter code in VS Code and initialize a git repository in the folder.

1. Add a class "MovieStore" to the module "movieStore.py."
2. Import the "movieStore" module in "movieRentalApp.py."
3. Create a method "run" in the "MovieRentalApp" class that prints a welcome message: "Movie Rental Store - your name" and creates an instance of the "MovieStore" class.
4. Write the code to invoke the "run" method.
5. Ensure the program compiles and runs and commit the changes with the message "**Project Setup Complete."**

Version Control: Make sure to commit the changes at each stage indicated in the requirements given below.

Part II (25): Entity Class

Modify the MovieStore class as described below (see UML Class Diagram). Choose good names that respect best practices and naming conventions. ***If required, additional attributes/properties/methods, may be added to the module/class.***

1. **(7 Points)** Create an initializer for the "MovieStore" class to initialize field variables. The initializer should accept parameters (wherever required) to set the attributes given below:
   a. The store name (string)
   b. The total number of movies available (int)
   c. The number of available movies (int), initialized to the total number of movies
   d. The rental fee per movie (float)
   e. The late fee per day (float)

Commit the changes done so far, with a message **"MovieStore constructor created"**

2. **(7 Points)** Define accessors and mutators:
   a. Define accessors (getters) for the store name, the total number of movies, the number of available movies, the rental fee per movie, and the late fee per day.
   b. Define a mutator (setter) for the rental fee, **making sure that new values lesser than 0 are rejected.**

Commit the changes done so far, with a message **"Created accessors and mutator for event attributes."**

3. **(3 Points)** Create a method "calculateLateFee" that calculates and returns the late fee amount as the late fee per day multiplied by the number of days a movie is overdue.
4. **(6 Points)** Create a method "rentMovie". The method should:
    a. accepts the number of movies to rent and updates the number of available movies accordingly. ***Make sure to handle cases where the requested number of movies is greater than the available movies.***
    b. Return True if the movies rented successfully, otherwise return False.
5. **(2 Points)** Create a method "returnMovie" that accepts the number of movies returned and updates the number of available movies. Additionally, if any movie is returned after a specified threshold number of days (i.e. number of overdue days >0), the late fee should be calculated and displayed.

Commit the changes done so far, with a message **"Created methods for renting and returning movies"**

Part III (20): User Interaction

1. **(8 Points)** Create an object of the "MovieStore" class by accepting the required data from the end-user.
2. **(2 Points)** Print the details of the movie store, including the store name, total number of movies, rental fee per movie, and late fee per day.

    Commit the changes done so far, with a message **"Created instance of MovieStore"**
3. **(8 Points)** Ask the user if they wish to rent or return movies.
    a. For renting, accept the number of movies to rent and update the available movies. Display the total fee due if renting was successful. If renting is not successful display, show an appropriate error message.
    b. For returning, accept the number of movies returned, and if any movies returns are overdue, calculate and display the late fee.
4. **(2 Points)** Update the movie rental fee to 75% of the original price.

Commit the changes done so far, with a message **"User interaction implemented."**

Part IV (5): Bonus

In each iteration, present a menu to the user with three options:

1. Rent Movies
2. Return Movies
3. Calculate Late Fees
4. Exit

Carry out the corresponding actions based on the user's choice.

Commit the changes done so far, with a message **"Bonus completed."**

NOTE: To be eligible for the bonus you must earn at least **85%** on the practical section without the bonus. **If without the bonus your grade is less than 85% your bonus marks will not be taken in consideration**.

Appendix 1: UML Class Diagram

Figure 1: Minimal Design. Attributes and/or methods may be added as neede