PROG 10004 - Final 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)

Problem Statement: Event Management System

For this exam, you are required to create a Python application for **managing events**. 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. You can state any assumptions you make, as Python comments in the submitted answer(s)**.

Part I (5): Project Setup: The starter code provided to you contains the following modules: eventManagerApp.py, event.py, and specialEvent.py containing the classes described below:

- ***ProductManagerApp***: implements the user interactivity and defines the entry point of the application.
- ***Event***: represents the details of an event.
- ***SpecialEvent***: a subclass of *Event.*

Modify the given application according to the following requirements:

1. Initialize a Git repository in the folder containing the starter code.
2. Import the event and specialEvent modules in eventManagerApp.py. Also, import the event module in specialEvent.py.
3. Write code in eventManagerApp.py to print a welcome message: "Event Manager – [Your Name]" in the run method. Call the run method.
4. Ensure the program runs without errors. Commit the changes with the message "Project Setup Complete."
5. Create a PRIVATE repository on GitHub and provide read access to sara.shaheen@sheridancollege.ca Push the changes to the remote repository. Creating a public repository will be treated as a **BREACH OF ACADEMIC INTEGRITY.**

Version Control (5): Make sure to commit changes at each stage indicated in the requirements given below.

Part II (20): Entity Classes. Modify the given classes as described below (see UML Class Diagram). Choose good names that respect best practices and naming conventions established in the course. ***If required, additional attributes/properties/methods, may be added to the given modules/classes.***

1. **Class Event:**
   a. **(1 point)** For the Event class create a constructor (initializer) that create fields for event including eventID (string e.g., 'E0001'), eventLocation (string e.g., 'redPalace'), category (string e.g., 'Wedding'),
   b. **(2 points)** Define the accessor method for the event ID.
   c. **(4 points)** Define the mutator method for the event location. The method should make sure that the location is not blank and contains only alphabets. Throw ValueError with appropriate message if the validation checks fail.
   d. (**1 point**) Use the mutator in the initializer, to enforce validation on location when an event instance is created (replace the currently given assignment statement for location).
   e. (**2.5 points**) Define a method, **calculateEventFee** that returns the fee of the event and payable tax @ 13.5%.

Commit the changes done so far, with a message "Implemented class Event"

2. Class SpecialEvent: This class is to manage VIP events.

a. (**1 points**) Inherit class SpecialEvent from Event.

b. (**5 points**) Create the constructor for SpecialEvent to set all the inherited fields using parameters. Also, define a field variable for the special event category.

c. **(3.5 points)** Override the method **calculateEventFee** to consider the special VIP events fee in calculation.

Commit the changes done so far, with a message “Implemented class SpecialEvent”

Part III (17): User Interaction Modify the run method in eventManagerApp.py to interact with the user for maintaining event details.

1. **(1 point)** Create an empty collection (list or dictionary) to hold instances of Event and its subclass SpecialEvent.
2. **(3 points)** Create a loop that presents menu options to the user in eventManagerApp.py. **Consider defining a method in eventManagerApp for action related to each option**
    1. Add a regular event to the collection.
    2. Add a special event to the collection.
    3. Calculate and display event fees for all events.
    4. Exit.

Commit changes with the message "Main menu created."

3. **(3 points)** Write an exception handler to prevent the app from crashing when an exception is encountered.
4. **(3points)** When the user selects 1, prompt them to input data for a regular event, create an instance of Event, and add it to the collection, add it to the empty collection you created in point 1.
5. **(3 points)** When the user selects 2, prompt them to input data for a special event, create an instance of SpecialEvent, and add it to the collection, add it to the empty collection you created in point 1.
6. **(3 points)** When the user selects 3, calculate and display event fees for all events in the collection.
7. **(1 point)** When the user selects 4, exit the loop.

Commit the changes done so far, with a message “User interaction implemented.”

Part IV (8): Data Persistence

1. **(3 points)** Write a method, saveDataToFile, in eventManagerApp.py, which saves the data contained in the entire collection of events and special events to a CSV/JSON file. This method should be called before the application terminates.

2. **(3 points)** Write a method, loadDataFromFile, in eventManagerApp.py, which reads data from the file created in Part IV back into the collection. Handle exceptions appropriately if the file is not found or if there's an error while reading an event. Log errors and proceed with reading the rest of the events.

3. **(2 points)** Update the run method to call loadDataFromFile at the beginning to load existing data from the file into the collection and proceed with user input after that.

Commit the changes with the message "Data Persistence Complete."

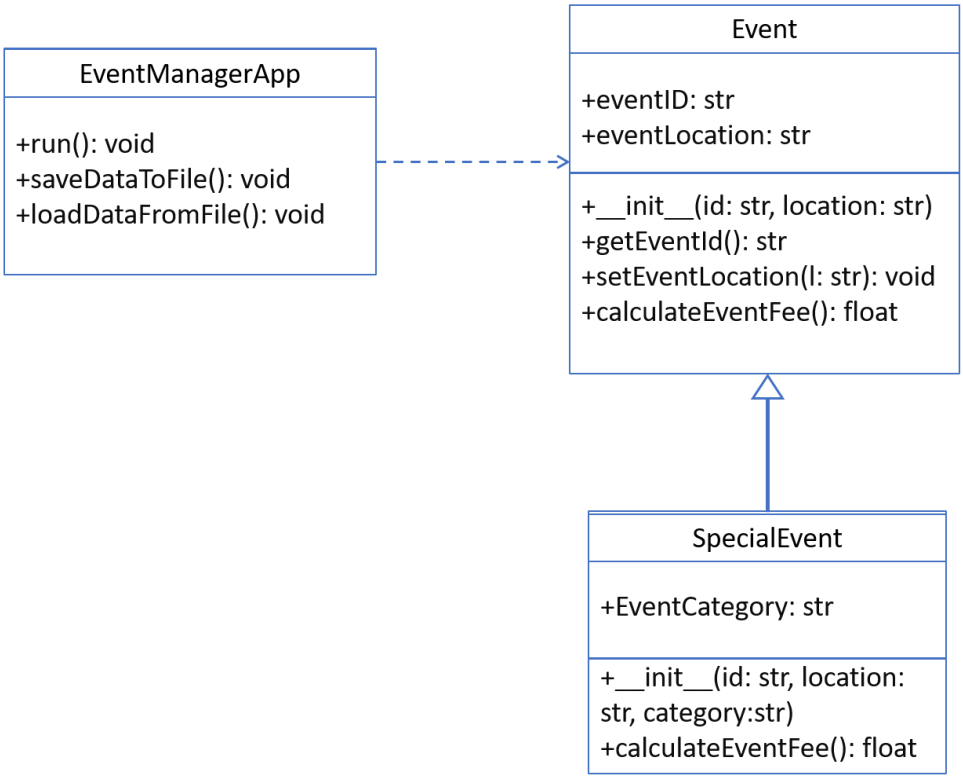

Figure1: Event Manager UML Diagram