HOMEWORK ASSIGNMENT #3

CS 586; Spring 2024
Due Date: **April 16, 2024**
Late homework 50% off
After **April 22,** the homework assignment will not be accepted.

This is an **individual** assignment. **Identical or similar** solutions will be penalized.

Submission: All homework assignments must be submitted on the Blackboard. The submission **must** be as one PDF file (otherwise, a 10% penalty will be applied).

PROBLEM #1 (35 points)
Consider the problem of designing a system using **Pipes and Filters** architecture. The system should provide the following functionality:

- Read student's test answers together with student's IDs.
- Read student's names together with their IDs.
- Read the correct answers for the test.
- Compute test scores.
- Compute test statistics: mean and standard deviation
- Report test scores in **descending score order** with student names and their IDs.
- Report test statistics

It was decided to use a Pipe and Filter architecture using the existing filters. The following existing filters are available:
Filter #1: this filter reads student's test answers together with student's IDs.
Filter #2: this filter reads correct answers for the test.
Filter #3: this filter reads student's names together with their IDs.
Filter #4: this filter computes test scores.
Filter #5: this filter prints test scores with student names in the order as they are read from an input pipe.

Part A:

Provide the Pipe and Filter architecture for the Grader system. In your design, you should use all existing filters. If necessary, introduce additional Filters in your design and describe their responsibilities. Show your Pipe and Filter architecture as a directed graph consisting of Filters as nodes and Pipes as edges in the graph.

Part B:

1. For the Pipe and Filter architecture of Part A, it is assumed that filters have different properties as shown below:
    a. Filter #1: active filter with buffered output pipe
    b. Filter #2: passive filter with un-buffered push pipes
    c. Filter #3: passive filter with un-buffered pull-out pipes
    d. Filter #4: passive filter with un-buffered pull-out pipes
    e. Filter #5: active filter with buffered input pipes
2. Use object-oriented design to refine your design. Each filter should be represented by a class. Provide a class diagram for your design. For each class identify operations supported by the class and its attributes. Describe each operation using **pseudo-code**. In your design, filters **should not be aware** of other filters.
3. Provide a sequence diagram for a typical execution of the system based on the class diagram of Step 2.

PROBLEM #2 (30 points)
There exist two inventory systems/servers (*Server-S1* and *Server-S2*) that maintain information about machine parts in warehouses, i.e., they keep track of the number of machine parts in warehouses. Machine parts may be added or removed from the warehouses and this should be reflected in the inventory system. Both servers (inventory systems) support the following services:

Services supported by **Server-S1**:

```
void AddPart (string w, string p)        //add part p to warehouse w, where p is a part ID
void DeletePart (string w, string p)     //deletes part p from warehouse w
int GetNumParts (string p)               //returns the total number of part p in all warehouses
int IsPart (string p)                    //returns 1, if part p exists; returns 0, otherwise
```

Services supported by **Server-S2**:

```
void Insert_Part(string p, string w)       //adds part p to warehouse w
void Remove_Part(string p, string w)       //deletes part p from warehouse w
int Get_Num_Of_Parts(string p)             //returns the total number of part p in all warehouses
int Is_Part(string p)                      //returns 1, if part p exists; returns 0, otherwise
```

The goal is to combine both inventory systems and provide a uniform interface to perform operations on both existing inventory systems using the **Strict Layered architecture**. The following top-layer interface should be provided:

```
void Add_Part (string p, string w)          //adds part p to warehouse w
void Remove_Part (string p, string w)       //deletes part p from warehouse w
int GetNumOfParts (string p)                //returns the total number of part p in all warehouses
int Is_Part (string p)                      //returns 1, if part p exists; returns 0, otherwise
RegisterCriticalPart(string p, int minimumlevel)
UnRegisterCriticalPart(string p)
ShowCriticalParts()
```

Notice that the top layer provides three additional services (*RegisterCriticalPart()*, *UnRegisterCriticalPart()*, and *ShowCriticalParts()*) that are not provided by the existing inventory systems. These services allow watching the status of critical parts. The user/application can register, *RegisterCriticalPart(string p, int minimumlevel)*, a critical part by providing its minimum level, i.e., a minimal number of parts of a specified part that should be present in all warehouses. When the number of parts of a critical part (a registered part) reaches the level below the minimum level, the system should store, e.g., in a buffer, the current status (number of parts) of the critical part. The current status of all critical parts whose level is below the minimum level can be displayed by invoking *ShowCriticalParts()* service. The service *UnRegisterCriticalPart()* allows removing a specified part from a list of critical parts.

Major assumptions for the design:

1. Users/applications that use the top layer interface should have the impression that there exists only one inventory system.
2. The bottom layer is represented by both inventory systems (i.e., inventory systems S1 and S2).
3. Both inventory systems should not be modified.
4. Your design should contain at least **four** layers. For each layer identify operations provided by the layer and its data structure(s).
5. Show call relationships between services of adjacent layers.
6. Each layer should be encapsulated in a class and represented by an object.
7. Provide a class diagram for the combined system. For each class list all operations supported by the class and major data structures. Briefly describe each operation in each class using **pseudo-code**.

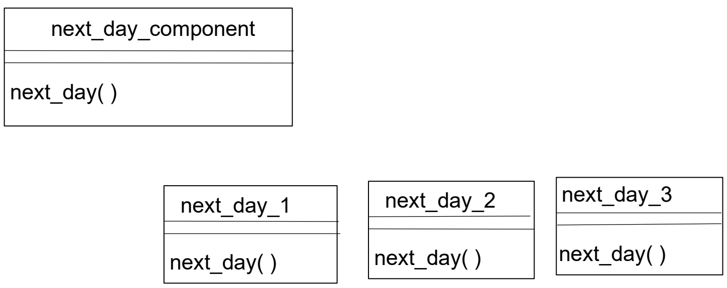
PROBLEM #3 (35 points)
Suppose that we would like to use a **fault-tolerant architecture** for *next_day_component* that is supposed to compute the date of the next day. The *next_day* operation of this component accepts as an input three integer variables: *month*, *day*, and *year,* and returns the date of the day after the input date. An interface of the *next_day* operation is as follows:

void next_day (**in**: int month, int day, int year; **out**: int month1, int day1, int year1)

where parameters *month*, *day*, and *year* represent an input date, and parameters *month1*, *day1*, and *year1* represent an output date (the date of the day after the input date).

For example, for the input date: *month*=12, *day*=15, *year*=2022, the *next_day* operation returns *month1*=12, *day1*=16, *year1*=2022.

Suppose that three versions of the *next_day* component have been implemented using different algorithms. Different versions are represented by classes: *next_day_1*, *next_day_2,* and *next_day_3*.



Provide two designs for the *next_day component* using the following types of fault-tolerant software architectures:

1. N-Version architecture
2. Recovery-Block architecture

For each design provide:

1. A class diagram. For each class identify operations supported by the class and its attributes. Specify in detail each operation using **pseudo-code** (you do not need to specify operations *next_day()* of the *next_day_i* classes; only new operations need to be specified).

2. A sequence diagram representing a typical execution of the *next_day component*.

Background for the *next_day* component:
Since a year is 365.2422 days long, leap years are used for the "extra day" problem. If we declared a leap year every fourth year, there would be a slight error. The Gregorian Calendar resolves this by adjusting leap years to century years. Thus a year is a leap year if it is divisible by 4, unless it is a century year. Century years are leap years only if they are multiples of 400, so 1992, 1996, and 2000 are leap years, while the year 1900 is a common year.