# HOMEWORK ASSIGNMENT #3

CS 586; Spring 2017
Due Date: **Monday, April 17, 2017**
Late homework 50% off
After **April 22,** the homework assignment will not be accepted.

The **hardcopy** of the assignment must be submitted. Electronic submissions are not acceptable. Notice that the Blackboard homework assignment submissions are only considered as a proof of submission on time (before the deadline). If the hardcopy is different than the electronic version submitted on the Blackboard, then **50% penalty** will be applied. If the assignment is submitted on the Blackboard on time, we must receive the hardcopy of the assignment by **noon on Wednesday, April 19**. If the hardcopy is received after this deadline, **20% 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 correct answers for the test.
- Compute test grades (A, B, C, E).
- Compute test statistics: # of A grades, # of B grades, etc.
- Report test's grades in a **descending grade order** with student names, i.e., first all students with grade A are listed, then all students with grade B, etc.
- 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 computes test grades with student ID.
Filter #4: this filter prints test grades with student names in the order as they are read from an input pipe.
Filter #5: this filter computes the test statistics

**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, assume that each pipe is an **un-buffered** pipe and all filters are **passive** filters with **push** 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.
3. Provide a sequence diagram for a typical execution of the system based on the class diagram of Step 2.

**Part C:**

1. For the Pipe and Filter architecture of Part A, assume that each pipe is a **buffered** pipe and all filters are **active** filters.
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.
3. Provide a sequence diagram for a typical execution of the system based on the class diagram of Step 2.

**PROBLEM #2 (35 points):**

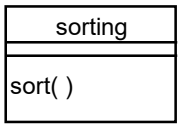
Suppose that we would like to use a fault-tolerant architecture for *sorting* component that is supposed to sort a list of integers in an ascending order considering only positive integers. The *sort()* operation of this component accepts as an input an integer parameter *n* and an integer array *L*. The output parameter is an integer array *SL* that contains the sorted list of positive integers. An interface of the *sort()* operation is as follows:

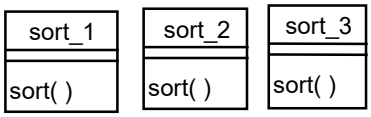
void sort (**in** int n, int L[]; **out** int m, int SL[])

*L* is an array of integers.
*n* is the number of elements in list *L*.
*SL* is an array of sorted (in ascending order) positive integers by the *sort()* operation.
*m* is the number of elements in list *SL*.
Notice: *L* and *n* are inputs to the *sort()* operation. *SL* and *m* are outputs of the *sort()* operation.

For example, for the following input:
n=7; L=(-1, 7, -2, 0, 4, 2, 1)
The component computes the following output:
m=4; SL=(1, 2, 4, 7)

Suppose that three versions of the *sorting* component have been implemented using different algorithms. Different versions are represented by classes: *sort_1*, *sort_2,* and *sort_3*.




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

1. N-version architecture
2. Recovery-Block architecture
3. N-Self Checking architecture

For each design provide 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 *sort()* of the *sort_i* classes; only new operations need to be specified). For each design provide a sequence diagram representing a typical execution of the *sorting* component.

**PROBLEM #3 (30 points):**
There exist two inventory software systems/servers S1 and S2 that maintain information about books in several warehouses, i.e., the functional core of each system is to keep track of books in warehouses. Books may be added or removed from the warehouses. Both systems/servers support the following services:

Services supported by **server S1**:

| | |
|---|---|
| void AddBooks (ISBN string, warehouse int, n int) | //adds *n* books identified by ISBN to a *warehouse* |
| void DeleteBooks (ISBN string, warehouse int, n int) | //deletes *n* books identified by ISBN from a *warehouse* |
| int GetNumBooks (ISBN string, warehouse int) | //returns the total number of an *ISBN* book in a *warehouse* |
| int IsBook (ISBN string, warehouse int) | //returns 1, if an *ISBN* book exists in a *warehouse*; returns 0, otherwise |

Services supported by **server-S2**:

| | |
|---|---|
| void InsertBook (ISBM string, warehouse int) | //adds a book to a *warehouse* |
| void RemoveBook (ISBN string, warehouse int) | //deletes a book from a *warehouse* |
| int GetNumBooks (ISBN string, warehouse int) | //returns the total number of an *ISBN* book in a *warehouse* |
| int IsBook (ISBN string, warehouse) | //returns 1, if an *ISBN* book exists in a *warehouse*; returns 0, otherwise |

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

| | |
|---|---|
| void InsertBooks (ISBN string, warehouse int, n int) | //adds *n* books identified by ISBN to a *warehouse* |
| void DeleteBooks (ISBN string, warehouse int, n int) | //deletes *n* books identified by ISBN from a *warehouse* |
| int GetNumBooks (ISBN string, warehouse int) | //returns the total number of an *ISBN* book in a *warehouse* |
| int IsBook (ISBN string, warehouse int) | //returns 1, if an *ISBN* book exists in a *warehouse*; returns 0, otherwise |
| int TotalNumBooks (ISBN string) | //returns the total number of an *ISBN* book in all warehouses |
| int IsBook (ISBN string) | // returns 1, if an *ISBN* book exists in any warehouse; returns 0, otherwise |
| int WhichWarehouse(ISBN string) | //returns a warehouse ID where an *ISBN* book is stored |

Major assumptions for the design:

1. Users/applications that use the top layer interface should have an 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.
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.