**Home exercise 8**

**Course name: Object oriented programming and design for engineering**

**Course number: 157109**

**Subject: Software frameworks – Dependency Injection, Aspect Oriented programming**

In this exercise we measure the runtime of different sorting algorithms.

You are provided with the interface SortingAlgorithm:

**public interface** SortingAlgorithm <T **extends** Comparable<T>>{  
 **void** sort(T[] array);  
}

and four algorithms implementing the interface:

* MergeSort
* QuickSort
* BubbleSort
* InsertionSort

You must not change these provided classes at all.

In addition, you are provided with the class AlgorithmRunner that runs a quadratic time-complexity algorithm and a linearithmic () time-complexity and in addition two randomly selected algorithms. For part A of this exercise the algorithms are set inside the class itself.

# Part A: Using Aspect Oriented Programming

We would like to measure the runtime of the algorithms and print additional information about the execution. Since time measurement and logging are not part of the responsibility of the AlgorithmRunner, we would like to have maximal flexibility with minimal changes to this class.

We would like to print **after *all*** sorting algorithms finished running the **total** runtime (in milliseconds) of **all** sorting algorithms (i.e., all sort functions that were called. In addition, we would like to print for **each** executed algorithm (i.e., an algorithm which its sort function was called at least once) the following information:

* How many times was the sort function called
* How much time in total it took to run, out of the total time of all sorting functions.

If, for example the sort function of QuickSort was called twice, the first call took 300ms and the second 312ms, we will print that this function ran 2 times and 612ms in total.

Advice: Add an aspect that measures the time of execution for any function called "sort" and prints information about the executed functions. For simplicity, use only the @Before and @After advices.

Write another advice that is executed based on the overall execution of all sort functions. Find which function should correspond to this advice.

Important: in this part you are not allowed to change any of the provided classes. You may only add new classes.

Possible output of MainApp (It is possible that each execution will result in a different output, depending on the random values generated for **randomAlgorithm1** and for **randomAlgorithm2**. For example, it is possible that the output will relate to MergeSort and will not relate to InsertionSort). Make sure your output has the same format as in this example.

Total time of running all sort functions was <timing> ms

In detail:  
Function sort in QuickSort ran <number> times and took in total <timing> ms

Function sort in BubbleSort ran <number> times and took in total <timing> ms

Function sort in InsertionSort ran <number> times and took in total <timing> ms

# Part B: Using Dependency Injection

In part A we violated several design principles:

* AlgorithmRunner should not be open to modification every time we change an algorithm or the array size (OCP violation)
* AlgorithmRunner should not be responsible for instantiating the algorithms but only for running them (SRP violation)
* AlgorithmRunner should not depend on concrete algorithms. Instead, it should depend on interfaces (DIP violation)

In order to fix these violations, we shall use Dependency Injection to initialize AlgorithmRunner’s fields. This way, it would be possible to change the sorting algorithms and the array size without opening AlgorithmRunner to modifications. Instead, the system initialization will be defined in a separate class (e.g., in the main class).

Change the declaration of the AlgorithmRunner class variables so that they do not contain initialization:

SortingAlgorithm<Integer> **quadraticAlgorithm**;  
SortingAlgorithm<Integer> **nlognAlgorithm**;

SortingAlgorithm<Integer> **randomAlgorithm1**;   
SortingAlgorithm<Integer> **randomAlgorithm2**;  
  
**int numberOfElements**;

Move the function makeRandomSortingAlgorithm to the MainApp class and change it so that it does not use the “new” operation. Instead use the WeldContainer to initialize the chosen algorithm.

Use dependency injection to update MainApp, so that it is possible to select which algorithms will be used by AlgorithmRunner and the array size.

Important: You may not add any code to AlgorithmRunner. You may only add annotations to this class. You may add code to the MainApp class and add more annotation/aspect classes as you need. Moreover, changes to MainApp should deal exclusively with dependency injection and not contain any time analysis code. Tha MainApp code must not contain the “new” operation.

Advice: In the preliminary work file you were provided with the code for the Student package, including the Logging aspect that prints a message every time a function is called. If you project contains this aspect it might impact the execution. Hence you are advised to remove the Logging aspect from the project (if you added it earlier). In addition, you are advised to use 'rebuild project' in order to let the framework identify any change you make to the aspects.

Alternatively, you could open a brand new project (according to the instructions in the preliminary work file)

You should submit the AlgorithmRunner, MainApp classes and also any aspect/annotation files you wrote. Add these files and a submitters.txt file to a zip archive and name this file based on the submitters IDs.

# Appendix: Frequently Asked questions

* Make sure you work with Intellij Ultimate (not the free version). Make sure you installed AspectJ according to the instructions in the preliminary work files.
* The Weld execution emits red lines as output, ending with “Weld shut down”. This is part of any normal execution and should not be regarded as an error.
* In some cases, the @Named annotation does not work. Use the alternative, shown in class, involving defining a custom annotation.
* When defining a custom annotation, remember to annotate it with @Qualifier and @Retention(RUNTIME)
* In the producer declaration, define the return value as SortingAlgorithm<Integer>, not SortingAlgorithm.
* Do not change any of the sorting algorithm files. You will not submit any of them.
* In some cases students experiment with many alternatives, which make their code full of obsolete code. Go over your files and make sure they do not contain errors from past experiments.
* If the execution emits an error about Ambiguous dependencies or Unsatisfied dependencies, it means the framework is working, but the injection is not well defined.
* Every time you change an annotation in a producer, make sure you change it also in the injection point.
* Every time you change the project, and before you run it, rebuild it using Build |Rebuild project

Good luck!