SOFT 437: Winter 2015

Project 3 due on March 16, 2015

Instrumentation

Date Assigned: March 6, 2015
Date Due: March 16, 2015 by 11:30 PM

[Note: This project is to be done individually – no partners]

In this project, you will design an instrumentation class to monitor the performance of a program and uncover any performance problems. This instrumentation class is to be developed in Java. The class tracks the time consumption of blocks, function calls, and statements. For example, the following code shows how to use an instance `ins` of the instrumentation class.

```
ins.startTiming("loop");
for (int i=0; i<100000; i++)
{
      int j=i;
}
ins->stopTiming("loop");
```

A sample output of the instrumentation log file looks like the following:

```
STARTTIMING: loop
STOPTIMING: loop 2ms
```

Class Description:

The instrumentation class contains the following methods:

`startTiming(string comment)` – start timing a method, or block of code

`stopTiming(string comment)` – stop timing a method, or block of code

`comment(string comment)` – place an additional comment in the output file (i.e. the instrumentation log)

`dump(string filename)` – dump() writes formatted/indented pairs of startTiming/stopTiming statements to a human readable log file. If you provide NULL as filename then a logfile will be created with the timestamp instrumentationddyyMMhhmmss.log. You should call this method ONCE at the end of a program that uses the instrumentation class.

`activate(bool onoff)` – activates/deactivates instrumentation. You should call this ONCE at the beginning of any program that uses the instrumentation class. Pass in false and the calls to all instrumentation methods will return immediately with no effect. The class behaves as if activate(false) was called by default.

Here's an example which ties all of these methods together:

```
...
Instrumentation ins=Instrumentation.Instance();

void main()
{
   ins.activate(true);
   ins.startTiming("main");
   ...
   ins.startTiming("loop");
   for (int i=0; i<5; i++)
   {
      doSomeStuff();
   }
   ins.stopTiming("loop");
   ...
   ins.comment("this is an example of a comment!");
   ins.stopTiming("main");
   ins.dump();

}

void doSomeStuff();
{
   ins.startTiming("doSomeStuff()");

   System.out.println("Hello there!");

   Ins.stopTiming("doSomeStuff()");
}
```

Will produce the following instrumentation log output:

```
STARTTIMING: main
|  STARTTIMING: loop
|  |  STARTTIMING: doSomeStuff()
|  |  STOPTIMING: doSomeStuff() 1ms
|  |  STARTTIMING: doSomeStuff()
|  |  STOPTIMING: doSomeStuff() 1ms
|  |  STATTIMING: doSomeStuff()
|  |  STOPTIMING: doSomeStuff() 1ms
|  |  STARTTIMING: doSomeStuff()
|  |  STOPTIMING: doSomeStuff() 1ms
|  |  STARTTIMING: doSomeStuff()
|  |  STOPTIMING: doSomeStuff() 1ms
|  STOPTIMING: loop 5ms
|  COMMENT: this is an example of a comment!
STOPTIMING: main 5ms
TOTAL TIME: 5ms
```

You instrumentation class should follow the Singleton Design Pattern as documented in "Design Patterns: Elements of Reusable Object-Oriented Software" by Gamma, Helm, Johnson and Vlissides, ISBN 0-201-63361-2. A singleton is a class for which there can be only one instance. The class ITSELF manages this through the use of a protected constructor and a static "Instance()" method. Logging classes are well suited to the singleton design pattern. More references can be found at http://userpages.umbc.edu/~tarr/dp/lectures/Singleton.pdf

Strategies for instrumentation:

1. startTiming/stopTiming statements should be paired in your code. They MUST match up in instrumented code. For example this is a good use of the instrumentation methods:

```
ins.startTiming("loop");
for (int i=0; i<100000; i++)
{
      int j=i;
}
ins->stopTiming("loop");
```

The following is an example of incorrect use:

```
int truncateToPositive(double theValue) {

   ins.startTiming("truncateToPositive");

   if (theValue < 0)  {
           System.out.println( "That number was negative");
           return 0;
   }

  int result = theValue;

  ins.stopTiming("truncateToPositive");
  return result;

}
```

The previous example is an incorrect use because if theValue is negative, then the stopTiming() for this method will not be called. The solution to this problem is to make a slight modification to the code:

```
int truncateToPositive(double theValue) {
   ins.startTiming("truncateToPositive");

   int result = 0;

   if (theValue < 0)  {
      System.out.println("That number was negative");
        result = 0;
   } else {
        result = theValue;
   }
```

```
  ins.stopTiming("truncateToPositive");
  return result;
}
```

2. Measure methods or functions with a pair of startTiming/stopTiming statements. One at the entry of the method and one at EACH exit of the method/function.
3. When measuring a method or function, make sure you include the time it takes to execute the return expression. For example:

```
bool isPrime(long theNumber)
{
   ins.startTiming("isPrime");

   if ((theNumber % 2) == 0) {
      System.out.println("even numbers are never prime!!!");
   }

   ins.stopTiming("isPrime");

   return lookupPrime(theNumber);

}
```

won't measure the cost of the return expression. Using a "result" variable pattern solves this problem:

```
bool isPrime(long theNumber)
{
   ins.startTiming("isPrime");

   if ((theNumber % 2) == 0) {
      System.out.println("even numbers are never prime!!!);
  }

  bool result = lookupPrime(theNumber);

  ins.stopTiming("isPrime");

  return result;
}
```

4. Don't measure the cost of frequently called methods, which are called thousands or tens of thousands of times within a relatively small time period. The overhead of your instrumentation object will begin to affect your measurements. For example, the following code illustrated the copy from one string, arg1, to another string, arg2.

```
String my_strcpy(String arg1, String arg2)
{
   ins.startTiming("my_strcpy()");
   ... code for the function ...
```

```
    ins.stopTiming("my_strcpy()");
}
```

If this method is called thousands of times, your instrumentation will affect the performance of the method. A good strategy for measuring the performance of frequently called functions is to write a loop with makes a fixed number of calls to the function, then measure the time it takes for the entire loop to execute. For example:

```
ins.startTiming("loop 10000 times for my_strcpy()");
char a[100] = "hello ";
for (int i=0; i<10000; i++) {
   my_strcpy(a,a);
}
ins.stopTiming("loop 10000 times for my_strcpy()");
```

5. Don't put a startTiming/stopTiming wrapper around the CALL of a method you have already instrumented.

6. Watch out when you instrument code that prompts the user for input. Users TAKE A LONG TIME relatively speaking to provide input to the system. These thinking delays can distract you from the true source of performance problems in the system.

Test drive your instrumentation class:

Once you are happy with your instrumentation class, you should try it out by answering these questions.

1) What is the overhead of the instrumentation itself? There are two resources consumed when calling startTiming and stopTiming which make up the bulk of the time consumed by your instrumentation calls. Design several tests which measure the overhead of these resources and report on them.
2) Download BubbleSort at http://www.cs.ubc.ca/spider/harrison/Java/BubbleSort2Algorithm.java.html, QuickSort at http://www.cs.ubc.ca/spider/harrison/Java/QSortAlgorithm.java.html, SortAlgorithm at http://www.cs.ubc.ca/spider/harrison/Java/SortAlgorithm.java.html, modify the code necessary to make them run.
3) Write a test class to run the BubbleSort class and QuickSort class. In the test class, write a method, called populateArray(), which generates an array and fills the array with random numbers from 1 to 99999. Then have another two methods to sort the array using the BubbleSort and the QuickSort algorithms.
4) What are the running times for the BubbleSort, and the QuickSort algorithms? Which algorithm takes more time? Why?
5) What are the running times for the main and populateArray functions?
6) Add instrumentation to the BubbleSort class. Based on your answer to (1), predict the new running time for the BubbleSort algorithm. Run the program. Is the new running time inline with the instrumentation overhead you predicted? Why or why not?
7) Add instrumentation to QuickSort. Run the program. Is the new running time inline with the instrumentation overhead you predicted? Is there a significant difference between the QuickSort running time and the BubbleSort running time?

DELIVERABLES:

* Submit a Zip file via moodle by March 16 11:30 PM, including
    - Your instrumentation class
    - Your test class
    - Your bubble sort class integrated with the instrumentation
    - Your quick sort class integrated with the instrumentation
    - Partial instrumentation logs for the test drive section of the project
* Submit a hard copy of your written report to class, your report should explain:
    - What instrumentation is
    - Define the purpose of your experiment
    - List the experiment variables
    - The design of your test cases to make your measurement result reproducible and representative
    - The analysis of the instrumentation logs and answers to each of the questions posed in the test drive section of the project