**SEG2105 Assignment 1**

**Course**: SEG2105

**Professor**: Miguel Garzon

**Group**: 19

**Students Name (Section, ID):** Devon Knight (C, 300325269); Cattuong Le (B, 300001854)

**Repository**: https://github.com/SEG2105BC-uOttawa/assignment-1-assignment1-300325369\_300001854.git

**Part 1: PointCP**

**E26. Hypothesis and Pro’s and Con’s**

**Table 1:** Hypothesis of the efficiency of all designs of PointCP

| **Design** | **Simplicity of Code**  **(Simple, medium, intermediate)** | **Efficiency when Creating Instances** | **Efficiency for Computations** | **Amount of Memory Used** |
| --- | --- | --- | --- | --- |
| **1** | Medium: Need method to check for flag | Moderate: Initially, only one coordinate type is stored along with the flag when creating an instance | Low or High: Dependent on what matches what is currently stored. If cartesian coordinates are stored and request getX(), values can be returned right away. But if getRho() called, system must compute values from stored Cartesian coordinates | High amount of memory used - large amount of code |
| **2** | Simple: Always deals with polar coordinates | Low: only stores polar coordinates so cartesian must be computed into polar | Moderate: Efficient for polar operations but will always require conversion for cartesian coordinates | Low: Stores only one set of coordinates |
| **3** | Simple: Always deals with cartesian coordinates | Low: only stores cartesian coordinates so polar must be computed into polar | Moderate: Efficient for cartesian operations but will always require conversion for polar coordinates | Low: Stores only one set of coordinates |
| **4** | Intermediate: more instance variables | High: efficient computationally in terms of both coordinate systems since they have separate instance variables and can simply be returned | High: Both coordinate systems are always available in memory | High: Stores both sets of coordinates |
| **5** | Simple: both coordinate systems are in the design | High - more efficient than designs 1, 2, and 3. With designs 2 and 3 implemented as subclasses, the appropriate coordinate system can be selected | High: Both coordinate systems are always available | High: both coordinate systems are incorporated |

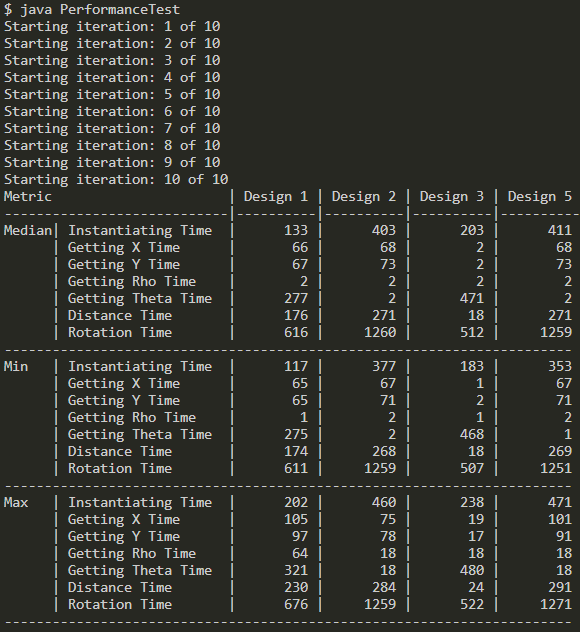
**E28, E29, and E30: Performance Analysis of Designs 1, 2, 3, and 5**

Tests were written in a sub-folder containing copies of all designs in order to run in a contained environment with an automated unit test. The test was written so that an array of 10,000,000 is created of random doubles corresponding to random polar or cartesian flags. Next, to calculate maximum, minimum, and median efficiency times, ann array for instantiatinmg, and for each method was created to hold as many efficiency times as there are maximum iterations. For each iteration, all designs were tested for the time it takes to be instatiated and run their class methods. The times were computed in milliseconds by taking the difference between the time before and after the test was run. The arrays of times were sorted and the maximum, minimum and median times were identified from benign the first, last, and middle times respectively. The results were recorded in a summary table in the terminal (**Fig. 1**).

The results (**Fig. 1**) indicate that our hypothesis stated earlier in Table 1 are close with soem amendments.

1. Design 1 showed the fastest median time for instantiation.
2. Design 2 showed the fastest median time for get requests for polar coordiantes
3. Design 3 showed the fastest median time for get requests for cartesian coordinates.
4. Design 3 showed the fastest median time for calculating distance and rotation time.

These results align with our hypothesis as design 1 will instantiate by simply holding cartesian or polar coordinates without any conversions int he constructor. Design 2’s results were anticipated as polar coordinates are handled right away and a get request is simply returning the value in memory. Design 3’s results were anticipated as cartesian coordinates are handled right away and a get request is simply returning the value in memory in addition to calculating distance and rotation as these methods depend on the readily accessible cartesian coordiantes in design 3’s memory. What was not anticipated in our hypothesis was the Rho get request being extremely efficient throughout the designs. We specuulate that this is because the Rho conversion is following basic math principles of square root and powers whereas converting to Theta requires an additional math conversion method of toDegrees().

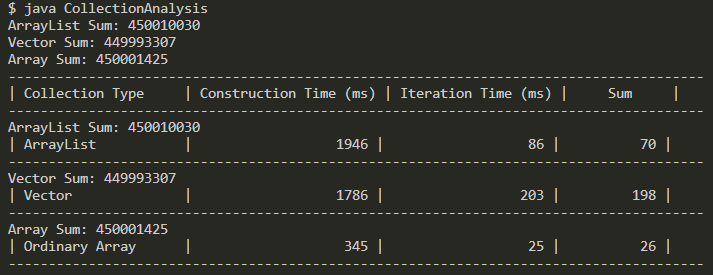


**Figure 1**: Summary Table of Performance Analysis. Time in milliseconds.

**Part 2: Arrays**

During this experiment, it was found that ordinary arrays have a efficient construction time, iteration time, and summation time when running the methods to do so (**Fig. 2**). However the same cannot be said for Vectors and ArrayLists. Vectors are the slowest in terms of iteration time and summation time, whereas the ArrayList is slowest in Consturcitontime by a mere 160 milliseconds. This may be due to the fact that during construction, these objects need to resolve memory allocation and pointers to create their data structures. Whereas a ordinary array has the memory allocated accordingly and occupies that memory immediately as each memory allocation is adjacent to the next. ArrayList is more efficient than a Vector int emrs of iteration time and summation time because a Vector defaults to doubling the size of its array, while the ArrayList increases its array size by 50 percent.

In the Java documentation: a Vector and an ArrayList are almost equivalent with a difference being that accessing a Vector is a synchronized process, whereas access to an ArrayList is not. This means is that only one thread can call methods on a Vector at a time, slowing its efficiency, whereas this is not the case for ArrayList.

****

**Figure 2:** Summary Table of Performance Analysis of Vectors, ArrayList and Array. Time in milliseconds.