**Harkaran Rangi**

**John Bute**

**Part 1**

E26

|  |  |  |
| --- | --- | --- |
|  | Pros | Cons |
| Design 1 | * Takes less memory due to there being only 1 instance | * Takes more time due to needing to convert from one system to another * Has more storage space |
| Design 2 | * Easier to code * Takes less time to create an instance * Less storage space needed | * Takes more time to get cartesian coordinates |
| Design 3 | * Easier to code * Takes less time to create an instance * Less storage space needed | * Takes more time to get polar coordinates |
| Design 5 | * We have subclasses that represent the two different systems so takes less time to create an instance * Easily interchangeable | * Harder to code due to there being subclasses |

**E30**

In order to conduct a proper performance analysis of designs 2, 3, and 5 of PointCP, we decided to test each one of their methods separately. We utilized the java.util.Random package to configure the parameters of each instantiation for each design. Thus, every time a method is tested, it will have different parameters. Secondly, we used System.Nanosecond() to record the time when the test began and when the test ended, then obtain the difference between the end time and start time and based the performance of the method off of said difference.

Here is a sample output from running our tests:

A screenshot of a computer screen

Description automatically generated

|  |  |  |  |
| --- | --- | --- | --- |
| Method | Design 2 | Design 3 | Design 5 |
| Instantiation | Max:143  Median:94  Min:80 | Max:145  Median:85  Min:83 | Max:114  Median:86  Min:83 |
| getX() | Max:146  Median:109  Min:103 | Max:84  Median:84  Min:84 | Max:87  Median:84  Min:82 |
| getY() | Max:106  Median:103  Min:102 | Max:91  Median85:  Min:82 | Max:82  Median:81  Min:79 |
| getRho | Max:85  Median:83  Min:80 | Max:98  Median:96  Min:91 | Max:92  Median:84  Min:82 |
| getTheta | Max:91  Median:85  Min:79 | Max:104  Median:99  Min:92 | Max:107  Median:102  Min:96 |
| convertStorageToCartesian() | Max:118  Median:117  Min:115 | Max:88  Median:86  Min: 83 | Max:93  Median:90  Min:88 |
| convertStorageToPolar() | Max:101  Median:90  Min:86 | Max:119  Median:116  Min:102 | Max:114  Median:109  Min:105 |
| rotatePoint() | Max:272  Median:268  Min:262 | Max:181  Median:175  Min:172 | Max:177  Median:176  Min:175 |
| getDistance() | Max:208  Median:197  Min:193 | Max:161  Median:159  Min:156 | Max:191  Median:190  Min:187 |
| toString() | Max:609  Median:602  Min:582 | Max:169  Median:166  Min:164 | Max:142  Median:139  Min:137 |

**Conclusion**

Instantiation: We found that while design 3 has a slightly inferior median instantiation time than design 5, we see that design 5 has a much tighter bound than design 3, thus confirming our hypothesis that since design 5 is the parent class of the subclasses in design 2 and 3, it takes less time to instantiate.

getX() and getY(): Firstly, design2 seemed to be the slowest when it came to getting the X and Y coordinate, since it had to convert its polar coordinates into an X and Y coordinate. However, both design 3 and 5 showed very similar results, which is to be expected as design 3 stores the x and Y coordinate while design 5 is the parent class of design 3.

getRho() and getTheta(): Design 2 and 5 yielded similar results for getRho(), but for getTheta, design 2 seemed to gain the upper hand. We can conclude that design 2 is more performant than design 5 when it comes to getting Rho and theta since it stores the values in its class. However, design 5 is far more performant than design 3, which lacks behind from the rest since it has to convert its values.

convertStorageToCartesian() and convertStorageToPolar(): The tests from these two methods showed us that design2 was the best at converting to polar since all it had to do was return what was stored in the object. Meanwhile design 3 was the worst, since it had to use mathematical operators to convert its coordinates into rho and theta. By contrast, design 3 was the best at converting to cartesian since it just had to return what was in its storage, while design 2 had to, just like design 3, use mathematical operators to convert its coordinates into x and y. However, design 5 is in the middle of the pack for both methods, thus showcasing its versatility. Thus, if you know if you will just be working in cartesian or in polar coordinates exclusively, then design 2 and design 3 are respectively the best choice. However, if you plan on switching from one to the other, than design 5 is the best choice.

rotatePoint(): Design 3 and 5 shared very similar results since polar coordinates are way more efficient at rotating points than cartesian coordinates, which explains why design 2 lags far behind.

getDistance():However, for getDistance(), design 3 seems to be the leader of the pack, while design 5 is slightly better than design 2.

toString(): design 5 takes the lead here, while design 2’s performance is very lackluster.

In conclusion, we consider design 5 to be the most performant out of the three for one singly reason: its methods can replicate the methods from design 2 and 3 that are the most efficient while taking up the same amount of storage space. However, it is worth noting that the complexity of the code of design 5 is significantly higher than that of design 3 and 2 since you have to write 3 classes (PointCP5, PointCP3, PointCP2)instead of 1 (PointCP2 or PointCP3).

**PART 2**

For part 2 we had to compute the performance of ArrayList, Vectors, and ordinary arrays. We calculated their construction time and iteration time with a collection size of 150000000. Here is the result of each data structure’s construction time and iteration time over 10 iterations.

|  |  |  |  |
| --- | --- | --- | --- |
| ITERATION # | Array List Construction Time/ Iteration Time (ms) | Vector Construction Time/ Iteration Time (ms) | Array Construction Time/ Iteration Time (ms) |
| 1 | 4111/772 | 4314/827 | 3685/62 |
| 2 | 4191/1114 | 4480/1071 | 3693/89 |
| 3 | 4151/826 | 4432/1303 | 4368/62 |
| 4 | 3824/880 | 3886/1009 | 2376/61 |
| 5 | 3868/730 | 3896/775 | 3382/81 |
| 6 | 3980/1077 | 3956/1382 | 2673/99 |
| 7 | 3874/1104 | 3989/775 | 2993/64 |
| 8 | 4239/633 | 4963/807 | 3088/61 |
| 9 | 4048/1088 | 4636/1068 | 2591/60 |
| 10 | 4281/683 | 4567/826 | 3252/66 |

Iteration Time:

At first glance, we notice that the iteration time for both Array List and Vectors are very similar since they both utilize iterators, while array has a much faster iteration time since it has a simple for loop. Furthermore, by accessing the first element of the array, we cache the entire array. Thus, we can sum elements extremely fast.

Construction Time:

Secondly, the construction times for ArrayList and Vectors are similar (with ArrayList being slightly better), since they dynamically resize as elements are added. However, it must be noted that by initializing ArrayList and Vectors at the collection size allows it to run slightly faster. Meanwhile, the array has the fastest construction time since it is created at full size from the start.

Recommendation:

For iteration and element access, the ordinary array is much faster than ArrayList and vectors as it allows for direct access to its elements and a cache-friendly structure. However, if you need to use an iterator-like interface, then Vectors and ArrayLists are good choices, but there will be a performance trade-off to keep in mind. For construction performance, if you do not know the size of the collection, then it is wiser to use an ArrayList or Vector data structure as they dynamically resize, unlike ordinary arrays. However, if you do know the exact size of the collection you are dealing with, then ordinary arrays will have the fastest construction time.