E26: Advantages and Disadvantages of the designs

Designs	Brief description	Advantages	Disadvantages
Design 2 - PointCP2	Stores Polar only and computes Cartesian	• Getting polar coordinates is fast; • Efficient when creating an instance by inputting polar coordinates. • If compared to design 1 (PointCP original), stores only 2 variables, hence less memory is required. • The majority of code is simple.	• Getting Cartesian coordinates is slow. • The functions that require Cartesian coordinates such as getDistance and Rotate are expected to be slow due to required computation before these methods can be implemented. • Some memory is used to store the coordinates. • Inefficient when an instance created with cartesian coordinates (slow because of computation of rho and theta)
Design 3 - PointCP3	Stores Cartesian only and computes Polar	• Getting Cartesian is fast; Methods that utilize Cartesian coordinates are expected to be fast. • Efficient when creating an instance by giving cartesian coordinates • Compared to original design 1 PointCP, less memory is required • Simple Code	• Getting Polar coordinates is expected to be slower; • Some memory is used to store the coordinates. • Inefficient when creating an instance with Polar coordinates (slower);
Design 5 - PointCP5	Is an abstract superclass for subclasses Design 2 and Design 3.	• Defines the methods and enforces the subclasses to implement them. • Improves code reusability • Allows to hide some code due to abstraction property	• Cannot be inherited multiple times. • As per assignment specifications the computation and getting the coordinates depend on the concrete classes. Hence, the abstract class is not expected to reduce the runtime.

E28-E30: Performance Analysis

Operations		Time for Design 2 *(milliseconds)*	Time for Design 3 *(milliseconds)*	Time for Design 5 (Concrete 2) *(milliseconds)*	Time for Design 5 (Concrete 3) *(milliseconds)*
Construction by giving Cartesian coordinates	**Ave**	**1801.9**	**29.7**	**1882.2**	**30.2**
	Min	1553	27	1597	28
	Max	2138	41	2234	37
Construction by giving Polar	**Ave**	**28.6**	**3836.4**	**29.5**	**3897.1**
	Min	27	3573	27	3609
	Max	32	4111	38	4152
getX()	**Ave**	**728.5**	**12.1**	**1130.3**	**20.6**
	Min	655	10	1107	16
	Max	971	14	1297	33
getY()	**Ave**	**921.8**	**10**	**1111.1**	**11.8**
	Min	900	6	1105	10
	Max	948	26	1127	15
getRho()	**Ave**	**7.5**	**11.4**	**12.9**	**15**
	Min	7	7	12	11
	Max	8	19	15	28
getTheta()	**Ave**	**7**	**218.6**	**12.4**	**211.3**
	Min	6	209	11	206
	Max	8	240	14	244
getDistance()	**Ave**	**4407.1**	**14.6**	**4456.2**	**29.8**
	Min	4393	9	4388	18
	Max	4421	41	4596	88
rotatePoint()	**Ave**	**8758.4**	**4624.5**	**6838.7**	**4664.9**
	Min	8726	4596	6774	4618
	Max	8804	4678	7050	4748

Description of how the test were done

First, the designs 2,3 and 5 were implemented by modifying PointCP according to the table provided in the Book. The PointCPTest was also modified to test each design. After confirming proper implementation of each design, the time efficiency analysis was performed.

For time efficiency analysis, we created several classes:

- **Test** - the class generates a large number of random instances for each design. In addition this class generates an array of *point B* necessary for testing the distance between points. It contains the method that constructs the points.
- **PointCPTesting** - the class contains methods to test each design's function and to track the time taken for each method. Here is an example of how a function typical for each design is tested:

```
//GETX TEST
long s_getX = System.currentTimeMillis();
for (int i=0; i<arrayCP2.length;i++ ){
    arrayCP2[i].getX();
}
long e_getX = System.currentTimeMillis();
System.out.println("Time taken for getX for " +arrayCP2[0].getClass()+" (milliseconds): "+ (e_getX-s_getX));
```

- **Analysis** - class that contains the Main and performs the time analysis. In this class, the user can input the number of instances to be created, as well as the number of tests to be run. It is important to note that the user can test all the designs in one test run for several times, or test each design separately for several times by simply removing the testing of other designs. Our group tested each design separately for 10 times.

To analyze the performance of designs, we have created 2 800 000 instances of points for each design and timed the methods of each design. The reason we chose 2 800 000 is to ensure that tests for each design run for at least 10 seconds to get a good measure of performance.

A total of 10 runs were performed to generate 10 time results per each design and each method. The code for this part of analysis can be found in *Analysis.java.*

Here is an example of the output for 1 test run for design 2 (PointCP2):

```
START OF TEST FOR PointCP2

Time taken to construct PointCP2 (milliseconds) giving Polar coordinates: 27
Time taken to construct PointCP2 (milliseconds) giving Cartesian coordinates: 1553
Time taken for getX for class PointCP2 (milliseconds): 697
Time taken for getY for class PointCP2 (milliseconds): 902
Time taken for getRho for class PointCP2 (milliseconds): 8
Time taken for getTheta for class PointCP2 (milliseconds): 7
Time taken for getDistance for class PointCP2 (milliseconds): 4407
Time taken for rotatePoint for class PointCP2 (milliseconds): 8804
```

At first we tried to run 10 test runs by simply specifying the test run number in Analysis.java. However, It was observed that the time was significantly different for the second test run of PointCP2. After some research, we believe such discrepancy is related to JVM performance. To further investigate this issue and possibly counteract, we have run tests one by one (by changing the functions we call in Analysis) for 10 times in the terminal and scripted the outputs into the files. These outputs can be found in **terminalTimeOutput.txt.* We have confirmed that running the Analysis in the terminal loop provided more consistent results as opposed to looping inside Analysis. Hence, for average, minimum and maximum results we used **terminalTimeOutput.txt.*

Once the output for 10 runs for each design were completed. The average, minimum and maximum were reported in the table above.

Discussion of Results

Design 2

As can be observed from the table, the fastest operations for Design 2 were constructed when Polar coordinates are given, getRho(), and getTheta, whereas getX() and getY() operations were more time-consuming. This is expected, because class PointCP2 stores Rho and Theta, hence they can be accessed directly without computation. However, to get x and y coordinates, PointCP2 has to perform computations which increases the run time.

When considering the functions getDistance() and rotate(), they are also considerably slower when compared to other operations or to the same methods in Design 3, and Design 5 with concrete class PointCP3. This is also expected since both of these functions call getY() and getX() in the body. In addition, rotate() returns PointCP2 that is constructed using cartesian coordinates (which was shown to be very slow, about 1801.9 ms).

Design 3

As shown in the table, the fastest operations included construction with Cartesian coordinates, getX(), getY(), getRho(), getDistance(). The slowest in increasing order were getTheta(), constructing with Polar coordinates, and rotate(). The results for getX() and getY() were expected as PointCP3 stores x and y coordinates, hence they can be accessed without any computation.

The outcome for getRho() was somewhat unexpected as we initially suggested that it would be considerably slower than getting the stored variables and perhaps it would be similar to getTheta() in terms of running time. This prompted us to further investigate and we came to a conclusion that math operations performed for getting Rho are faster than the operations necessary to compute Theta. In addition, the construction of PointCP3 with Polar coordinates uses Math.toRadians() and sin/cos functions to get x and y; therefore making the construction slower compared to the one which uses cartesian coordinates.

The difference observed in math operations is summarized in the figure below. We tested the time (milliseconds) needed to perform the math functions used in the designs for 1 000 000 random numbers.

```
Math.toDegrees(Math.atan2(a, b)) takes 53
Math.sqrt(Math.pow(a, 2) + Math.pow(b, 2)) takes 6
(Math.sin(Math.toRadians(a)) * b) takes 400
```

It should be noted that getDistance() was fast compared to design 2; and such outcome is attributed to the fact that cartesian coordinates are stored within PointCP3. As for rotate, the method computes trigonometric function and constructs a Point, thus taking more time than other operations. Design 2 rotatePoint() is two times slower than Design 3 rotatePoint() and this is because the Design 2 rotatePoint(), apart from calling getX() and getY(), constructs PointCP2 with Cartesian coordinates (known to be slow from the performance analysis).

Design 3 outperformed design 2 in such methods: getX, getY, Construction with Cartesian coordinates, getDistance(), and rotatePoint().

Design 5

Since PointCP5 is abstract and computation of coordinates depends on concrete classes, we have tested PointCP5 with concrete class PointCP2_5 and with concrete class PointCP3_5. Overall, PointCP3_5 follows the same time trend as PointCP3. This is expected as the computation of coordinates depends on the concrete class used. PointCP2_5 also has a similar time trend as PointCP2: same operations are slowest in both designs.

We implemented rotatePoint() and getDistance() in the abstract superclass and compared the performance of these methods in each design. According to our design 5, PointCP5 abstract superclass implements the rotatePoint() that returns PointCP3 constructed with Cartesian coordinates; this approach is expected to be faster than rotatePoint() in design 2 where a PointCP2 was returned after construction with cartesian coordinates (time-consuming as discussed in Design 2). Therefore, we observe that rotatePoint() in design 5 is slightly faster than in design 2.

Appendix

Test results

PointCP2	Const P	Const C	getX	getY	getRho	getTheta	getDistance	rotatePoint
1	32	1703	694	941	7	7	4401	8737
2	27	1553	697	902	8	7	4407	8804
3	29	1893	681	907	8	7	4395	8797
4	29	2136	694	922	7	7	4407	8747
5	27	1628	732	906	7	7	4402	8738
6	30	1571	971	935	8	7	4419	8765
7	29	1758	655	943	8	6	4418	8768
8	28	1592	707	914	7	7	4408	8726
9	28	2138	675	900	7	8	4421	8733
10	27	2047	779	948	8	7	4393	8769
Avg	**28.6**	**1801.9**	**728.5**	**921.8**	**7.5**	**7**	**4407.1**	**8758.4**
Min	**27**	**1553**	**655**	**900**	**7**	**6**	**4393**	**8726**
Max	**32**	**2138**	**971**	**948**	**8**	**8**	**4421**	**8804**

PointCP3	Const P	Const C	getX	getY	getRho	getTheta	getDistance	rotatePoint
1	3768	33	13	7	19	213	10	4678
2	4090	27	14	7	19	213	13	4668
3	4111	28	14	7	7	240	13	4655
4	3730	29	12	14	13	217	12	4609
5	3633	28	10	11	7	218	12	4608
6	4064	28	10	6	14	212	9	4608
7	3573	28	11	8	10	216	15	4596
8	4074	41	13	7	9	209	41	4601
9	3719	28	10	26	7	218	11	4617
10	3602	27	14	7	9	230	10	4605
Avg	**3836.4**	**29.7**	**12.1**	**10**	**11.4**	**218.6**	**14.6**	**4624.5**
Min	**3573**	**27**	**10**	**6**	**7**	**209**	**9**	**4596**
Max	**4111**	**41**	**14**	**26**	**19**	**240**	**41**	**4678**

PointCP2_5	Const P	Const C	getX	getY	getRho	getTheta	getDistance	rotatePoint
1	38	1914	1107	1106	12	11	4388	6843
2	29	1609	1117	1117	15	12	4497	6785
3	29	2123	1110	1105	12	13	4426	6818
4	28	1909	1109	1105	13	13	4392	6813
5	30	1597	1297	1114	13	14	4451	6787
6	27	2234	1113	1109	12	12	4402	6877
7	28	1650	1110	1111	13	11	4596	6774
8	30	2095	1108	1106	12	12	4586	7050
9	28	1651	1119	1127	14	12	4423	6855
10	28	2040	1113	1111	13	14	4401	6785
Avg	**29.5**	**1882.2**	**1130.3**	**1111.1**	**12.9**	**12.4**	**4456.2**	**6838.7**
Min	**27**	**1597**	**1107**	**1105**	**12**	**11**	**4388**	**6774**
Max	**38**	**2234**	**1297**	**1127**	**15**	**14**	**4596**	**7050**

PointCP3_5	Const P	Const C	getX	getY	getRho	getTheta	getDistance	rotatePoint
1	3909	37	16	10	28	207	22	4625
2	4130	28	19	13	12	207	32	4704
3	3609	28	33	15	16	210	26	4748
4	4152	34	16	11	12	207	18	4670
5	3642	28	21	12	12	208	22	4663
6	4037	30	25	12	17	209	21	4653
7	4126	29	20	11	12	206	19	4677
8	3982	29	16	11	12	244	88	4659
9	3694	30	19	12	11	206	30	4618
10	3690	29	21	11	18	209	20	4632
Avg	**3897.1**	**30.2**	**20.6**	**11.8**	**15**	**211.3**	**29.8**	**4664.9**
Min	**3609**	**28**	**16**	**10**	**11**	**206**	**18**	**4618**
Max	**4152**	**37**	**33**	**15**	**28**	**244**	**88**	**4748**