SEG2105 Assignment 1

Natasa Bolic & Angeleeca Jocson

Question E26

Note: + represents an advantage and - represents a disadvantage.

	Simplicity of code	Efficiency when creating instances	Efficiency when doing computations that require both coordinate systems	Amount of memory used
Design 1: Store one type of coordinates using a single pair of instance variables, with a flag indicating which type is stored	- The code is not very simple, as it must take into account that the same pair of instance variables can represent two different things. This leads to extra boolean logic required to verify the type before the variables are used, making the code more complex.	- The constructor must verify whether the type flag has the correct input (if the flag is implemented as a character; this could be avoided if the flag was a boolean variable but that is less readable). + The constructor initiates the 3 instance variables, without any extra computations.	- Doing computations that require both coordinate systems is inefficient because it requires checking which values are currently stored and computing the values of the one that is not stored.	+ Only one pair of instance variables is needed even though both types of variables can be represented, meaning less memory is used. - There is an extra variable for the flag, which entails additional memory use.
Design 2: Store polar coordinates only	+ This code is simpler as the coder always knows what the pair of instance variables represent (polar coordinates).	+ The constructor initiates the 2 instance variables, without any extra computations.	- It is less efficient to do computations with cartesian coordinates because they are not stored so they must be calculated every time that they	+ Less memory is used because only one pair of instance variables is stored.

			are used.	
Design 3: Store cartesian coordinates only	+ This code is simpler as the coder always knows what the pair of instance variables represent (cartesian coordinates).	+ The constructor initiates the 2 instance variables, without any extra computations.	- It is less efficient to do computations with polar coordinates because they are not stored so they must be calculated every time that they are used.	+ Less memory is used because only one pair of instance variables is stored.
Design 4: Store both types of coordinates, using four instance variables	+ This code is simple because the user has variables representing all of the types, so no additional conversion calculations ever have to be done, and it is always clear which variables are being dealt with.	+ The constructor initiates the 4 instance variables, without any extra computations.	+ It is efficient to do calculations with both systems because both are stored, so they can be accessed directly.	- More memory is used to store all four instance variables.
Design 5: Abstract superclass with designs 2 and 3 as subclasses	- The code is less simple because 3 classes must be implemented and the inheritance relationship between them must be set up properly. + The code is more structured, and less cluttered because the superclass can	+ The constructor initiates the instance variables in the subclasses, without any extra computations.	- In order to do computations with both coordinate systems, two instances of classes must be created which is not initially efficient. However, once the two subclasses have been initialized, it is easy to access both types of	+ Less memory is used because only one pair of instance variables is stored.

	contain methods that are shared between both subclasses.		coordinates as they are both stored in their respective classes, but doing a computation on one of the points does not translate to the other point, so it must be repeated, making it less efficient.	

Question E28-E30

Note: each block contains data in the form 'Median Runtime; Max Runtime; Min Runtime', where the runtime is measured in milliseconds and summed over 100000000 iterations.

	Design 2	Design 3	Design 5	
			Design 2	Design 3
constructor	238; 273; 213	249; 293; 224	253; 263; 222	386; 436; 327
getX	1264; 1313; 1088	286; 327; 222	1302; 1424; 1062	338; 375; 276
getY	1914; 1950; 1647	306; 329; 250	1924; 2132; 1668	314; 347; 292
getRho	286; 301; 248	385; 431; 340	245; 295; 229	285; 367; 268
getTheta	285; 305; 246	8504; 9251; 7448	272; 296; 238	8084; 8641; 6924
convertStorageToCartesian	2579; 2956; 2386	N/A	3563; 3946; 3131	N/A
convertStorageToPolar	N/A	6696; 7625; 6256	N/A	7410; 8130; 6539
getDistance	5968; 6218;	262; 279; 242	4944; 5520	267; 300; 256

	Design 2	Design 3	Design 5	
			Design 2	Design 3
constructor	238; 273; 213	249; 293; 224	253; 263; 222	386; 436; 327
	5469		4396	
rotatePoint	8654; 9022; 7857	5518; 5912; 4548	8681; 9291; 7530	5213; 5476; 4396
toString	56904; 60461; 51581	55169; 62490; 52995	56924; 60708; 50331	57816; 61037; 51350

The tests were implemented by creating a test method for every method in the file in which timestamps are taken before and after calling the method to be tested. These timestamps are subtracted and the time elapsed is returned as a long value. Then in the main function, in a loop of 100000000 iterations, these methods are called on points generated with random double variables as their input. The sum of the time elapsed at each iteration for every method is calculated and printed as output. Below is a sample of the output (for TestRunTime3.java in the design3 folder).

```
C:\Users\Bolic\Documents\SEG2105\assignment1_300234878_300241734\pointcp>java design3/TestRunTime3
224
222
250
340
7448
6256
242
4548
52995
```

This process was repeated 5 times for every design, with the results pasted into an Excel file. Then the median, max and min values were called in Excel and formatted in the table above.

Comparing designs 2 and 3 with design 5 reveals that all values are comparable, meaning that having a superclass does not greatly improve or worsen the running time. This observation implies that we can choose to implement a superclass for better code structure without worrying about the efficiency of the code.

We note several other interesting observations from the results. Firstly, for design 2, the getX and getY methods have significantly larger running time than the methods getRho and getTheta. This is expected because rho and theta are stored but x and y must be computed. For design 3, it is the opposite since x and y are stored, but rho and theta are computed; however, the running time for getRho is more comparable to getX and getY, while getTheta is much larger. This suggests that the functions used to compute theta (Math.toDegrees and Math.atan2) may be more complex for the random input data used than the functions used to compute rho (Math.pow

and Math.sqrt). When comparing the two classes to each other, we observe that getX in design 2 is about 4 times the running time of getX in design 3, and getY in design 2 is about 6 times the running time in design3. Furthermore, getTheta in design 2 is almost 30 times the running time of getTheta in design 2 (a much larger magnitude increase), but getRho is comparable for both classes. Overall, if a problem requires the use of polar coordinates more frequently, it is more efficient to use design 2 and the same for cartesian coordinates and design 3. If both types are needed, it may make more sense to use design 2 because of the large increase of the running time of getTheta in design 3.

Secondly, the getDistance and rotatePoint methods are both faster for design 3 implementations than design 2 implementations because, as in design 1, both methods use getX and getY to compute the return value. As discussed in the first point, getX and getY are more efficient for design 3 because x and y can be accessed directly.

Thirdly, convertStorageToCartesian in design 2 performs better (approximately half the running time) than convertStorageToPolar in design 3 for all classes. This could be explained by the significantly large running time of getTheta in design 3 (of magnitude at least 4 times that of getY and getX in design 2), which is used to create and return an instance of PointCP2.

Fourthly, it is interesting to note that the toString methods have a drastically large running time compared to the other methods across all classes. This is due to java Strings being immutable, making their concatenation a slow process.