CMPT 213 Midterm

February 28, 2020

SOLUTION

DO NOT TURN THE PAGE UNTIL TOLD TO DO SO

You will have 50 minutes to complete this test once told to begin.

The test has 7 (multi-part) questions totalling 47 points. Points shown in square brackets.

Each question provides space for the answer, and your answer should appear in that space. **Multiple choice must be answered on bubble sheet** (detachable last page). Clarity and legibility count.

No aids are allowed: no crib notes, calculators, smart watches, textbooks, electronic aids, or dictionaries. If you do not understand the meaning of a word, ask the instructor or TA. **<u>Reread</u> the question carefully before asking questions during the exam.**

[10] Multiple Choice, worth **2 mark each.**

Circle the one correct answer.

1. Which of the following statements is true about static methods?
   a) Static methods cannot access static final fields.
   b) Static methods cannot be private.
   c) <u>Static methods cannot change instance data.</u>
   d) Static methods cannot call other static methods.
   e) None of the above; they are all false.

2. The *best reason* to make a class like `CarCollection` `Iterable` is...
   a) it makes `CarCollection` immutable.
   b) it allows `Collections.sort()` to be used on a `CarCollection`.
   c) **<u>it allows client code to use a for-each loop over a `CarCollection`.</u>**
   d) it allows `CarCollection` to reduce its coupling to `Iterator`.
   e) it allows client code to break `CarCollection`'s encapsulation and access its private fields.

3. To support sorting instances of a class by multiple different sort orders we use:
   a) `Sortable`
   b) <u>`Comparator`</u>
   c) `Object.compareTo()`
   d) `Comparable`
   e) Two of the above.

4. CRC cards are useful because they...
   a) formally record the decisions, and the reasons for these decisions, made during OOD.
   b) record the steps a user does while using the software.
   c) show the low-level implementation details of each class, such as method and field names.
   d) support refactoring code late in development.
   e) <u>none of the above.</u>

5. Which of the following best describes the argument in the following Java code? Assume `processFile()` is a function which accepts one argument: a `File` object.

```
processFile(new File("C:\output.txt"));
```

   a) <u>Anonymous object</u>
   b) Anonymous method
   c) Anonymous class
   d) Anonymous field
   e) Two of the above

Short answer questions. Points as indicated.

6. **[10]** Analyze the quality of the interface of `Maze`. Provide five (5) different ways that the quality of the interface can be improved, as discussed in lecture. For each improvement, **state the general name of the improvement**, and briefly **explain with an example** in the interface of where it applies. ***Do not use the same named improvement more than once.***

```
/**
 * Store a maze of walls for use in a cat-and-mouse game.
 */
public interface Maze {
      // Access size of the maze
      int getWidth();
      int getNumRows();

      // Access the elements in the maze; true indicates a wall.
      boolean[][] getMaze();

      // Count number of cheese pieces the mouse has eaten.
      int getNumCheeseEaten();

      // Checks validity of the row index.
      // Returns 1 for valid, 0 for invalid.
      int checkValidRowIndex(int row);

      // Make the cell a wall, returning if it was already a wall.
      boolean makeWall(int x, int y);
}
```

	Improvement name (use each at most once)	**Explain with an example in code of where to apply it**
a)	Cohesion	getNumCheeseEaten() does not fit.
b)	Consistency or One Name for an Idea	width vs row vs x/y.
c)	Clarity	checkValidRowIndex is not a yes/no name, plus should return true or false.
d)	Completeness or Convenience	Should have checkValidColIndex() too
e)	Encapsulation	Should not return the full maze.
f)	Command-Query	makeWall() both changes state and returns value.
g)	Not implement iterable	Implement iterable to access elements.
	Others that may apply...	
h)	Incomplete Object	

Marking Guide:
2 mark per row.

Only able to use each named improvement once.
Using one java line for two improvements OK.

7. An `Alien` is searching the `Universe` to find a good `Planet`. Complete the `Universe` and `Alien` classes below to help! Each `Planet` has a maximum temperature, and each `Alien` has a maximum temperature he/she likes.
   - `Planet` and `PlanetRater` are shown on page 7 (which is removable).
   - Include Java annotations as appropriate.
   - You need not write any comments or import statements.
   - **Use intention revealing names.**

   a) **[8]** Create a well-named new function in the `Universe` class which finds the "best" planet. Your method must accept one parameter, a `PlanetRater` object, and returns a `Planet` which is rated the highest by the `PlanetRater`. If there are no planets in the `Universe`, return `null`. **You must use a for-each loop.**
   Hint: Start by looking at the `PlanetRater` interface (page 7).

```
public class Universe {

   private List<Planet> planets = new ArrayList<>();

   // Create a new Universe, reading it in from the file fileName
   public static Universe makeFromFile(String fileName) {
       // CODE OMITTED; don't write it, assume it's here and works.
   }



1 {  public Planet getBestPlanet(PlanetRater rater) {
        if (planets.isEmpty()) {
1 {         return null;
        }

1 {     Planet best = planets.get(0);
2 {     for (Planet planet : planets) {

2 {         double score = rater.ratePlanet(planet);
            if (score > rater.ratePlanet(best)) {

0.5{            best = planet;
            }
        }
0.5{    return best;
     }

     // max 4 marks if not using PlanetRater to rate the planet.


}
```

b) **[12]** Complete the client code below for an `Alien` searching for the best `Planet`.

i. Have `main()` create a `Universe` from the filename "`data.txt`".
Hint: `Universe` already has a method to create a `Universe` from a file.

ii. Find the `Planet` in the `Universe` which has its maximum temperature closest to the `Alien`'s maximum temperature by calling the function you wrote in part a). Hints:

- Compute the absolute value of the *difference* between the planet's maximum temperature and the alien's desired maximum temperature.
  You may need: `double Math.abs(double);`
- The *score* is (1 / *difference*).

iii. If a best `Planet` is found, then print its name; otherwise print "`None`".

```
public class Alien {

  private static final int DESIRED_MAX = 45;

     public static void main(String[] args) {

2 {      Universe universe = Universe.makeFromFile("data.txt");

         // Full marks if they correctly use a Lambdas
1 {      PlanetRater rater = new PlanetRater() {
1 {         @Override
1 {         public double ratePlanet(Planet planet) {

               double difference =
3 {                    Math.abs(DESIRED_MAX - planet.getMaxTempC());
               return 1 / difference;

            }
         };

2 {      Planet best = universe.getBestPlanet(rater);
         if (best != null) {
2 {         System.out.println(best.getName());
         }
     }


}
```

c) **[2]** State just the name of the design pattern for...

i. the method `Universe.makeFromFile(...)`:

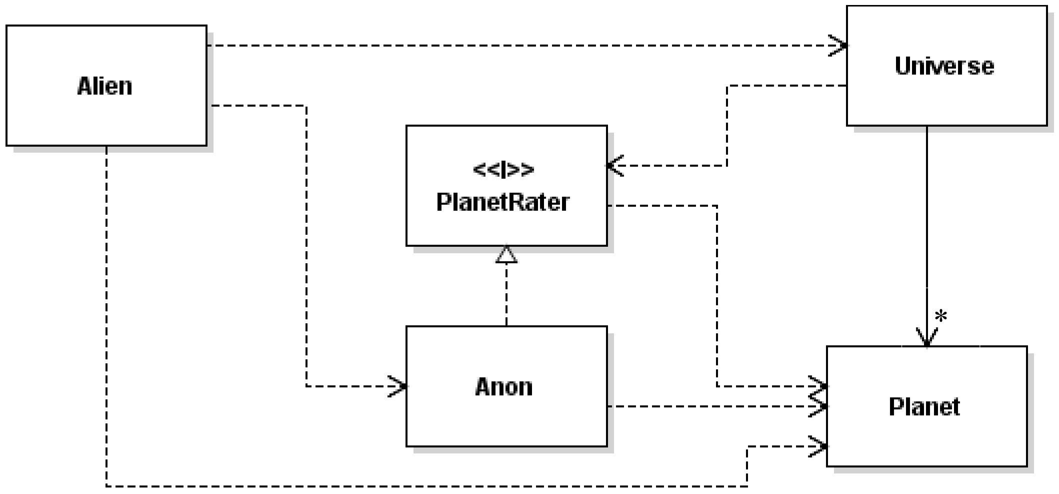
Static Factory Method **[1]**

ii. finding the best `Planet` with a `PlanetRater`:

Strategy Pattern (or just "Strategy") **[1]**

d) **[5]** Complete the UML class diagram below, as relating to the code in the rest of this question.

- Draw any anonymous classes (name them "Anon").
- Draw all **class relationships** and **decorations** (such as << ... >>) for the classes and interfaces shown.
- Do *not* show any fields or methods.



<u>Marking Guide:</u>
[0.5] Anon
[0.5] <<I>> or <<interface>> annotations

[4] Relationships
(-1 per missing has-a / implements)
(-.5 per missing dependency)
(-.5 if incorrect arrow/line)
(-.5 if missing * on Universe -> Planet)

OK if missing 1 dependency on Planet
OK if Alien depends on interface instead of Anon

Detachable Information Page

You may remove this page but it must be turned in with your exam.

```
/**
 * Store info about a planet.
 */
public class Planet {

      private String name;
      private double maxTempC;


      public Planet(String name, double maxTempC) {...}

      public String getName() {...}
      public double getMaxTempC() {...}
}
```

Code 1: `Planet` *class skeleton (method implementation omitted).*

```
/**
 * Rate how desirable a planet is.
 */
public interface PlanetRater {

      // Returns the score for a Planet:
      // larger numbers are more desirable (better).
      double ratePlanet(Planet planet);

}
```

Code 2: `PlanetRater` *interface for rating the desirability of a planet.*