Java Programming
MTech Preparatory Semester 2016
Practice programs – 4

A Taxi App

The objective is to get familiar with some of the issues in developing a simple web-based app and building on the object-oriented design approach. The app could be a desktop, browser or mobile app that connects to a web service. We will develop the app in stages, starting with a standalone program to a full multi-user app.

Consider a taxi service that runs a fleet of cars. Users can request a taxi at a specific location. The service allocates a car if one is available close by. The user also decides when and where to get off and release the cab.

The basic functionality of this service (managing the location and status of a set of taxis) is implemented in a **TaxiFleet** class, which is provided to you as a jar file. The public methods are listed later.

On top of this class, we would like you to develop a value-added service **TaxiService** that charges users for this service: this service charges the user for the time travelled in a taxi, as well as for requests that cannot be satisfied (to avoid frivolous users). To encourage people to join, the taxi service puts money into your account at the beginning and again for frequent users. TaxiService maintains the current balance of each user. TaxiService should build on TaxiFleet either by inheritance or composition. For simplicity, each client app will have its own version of TaxiService, and hence will effectively maintain the balance of only 1 user. However, you should set it up so that it can service multiple users.

DB required for multi user??

For multi user is the TaxiService class at server end??

Phase 1: A standalone app

Develop an interactive front-end to help the user request and release a taxi. This can be implemented in Swing, FX, or any other Java-based framework you would like, including Android. Swing is recommended unless you are already familiar with one of the other frameworks (or are keen on learning it in the next couple of days!) This should use the services of the class TaxiService that you implement (which will in turn use TaxiFleet, provided to you as a jar file, in the package **taxiservice**).

The jar doesnt have a TaxiFleet.class file

Functionally, the app has the following front-end:

1. A rectangular area that will show the current location of all the taxis in the fleet. Busy taxis are shown in red, and others in yellow (or your favourite colour). You can use a rectangle or circle to indicate a taxi. Also, try to show the id of the taxi next to its image, as well as the id of the rider if the taxi is busy. For simplicity, the taxi service runs in a rectangular region, on a 1000x1000 grid.
2. Clicking anywhere on the screen (within this rectangular region) indicates you are requesting a taxi at that location. The system will try to allocate one if it is close by and not being used. An allocated taxi is marked as in-use.
3. Once a taxi is allocated to you, clicking again will “drop” you off at that place, even if it is far from where you are now.
4. The screen also has an area where it shows your current status (riding or idle) and your current balance.

5. Since the taxis are all moving around constantly, the front end has to periodically update the view by querying TaxiService for the location and status of all taxis. The current balance is also obtained by querying the TaxiService for your balance.
6. The UI should be responsive – that is, portions of the UI should still be usable even if some part of the app is busy on some activity. For example, if the TaxiService takes some amount of time to respond from a request, the user should still be able to use other parts of the interface, and updates to taxi locations should continue in the background.

Some rules that the TaxiService enforces:

- You should pass in the user id for each request to obtain/release a taxi. For our purposes, your user-id is the same as the last 3 digits of your roll number. (Note: please do not use any other roll number except for testing)
- TaxiService pre-loads the wallet of each user with 100 units of cash
- TaxiService uses the methods of TaxiFleet to request a car at a given location. If no taxi is found, it deducts 10 units from that user's balance
- When a ride is over, TaxiService deducts the ride charge for the user - 6 times the amount of time that the ride lasted
- If a user completes 5 trips, his account is credited with 50 units cash
- Taxi's cannot be requested if the balance is 5 units or less. So, in effect the game is over for a user if the balance drops to 5 or below.

Phase 2: Using a web service

An implementation of TaxiFleet will be made available as a REST web service (running on a server in the data center), returning information in JSON format. You will need to modify your front end to invoke these services. This will involve the following:

1. Implement a proxy version of TaxiFleet for the client end. That is, this class should have the same public interface as the earlier TaxiFleet, but each method should invoke the relevant services in the web service using http, and converting from JSON to the information returned by the TaxiFleet methods. Hence, your implementation of TaxiService should run un-modified, except that the app connects to the web server to perform a request/release or to get status of taxis.

We can now turn this into a multi-layer game, since many clients can connect to the same server simultaneously and compete for the same taxis.See how many simultaneous players can be supported.

Phase 3: Building the web service

For those who are interested, build your own REST server using the TaxiFleet class provided (as jar file). Ideally extend this to also provide the TaxiService services.
Modify the clients to use this TaxiService, again through http calls. You will likely need to use a framework like Jersey.

```
public class Location {
        public Location(int x, int y) {

        }

        public int getX() {
```

```
        }

        public int getY() {

        }
}

public class Taxi {
        public Taxi() {

        }

        public Location getLocation() {

        }

        public boolean isBusy() {

        }

}

public class TaxiFleet {

        public TaxiFleet() {   // number of taxis in the fleet

        }

        // get the fleet going. num is the number of taxis in the fleet.
        //  Should be called once before any use of TaxiFleet
        public void start(int num) {

        }

        // request a taxi for a user at a specific x,y location. Returns the id of a taxi if
        // any found. Otherwise returns -1
        public int request (int userId, Location loc) {

        }

        // release a taxi. Returns the time for which the taxi was occupied (since it was
requested)
        public int release(int userId, Location loc) {

        }

        // get the list of taxis (with their current state)
        public ArrayList<Taxi> getTaxis() {

        }
}
.
```

Using Swing

If you are using Swing, please keep the following in mind (similar issues with use of other frameworks):

- The UI should not be blocked, So, UI events that trigger a server request should NOT be run in the main UI thread.
- You will need to set a timer to be able to update the view periodically (say, every second). See if you can use the Timer class in Swing (which is different from the Timer class in Java utilities)
- Keep the UI simple.

http://stackoverflow.com/questions/20844144/how-do-i-make-a-rectangle-move-across-the-screen-with-key-bindings