Practice 4

Deadline: 2 weeks from now. Should be checked onsite (during labs).

Task 1: Implementing the Observer Pattern Using the `Consumer` Functional Interface

The Observer Pattern is a software design pattern in which an object, named the *subject*, maintains a list of its dependents, called *observers*, and notifies them automatically of any *state* changes, usually by calling one of their methods. Imagine that the subject is a youtube channel, while the observers are a list of subscribers of the channel. Whenever a new video is uploaded to the channel, it goes over all its subscribers and notifies them.

In this task, you will implement the Observer Pattern in Java using the `Consumer` functional interface. We provide an incomplete `ObserverPattern.java`. You need to complete the four **TODOs** in `ObserverPattern.java`, which:

1. **Define the Subject Class and Implement the methods**

   You should complete the `Subject` class that includes the following attributes and methods:

   - **Attributes:**
     - `observers`: A list that stores all subscribed observers. Observers are defined using the `Consumer<String>` functional interface, enabling them to process state updates delivered by the subject.
     - `state`: Represents the current state of the subject.
   - **Methods:**
     - `subscribe(Consumer<String> observer)`: Adds a new observer to the list.
     - `unsubscribe(Consumer<String> observer)`: Removes an observer from the list.
     - `changeState(String newState)`: Updates the state attribute with the provided newState and calls notifyObservers() to inform all subscribed observers of the change.
     - `notifyObservers()`: Iterates through all observers and invokes their `accept` method, passing the updated state.

2. **Complete Concrete Observer 1**
   - simply prints the received state update.
3. **Complete Concrete Observer 2**
   - simulates logging the state change.
4. **Complete Concrete Observer 3**
   - performs specific actions based on the new state (e.g., starting or stopping a service).

Sample Output

```
Changing state to 'START'.
Observer 1 received state update: START
Observer 2 logged: State changed to START
Observer 3 performing START action.

Changing state to 'STOP'.
Observer 1 received state update: STOP
Observer 2 logged: State changed to STOP
Observer 3 performing STOP action.

Changing state to 'RESTART', observer 2 is unsubscribed.
Observer 1 received state update: RESTART
Observer 3 performing RESTART action.
```

Task 2:

In this task, you'll be using Java 8 Stream to perform simple data analysis. Specifically, we provide a `cities.txt` and an incomplete `CityAnalysis.java` which reads `cities.txt`. You need to complete the four **TODOs** in `CityAnalysis.java`, which:

1. Count how many cities there are for each state. The result is `Map<String, Long>`, where the key is the name of the state while the value is the number of cities in that state.
2. Count the total population for each state. The result is `Map<String, Integer>`, where the key is the name of the state while the value is the population of that state (i.e., sum of the population of each city in that state).
3. For each state, get the city of the longest name. The result is `Map<String, String>` or `Map<String, Optional<String>>`.
4. For each state, get the set of cities with >500,000 population. The result is `Map<String, Set<City>>`.

Note

- Check out `Collectors.groupingBy` and other available collectors in java.util.stream.Collectors.
- You may want to override the `toString` method for `City` to facilitate printing `City` objects.
- Your output doesn't have to have the exact same format as our sample output.

Sample Output

```
Q1: # of cities per state:
{DE=1, HI=1, TX=63, MA=22, MD=5, ME=1, IA=10, ID=5, MI=24, UT=12, I

Q2: population per state:
{DE=71292, HI=345610, TX=13748465, MA=2403297, MD=869891, ME=66214

Q3: longest city name per state:
DE:Wilmington,  HI:Honolulu,  TX:North Richland Hills,  MA:Springf

Q4: cities with >500,000 population for each state:
TX: [City{name='Fort Worth', state='TX', population=777992}, City{
MA: [City{name='Boston', state='MA', population=636479}]
MD: [City{name='Baltimore', state='MD', population=621342}]
MI: [City{name='Detroit', state='MI', population=701475}]
IL: [City{name='Chicago', state='IL', population=2714856}]
IN: [City{name='Indianapolis', state='IN', population=834852}]
NC: [City{name='Charlotte', state='NC', population=775202}]
AZ: [City{name='Tucson', state='AZ', population=524295}, City{name=
NM: [City{name='Albuquerque', state='NM', population=555417}]
```

Evaluation

The practice will be checked by teachers or SAs. What will be tested:

1. That you understand every line of your own code, not just copy from somewhere
2. That your program compiles correctly (javac)
3. Correctness of the program logic
4. That the result is obtained in a reasonable time

Late submissions after the deadline will incur a 20% penalty, meaning that you can only get 80% of this practice's score.