Ex 1: Simulation I

Basic Information

- **Deadline:** 3 September 2024, Tuesday, 23:59 SGT
- **Difficulty:** ★★★

Prerequisite

- Completed Programming Exercise 0.
- Caught up to Unit 18 of Lecture Notes.

Goal

The goal of Programming Exercise 1 is for you to practice the basic OOP principles: encapsulation (*including tell-don't-ask and information hiding*), abstraction, inheritance, and polymorphism.

You are given six classes: five Java classes and one main `Ex1` class. Two of them are poorly written without applying any of the OOP principles. Using the OO principles that you have learned, you should rewrite, remove, or add new classes as needed.

Background: Discrete Event Simulator

A discrete event simulator is a piece of software that simulates a system (*often modeled after the real world*) with events and states. An event occurs at a particular time and each event alters the states of the system and may generate more events. A discrete event simulator can be used to study many complex real-world systems. The term discrete refers to the fact that the states remain unchanged between two events, and therefore, the simulator can jump from the time of one event to another, instead of following the clock in real time.

In Programming Exercise 1, we provide you with three very generic classes:

- `Simulator`: Implements a discrete event simulator.

- `Event`: Encapsulates an event (*with a time*)
- `Simulation`: Encapsulates the states we are simulating.

The Event and Simulation class can be extended to implement any actual simulation (*network, road traffic, weather, pandemic, etc*).

Simulating a Coffee Shop

In Exercise 1, we wish to extend the `Simulation` class to simulate customers in a coffee shop.

Our coffee shop can have one or more counters to make orders. In the beginning, all counters are available. A counter becomes unavailable when it is serving a customer, and becomes available again after servicing a customer.

A customer, upon arrival at the coffee shop, goes to the first available counter. If no counter is available, the customer departs (*we are simulating a neighborhood coffee shop with no space for waiting*). Otherwise, the customer is served by a counter. After being served for a given amount of time (*called service time*), the customer departs.

Two classes, `CoffeeSimulation` (*a subclass of* `Simulation`) and `CoffeeEvent` (*a subclass of* `Event`) are provided. The two classes implement the simulation above.

The `Event` Class

Do NOT Edit

You should **NOT** edit this class. The following is for your info only.

The `Event` class is an abstract class with a single field `time`, which indicates the time the event occurs. The `Event::toString` method returns the time as a string and the `Event::getTime` method returns the time.

The most important thing to know about the `Event` class is that it has an abstract method `simulate` that needs to be overridden by its subclass to concretely define the action to be taken when this event occurs.

Simulating an event can lead to more events being created. `Event::simulate` returns an array of `Event` instances.

The `Simulation` Class

❌ **Do NOT Edit**

You should **NOT** edit this class. The following is for your info only.

The `Simulation` class is an abstract class with a single method `getInitialEvents`, which returns an array of events to simulate. Each of these events may generate more events.

The `Simulator` Class

❌ **Do NOT Edit**

You should **NOT** edit this class. The following is for your info only.

The `Simulator` class is a class with only two methods and it is what drives the simulation. To run the simulator, we initialize it with a `Simulation` instance, and then call run:

```
Simulation sim = new SomeSimulation();
new Simulator(sim).run();
```

The `Simulation::run` method simply does the following:

- It gets the list of initial `Event` objects from the `Simulation` object;
- It then simulates the pool of events, one by one in the order of increasing time, by calling `Event::simulate`;
- If simulating an event results in one or more new events, the new events are added to the pool.
- Before each event is simulated, `Event::toString` is called and a message is printed
- The simulation stops running if there are no more events to simulate.

For those of you taking CS2040S, you might be interested to know that the `Simulator` class uses a priority queue to keep track of the events with their time as the key.

The `CoffeeSimulation` Class

✓ **Edit**

You are expected to *edit* this class and create new classes.

The `CoffeeSimulation` class is a concrete implementation of a `Simulation`. This class is responsible for:

- reading the inputs from the standard inputs,
- initialize the coffee shop counters (*represented with boolean* `available` *arrays*)
- initialize the events corresponding to customer arrivals
- return the list of customer arrival events to the `Simulator` object when `getInitialEvent` is called.

Each customer has an ID. The first customer has id **0**, the next one is **1**, and so on.

Each counter has an ID, numbered from **0**, **1**, **2**, and onwards.

The `CoffeeEvent` Class

✓ **Replace**

You are expected to *replace* this class with new classes. You must *remove* the file `CoffeeEvent.java` before submission if the file is no longer used in your solution.

The `CoffeeEvent` class is a concrete implementation of `Event`. This class overrides the simulate method to simulate the customer and counter behavior.

A `CoffeeEvent` instance can be tagged as either an arrival event, service-begin event, service-end event, or departure event.

- **Arrival:** the customer arrives. It finds the first available coffee shop counter (*scanning from* ID **0** *upwards*) and goes to the counter for service immediately. A service-begin event is generated. If no counter is available, it departs. A departure event is generated.
- **Service-begin:** the customer is being served. A service-end event scheduled at the time (*current time* + *service time*) is generated.
- **Service-end:** the customer is done being served and departs immediately. A departure event is generated.

- **Departure:** the customer departs.

Inputs and Outputs

The main program `Ex1.java` reads the following, from the standard inputs.

- An integer n, for the number of customers to simulate.
- An integer k, for the number of coffee shop counters the coffee shop has.
- n pairs of double values, each pair corresponds to a customer. The first value indicates the arrival time, and the second indicates the service time for the customer.

The customers are sorted in increasing order of arrival time.

Assumptions

We assume that no two events at the same time will ever be in the collection of events and no customer will arrive exactly at the same time as another customer is leaving. As per all exercises, we assume that the input is correctly formatted.

Tasks

The two classes, `CoffeeSimulation` and `CoffeeEvent`, are poorly written. They do not fully exploit OOP features and apply the OO principles such as abstraction, encapsulation (*including information hiding and tell-don't-ask*), composition, inheritance, polymorphism, and Liskov substitution principle.

Task 1: Rewrite

Rewrite these two classes (*adding new ones as needed*) with the OOP principles that you have learned:

- encapsulation to group relevant fields and methods into new classes;
- inheritance and composition to model the relationship between the classes;
- information hiding to hide internal details; and
- using polymorphism to make the code more succinct and extendable in the future, while adhering to the LSP.

Here are some hints:

- Think about the problem that you are solving: what are the nouns? These are good candidates for new classes.
- For each class, what are the attributes/properties relevant to the class? These are good candidates for fields in the class.
- Do the classes relate to each other via IS-A or HAS-A relationship?
- For each class, what are their responsibilities? What can they do? These are good candidates for methods in the class.
- How do the objects of each class interact? These are good candidates for public methods.
- What are some behavior that changes depending on the specific type of object?

Note that the goal of this exercise, and CS2030S in general, is **NOT** *to solve the problem with the cleverest and the shortest piece of code possible*. For instance, you might notice that you can solve Programming Exercise 1 with only a few variables and an array. But such a solution is hard to extend and modify. In CS2030S, our goal is to produce software that can easily evolve and be modified, with a reduced risk of introducing bugs while doing so.

Note that Programming Exercise 1 is the first of a series of exercises, where we introduce new requirements or modify existing ones in every exercise (*not unlike what software engineers face in the real world*). We will modify the behavior of the coffee shop, the counters, and the customers. In particular, in the future,

- a customer may order different kinds of coffee such as Latte, *etc*.
- there may be different counters that only allow a customer to order specific coffee, such as only Cold Brew, only Latte, or only Espresso.
- counters may open and close at different times because some orders are more popular at certain times.
- the coffee shop might expand and be able to handle more customers and more counters.

Thus, making sure that your code will be able to adapt to new problem statements is the key. If you solve the exercise without considering this, you will likely find yourself painted into a corner and have to re-write much of your solution to handle any new requirements.

Testing

You will find the sub-directories `inputs` and `outputs` in the given skeleton. These directories contain the test cases. The file `Ex1.x.in` contains the input for Test Case ; the file `Ex1.x.out` contains the expected output for Test Case .

A script `test.sh` has been given to you for testing. To test your program, run

```
username@pe111:~/ex1-github-username$ ./test.sh Ex1
```

You should see:

```
test 1: passed
test 2: passed
test 3: passed
test 4: passed
test 5: passed
Ex1: passed everything 🎉
```

Note that the given skeleton is a fully working program. Thus, it already passed the test cases.

If you want to test your program with your own test cases, run the following:

```
username@pe111:~/ex1-github-username$ java Ex1 < FILE
```

where `FILE` is the file that contains your own test case. If you are not familiar with `<`, take a look at our Unix CLI notes on standard input and output.

Comparing Outputs

If the output of your program is different from the expected output, you can redirect the output of your program to a file for inspection. For instance, suppose that your code failed Test Case 3. The following would create a file `OUT` that contains the output from your program for Test Case 3.

```
username@pe111:~/ex1-github-username$ java Ex1 < inputs/Ex1.3.in > OUT
```

Use `vim -d` to compare your output with the expected output.

```
username@pe111:~/ex1-github-username$ vim -d OUT outputs/Ex1.3.out
```

Style (Optional)

Make sure that your code following our coding style is optional for Programming Exercise 1. If you would like to keep your code neat, tidy, and adhere to the CS2030S style, you can run

```
username@pe111:~/ex1-github-username$ java -jar
~cs2030s/bin/checkstyle.jar -c ex1_style.xml *.java
```

You should see the following output, with nothing in between the two lines. Any style errors would appear in between the two.

```
Starting audit...
Audit done.
```

Documentation (Optional)

Documenting your code with Javadoc is optional for Programming Exercise 1. It is, however, always a good practice to include comments to help readers understand your code.