Lab 1: Simulation I

- Deadline: 31 August, 2022, 23:59
- Marks: 3%

Lab 2 (to be released in Week 4) is an extension of Lab 1.

Prerequisite:

- Completed Lab 0
- Caught up to Unit 17 of Lecture Notes

Goal

The goal of Lab 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 `Lab1` 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 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 Lab 1, we provide you with three very generic classes: `Simulator`, which implements a discrete event simulator, `Event`, which encapsulates an event (with a time), and `Simulation`, which 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). More details of these classes can be found below.

Simulating a Shop

In Lab 1, we wish to simulate a shop. A shop can have one or more service counters.

In the beginning, all service 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 shop, goes to the first available counter. If no counter is available, the customer departs (due to COVID-19, no waiting is allowed). Otherwise, the customer is served, and after being served for a given amount of time (called service time), the customer departs.

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

The `Event` class

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

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

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`:

```
1   Simulation sim = new SomeSimulation();
2   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 resulted 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 `ShopSimulation` class

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

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

- reading the inputs from the standard inputs,
- initialize the service 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, onwards.

The `ShopEvent` class

You are expected to *replace* this class with new classes.

The `ShopEvent` class is a concrete implementation of `Event`. This class overrides the

`simulate` method to simulate the customer and counter behavior.

A `ShopEvent` 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 service counter (scanning from id 0 upwards) and go 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 `Lab1.java` reads the following, from the standard inputs.

- An integer n, indicating the number of customers to simulate.
- An integer k, indicating the number of service counters the shop has.
- n pairs double values, each pair corresponds to a customer.
  The first value indicates the arrival time, 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 ever occur at the same time. As per all labs, we assume that the input is correctly formatted.

Your Task

The two classes, `ShopSimulation` and `ShopEvent`, 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 LSP.

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
- using polymorphism to make the code more succinct and extendable in the future, while adhering to 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 objects?

Note that the goal of this lab and, and of 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 Lab 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 Lab 1 is the first of a series of labs, where we introduce new requirements or modify existing ones in every lab (not unlike what software engineers face in the real world). We will modify the behavior for the shop, the counters, and the customers. In particular, in the future,

- a customer may have different behavior in selecting the counters to join.
- a shop may have different rules about managing customers when the COVID-19 pandemic is over, including introducing different types of counters.

Thus, making sure that your code will be able to adapt to new problem statements is the key. Trying to solve the lab without considering this and you will likely find yourself

painted into a corner and have to re-write much of your solution to handle the new requirement.

Submission

Upload the following files to CodeCrunch:

1. `Lab1.java`
2. *any other* `.java` *files you use*