11.5 Creating a Discrete-Event Simulation

Let's put together all of the classes we've designed over the course of this chapter to create a full simulation our of food delivery system. In this section, we'll first learn about how the main simulation loop works. Then, we'll turn our attention to the possible ways a simulation can be configured, and how to incorporate these configuration options as part of the public interface of a class.

The main simulation loop

Before we get to creating a full simulation class, we'll discuss how our simulation works. The type of simulation we're learning about is called a **discrete-event simulation**, because it is driven by individual events occurring at specified periods of time.

A discrete-event simulation runs as follows:

1. It keeps track of a collection of events, which begins with some initial events. The collection is a *priority queue*, where an event's priority is its timestamp (earlier timestamps mean higher priority).
2. The highest-priority event (i.e., the one with the earliest timestamp) is removed and processed. Any new events it generates are added to the priority queue.
3. Step 2 repeats until there are no events left.

The algorithm is remarkably simple, though it does rely on a slightly modified version of our *priority queue* implementation from 10.8 Priority Queues.[1] Assuming we have such an implementation called `EventQueueList`, here is how we could write a simple function that runs this simulation loop:

[1] In that section, we used `int`s to represent priority, while here we're using `datetime.datetime` values.

```python
def run_simulation(initial_events: list[Event], system: FoodDeliverySystem) -> None:
    events = EventQueueList()  # Initialize an empty priority queue of events
    for event in initial_events:
        events.enqueue(event)

    # Repeatedly remove and process the next event
    while not events.is_empty():
        event = events.dequeue()

        new_events = event.handle_event(system)
        for new_event in new_events:
            events.enqueue(new_event)
```

The main reason for this implementation's simplicity is abstraction. Remember that `Event` is an abstract class; the complex behaviour of how different events are handled is deferred to its concrete subclasses via our calls to `event.handle_event`. Our `run_simulation` function is *polymorphic*: it works regardless of what `Event` instances are contained in `initial_events`, or what new events are generated and stored in `new_events`. The only thing our function needs to be able to do is call the `handle_event` method on each event object, which we can assume is present because it is defined in the `Event` public interface.

A simulation class

Next, we will take our `run_simulation` in the previous section and "wrap" it inside a new class. This isn't necessary to the running of the simulation, but is a standard practice in an object-oriented design, and makes it easier to both configure the simulation parameters and report results after the simulation is complete.

We're going to begin with a sketch of a class to represent our simulation:

```python
class FoodDeliverySimulation:
    """A simulation of the food delivery system.
    """
    # Private Instance Attributes:
    #   - _system: The FoodDeliverySystem instance that this simulation uses.
    #   - _events: A collection of the events to process during the simulation.
    _system: FoodDeliverySystem
    _events: EventQueue

    def __init__(self, start_time: datetime.datetime, num_days: int,
                 num_couriers: int, num_customers: int,
                 num_restaurants: int) -> None:
        """Initialize a new simulation with the given simulation parameters.

        start_time: the starting time of the simulation
        num_days: the number of days that the simulation runs
        num_couriers: the number of couriers in the system
        num_customers: the number of customers in the system
        num_restaurants: the number of restaurants in the system
        """
        self._events = EventQueueList()
        self._system = FoodDeliverySystem()

        self._populate_initial_events(start_time, num_days)
        self._generate_system(num_couriers, num_customers, num_restaurants)

    def _populate_initial_events(self, start_time: datetime.datetime, num_days: int) -> None:
        """Populate this simulation's Event priority queue with GenerateOrdersEvents.

        One new GenerateOrderEvent is generated per day, starting with start_time and
        repeating num_days times.
        """

    def _generate_system(self, num_couriers: int, num_customers: int, num_restaurants: int) ->
        """Populate this simulation's FoodDeliverySystem with the specified number of entities.
        """

    def run(self) -> None:
        """Run this simulation.
        """
        while not self._events.is_empty():
            event = self._events.dequeue()

            new_events = event.handle_event(self._system)
            for new_event in new_events:
                self._events.enqueue(new_event)
```

There are a few key items to note in this (incomplete) implementation:

1. The `run_simulation` method has been renamed to simply `run`, since it's a method in the `FoodDeliverySimulation` class.

2. The local variable `events` and parameter `system` from the function are now instance attributes for the `FoodDeliverySimulation` class, and have been moved out of the `run` method entirely. It's the job of `FoodDeliverySimulation.__init__` to initialize these objects.

3. The initializer takes in several parameters representing *configuration values* for the simulation. It then uses these values in two helper methods to initialize the `_system` and `_events` objects. These methods are marked private (named with a leading underscore) because they're only meant to be called by the initializer, and not code outside of the class.

   Here is how we could use the `FoodDeliverySimulation` class:

   ```python
   >>> simulation = FoodDeliverySimulation(datetime.dateti
   >>> simulation.run()
   ```

Next, we'll briefly discuss one way to implement each of the two key helper methods for the initializer, `_populate_initial_events` and `_generate_system`.

Populating initial events

The key idea for our first helper method is that given a start time and a number of days, our initial events will be a series of `GenerateOrderEvents` that will generate `NewOrderEvents` when they are processed. Here is the basic skeleton, which will be leave as an exercise for you to complete:

```python
    def _populate_initial_events(self, start_time: datetim
        """Populate this simulation's Event priority queue

        One new GenerateOrderEvent is generated per day, s
        repeating num_days times.

        Preconditions:
        - num_days >= 0
        """
        for day in range(0, num_days):
            # 1. Create a GenerateOrderEvent for the given

            # 2. Enqueue the new event.
```

Populating the system entities

The way that our simulation is currently set up, our `FoodDeliverySystem` instance will contain all vendors, customers, and couriers before the events start being processed. That is, we are assuming that only *orders* are dynamic in our system; the vendors, customers, and couriers do not change over time.

The easiest way to populate these three entity types is to randomly generate new instances of each of these classes. We've shown an example with `Customer`s below.

```python
    def _generate_system(self, num_couriers: int, num_cust
        """Populate this simulation's FoodDeliverySystem w
        """
        for i in range(0, num_customers):
            location = _generate_location()
            customer = Customer(f'Customer {i}', location)
            self._system.add_customer(customer)

        # Couriers and Restaurants are similar
        ...


# Outside the class: helper for generating random location
TORONTO_COORDS = (43.747743, 43.691170, -79.633951, -79.17


def _generate_location() -> tuple[float, float]:
    """Return a randomly-generated location (latitude, lon
    """
    return (random.uniform(TORONTO_COORDS[0], TORONTO_COOR
            random.uniform(TORONTO_COORDS[2], TORONTO_COOR
```

Putting it all together

After completing the implementation of these two helper methods, you are ready to run the simulation! Try doing the following in the Python console:

```python
>>> simulation = FoodDeliverySimulation(datetime.datetime(
>>> simulation.run()
```

Of course, we aren't printing anything out, and the `FoodDeliverySimualtion.run` method doesn't actually return anything. You are free to insert some `print` calls to see whether events are actually being processed, but that's not the only way to see the results of the simulation.

Once the simulation is complete, `self._system` will have accumulated several completed orders, as a `list[Order]`. We can access these values and perform any kind of computation on them we want, using comprehensions and/or for loops like we've done throughout this course!

For example, we might ask:

- How many orders were delivered in total?
- What was the average number of orders delivered per courier?
- For a given vendor, which menu items were most popular?
- *What else can you come up with?*