CSC110 Lecture 30: Discrete-Event Simulations

Hisbaan Noorani

January 14, 2021

Contents

Note: like yesterday, we've provided a starter file `events.py` for you to complete your work for today's worksheet (Exercises 1 and 2).

1 Exercise 1: Representing events

In lecture, you learned about the `Event` abstract class, used to represent a single change in the state of our food delivery system.

```
class Event:
    """An abstract class representing an event in a food delivery simulation.

    Instance Attributes:
        - timestamp: the start time of the event
    """
    timestamp: datetime.datetime

    def __init__(self, timestamp: datetime.datetime) -> None:
        """Initialize this event with the given timestamp."""
        self.timestamp = timestamp

    def handle_event(self, system: FoodDeliverySystem) -> None:
        """Mutate the given food delivery system to process this event.
        """
        raise NotImplementedError
```

1. First, please review both this class and the `NewOrderEvent` we developed together in lecture. Make sure you understand both of them (and the relationship between them) before moving on.
2. Your main task here is to implement a new event class called `CompleteOrderEvent` that represents when a courier has completed a delivery to a customer.

   Its structure should be very similar to `NewOrderEvent`, except:

   (a) Its initializer needs an explicit `timestamp` parameter (to represent when the order is completed).

   (b) The implementation of `handle_event` needs to call a different `FoodDeliverySystem` method—please review yesterday's code for this.

```python
class CompleteOrderEvent(Event):
    """An event representing when an order is delivered to a customer by a courier."""
    _order: Order

    def __init__(self, timestamp: datetime.datetime, order: Order) -> None:
        Event.__init__(self, timestamp)
        self.order = order

    def handle_event(self, system: FoodDeliverySystem) -> List[Event]:
        system.complete_order(self._order, self.timestamp)
        return []
```

2 Exercise 2: The `GenerateOrdersEvent`

Consider the `GenerateOrdersEvent` we covered in lecture (atttributes and initializer shown):

```python
class GenerateOrdersEvent(Event):
    """An event that causes a random generation of new orders.
    """
    # Private Instance Attributes:
    #   - _duration: the number of hours to generate orders for
    _duration: int

    def __init__(self, timestamp: datetime.datetime, duration: int) -> None:
        """Initialize this event with timestamp and the duration in hours.

        Preconditions:
            - duration > 0
        """
        Event.__init__(self, timestamp)
        self._duration = duration
```

Your task here is to implement its `handle_event` method, which does not mutate the given `FoodDeliverySystem`, but instead randomly generates a list of =NewOrderEvent=s using the following algorithm:

1. Initialize a variable `current_time` to be this event's `timestamp`.
2. Create a new `Order` by randomly choosing a customer and restaurant, an empty `food_items` dictionary, and the `current_time`.
3. Create a new `NewOrderEvent` based on the `Order` from Step 2, and add it to a list accumulator.
4. Increase the `current_time` by a random number of minutes, from 1 to 60 inclusive.
5. Repeat Steps 2–4 until the `current_time` is greater than the `GenerateOrderEvent`'s `timestamp` plus its `_duration` (in hours).

We've started this method for you; you only need to fill in the body of the while loop. This is good practice with the `random` module!

```python
def handle_event(self, system: FoodDeliverySystem) -> List[Event]:
    """Generate new orders for this event's timestamp and duration."""
    # Technically the lines below access a private attribute of system,
    # which is a poor practice. We'll discuss an alternate approach in class.
```

```
    customers = [system._customers[name] for name in system._customers]
    restaurants = [system._restaurants[name] for name in system._restaurants]

    events = []  # Accumulator

    current_time = self.timestamp
    end_time = self.timestamp + datetime.timedelta(hours=self._duration)

    while current_time < end_time:
        customer = random.choice(customers)
        restaurant = random.choice(restaurants)

        random_order = Order(customer, restaurant, {}, current_time)
        new_order_event = NewOrderEvent(random_order)
        events.append(new_order_event)

        current_time = current_time + datetime.timedelta(minutes=random.randint(1, 60))

    return events
```

```
def handle_event(self, system: FoodDeliverySystem) -> List[Event]:
    """Generate new orders for this event's timestamp and duration."""
    # FIXME: Create some public methods in FoodDeliverySystem to access
    #        a list of customers and a list of restaurants
    customers = [system._customers[name] for name in system._customers]
    restaurants = [system._restaurants[name] for name in system._restaurants]

    events = []  # Accumulator

    current_time = self.timestamp
    end_time = self.timestamp + datetime.timedelta(hours=self._duration)

    while current_time < end_time:
        customer = random.choice(customers)
        restaurant = random.choice(restaurants)

        random_order = Order(customer, restaurant, {}, current_time)
        new_order_event = NewOrderEvent(random_order)
        events.append(new_order_event)

        current_time = current_time + datetime.timedelta(minutes=random.randint(1, 60))
```

3 Exercise 3: Understanding the main simulation loop

Recall the main simulation loop from lecture:

```
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)

```

```
6       # Repeatedly remove and process the next event
7       while not events.is_empty():
8           event = events.dequeue()
9
10          new_events = event.handle_event(system)
11          for new_event in new_events:
12              events.enqueue(new_event)
```

Your goal for this exercise is to review the three `Event` subclasses we've seen so far and see how to trace the execution of this loop.

Suppose we call `run_simulation` with a single initial event:

- type `OrderGenerateEvent`, timestamp `December 1 2020, 11:00am`, duration 1 hour

Complete the following table, showing the state of the priority queue `events` after each loop iteration. For each event, only show its class name and the time from the timestamp, not the day (all events for this example will take place on the same day). We've given an example in the first two rows.

(Note that since there's some randomness in `OrderGenerateEvent.handle_event`, we assumed that it creates *three* `NewOrderEvents` that occur at 11:00, 11:07, and 11:20.)

Loop Iteration	Events stored in `events`
0	`OrderGenerateEvent(11:00)`
1	`NewOrderEvent(11:00)`, `NewOrderEvent(11:07)`, `NewOrderEvent(11:20)`
2	`NewOrderEvent(11:07)`, `CompleteOrderEvent(11:10)`, `NewOrderEvent(11:20)`,
3	`CompleteOrderEvent(11:10)`, `CompleteOrderEvent(11:17)`, `NewOrderEvent(11:20)`
4	`CompleteOrderEvent(11:17)`, `NewOrderEvent(11:20)`
5	`NewOrderEvent(11:20)`
6	`CompleteOrderEvent(11:30)`
7	`<empty>`

While loop stops since `events` is empty