Lecture 30: Discrete-Event Simulations

Print this handout

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. We also implemented one subclass of `Event` called `NewOrderEvent`, representing a new order being placed.

```
from __future__ import annotations

import datetime

from entities import Order
from food_delivery_system import FoodDeliverySystem


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


class NewOrderEvent(Event):
    """An event representing when a customer places an order at a vendor."""
    # Private Instance Attributes:
    #   _order: the new order to be added to the FoodDeliverySystem
    _order: Order

    def __init__(self, order: Order) -> None:
        """Initialize a NewOrderEvent for the given order."""
        Event.__init__(self, order.start_time)
        self._order = order

    def handle_event(self, system: FoodDeliverySystem) -> None:
        """Mutate system by placing an order.
        """
        system.place_order(self._order)
```

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.

1. 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 last class's code (`food_delivery_system.py`) for this.

Exercise 2: The `GenerateOrdersEvent`

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

```
class GenerateOrdersEvent(Event):
    """An event that causes a random generation of new orders.

    Representation Invariants:
    - self._duration > 0
    """
    # 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`.
   - *Hint*: You can use the `random.choice` function to take a list and randomly select one of its elements. (You'll need to import the `random` module.)
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.
   - *Hint*: You can use `random.randint` to make this choice.
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 complete the while loop. This is good practice with the `random` module!

```
class GenerateOrdersEvent(Event):
    ...

    def handle_event(self, system: FoodDeliverySystem) -> list[Event]:
        """Generate new orders for this event's timestamp and duration."""
        customers = [system._customers[name] for name in system._customers]
        restaurants = [system._restaurants[name] for name in system._restaurants]

        events = []  # Event accumulator

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

        while                                   :

            # Create a randomly-generated Order called new_order









            new_order_event = NewOrderEvent(new_order)
            events.append(new_order_event)

            # Update current_time




        return events
```

Additional exercise: Understanding the main simulation loop

Recall the main simulation loop from lecture:

```
def run_simulation(initial_events: list[Event], system: FoodDeliverySystem) -> None:
    events = EventQueueList()
    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)
```

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 `GenerateOrdersEvent`, timestamp `December 1 2022, 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 `GenerateOrdersEvent.handle_event`, we assumed that it creates *three* `NewOrderEvent`s that occur at 11:00, 11:07, and 11:20.)

Loop Iteration	Events stored in `events`
0	`GenerateOrdersEvent(11:00)`
1	`NewOrderEvent(11:00)`, `NewOrderEvent(11:07)`, `NewOrderEvent(11:20)`

Additional exercise: Other event types

Try implementing the following types of events, which model different types of changes to our food delivery system.

- A *new customer event*, which causes the customer to place orders for five different food vendors on consecutive days.
- A *new food vendor event*, which causes 10 customer to place orders for the vendor within 1 hour of when the vendor joins.
- A *special sale event*, which causes every vendor to reduce their prices by 50% on a given day, and causes 100 customers to place orders at random vendors on that day. (You'll need to add a method to `FoodDeliverySystem` to modify vendor prices—and how will you signal the *end* of the sale on the next day?)