CSC110 Lecture 29: Object-Oriented Modelling

Hisbaan Noorani

November 30, 2020

Contents

1 Exercise 1: Designing a `Courier` data class

In this week's prep, you read about four data classes we'll use this week to model the different entities in our food delivery system: `Restaurant`, `Customer`, `Courier`, and `Order`. We gave the full design for `Restaurant` and `Order` in the Course Notes, and guided you through an implementation of `Customer` in the prep. However, we left `Courier` blank:

```
@dataclass
class Courier:
    """A person who delivers food orders from restaurants to customers.

    Instance Attributes:
        - name: name of the courier.
        - location: the current location of the lattitude.
        - order: the order currently assigned to the courier.

    Representation Invariants:
        - self.name != ''
        - -90 <= self.location[0] <= 90
        - -180 <= self.location[1] <= 180
        - self.order.courier is self or self.order.courier is None
    """
    name: str
    location: Tuple[float, float]
    order: Optional[Order] = None
```

In this exercise, you will design this class. We recommend completing this exercise in the starter file `entities.py` we've posted under Week 11 on Quercus.

1. First, we want all couriers to have these three attributes:

   - A name (which should not be empty)
   - A location (latitude and longitude, just like restaurants and customers)

- A *current order*, which is either `None` (if they have no order currently assigned to them) or an `Order` instance (if they have an order assigned to them).

  The *default value* for this attribute should be `None`—review the `Order` data class for how to set a default value for an instance attribute.

Add to the given definition of the `Courier` data class to include these three instance attributes. Make sure to include type annotations, descriptions, and representation invariants for these attributes. Also write an *example use* of this data class as a doctest example.

2. One thing to note for this design is that every `Order` instance has an associated `Courier` attribute, and every `Courier` has an associated `Order` attribute. This leads to a new representation invairant:

   - If `self` has a non-`None` current order, then that `Order` object's `courier` attribute is equal to `self`.

   Translate this representation invariant into Python code; use `is` to check for reference equality between `self` and the order's `courier`.

3. Can two `Order` objects refer to the same `Courier` instance? Why or why not?

   Yes, when a `Courier` finishes an `Order`, they can be assigned to another `Order`. Just not at the same time. You cannot have two incomplete orders refferring to the same `Courier`, only one `Courier` can be reffered to by an uncompleted (outstanding) `Order`.

4. Brainstorm two or three other instance attributes you could add to the `Courier` data class to better model "real world" food delivery systems. Pick meaningful names and type annotations for these instance attributes.

   There are no right or wrong answers here! You are practicing brainstorming a small part of object-oriented design.

   - `phone_number: str`
   - `mode_of_transport: str`

2 Exercise 2: Developing the `FoodDeliverySystem` class

In lecture we introduced the start of a new class to act as a "manager" of all the entities in the network.

```
class FoodDeliverySystem:
    """A system that maintains all entities (restaurants, customers, couriers, and orders).

    Representation Invariants:
        - self.name != ''
        - all(r == self._restaurants[r].name for r in self._restaurants)
        - all(c == self._customers[c].name for c in self._customers)
        - all(c == self._couriers[c].name for c in self._couriers)
    """
    # Private Instance Attributes:
    #   - _restaurants: a mapping from restaurant name to Restaurant object.
    #       This represents all the restaurants in the system.
    #   - _customers: a mapping from customer name to Customer object.
    #       This represents all the customers in the system.
    #   - _couriers: a mapping from courier name to Courier object.
    #       This represents all the couriers in the system.
    #   - _orders: a list of all orders (both open and completed orders).

    _restaurants: Dict[str, Restaurant]
```

```
    _customers: Dict[str, Customer]
    _couriers: Dict[str, Courier]
    _orders: List[Order]
```

Now, we're going to ask you to implement two different methods for this class. We recommend completing this exercise in the starter file `food_delivery_system.py` we've posted under Week 11 on Quercus.

1. Implement the `FoodDeliverySystem` initializer, which simply initializes all of the instance attributes to be empty collections of the appropriate type.

```
def __init__(self) -> None:
    """Initialize a new food delivery system.

    The system starts with no entities.
    """
    self._restaurants = {}
    self._customers = {}
    self._couriers = {}
    self._orders = []
```

2. Implement the `FoodDeliverySystem.add_restaurant` method, which adds a new restaurant to the system. Because the `FoodDeliverySystem` keeps track of all entities, it can check uniqueness constraints across all the restaurants—something that individual `Restaurant` instances can't check for.

```
def add_restaurant(self, restaurant: Restaurant) -> bool:
    """Add the given restaurant to this system.

    Do NOT add the restaurant if one with the same name already exists.

    Return whether the restaurant was successfully added to this system.
    """
    if restaurant.name in self._restaurants:
        return False

    self._restaurants[restaurant.name] = restaurant
    return True
```

For extra practice later, implement the analogous `add_customer` and `add_courier` methods to this class.

```
def add_customer(self, customer: Customer) -> bool:
    """Add the given customer to this system.

    Do NOT add the customer if one with the same name already exists.

    Return whether the customer was successfully added to this system.
    """
    if customer.name in self._customers:
        return False

    self._customers[customer.name] = customer
    return True
```

```python
def add_courier(self, courier: Courier) -> bool:
    """Add the given courier to this system.

    Do NOT add the courier if one with the same name already exists.

    Return whether the courier was successfully added to this system.
    """
    if courier.name in self._couriers:
        return False

    self._couriers[courier.name] = courier
    return True
```

3 Exercise 3: Handling orders

Handling new orders is more complex than the other entities, since there are two steps involved.

- First, a new order is assigned an available courier.
- Second, at a later time the order is marked as complete.

Your task for this exercise is to implement each of the two methods below. You should do this in the same file you used for Exercise 2.

Note: When choosing a particular courier to assign, you may choose how you want to make the choice: e.g., the first courier in `self._couriers` who is available, or perhaps the one that is closest to the restaurant or customer?

```python
def place_order(self, order: Order) -> bool:
    """Try to add an order to this system.

    Do NOT add the order if no couriers are available (i.e., are already assigned orders).

    - If a courier is available, add the order and assign it a courier, and return True.
    - Otherwise, do not add the order, and return False.

    Preconditions:
        - order not in self._orders
    """
    available_couriers = [courier for courier in self._couriers
                          if self._couriers[courier].order is None]

    if not available_couriers:
        return False

    # Set the courier's order
    courier = available_couriers[0]
    courier.order = order

    # Set the order's courier
    order.courier = courier
    self._orders.append(order)
    return True

```

```python
def complete_order(self, order: Order, timestamp: datetime.datetime) -> None:
    """Record that the given order has been delivered successfully at the given timestamp.

    Make the courier who was assigned this order available to take a new order.

    In addition to implementing the function, add the following preconditions:
        - the order is not already complete
        - the given timestamp is after the order's start time
          (you can use < to compare datetime.datetimes)

    Preconditions:
        - order.end_time is None
        - order.start_time < timestamp
    """
    order.end_time = timestamp
    order.courier.order = None

    # Don't do:
    # order.courier = None
    # self._orders.remove(order)
    #
    # These were not specified in the docstring and it would be useful to keep this information.
```