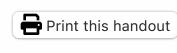
CSC110 Lecture 29: Object-Oriented Modelling



Exercise 1: Designing a Courier data class

In lecture, we introduced four data classes we'll use this week to model the different entities in our food delivery system: Vendor, Customer, Courier, and Order. We covered the design for Vendor, Order, and Customer in lecture. However, we left Courier blank. Here is starter code you can copy into a new Python file, with an empty Courier class at the end of the file.

```
empty Courier class at the end of the file.
 from __future__ import annotations
                                                                                     from dataclasses import dataclass
 import datetime
from typing import Optional
 @dataclass
 class Vendor:
     """A vendor that sells groceries or meals.
     This could be a grocery store or restaurant.
     Instance Attributes:
       - name: the name of the vendor
       - address: the address of the vendor
       - menu: the menu of the vendor with the name of the food item mapping to
               its price
       - location: the location of the vendor as (latitude, longitude)
     Representation Invariants:
       - self.name != ''
       - self.address != ''
       - all(self.menu[item] >= 0 for item in self.menu)
       - -90.0 <= self.location[0] <= 90.0
       - -180.0 <= self.location[1] <= 180.0
     name: str
     address: str
     menu: dict[str, float]
     location: tuple[float, float] # (lat, lon) coordinates
 @dataclass
 class Customer:
     """A person who orders food.
     Instance Attributes:
      - name: the name of the customer
       - location: the location of the customer as (latitude, longitude)
     Representation Invariants:
      - self.name != ''
       - -90 <= self.location[0] <= 90
       - -180 <= self.location[1] <= 180
     name: str
     location: tuple[float, float]
 @dataclass
 class Order:
     """A food order from a customer.
     Instance Attributes:
      - customer: the customer who placed this order
      - vendor: the vendor that the order is placed for
       - food_items: a mapping from names of food to the quantity being ordered
       - start time: the time the order was placed
       - courier: the courier assigned to this order (initially None)
       - end_time: the time the order was completed by the courier (initially None)
     Representation Invariants:
       - all(self.food items[item] >= 1 for item in self.food items)
     customer: Customer
     vendor: Vendor
     food_items: dict[str, int]
     start_time: datetime.datetime
     courier: Optional[Courier] = None
     end_time: Optional[datetime.datetime] = None
 @dataclass
 class Courier:
     """A person who delivers food orders from vendors to customers.
     Instance Attributes:
     Representation Invariants:
```

A name (which should not be empty)
 A location (latitude and longitude, just

design.

class FoodDeliverySystem:

In this exercise, you will design this class.

A location (latitude and longitude, just like vendors and customers)
 A current order, which is either None (if they have no order currently assigned to them) or an Order

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

- A *current order*, which is either **None** (if they have 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.

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.

2. Write an example use of this data class as a doctest example.

- 4. Can two Order objects refer to the same Courier instance? Why or why not?
 - "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

Brainstorm two or three other instance attributes you could add to the Courier data class to better model

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.

Representation Invariants: - self.name != '' - all(vendor == self._vendors[vendor].name for vendor in self._vendors)

- all(customer == self._customers[customer].name for customer in self._customers

- all(courier == self._couriers[courier].name for courier in self._couriers)

"""A system that maintains all entities (vendors, customers, couriers, and orders).

```
# Private Instance Attributes:
# - _vendors: a mapping from vendor name to Vendor object.
# This represents all the vendors 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).
```

_vendors: dict[str, Vendor]
 _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.

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

The system starts with no entities.
 """

1. Implement the FoodDeliverySystem initializer, which simply initializes all of the instance attributes to

2. Implement the FoodDeliverySystem.add_vendor method, which adds a new food vendor to the

system. Because the FoodDeliverySystem keeps track of all entities, it can check uniqueness constraints

across all the vendors—something that individual Vendor instances can't check for.

def add_vendor(self, vendor: Vendor) -> bool:

"""Add the given vendor to this system.

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

Return whether the vendor was successfully added to this system.

If you have time, implement analogous methods add_customer and add_courier for this class.