CSC110 Lecture 30: Discrete-Event Simulations

David Liu and Tom Fairgrieve, Department of Computer Science

*Navigation tip for web slides: press **?** to see keyboard navigation controls.*

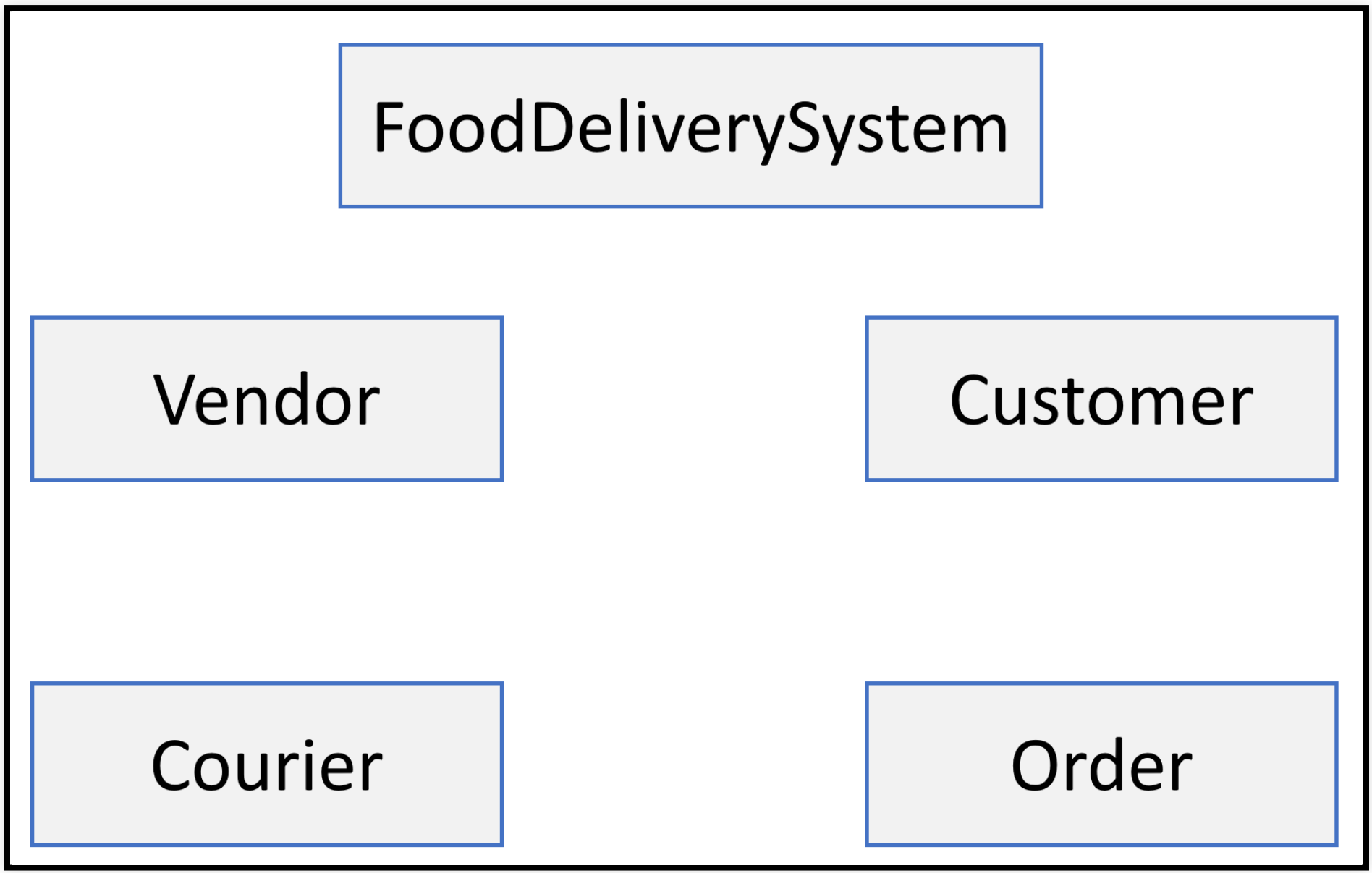
Announcements and Today's plan

Announcements

- Please complete the PythonTA Survey 2
  - Due December 8
- Last tutorial tomorrow
- Last lecture on Tuesday, December 6

Recapping our food delivery system

Seeing the proliferation of various food delivery apps, you have decided to create a food and grocery delivery app that focuses on students. Your app will allow student users to order groceries and meals from local grocery stores and restaurants. The deliveries will be made by couriers to deliver these groceries and meals—and you'll need to pay the couriers, of course!


FoodDeliverySystem
Vendor
Customer
Courier
Order

entities.py

```python
@dataclass
class Vendor:
    name: str
    address: str
    menu: dict[str, float]
    location: tuple[float, float]


@dataclass
class Customer:
    name: str
    location: tuple[float, float]
```

```python
@dataclass
class Courier:
    name: str
    location: tuple[float, float]
    current_order: Optional[Order] = None


@dataclass
class Order:
    customer: Customer
    vendor: Vendor
    food_items: dict[str, int]
    start_time: datetime.datetime
    courier: Optional[Courier] = None
    end_time: Optional[datetime.datetime]
```

food_delivery_system.py

```
class FoodDeliverySystem:
    _vendors: dict[str, Vendor]
    _customers: dict[str, Customer]
    _couriers: dict[str, Courier]
    _orders: list[Order]

    ...

    def place_order(self, order: Order) -> bool:
        """Add an order to this system."""

    def complete_order(self, order: Order,
                       timestamp: datetime.datetime) -> None:
```

Today you'll learn to...

1. Define event classes to represent individual changes in a system.
2. Create a discrete-event simulation to represent several changes to a system over a period of time.

Prompting mutation

`FoodDeliverySystem` has methods to mutate its state (e.g., `add_customer`, `place_order`).

But where are these methods actually called?

Discrete-event simulation: a computational model of system over time, where changes occur due to individual events.

1. What are the events?
2. How do we process many events over time?

The Event public interface (1)

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

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

Each `Event` has a method to mutate the underlying `FoodDeliverySystem` (by calling `FoodDeliverySystem` methods).

But also: every event should have a timestamp representing when that event takes place.

The Event public interface (2)

```
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 ev
        """
        raise NotImplementedError
```

Demo: `NewOrderEvent`

Goal: implement an event class representing when a customer places a new order.

Calling the superclass initializer (summary)

When `B` is a suclass of `A`, and `A` defines its own `__init__` method:

1. `B.__init__` must call `A.__init__` to initialize all common attributes.
2. `B.__init__` is responsible for initializing any additional attributes that are specific to `B`.

Exercise 1: Representing events

Generating new events

Events trigger `FoodDeliverySystem` mutation. But where do these events come from in a simulation?

Key idea: in a discrete event simulation, handling an event can cause future events to occur.

Examples:

- After handling a `NewOrderEvent`, we expect a corresponding `CompleteOrderEvent` to happen in the future.
- After a new customer joins, they place a few different orders to try out some food vendors.

Changing the `Event` interface

```
class Event:
    ...

    def handle_event(self, system: FoodDeliverySystem) -> list[Even
        """Mutate the given food delivery system to process this ev

        (NEW) Return a new list of new events created by processing
        this event.
        """
        raise NotImplementedError
```

To PyCharm!

A new event type

A `GenerateOrdersEvent` is our simulation's "initial" event type. Its purpose is to randomly create `NewOrderEvents` over a set time period. (E.g., around dinner time many new orders get created.)

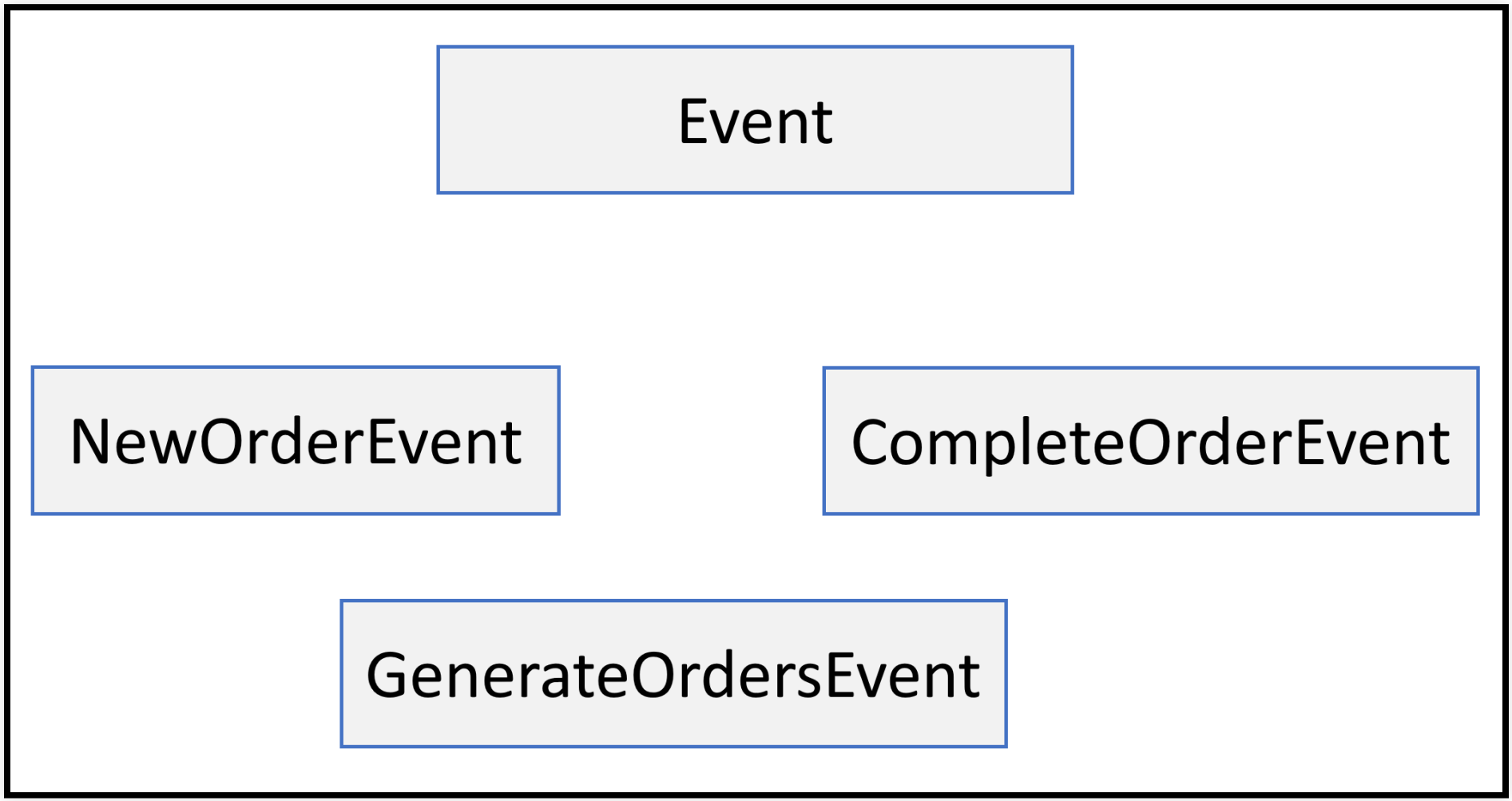
```
class GenerateOrdersEvent(Event):
    """An event that causes a random generation of new orders.

    Private Representation Invariants:
        - self._duration > 0
    """
    # Private Instance Attributes:
    #   - _duration: the number of hours to generate orders for
    _duration: int
```

To PyCharm!

Exercise 2: The `GenerateOrdersEvent`

The main simulation loop


Event
NewOrderEvent
CompleteOrderEvent
GenerateOrdersEvent

So we have:

- `GenerateOrdersEvent` can create `NewOrderEvent`s
- `NewOrderEvent`s can create `CompleteOrderEvent`s

But for this to happen, the `handle_event` method needs to be called!

There needs to be some code that calls `handle_event` on every event.

If the events were given all at once, we could just do:

```
for event in events:
    event.handle_event(system)
```

But we don’t start with all events…

Idea: keep track of events in a collection. Every time an event is handled, add the events it generates into the collection.

```
while events is not empty:
    event = get next event from events
    new_events = event.handle_event(system)

    for new_event in new_events:
        add new_event to events
```

What abstract data type should we use to store the events?

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

```
    events = EventQueueList()
    for event in initial_events:
        events.enqueue(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)
```

`EventQueueList` is a priority queue that uses the `timestamp` attribute as the priority. **Demo!**

Summary

Today you learned to...

1. Define event classes to represent individual changes in a system.
2. Create a discrete-event simulation to represent several changes to a system over a period of time.

Homework

- Readings:
  - From last class: 11.1, 11.2, 11.3
  - From today: 11.4, 11.5
  - For next class: review the posted code
- **Last tutorial** on Friday!

Good luck with your MAT137 test!