```
class Container:
    """A container that holds objects.

    This is an abstract class.  Only child classes should be
instantiated.
    """

    def add(self, item):
        """Add <item> to this Container.

        @type self: Container
        @type item: Object
        @rtype: None
        """
        raise NotImplementedError("Implemented in a subclass")

    def remove(self):
        """Remove and return a single item from this Container.

        @type self: Container
        @rtype: Object
        """
        raise NotImplementedError("Implemented in a subclass")

    def is_empty(self):
        """Return True iff this Container is empty.

        @type self: Container
        @rtype: bool
        """
        raise NotImplementedError("Implemented in a subclass")


class PriorityQueue(Container):
    """A queue of items that operates in priority order.

    Items are removed from the queue according to priority; the item with
the
    highest priority is removed first. Ties are resolved in FIFO order,
    meaning the item which was inserted *earlier* is the first one to be
    removed.

    Priority is defined by the rich comparison methods for the objects in
the
    container (__lt__, __le__, __gt__, __ge__).

    If x < y, then x has a *HIGHER* priority than y.

    All objects in the container must be of the same type.
    """

    # === Private Attributes ===
    # @type _items: list
    #     The items stored in the priority queue.
```

```
#
# === Representation Invariants ===
# _items is a sorted list, where the first item in the queue is the
# item with the highest priority.

def __init__(self):
    """Initialize an empty PriorityQueue.

    @type self: PriorityQueue
    @rtype: None
    """
    self._items = []

def remove(self):
    """Remove and return the next item from this PriorityQueue.

    Precondition: <self> should not be empty.

    @type self: PriorityQueue
    @rtype: object

    >>> pq = PriorityQueue()
    >>> pq.add("red")
    >>> pq.add("blue")
    >>> pq.add("yellow")
    >>> pq.add("green")
    >>> pq.remove()
    'blue'
    >>> pq.remove()
    'green'
    >>> pq.remove()
    'red'
    >>> pq.remove()
    'yellow'
    """
    return self._items.pop(0)

def is_empty(self):
    """
    Return true iff this PriorityQueue is empty.

    @type self: PriorityQueue
    @rtype: bool

    >>> pq = PriorityQueue()
    >>> pq.is_empty()
    True
    >>> pq.add("thing")
    >>> pq.is_empty()
    False
    """
    return len(self._items) == 0

def add(self, item):
```

```
        """Add <item> to this PriorityQueue.

        @type self: PriorityQueue
        @type item: object
        @rtype: None

        >>> pq = PriorityQueue()
        >>> pq.add("yellow")
        >>> pq.add("blue")
        >>> pq.add("red")
        >>> pq.add("green")
        >>> pq.items
        ['blue', 'green', 'red', 'yellow']
        """
        queue = self._items
        if len(queue) == 0:
            queue.append(item)
        else:
            i = 0
            while i < len(queue) and item.__gt__(queue[i]):
                i += 1
            queue.insert(i, item)

    @property
    def items(self):
        return self._items


if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

```
from driver import Driver
from rider import Rider
from location import Location


class Dispatcher:
    """A dispatcher fulfills requests from riders and drivers for a
    ride-sharing service.

    When a rider requests a driver, the dispatcher assigns a driver to
the
    rider. If no driver is available, the rider is placed on a waiting
    list for the next available driver. A rider that has not yet been
    picked up by a driver may cancel their request.

    When a driver requests a rider, the dispatcher assigns a rider from
    the waiting list to the driver. If there is no rider on the waiting
list
    the dispatcher does nothing. Once a driver requests a rider, the
driver
    is registered with the dispatcher, and will be used to fulfill future
    rider requests.
    """

    def __init__(self):
        """Initialize a Dispatcher.

        @type self: Dispatcher
        @rtype: None
        """
        self.waiting_riders = []
        self.available_drivers = []

    def __str__(self):
        """Return a string representation.

        @type self: Dispatcher
        @rtype: str
        """
        return "Waiting riders:{}, Drivers available {}".\
            format(self.waiting_riders, self.available_drivers)

    def request_driver(self, rider):
        """Return a driver for the rider, or None if no driver is
available.

        Add the rider to the waiting list if there is no available
driver.

        @type self: Dispatcher
        @type rider: Rider
        @rtype: Driver | None
        >>> dis = Dispatcher()
        >>> dis.available_drivers = []
```

```
        >>> r1 = Rider("C", Location(0, 0), Location(10, 23), 10)
        >>> dis.request_driver(r1) is None
        True
        >>> dis1 = Dispatcher()
        >>> d1 = Driver("C", Location(0, 0), 10)
        >>> d2 = Driver("B", Location(3, 3), 10)
        >>> dis1.available_drivers = [d1, d2]
        >>> r1 = Rider("C", Location(0, 0), Location(10, 23), 10)
        >>> dis1.request_driver(r1) == d1
        True
        """
        if len(self.available_drivers) == 0:
            self.waiting_riders.append(rider)
            return None
        else:
            if len(self.available_drivers) == 1:
                fastest_driver = self.available_drivers[0]
                fastest_driver.is_idle = False
                return fastest_driver
            else:
                fastest_driver = self.available_drivers[0]
                for available_driver in self.available_drivers[1:]:
                    if available_driver.start_drive(rider.origin) <=\
                            fastest_driver.start_drive(rider.origin):
                                fastest_driver = available_driver
                fastest_driver.is_idle = False
                return fastest_driver

    def request_rider(self, driver):
        """Return a rider for the driver, or None if no rider is
available.

        If this is a new driver, register the driver for future rider
requests.

        @type self: Dispatcher
        @type driver: Driver
        @rtype: Rider | None
        >>> dis = Dispatcher()
        >>> dis.waiting_riders = []
        >>> dis.request_rider(Driver("A", Location(1,1), 3)) is None
        True
        >>> dis1 = Dispatcher()
        >>> r1 = Rider("C", Location(0, 0), Location(10, 23), 10)
        >>> r2 = Rider("B", Location(3, 3), Location(10, 22), 10)
        >>> dis1.waiting_riders = [r1, r2]
        >>> d1 = Driver("D", Location(3, 3), 10)
        >>> dis1.request_rider(d1) == r1
        True
        """
        if driver not in self.available_drivers:
            self.available_drivers.append(driver)
        if len(self.waiting_riders) == 0:
            return None
```

```
    else:
        selected_rider = self.waiting_riders.pop(0)
        driver.is_idle = False
        return selected_rider

def cancel_ride(self, rider):
    """Cancel the ride for rider.

    @type self: Dispatcher
    @type rider: Rider
    @rtype: None
    """
    if rider in self.waiting_riders:
        self.waiting_riders.remove(rider)
```

```
from location import Location, manhattan_distance
from rider import Rider


class Driver:
    """A driver for a ride-sharing service.

    === Attributes ===
    @type id: str
        A unique identifier for the driver.
    @type location: Location
        The current location of the driver.
    @type is_idle: bool
        A property that is True if the driver is idle and False
otherwise.
    """

    def __init__(self, identifier, location, speed):
        """Initialize a Driver.

        @type self: Driver
        @type identifier: str
        @type location: Location
        @type speed: int
        @rtype: None
        """
        self.id = identifier
        self.location = location
        self.speed = speed
        self.is_idle = True
        self._loc_end_drive = None
        self._loc_end_ride = None

    def __str__(self):
        """Return a string representation.

        @type self: Driver
        @rtype: str

        >>> d = Driver("Chris",Location(3,1),10)
        >>> print(d)
        Chris 3,1 10
        """
        return "{} {} {}".format(self.id, self.location, self.speed)

    def __eq__(self, other):
        """Return True if self equals other, and false otherwise.

        @type self: Driver
        @rtype: bool

        >>> d1 = Driver("Chris",Location(3,1),10)
        >>> d2 = Driver("Chris",Location(3,1),10)
        >>> d3 = Driver("Jack",Location(4,5),10)
```

```
        >>> d1 == d2
        True
        >>> d1 == d3
        False
        """
        return (type(self) == type(other) and
                self.id == other.id and
                self.location == other.location and
                self.speed == other.speed)

    def get_travel_time(self, destination):
        """Return the time it will take to arrive at the destination,
        rounded to the nearest integer.

        @type self: Driver
        @type destination: Location
        @rtype: int

        >>>
Driver("Chris",Location(3,1),10).get_travel_time(Location(10,14))
        2
        >>>
Driver("Jack",Location(20,1),2).get_travel_time(Location(10,7))
        8
        """
        d = manhattan_distance(self.location, destination)
        travel_time = d / self.speed
        return round(travel_time)

    def start_drive(self, location):
        """Start driving to the location and return the time the drive
will take.

        @type self: Driver
        @type location: Location
        @rtype: int

        >>> Driver("D",Location(0,0),1).start_drive(Location(10,11))
        21
        >>> Driver("Vic",Location(10,20),5).start_drive(Location(0,5))
        5
        """
        self.is_idle = False
        drive_distance = manhattan_distance(self.location, location)
        drive_time = float(drive_distance) / self.speed
        self._loc_end_drive = location
        return round(drive_time)

    def end_drive(self):
        """End the drive and arrive at the destination.

        Precondition: self.destination is not None.

        @type self: Driver
```

```
        @rtype: None
        """
        self.is_idle = False
        self.location = self._loc_end_drive

    def start_ride(self, rider):
        """Start a ride and return the time the ride will take.

        @type self: Driver
        @type rider: Rider
        @rtype: int

        >>> d = Driver("D",Location(0,0),1)
        >>> r = Rider("C",Location(0,0),Location(10,10),10)
        >>> d.start_ride(r)
        20
        >>> d.is_idle
        False
        """
        self.is_idle = False
        ride_distance = manhattan_distance(self.location,
rider.destination)
        ride_time = float(ride_distance) / self.speed
        self._loc_end_ride = rider.destination
        return round(ride_time)

    def end_ride(self):
        """End the current ride, and arrive at the rider's destination.

        Precondition: The driver has a rider.
        Precondition: self.destination is not None.

        @type self: Driver
        @rtype: None

        """
        self.is_idle = True
        self.location = self._loc_end_ride

if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

```
"""Simulation Events

This file should contain all of the classes necessary to model the
different
kinds of events in the simulation.
"""
from rider import Rider, WAITING, CANCELLED, SATISFIED
from dispatcher import Dispatcher
from driver import Driver
from location import deserialize_location
from monitor import Monitor, RIDER, DRIVER, REQUEST, CANCEL, PICKUP,
DROPOFF


class Event:
    """An event.

    Events have an ordering that is based on the event timestamp: Events
with
    older timestamps are less than those with newer timestamps.

    This class is abstract; subclasses must implement do().

    You may, if you wish, change the API of this class to add
    extra public methods or attributes. Make sure that anything
    you add makes sense for ALL events, and not just a particular
    event type.

    Document any such changes carefully!

    === Attributes ===
    @type timestamp: int
        A timestamp for this event.
    """

    def __init__(self, timestamp):
        """Initialize an Event with a given timestamp.

        @type self: Event
        @type timestamp: int
            A timestamp for this event.
            Precondition: must be a non-negative integer.
        @rtype: None

        >>> Event(7).timestamp
        7
        """
        self.timestamp = timestamp

    # The following six 'magic methods' are overridden to allow for easy
    # comparison of Event instances. All comparisons simply perform the
    # same comparison on the 'timestamp' attribute of the two events.
    def __eq__(self, other):
        """Return True iff this Event is equal to <other>.
```

```
    Two events are equal iff they have the same timestamp.

    @type self: Event
    @type other: Event
    @rtype: bool

    >>> first = Event(1)
    >>> second = Event(2)
    >>> first == second
    False
    >>> second.timestamp = first.timestamp
    >>> first == second
    True
    """
    return self.timestamp == other.timestamp

def __ne__(self, other):
    """Return True iff this Event is not equal to <other>.

    @type self: Event
    @type other: Event
    @rtype: bool

    >>> first = Event(1)
    >>> second = Event(2)
    >>> first != second
    True
    >>> second.timestamp = first.timestamp
    >>> first != second
    False
    """
    return not self == other

def __lt__(self, other):
    """Return True iff this Event is less than <other>.

    @type self: Event
    @type other: Event
    @rtype: bool

    >>> first = Event(1)
    >>> second = Event(2)
    >>> first < second
    True
    >>> second < first
    False
    """
    return self.timestamp < other.timestamp

def __le__(self, other):
    """Return True iff this Event is less than or equal to <other>.

    @type self: Event
```

```
        @type other: Event
        @rtype: bool

        >>> first = Event(1)
        >>> second = Event(2)
        >>> first <= first
        True
        >>> first <= second
        True
        >>> second <= first
        False
        """
        return self.timestamp <= other.timestamp

    def __gt__(self, other):
        """Return True iff this Event is greater than <other>.

        @type self: Event
        @type other: Event
        @rtype: bool

        >>> first = Event(1)
        >>> second = Event(2)
        >>> first > second
        False
        >>> second > first
        True
        """
        return not self <= other

    def __ge__(self, other):
        """Return True iff this Event is greater than or equal to
<other>.

        @type self: Event
        @type other: Event
        @rtype: bool

        >>> first = Event(1)
        >>> second = Event(2)
        >>> first >= first
        True
        >>> first >= second
        False
        >>> second >= first
        True
        """
        return not self < other

    def __str__(self):
        """Return a string representation of this event.

        @type self: Event
        @rtype: str
```

```
        """
        raise NotImplementedError("Implemented in a subclass")

    def do(self, dispatcher, monitor):
        """Do this Event.

        Update the state of the simulation, using the dispatcher, and any
        attributes according to the meaning of the event.

        Notify the monitor of any activities that have occurred during
the
        event.

        Return a list of new events spawned by this event (making sure
the
        timestamps are correct).

        Note: the "business logic" of what actually happens should not be
        handled in any Event classes.

        @type self: Event
        @type dispatcher: Dispatcher
        @type monitor: Monitor
        @rtype: list[Event]
        """
        raise NotImplementedError("Implemented in a subclass")


class RiderRequest(Event):
    """A rider requests a driver.

    === Attributes ===
    @type rider: Rider
        The rider.
    """

    def __init__(self, timestamp, rider):
        """Initialize a RiderRequest event.

        @type self: RiderRequest
        @type rider: Rider
        @rtype: None
        """
        super().__init__(timestamp)
        self.rider = rider

    def do(self, dispatcher, monitor):
        """Assign the rider to a driver or add the rider to a waiting
list.
        If the rider is assigned to a driver, the driver starts driving
to
        the rider.
```

```
        Return a Cancellation event. If the rider is assigned to a
driver,
        also return a Pickup event.

        @type self: RiderRequest
        @type dispatcher: Dispatcher
        @type monitor: Monitor
        @rtype: list[Event]
        """
        monitor.notify(self.timestamp, RIDER, REQUEST,
                       self.rider.id, self.rider.origin)

        events = []
        driver = dispatcher.request_driver(self.rider)
        if driver is not None:
            travel_time = driver.start_drive(self.rider.origin)
            events.append(Pickup(self.timestamp + travel_time,
self.rider,
                                 driver))
        events.append(Cancellation(self.timestamp + self.rider.patience,
                                   self.rider))
        return events

    def __str__(self):
        """Return a string representation of this event.

        @type self: RiderRequest
        @rtype: str
        """
        return "{} -- {}: Request a driver".format(self.timestamp,
self.rider)


class DriverRequest(Event):
    """A driver requests a rider.

    === Attributes ===
    @type driver: Driver
        The driver.
    """

    def __init__(self, timestamp, driver):
        """Initialize a DriverRequest event.

        @type self: DriverRequest
        @type driver: Driver
        @rtype: None
        """
        super().__init__(timestamp)
        self.driver = driver

    def do(self, dispatcher, monitor):
        """Register the driver, if this is the first request, and
        assign a rider to the driver, if one is available.
```

```
        If a rider is available, return a Pickup event.

        @type self: DriverRequest
        @type dispatcher: Dispatcher
        @type monitor: Monitor
        @rtype: list[Event]
        """
        # Notify the monitor about the request.
        monitor.notify(self.timestamp, DRIVER, REQUEST,
                       self.driver.id, self.driver.location)
        # Request a rider from the dispatcher.
        events = []
        rider = dispatcher.request_rider(self.driver)
        # If there is one available, the driver starts driving towards
the
        # rider, and the method returns a Pickup event for when the
driver
        # arrives at the riders location.
        if rider is not None:
            travel_time = self.driver.start_drive(rider.origin)
            events.append(Pickup(self.timestamp + travel_time, rider,
                                 self.driver))

        return events

    def __str__(self):
        """Return a string representation of this event.

        @type self: DriverRequest
        @rtype: str
        """
        return "{} -- {}: Request a rider".format(self.timestamp,
self.driver)


class Cancellation(Event):
    """Change a waiting rider to a cancelled rider

    === Attributes ===
    @type rider: Rider
    """
    def __init__(self, timestamp, rider):
        """

        @type self: Cancellation
        @type rider: Rider
        """
        super().__init__(timestamp)
        self.rider = rider

    def do(self, dispatcher, monitor):
        """Cancel the ride if waiting time is greater than rider's
patience
```

```
        @type self: Cancellation
        @type dispatcher: Dispatcher
        @type monitor: Monitor
        @rtype: None
        """
        if self.rider.status == WAITING:
            driver = dispatcher.request_driver(self.rider)
            if driver is None:
                pass
            else:
                travel_time = driver.start_drive(self.rider.origin)
                if self.timestamp < self.timestamp - self.rider.patience\
                        + travel_time:
                    self.rider.status = CANCELLED
                    driver.is_idle = True
                    dispatcher.cancel_ride(self.rider)
                    monitor.notify(self.timestamp, RIDER, CANCEL,
                                   self.rider.id, self.rider.origin)
                    monitor.notify(self.timestamp, DRIVER, CANCEL,
                                   driver.id, self.rider.origin)

    def __str__(self):
        """Return a string representation of cancellation event
        @type self: Cancellation
        @rtype: str
        """
        return "{} -- {}: Cancel the ride".format(self.timestamp,
self.rider)


class Pickup(Event):
    """The driver picks up the rider.
    === Attributes ===
    @type driver: Driver
    @type rider: Rider
    """

    def __init__(self, timestamp, rider, driver):
        """Initialize Pickup

        @type self: Pickup
        @type driver: Driver
        @type rider: Rider
        """
        super().__init__(timestamp)
        self.driver = driver
        self.rider = rider

    def do(self, dispatcher, monitor):
        """Modify driver's location and rider's status when driver
arrives at
        riders location
```

```
        @type self: Pickup
        @type dispatcher: Dispatcher
        @type monitor: Monitor
        @rtype: list[Events]
        """
        events = []
        if self.rider.status == WAITING:
            travel_time = self.driver.start_drive(self.rider.origin)
            if self.timestamp < self.timestamp - travel_time\
                    + self.rider.patience:
                self.driver.location = self.rider.origin
                monitor.notify(self.timestamp, RIDER, PICKUP,
                               self.rider.id, self.rider.origin)
                monitor.notify(self.timestamp, DRIVER, PICKUP,
                               self.driver.id, self.driver.location)
                dispatcher.available_drivers.remove(self.driver)
                # Append dropoff event
                travel_time = \
                    self.driver.get_travel_time(self.rider.destination)
                self.driver.location = self.rider.destination
                self.rider.status = SATISFIED
                events.append(Dropoff(self.timestamp + travel_time,
self.rider,
                                      self.driver))
                return events

        if self.rider.status == CANCELLED:
            self.driver.location = self.rider.origin
            self.driver.is_idle = True
            events.append(DriverRequest(self.timestamp, self.driver))
            monitor.notify(self.timestamp, DRIVER, REQUEST,
                           self.driver.id, self.driver.location)
            return events

    def __str__(self):
        """Return a string representation of pickup event
        @type self: Pickup
        @rtype: str
        """
        return "{} -- {} -- {}: Pick up the rider".format(self.timestamp,
                                                          self.driver,
                                                          self.rider)


class Dropoff(Event):
    """The driver drops off the rider and requests a new rider

    === Attributes ===
    @type driver: Driver
    @type rider: Rider
    """

    def __init__(self, timestamp, rider, driver):
        """Initialize Dropoff
```

```
        @type self: Dropoff
        @type driver: Driver
        @type rider: Rider
        """
        super().__init__(timestamp)
        self.rider = rider
        self.driver = driver

    def do(self, dispatcher, monitor):
        """Drop off the rider and requests for a new rider

        @type self: Dropoff
        @type dispatcher: Dispatcher
        @type monitor: Monitor
        @rtype: list[Events]
        """
        events = []
        self.driver.location = self.rider.destination
        monitor.notify(self.timestamp, RIDER, DROPOFF,
                       self.rider.id, self.driver.location)
        monitor.notify(self.timestamp, DRIVER, DROPOFF,
                       self.driver.id, self.driver.location)
        events.append(DriverRequest(self.timestamp, self.driver))
        self.driver.is_idle = True
        dispatcher.available_drivers.append(self.driver)

    def __str__(self):
        """Return a string representation of dropoff event
        @type self: Dropoff
        @rtype: str
        """
        return "{} -- {} -- {}: Dropoff the rider".format(self.timestamp,
                                                          self.driver,
                                                          self.rider)


def create_event_list(filename):
    """Return a list of Events based on raw list of events in <filename>.

    Precondition: the file stored at <filename> is in the format
specified
    by the assignment handout.

    @param filename: str
        The name of a file that contains the list of events.
    @rtype: list[Event]
    """
    events = []
    with open(filename, "r") as file:
        for line in file:
            line = line.strip()

            if not line or line.startswith("#"):
```

```
                # Skip lines that are blank or start with #.
                continue

            # Create a list of words in the line, e.g.
            # ['10', 'RiderRequest', 'Cerise', '4,2', '1,5', '15'].
            # Note that these are strings, and you'll need to convert
some
            # of them to a different type.
            tokens = line.split()
            timestamp = int(tokens[0])
            event_type = tokens[1]

            # HINT: Use Location.deserialize to convert the location
string to
            # a location.
            if event_type == "DriverRequest":
                # Create a DriverRequest event.
                driver = Driver(tokens[2],
deserialize_location(tokens[3]),
                                int(tokens[4]))
                event = DriverRequest(timestamp, driver)
                events.append(event)

            elif event_type == "RiderRequest":
                # Create a RiderRequest event.
                rider = Rider(tokens[2], deserialize_location(tokens[3]),
                              deserialize_location(tokens[4]),
int(tokens[5]))
                event = RiderRequest(timestamp, rider)
                events.append(event)
    return events
```

```
# Sample Event List

# The parser will skip empty lines, lines with whitespace only,
# or those that start with '#'.

# The format for DriverRequest events is:
# <timestamp> DriverRequest <driver id> <location> <speed>
# <location> is <row>,<col>

0 DriverRequest Amaranth 1,1 1
0 DriverRequest Bergamot 1,2 1
0 DriverRequest Crocus 3,1 1
0 DriverRequest Dahlia 3,2 1
0 DriverRequest Edelweiss 4,2 1
0 DriverRequest Foxglove 5,2 1

# The format for RiderRequest events is:
# <timestamp> RiderRequest <rider id> <origin> <destination> <patience>
# <origin>, <destination> are <row>,<col>

0 RiderRequest Almond 1,1 5,5 10
5 RiderRequest Bisque 3,2 2,3 5
10 RiderRequest Cerise 4,2 1,5 15
15 RiderRequest Desert 5,1 4,3 5
20 RiderRequest Eggshell 3,4 3,1 1
25 RiderRequest Fallow 2,1 2,5 10
```

```
#At time 1, Dan exists
#Dan is at location 1,1, requests a driver, and is willing
#to wait 15 units of time for pickup before he cancels
1 RiderRequest Dan 1,1 6,6 15

#At time 10, Arnold exists
#Arnold is at location 3,3, requests a rider,
#and Arnold's car moves 2 units of distance per unit time
10 DriverRequest Arnold 3,3 2
```

```
class Location:
    """
    Our simulation plays out on a simplified grid of city blocks.
    Each location is specified by a pair of (row, column)

    Attribute:
    ==========
    @type row: non-negative integer
         number of blocks from the bottom
    @type column: non-negative integer
         number of blocks from the left
    """

    def __init__(self, row, column):
        """Initialize a location.

        @type self: Location
        @type row: int
        @type column: int
        @rtype: None
        """
        self.row = row
        self.column = column

    def __str__(self):
        """Return a string representation.

        @type self: Location
        @rtype: str

        >>> l = Location (2,3)
        >>> print(l)
        2,3
        """
        return "{},{}".format(self.row, self.column)

    def __eq__(self, other):
        """Return True if self equals other, and false otherwise.

        @type self:Location
        @type other: Location | Any
        @rtype: bool

        >>> l1 = Location (3,4)
        >>> l2 = Location (3, 4)
        >>> l3 = Location (5,6)
        >>> l1 == l2
        True
        >>> l1 == l3
        False
        """
        return (type(self) == type(other) and
                self.column == other.column and
                self.row == other.row)
```

```
def manhattan_distance(origin, destination):
    """Return the Manhattan distance between the origin and the
destination.

    @type origin: Location
    @type destination: Location
    @rtype: int

    >>> manhattan_distance(Location(2,3),Location(5,7))
    7
    >>> manhattan_distance(Location(2,3),Location(2,7))
    4
    """
    return (abs(origin.column - destination.column) +
            abs(origin.row - destination.row))


def deserialize_location(location_str):
    """Deserialize a location.

    @type location_str: str
        A location in the format 'row,col'
    @rtype: Location

    >>> d1 = deserialize_location('24,35')
    >>> print (d1)
    24,35
    >>> d2 = deserialize_location('6,4')
    >>> print(d2)
    6,4
    """
    l = Location(0, 0)
    index_comma = location_str.find(',')
    l.row = int(location_str[:index_comma])
    l.column = int(location_str[index_comma+1:])
    return l


if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

```
from location import Location
from location import manhattan_distance

"""
The Monitor module contains the Monitor class, the Activity class,
and a collection of constants. Together the elements of the module
help keep a record of activities that have occurred.

Activities fall into two categories: Rider activities and Driver
activities. Each activity also has a description, which is one of
request, cancel, pickup, or dropoff.

=== Constants ===
@type RIDER: str
    A constant used for the Rider activity category.
@type DRIVER: str
    A constant used for the Driver activity category.
@type REQUEST: str
    A constant used for the request activity description.
@type CANCEL: str
    A constant used for the cancel activity description.
@type PICKUP: str
    A constant used for the pickup activity description.
@type DROPOFF: str
    A constant used for the dropoff activity description.
"""

RIDER = "rider"
DRIVER = "driver"

REQUEST = "request"
CANCEL = "cancel"
PICKUP = "pickup"
DROPOFF = "dropoff"


class Activity:
    """An activity that occurs in the simulation.

    === Attributes ===
    @type timestamp: int
        The time at which the activity occurred.
    @type description: str
        A description of the activity.
    @type identifier: str
        An identifier for the person doing the activity.
    @type location: Location
        The location at which the activity occurred.
    """

    def __init__(self, timestamp, description, identifier, location):
        """Initialize an Activity.

        @type self: Activity
```

```
        @type timestamp: int
        @type description: str
        @type identifier: str
        @type location: Location
        @rtype: None
        """
        self.description = description
        self.time = timestamp
        self.id = identifier
        self.location = location


class Monitor:
    """A monitor keeps a record of activities that it is notified about.
    When required, it generates a report of the activities it has
recorded.
    """

    # === Private Attributes ===
    # @type _activities: dict[str, dict[str, list[Activity]]]
    #       A dictionary whose key is a category, and value is another
    #       dictionary. The key of the second dictionary is an identifier
    #       and its value is a list of Activities.

    def __init__(self):
        """Initialize a Monitor.

        @type self: Monitor
        """
        self._activities = {
            RIDER: {},
            DRIVER: {}
        }
        """@type _activities: dict[str, dict[str, list[Activity]]]"""

    def __str__(self):
        """Return a string representation.

        @type self: Monitor
        @rtype: str

        >>> m = Monitor()
        >>> A1 = Activity(1,"request","Chris",Location(0,0))
        >>> A2 = Activity(3,"pickup","Chris",Location(10,2))
        >>> A3 = Activity(5,"request","Jack",Location(1,3))
        >>> A4 = Activity(6,"pickup","Louis",Location(6,7))
        >>> A5 = Activity(2,"request","Chen",Location(3,3))
        >>> A6 = Activity(6,"cancel","Chen",Location(3,3))
        >>> A7 = Activity(10,"pickup","LK",Location(0,0))
        >>> m._activities = {RIDER:{"Chen":[A5,A6],"Jack":[A3]},\
        DRIVER:{"Chris":[A1,A2],"Louis":[A4],"LK":[A7]}}
        >>> print(m)
        Monitor (3 drivers, 2 riders)
        """
```

```
        return "Monitor ({} drivers, {} riders)".format(
                len(self._activities[DRIVER]),
len(self._activities[RIDER]))

    def notify(self, timestamp, category, description, identifier,
location):
        """Notify the monitor of the activity.

        @type self: Monitor
        @type timestamp: int
            The time of the activity.
        @type category: DRIVER | RIDER
            The category for the activity.
        @type description: REQUEST | CANCEL | PICKUP | DROP_OFF
            A description of the activity.
        @type identifier: str
            The identifier for the actor.
        @type location: Location
            The location of the activity.
        @rtype: None

        """
        if identifier not in self._activities[category]:
            self._activities[category][identifier] = []

        activity = Activity(timestamp, description, identifier, location)
        self._activities[category][identifier].append(activity)

    def report(self):
        """Return a report of the activities that have occurred.

        @type self: Monitor
        @rtype: dict[str, object]
        """
        return {"rider_wait_time": self._average_wait_time(),
                "driver_total_distance": self._average_total_distance(),
                "driver_ride_distance": self._average_ride_distance()}

    def _average_wait_time(self):
        """Return the average wait time of riders that have either been
picked
        up or have cancelled their ride.

        @type self: Monitor
        @rtype: float

        >>> m1 = Monitor()
        >>> A1 = Activity(1,"request","Chris",Location(0,0))
        >>> A2 = Activity(3,"pickup","Chris",Location(10,2))
        >>> A3 = Activity(5,"request","Jack",Location(1,3))
        >>> A4 = Activity(6,"request","Louis",Location(6,7))
        >>> A5 = Activity(2,"request","Chen",Location(3,3))
        >>> A6 = Activity(6,"cancel","Chen",Location(3,3))
        >>> A7 = Activity(9,"pickup","Louis",Location(0,0))
```

```
        >>> m1._activities = {RIDER:{"Chris":[A1,A2],"Jack":[A3],\
        "Louis":[A4,A7],"Chen":[A5,A6]},DRIVER:{}}
        >>> m1.average_wait_time()
        3.0
        """
        wait_time = 0
        count = 0
        for activities in self._activities[RIDER].values():
            # A rider that has less than two activities hasn't finished
            # waiting (they haven't cancelled or been picked up).
            if len(activities) >= 2:
                # The first activity is REQUEST, and the second is PICKUP
                # or CANCEL. The wait time is the difference between the
two.
                wait_time += activities[1].time - activities[0].time
                count += 1
        return wait_time / count

    @property
    def average_wait_time(self):
        return self._average_wait_time()

    def _average_total_distance(self):
        """Return the average distance drivers have driven.

        @type self: Monitor
        @rtype: float

        >>> m2 = Monitor()
        >>> A1 = Activity(1,"request","Chris",Location(0,0))
        >>> A2 = Activity(3,"pickup","Chris",Location(10,2))
        >>> A3 = Activity(5,"dropoff","Chris",Location(1,3))
        >>> A4 = Activity(6,"request","Louis",Location(6,7))
        >>> A5 = Activity(2,"request","Chen",Location(3,3))
        >>> A6 = Activity(6,"cancel","Chen",Location(4,3))
        >>> A7 = Activity(9,"cancel","Louis",Location(0,0))
        >>> m2._activities = {RIDER:{},DRIVER:{"Chris":[A1,A2,A3],\
        "Louis":[A4,A7],"Chen":[A5,A6]}}
        >>> m2.average_total_distance()
        12.0
        """
        total_distance = 0
        num_drivers = len(self._activities[DRIVER])
        for activities in self._activities[DRIVER].values():
            i = 1
            while i < len(activities):
                total_distance +=
manhattan_distance(activities[i].location,
                                                   activities[i-
1].location)
                i += 1
        return total_distance / num_drivers

    @property
```

```
    def average_total_distance(self):
        return self._average_total_distance()

    def _average_ride_distance(self):
        """Return the average distance drivers have driven on rides.

        @type self: Monitor
        @rtype: float

        >>> m3 = Monitor()
        >>> A1 = Activity(1,"request","Chris",Location(0,0))
        >>> A2 = Activity(3,"pickup","Chris",Location(10,2))
        >>> A3 = Activity(5,"dropoff","Chris",Location(1,3))
        >>> A4 = Activity(6,"request","Louis",Location(6,7))
        >>> A5 = Activity(2,"request","Chen",Location(3,3))
        >>> A6 = Activity(6,"pickup","Chen",Location(4,0))
        >>> A7 = Activity(10,"dropoff","Chen",Location(13,20))
        >>> A8 = Activity(9,"cancel","Louis",Location(0,0))
        >>> m3._activities = {RIDER:{},DRIVER:{"Chris":[A1,A2,A3],\
        "Louis":[A4,A8],"Chen":[A5,A6,A7]}}
        >>> m3.average_ride_distance()
        13.0
        """
        ride_distance = 0
        num_drivers = len(self._activities[DRIVER])
        for activities in self._activities[DRIVER].values():
            for activity in activities:
                if activity.description == PICKUP:
                    i = activities.index(activity)
                    ride_distance += manhattan_distance(
                        activities[i + 1].location,
activities[i].location)
        return ride_distance / num_drivers

    @property
    def average_ride_distance(self):
        return self._average_ride_distance()
```

```
from location import Location
"""
The rider module contains the Rider class. It also contains
constants that represent the status of the rider.

=== Constants ===
@type WAITING: str
    A constant used for the waiting rider status.
@type CANCELLED: str
    A constant used for the cancelled rider status.
@type SATISFIED: str
    A constant used for the satisfied rider status
"""

WAITING = "waiting"
CANCELLED = "cancelled"
SATISFIED = "satisfied"


class Rider:
    """A rider for a ride-sharing service.

    === Attributes ===
    @type id: str
        A unique identifier for the rider.
    @type origin: Location
        The starting location
    @type destination: Location
        The ending location
    @type status: str
        WAITING, CANCELLED or SATISFIED
    @type patience: int
    The number of time units the rider will wait to be picked up before
they
    cancel their ride
    """

    def __init__(self, identifier, origin, destination, patience):
        """Initialize a Driver.

        @type self:Rider
        @type identifier: str
        @type origin: Location
        @type destination: Location
        @type patience: int
        @rtype: None
        """
        self.id = identifier
        self.origin = origin
        self.destination = destination
        self.patience = patience
        self.status = WAITING

    def __str__(self):
```

```
        """Return a string representation.

        @type self: Rider
        @rtype: str

        >>> r = Rider("Alex",Location(1,2),Location(3,4), 12)
        >>> print(r)
        Alex 1,2 3,4 12
        """
        return "{} {} {} {}".format(self.id, self.origin,
self.destination,
                                    self.patience)

    def __eq__(self, other):
        """Return True if self equals other, and false otherwise.

        @type self: Rider
        @type other: Rider | Any
        @rtype:bool

        >>> r1 = Rider("Alex",Location(1,2),Location(3,4), 10)
        >>> r2 = Rider("Alex",Location(1,2),Location(3,4),10)
        >>> r3 = Rider("Shawn",Location(3,2),Location(2,4), 15)
        >>> r1 == r2
        True
        >>> r1 == r3
        False
        """
        return (type(self) == type(other) and
                self.id == other.id and
                self.origin == other.origin and
                self.destination == other.destination and
                self.patience == other.patience)

if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

```
from container import PriorityQueue
from dispatcher import Dispatcher
from event import create_event_list
from monitor import Monitor


class Simulation:
    """A simulation.

    This is the class which is responsible for setting up and running a
    simulation.

    The API is given to you: your main task is to implement the run
    method below according to its docstring.

    Of course, you may add whatever private attributes and methods you
want.
    But because you should not change the interface, you may not add any
public
    attributes or methods.

    This is the entry point into your program, and in particular is used
for
    auto-testing purposes. This makes it ESSENTIAL that you do not change
the
    interface in any way!
    """

    # === Private Attributes ===
    # @type _events: PriorityQueue[Event]
    #     A sequence of events arranged in priority determined by the
event
    #     sorting order.
    # @type _dispatcher: Dispatcher
    #     The dispatcher associated with the simulation.

    def __init__(self):
        """Initialize a Simulation.

        @type self: Simulation
        @rtype: None
        """
        self._events = PriorityQueue()
        self._dispatcher = Dispatcher()
        self._monitor = Monitor()

    def run(self, initial_events):
        """Run the simulation on the list of events in <initial_events>.

        Return a dictionary containing statistics of the simulation,
        according to the specifications in the assignment handout.

        @type self: Simulation
        @type initial_events: list[Event]
```

```
            An initial list of events.
        @rtype: dict[str, object]
        """
        # Add all initial events to the event queue.
        for event in initial_events:
            self._events.add(event)
            while self._events.is_empty() is False:
                executed_event = self._events.remove()
                result_events = executed_event.do(self._dispatcher,
                                                  self._monitor)
                # this warning can be ignored
                if result_events is not None:
                    for result_event in result_events:
                        self._events.add(result_event)
        # Until there are no more events, remove an event
        # from the event queue and do it. Add any returned
        # events to the event queue.
        return self._monitor.report()


if __name__ == "__main__":
    events = create_event_list("events.txt")
    sim = Simulation()
    final_stats = sim.run(events)
    print(final_stats)
```