Software Development for Unmanned Aerial Systems

Instructor:

Jane Cleland-Huang, PhD

JaneClelandHuang@nd.edu

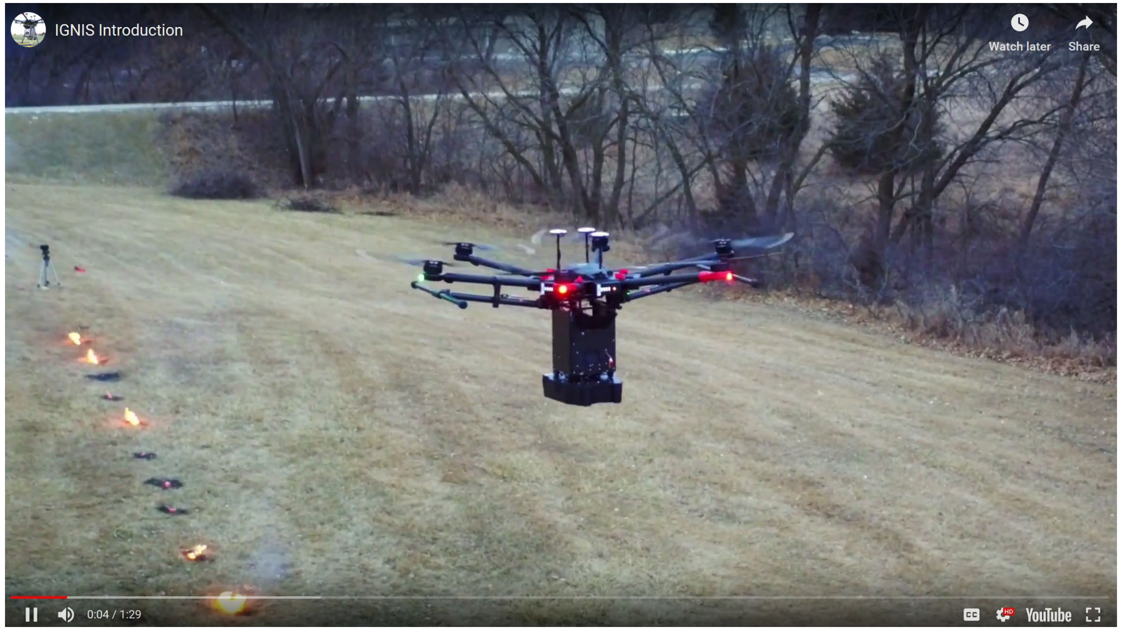
Department of Computer Science and Engineering

University of Notre Dame

Goals for Today's Class

1. Overview of Dronology's Core architecture
2. Shallow dive into Dronology's Ground Control Station
    - Messaging system
    - Functionality
3. Techniques for reverse engineering code
    - Walk throughs
    - CRC Cards
    - Sequence Diagrams
4. Hands-on execution of a GCS with Dronology Map

UAV Projects



https://www.youtube.com/watch?v=Gbzn8WBnRvI&authuser=0

https://dronology.info/defibrillator-delivery/

CRC Cards

CRC cards are typically created on index cars. Team members create one CRC card for each key class/object in their design.

The card is divided into three areas:

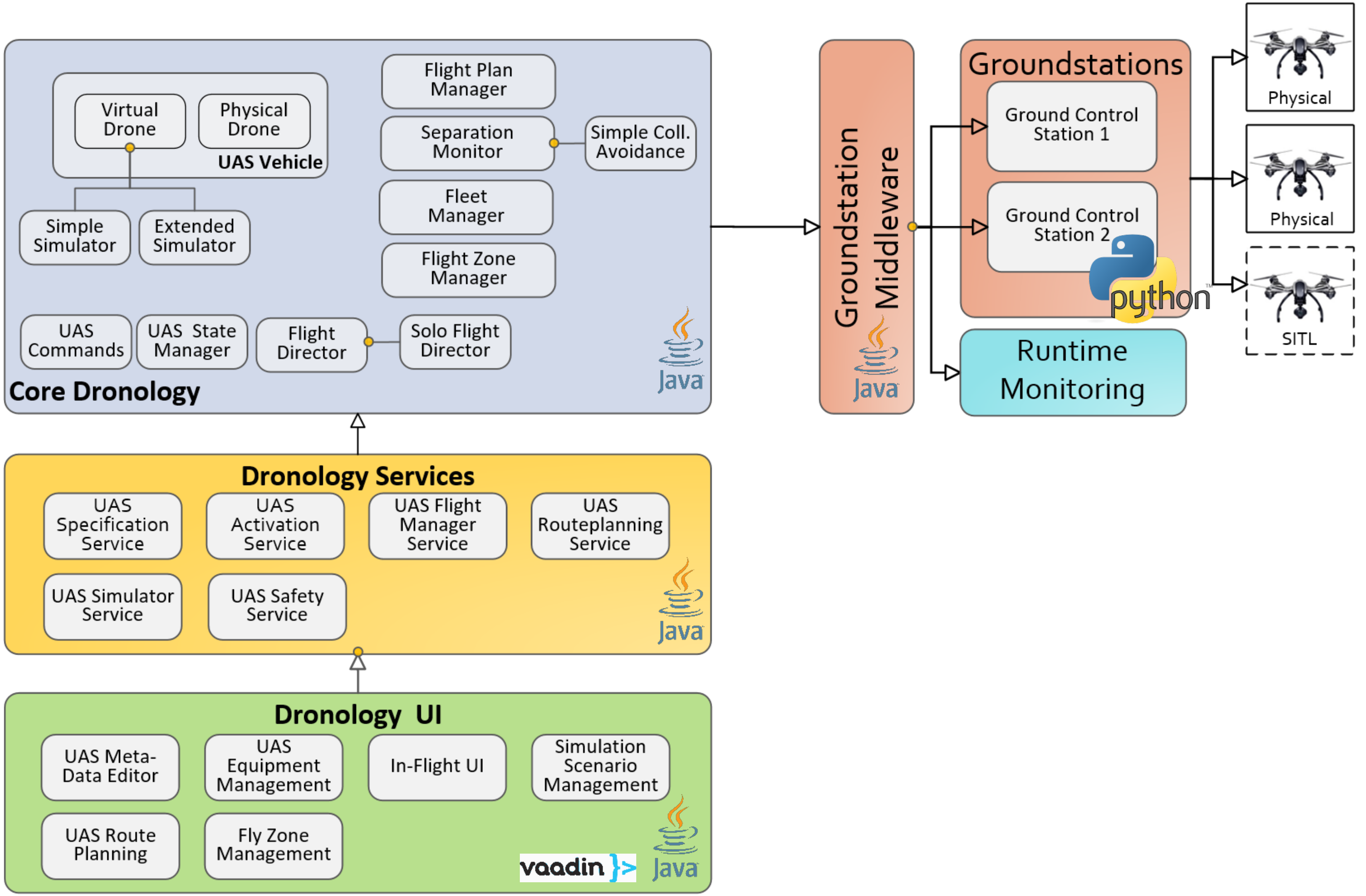
1. The **class** name at the top
2. **Responsibilities** of the class on the left
3. **Collaborators** (other classes) with which this class interacts to fulfill its responsibilities on the right.

As we look at ground control station code – we'll make **frequent stops to create CRC cards** for the key classes that we come across.

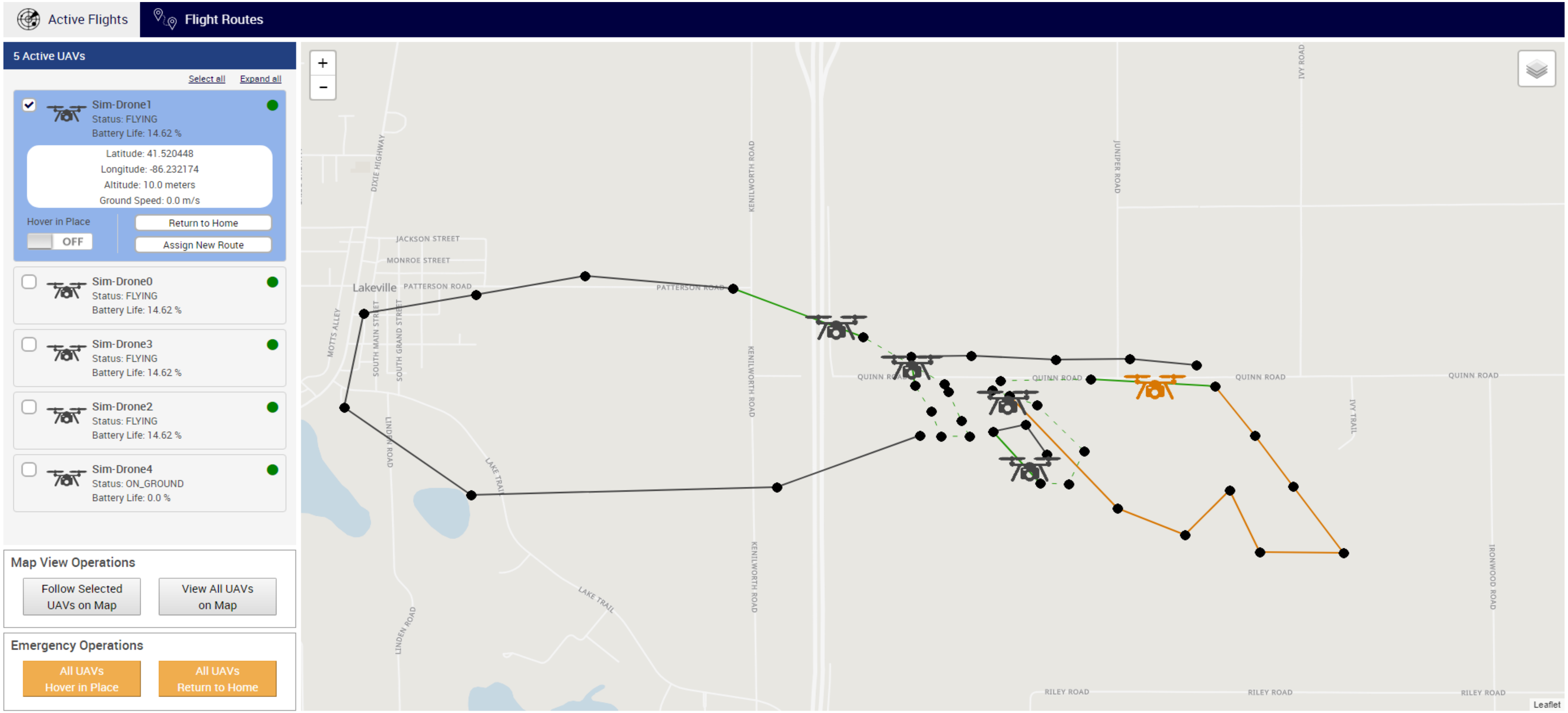
Class Name	
Responsibilities	**Collaborators**

ATM (Automatic Teller)	
Responsibility	**Collaborations**
Access and modify account balance	Account Balance Inquiry Deposit Transaction Funds Transfer Withdrawal Transaction

Dronology Architecture



Dronology in Action: Demo



☑ Sim-Drone1 ●
Status: FLYING
Battery Life: 14.62 %

Latitude: 41.520448
Longitude: -86.232174
Altitude: 10.0 meters
Ground Speed: 0.0 m/s

Hover in Place: OFF

□ Sim-Drone0 ●
Status: FLYING
Battery Life: 14.62 %

□ Sim-Drone3 ●
Status: FLYING
Battery Life: 14.62 %

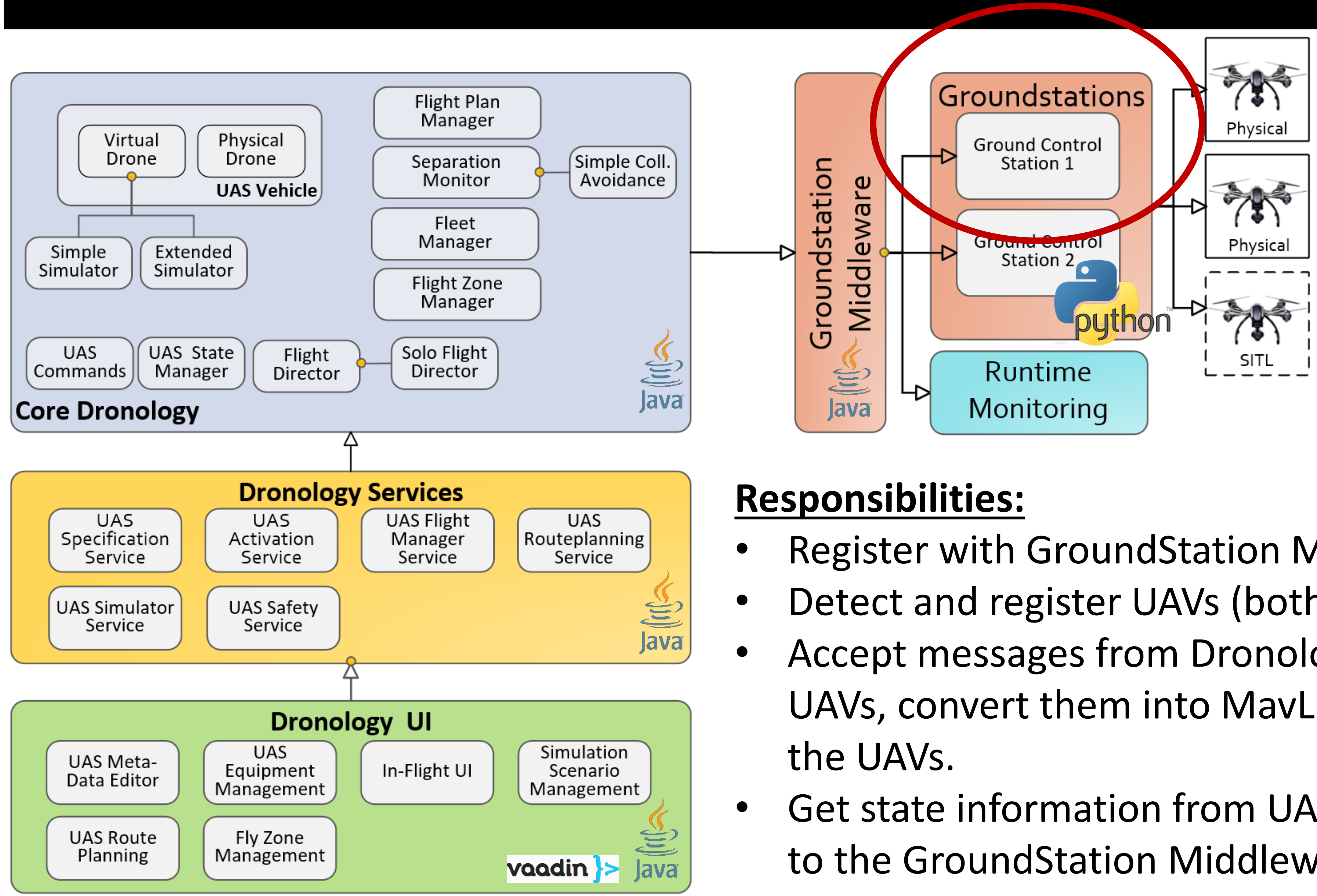
□ Sim-Drone2 ●
Status: FLYING
Battery Life: 14.62 %

□ Sim-Drone4 ●
Status: ON_GROUND
Battery Life: 0.0 %

Homework

1. Refactor your own ground control station to make it reusable for future assignments.
2. Connect it to Dronology so that you can leverage Dronology's UI interface.
3. Search and Rescue challenge

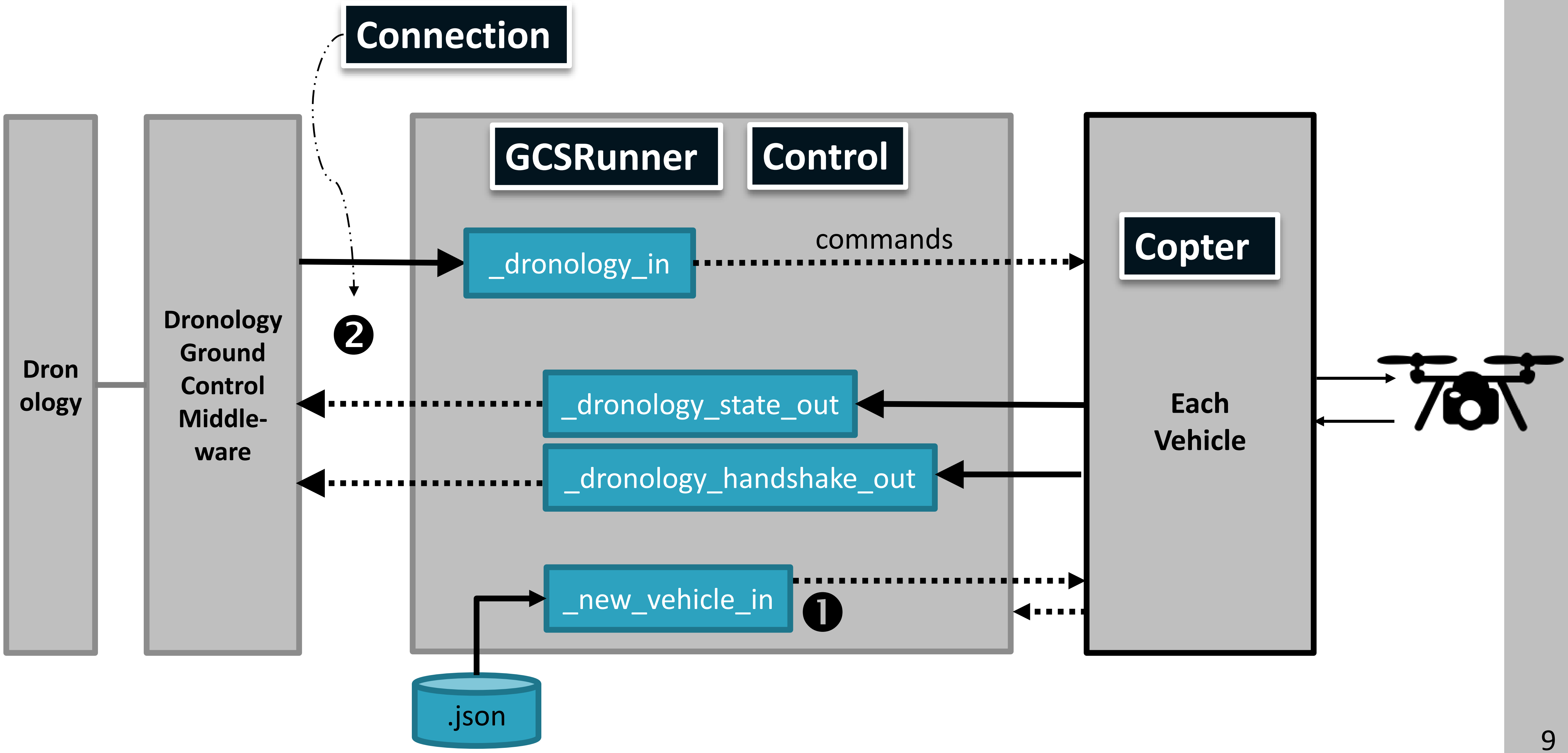
Ground Station



Responsibilities:

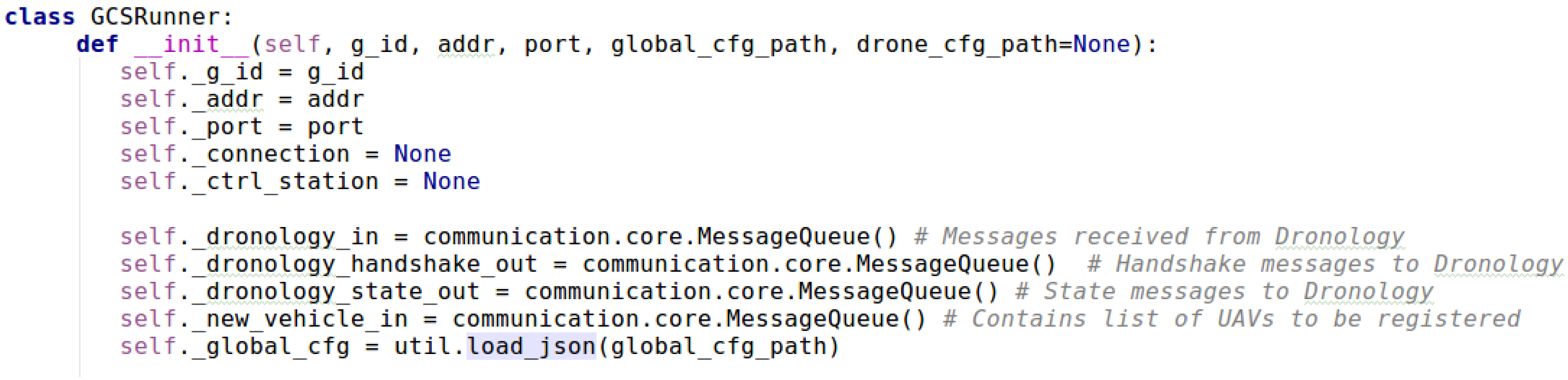
- Register with GroundStation Middleware
- Detect and register UAVs (both physical and SITL)
- Accept messages from Dronology for individual UAVs, convert them into MavLink, and forward to the UAVs.
- Get state information from UAVs and forward it to the GroundStation Middleware..

Dronology Messages



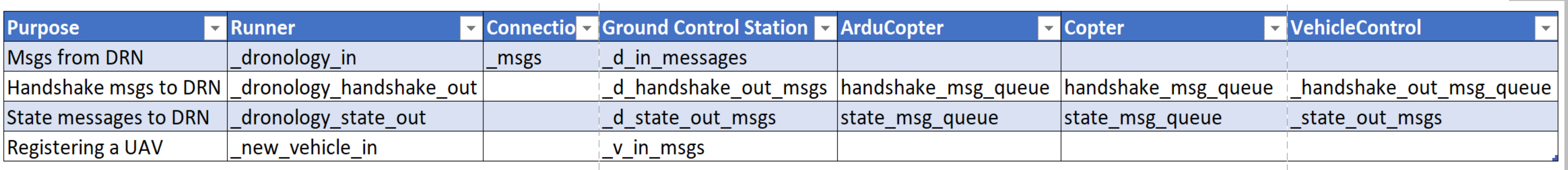
GCSRunner:

Sets up four message queues



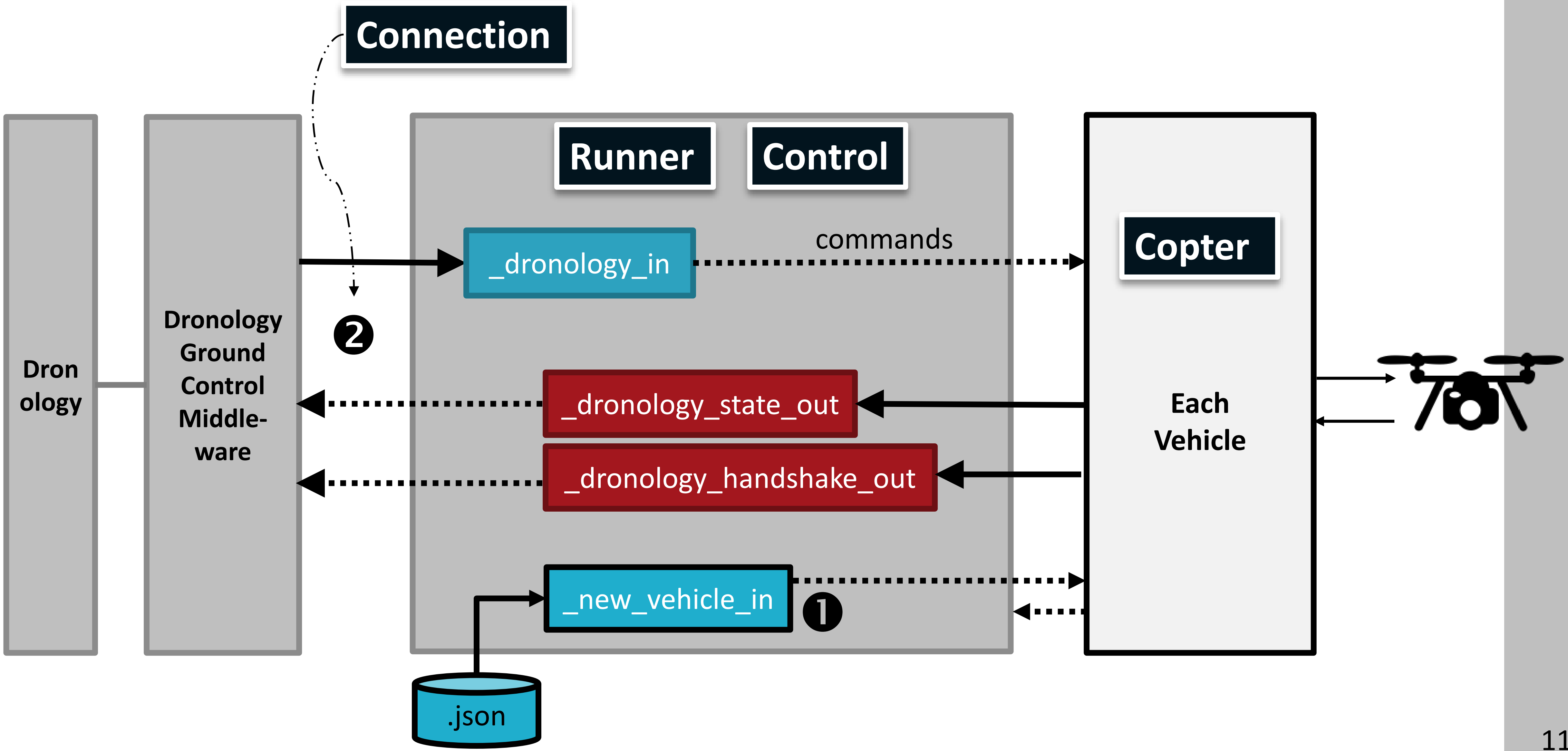
```
class GCSRunner:
    def __init__(self, g_id, addr, port, global_cfg_path, drone_cfg_path=None):
        self._g_id = g_id
        self._addr = addr
        self._port = port
        self._connection = None
        self._ctrl_station = None

        self._dronology_in = communication.core.MessageQueue() # Messages received from Dronology
        self._dronology_handshake_out = communication.core.MessageQueue()  # Handshake messages to Dronology
        self._dronology_state_out = communication.core.MessageQueue() # State messages to Dronology
        self._new_vehicle_in = communication.core.MessageQueue() # Contains list of UAVs to be registered
        self._global_cfg = util.load_json(global_cfg_path)
```

We can follow each message queue as it is referenced by other key classes (and the names they assign it)



Purpose	Runner	Connectio	Ground Control Station	ArduCopter	Copter	VehicleControl
Msgs from DRN	_dronology_in	_msgs	_d_in_messages			
Handshake msgs to DRN	_dronology_handshake_out		_d_handshake_out_msgs	handshake_msg_queue	handshake_msg_queue	_handshake_out_msg_queue
State messages to DRN	_dronology_state_out		_d_state_out_msgs	state_msg_queue	state_msg_queue	_state_out_msgs
Registering a UAV	_new_vehicle_in		_v_in_msgs			

What we will be using for this assignment



Running Dronology for our own GCS

First time only:

Build the project:

Open a new terminal

cd /home/uav/git/Dronology-Community/

mvn install

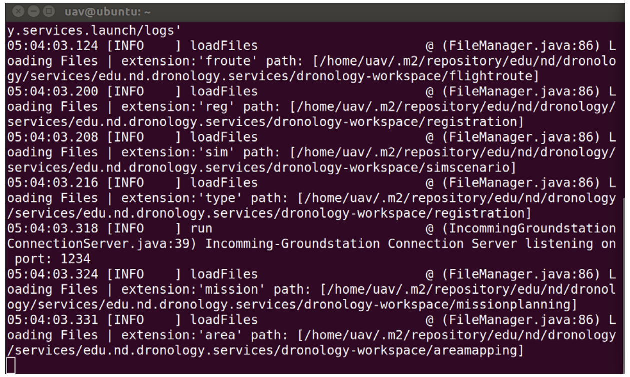
Every time:

1. **Start the Dronology Server:**
   Open a new terminal
   cd /home/uav/git/Dronology-Community/edu.nd.dronology.services.launch
   mvn exec:java
2. **Start the User Interface server:**
   Open a new terminal
   cd git/Dronology-Community/edu.nd.dronology.ui.vaadin
   mvn jetty:run
3. **Display the map:**
   Open your browser to bring up the Dronology map:
   http://localhost:8080/vaadinui

Write scripts for these repetitive tasks

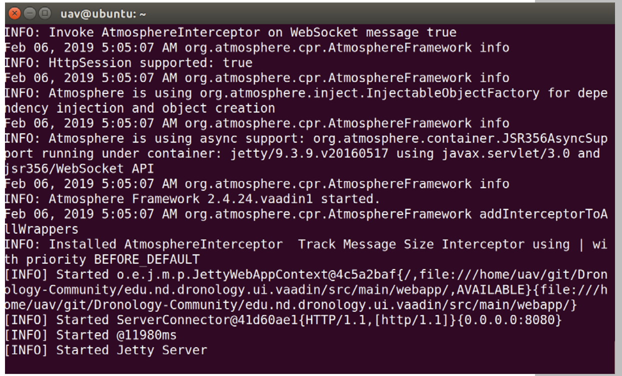
Normally to run Dronology we also start up the GCS. However, you'll be building your own in PyCharm (or whatever IDE you are using)

❶



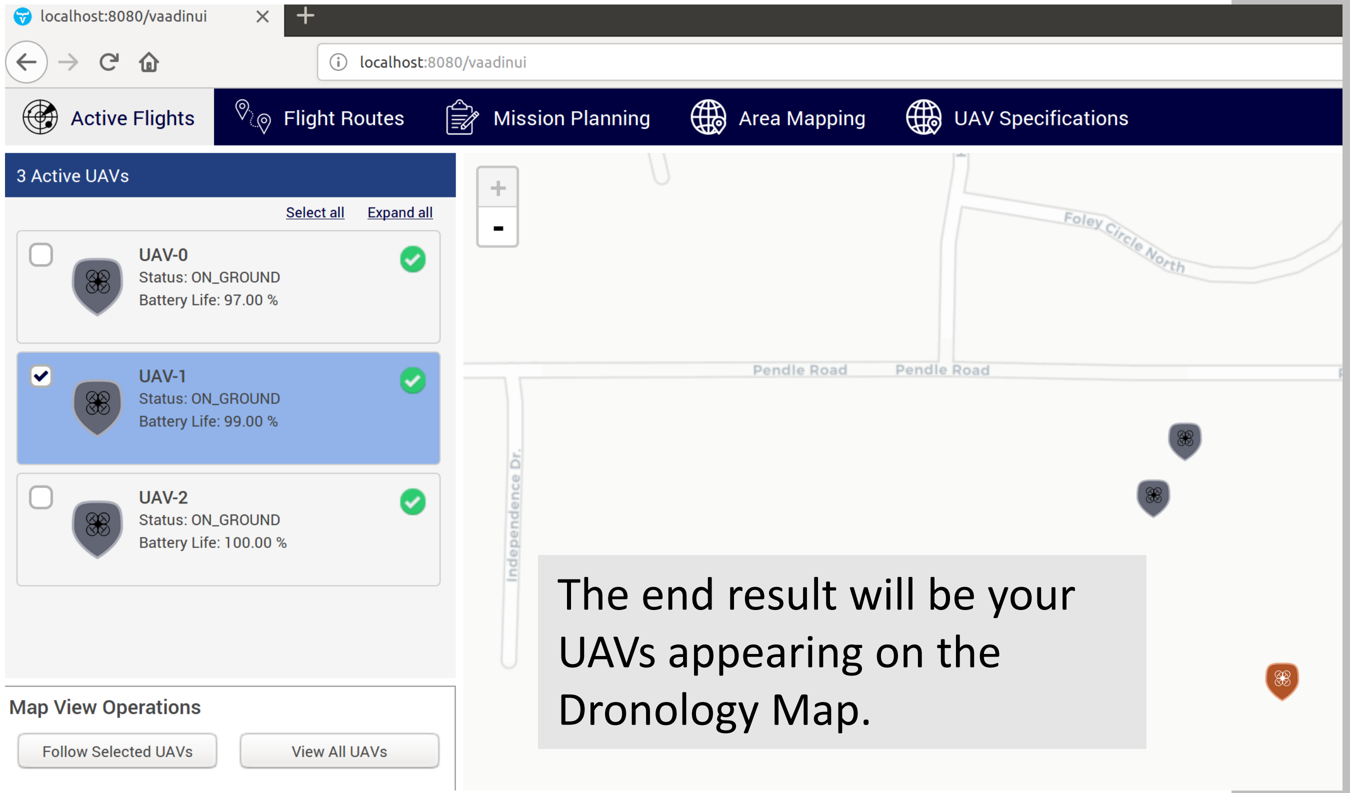
```
uav@ubuntu: ~
y.services.launch/logs'
05:04:03.124 [INFO    ] loadFiles                        @ (FileManager.java:86) L
oading Files | extension:'froute' path: [/home/uav/.m2/repository/edu/nd/dronolo
gy/services/edu.nd.dronology.services/dronology-workspace/flightroute]
05:04:03.200 [INFO    ] loadFiles                        @ (FileManager.java:86) L
oading Files | extension:'reg' path: [/home/uav/.m2/repository/edu/nd/dronology/
services/edu.nd.dronology.services/dronology-workspace/registration]
05:04:03.208 [INFO    ] loadFiles                        @ (FileManager.java:86) L
oading Files | extension:'sim' path: [/home/uav/.m2/repository/edu/nd/dronology/
services/edu.nd.dronology.services/dronology-workspace/simscenario]
05:04:03.216 [INFO    ] loadFiles                        @ (FileManager.java:86) L
oading Files | extension:'type' path: [/home/uav/.m2/repository/edu/nd/dronology
/services/edu.nd.dronology.services/dronology-workspace/registration]
05:04:03.318 [INFO    ] run                              @ (IncommingGroundstation
ConnectionServer.java:39) Incomming-Groundstation Connection Server listening on
 port: 1234
05:04:03.324 [INFO    ] loadFiles                        @ (FileManager.java:86) L
oading Files | extension:'mission' path: [/home/uav/.m2/repository/edu/nd/dronol
ogy/services/edu.nd.dronology.services/dronology-workspace/missionplanning]
05:04:03.331 [INFO    ] loadFiles                        @ (FileManager.java:86) L
oading Files | extension:'area' path: [/home/uav/.m2/repository/edu/nd/dronology
/services/edu.nd.dronology.services/dronology-workspace/areamapping]
```

❷



```
uav@ubuntu: ~
INFO: Invoke AtmosphereInterceptor on WebSocket message true
Feb 06, 2019 5:05:07 AM org.atmosphere.cpr.AtmosphereFramework info
INFO: HttpSession supported: true
Feb 06, 2019 5:05:07 AM org.atmosphere.cpr.AtmosphereFramework info
INFO: Atmosphere is using org.atmosphere.inject.InjectableObjectFactory for depe
ndency injection and object creation
Feb 06, 2019 5:05:07 AM org.atmosphere.cpr.AtmosphereFramework info
INFO: Atmosphere is using async support: org.atmosphere.container.JSR356AsyncSup
port running under container: jetty/9.3.9.v20160517 using javax.servlet/3.0 and
jsr356/WebSocket API
Feb 06, 2019 5:05:07 AM org.atmosphere.cpr.AtmosphereFramework info
INFO: Atmosphere Framework 2.4.24.vaadin1 started.
Feb 06, 2019 5:05:07 AM org.atmosphere.cpr.AtmosphereFramework addInterceptorToA
llWrappers
INFO: Installed AtmosphereInterceptor  Track Message Size Interceptor using | wi
th priority BEFORE_DEFAULT
[INFO] Started o.e.j.m.p.JettyWebAppContext@4c5a2baf{/,file:///home/uav/git/Dron
ology-Community/edu.nd.dronology.ui.vaadin/src/main/webapp/,AVAILABLE}{file:///h
ome/uav/git/Dronology-Community/edu.nd.dronology.ui.vaadin/src/main/webapp/}
[INFO] Started ServerConnector@41d60ae1{HTTP/1.1,[http/1.1]}{0.0.0.0:8080}
[INFO] Started @11980ms
[INFO] Started Jetty Server
```

❸



3 Active UAVs

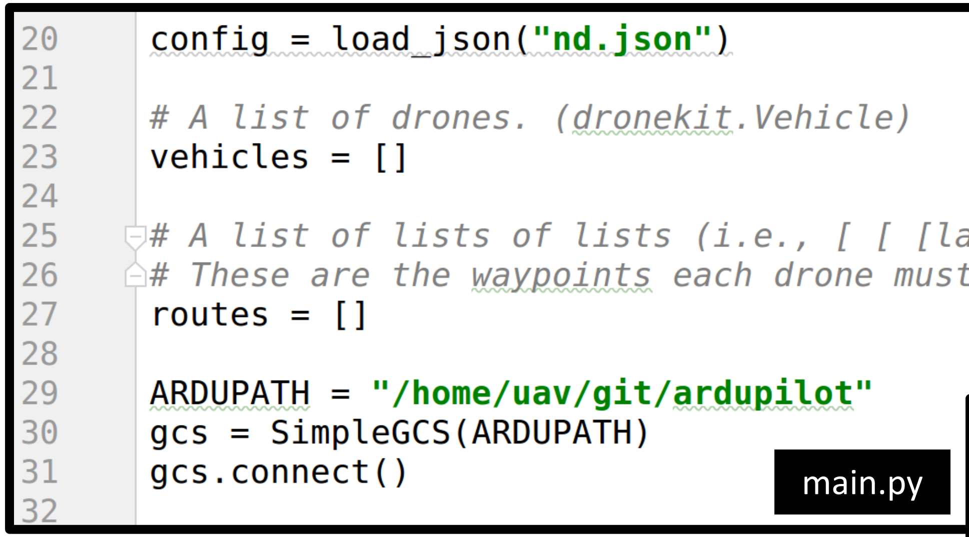
□ UAV-0
Status: ON_GROUND
Battery Life: 97.00 %

☑ UAV-1
Status: ON_GROUND
Battery Life: 99.00 %

□ UAV-2
Status: ON_GROUND
Battery Life: 100.00 %

The end result will be your UAVs appearing on the Dronology Map.

Our Simple GCS



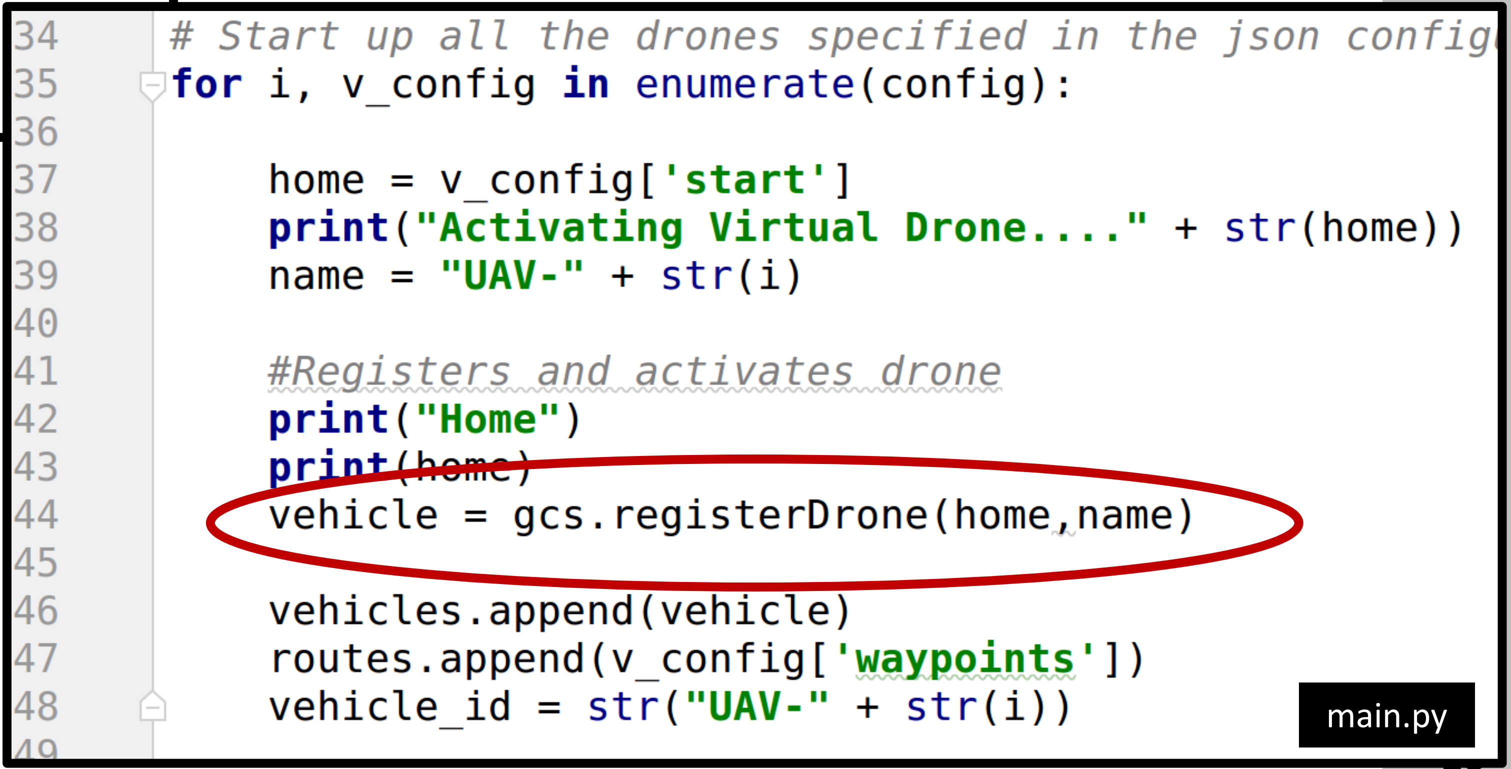
```
config = load_json("nd.json")

# A list of drones. (dronekit.Vehicle)
vehicles = []

# A list of lists of lists (i.e., [ [ [lat0, lon0
# These are the waypoints each drone must go to!
routes = []

ARDUPATH = "/home/uav/git/ardupilot"
gcs = SimpleGCS(ARDUPATH)
gcs.connect()
```

main.py

❶ Remember this from last week?

Where have copters[] and sitls[] gone?

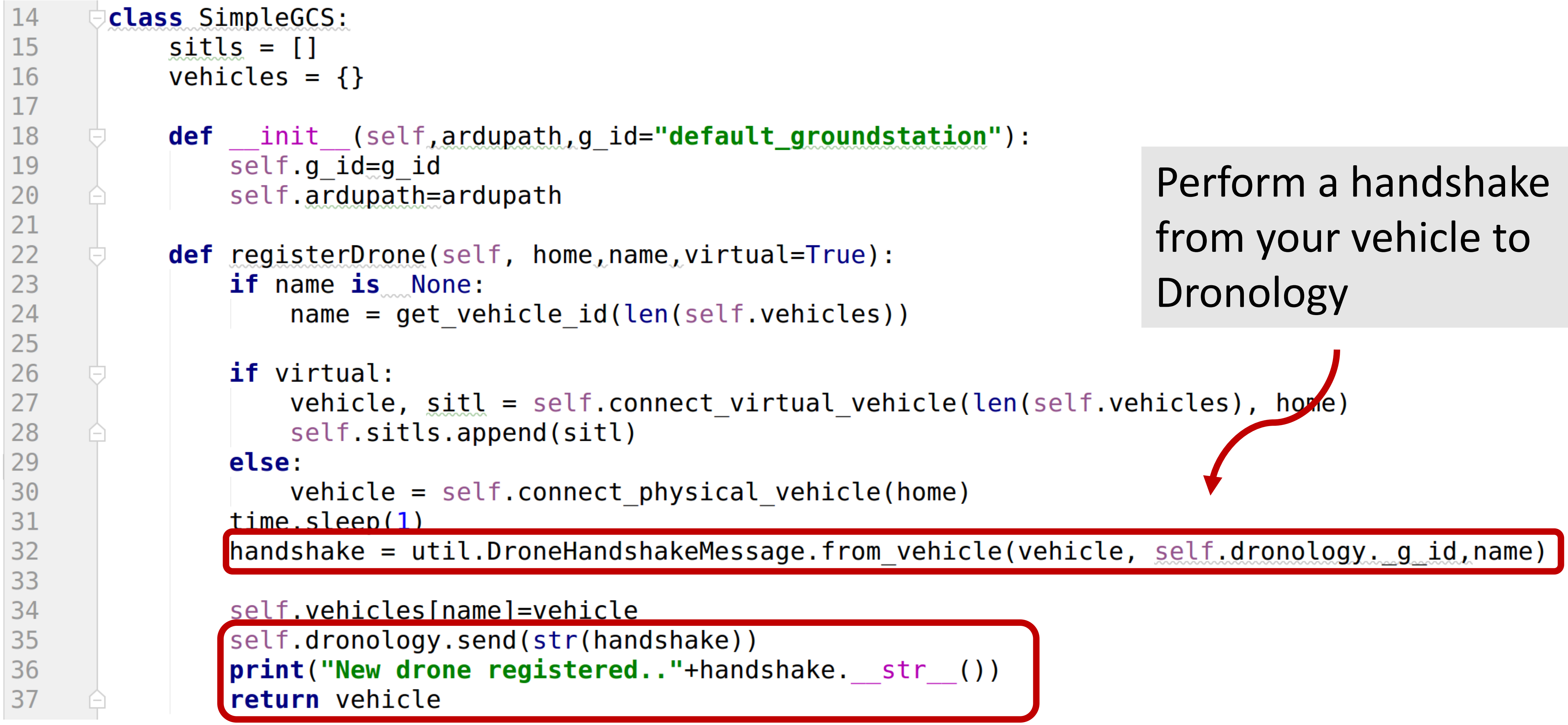
❷ We replace the Dronology GCS with our own simple version. It sends information in only one direction from our program (and our UAVs) to Dronology.



```
# Start up all the drones specified in the json config
for i, v_config in enumerate(config):

    home = v_config['start']
    print("Activating Virtual Drone...." + str(home))
    name = "UAV-" + str(i)

    #Registers and activates drone
    print("Home")
    print(home)
    vehicle = gcs.registerDrone(home,name)

    vehicles.append(vehicle)
    routes.append(v_config['waypoints'])
    vehicle_id = str("UAV-" + str(i))
```

main.py

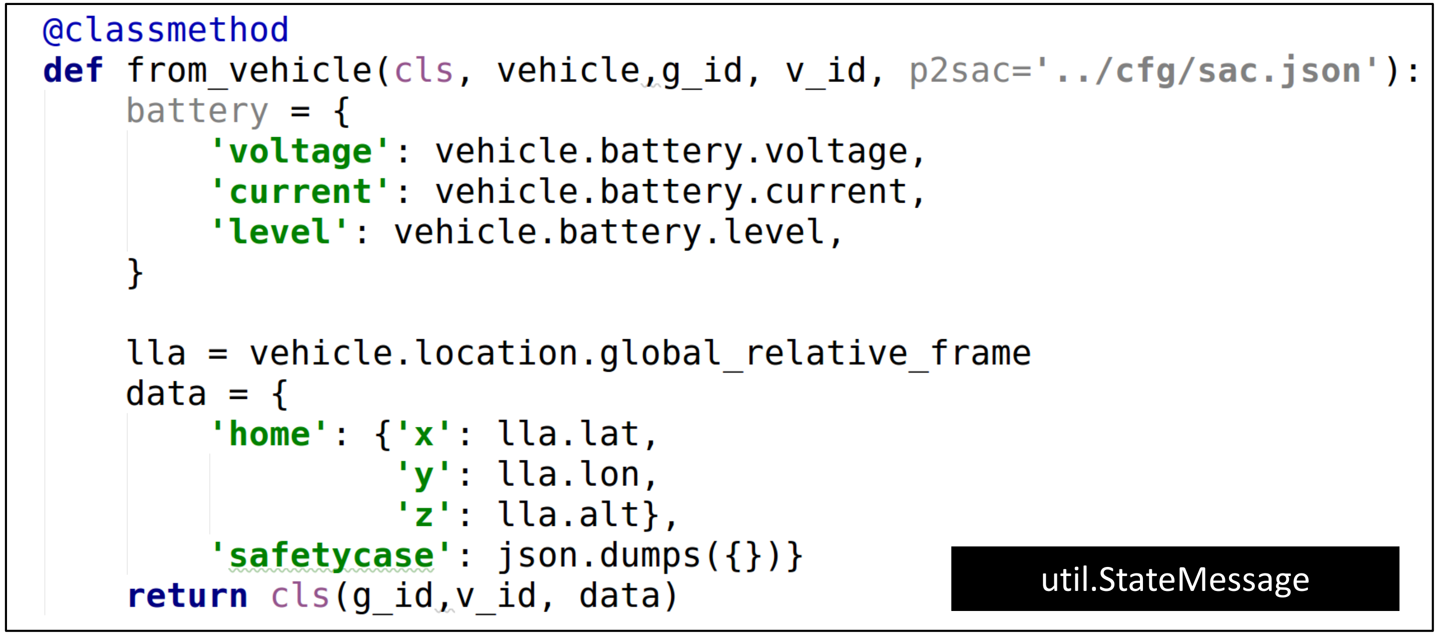
Our Simple GCS



```
class SimpleGCS:
    sitls = []
    vehicles = {}

    def __init__(self,ardupath,g_id="default_groundstation"):
        self.g_id=g_id
        self.ardupath=ardupath

    def registerDrone(self, home,name,virtual=True):
        if name is  None:
            name = get_vehicle_id(len(self.vehicles))

        if virtual:
            vehicle, sitl = self.connect_virtual_vehicle(len(self.vehicles), home)
            self.sitls.append(sitl)
        else:
            vehicle = self.connect_physical_vehicle(home)
        time.sleep(1)
        handshake = util.DroneHandshakeMessage.from_vehicle(vehicle, self.dronology._g_id,name)

        self.vehicles[name]=vehicle
        self.dronology.send(str(handshake))
        print("New drone registered.."+handshake.__str__())
        return vehicle
```

Perform a handshake from your vehicle to Dronology

DroneShakeMessage.from_vehicle

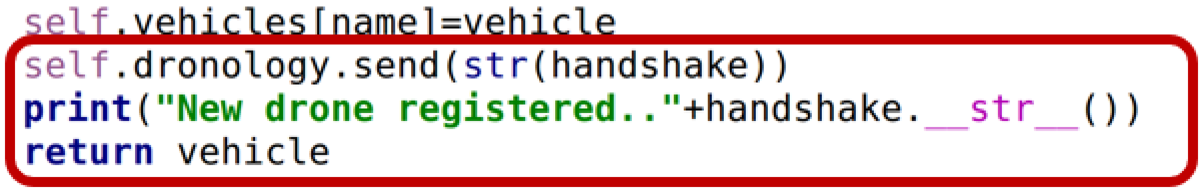


```
@classmethod
def from_vehicle(cls, vehicle,g_id, v_id, p2sac='../cfg/sac.json'):
    battery = {
        'voltage': vehicle.battery.voltage,
        'current': vehicle.battery.current,
        'level': vehicle.battery.level,
    }

    lla = vehicle.location.global_relative_frame
    data = {
        'home': {'x': lla.lat,
                 'y': lla.lon,
                 'z': lla.alt},
        'safetycase': json.dumps({})}
    return cls(g_id,v_id, data)
```

util.StateMessage

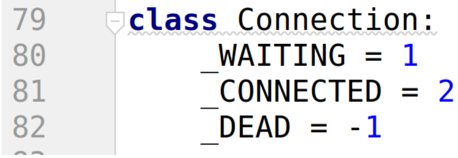
Returns GCS ID, Vehicle ID, and state data.

data": {"home": {"y": -86.2423008, "x": 41.7148673, "z": 0.0}, "safetycase": "{}"}, "groundstationid": "default_groundstation", "type": "handshake", "uavid": "UAV-1", "sendtimestamp": 1549458848503}



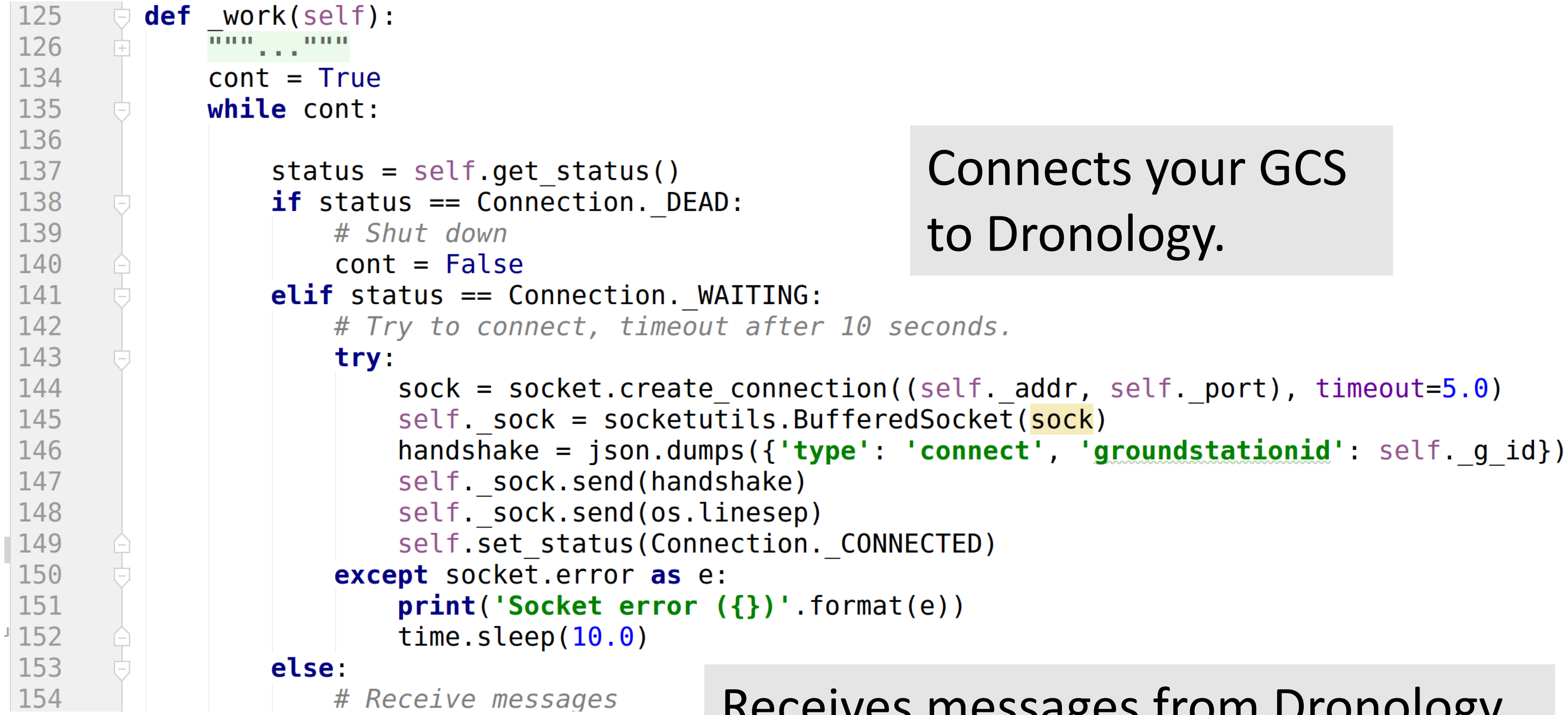
```
self.vehicles[name]=vehicle
self.dronology.send(str(handshake))
print("New drone registered.."+handshake.__str__())
return vehicle
```

Thread: util.Connection _work



```python
class Connection:
    _WAITING = 1
    _CONNECTED = 2
    _DEAD = -1
```

Manages the connection state.

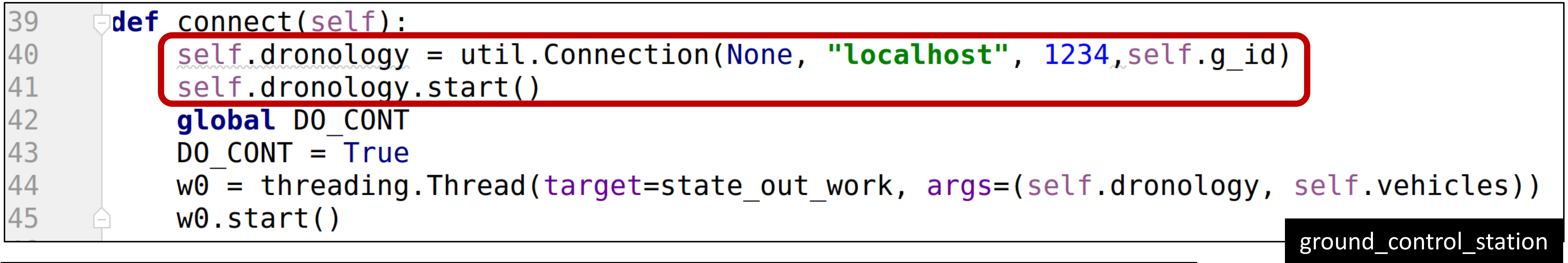


```python
def _work(self):
    """..."""
    cont = True
    while cont:

        status = self.get_status()
        if status == Connection._DEAD:
            # Shut down
            cont = False
        elif status == Connection._WAITING:
            # Try to connect, timeout after 10 seconds.
            try:
                sock = socket.create_connection((self._addr, self._port), timeout=5.0)
                self._sock = socketutils.BufferedSocket(sock)
                handshake = json.dumps({'type': 'connect', 'groundstationid': self._g_id})
                self._sock.send(handshake)
                self._sock.send(os.linesep)
                self.set_status(Connection._CONNECTED)
            except socket.error as e:
                print('Socket error ({})'.format(e))
                time.sleep(10.0)
        else:
            # Receive messages
```

Connects your GCS to Dronology.

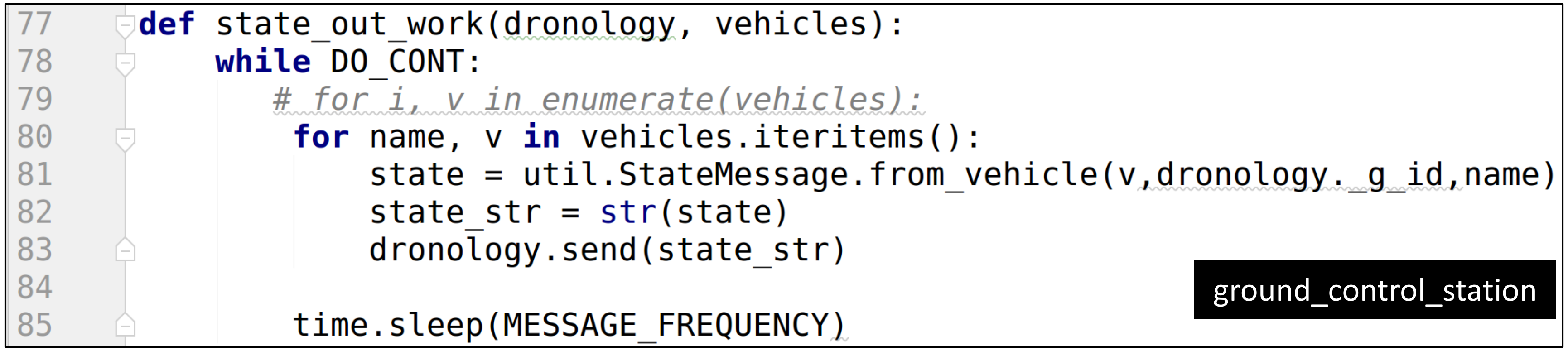
Receives messages from Dronology (but we aren’t using this part)

Thread: state_out_work



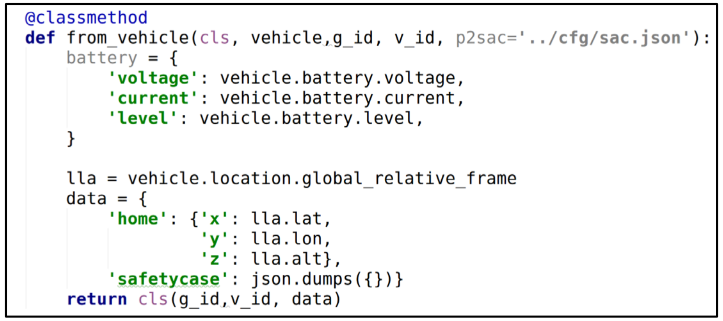
```python
def connect(self):
    self.dronology = util.Connection(None, "localhost", 1234,self.g_id)
    self.dronology.start()
    global DO_CONT
    DO_CONT = True
    w0 = threading.Thread(target=state_out_work, args=(self.dronology, self.vehicles))
    w0.start()
```

ground_control_station



```python
def state_out_work(dronology, vehicles):
    while DO_CONT:
        # for i, v in enumerate(vehicles):
         for name, v in vehicles.iteritems():
            state = util.StateMessage.from_vehicle(v,dronology._g_id,name)
            state_str = str(state)
            dronology.send(state_str)

         time.sleep(MESSAGE_FREQUENCY)
```

ground_control_station

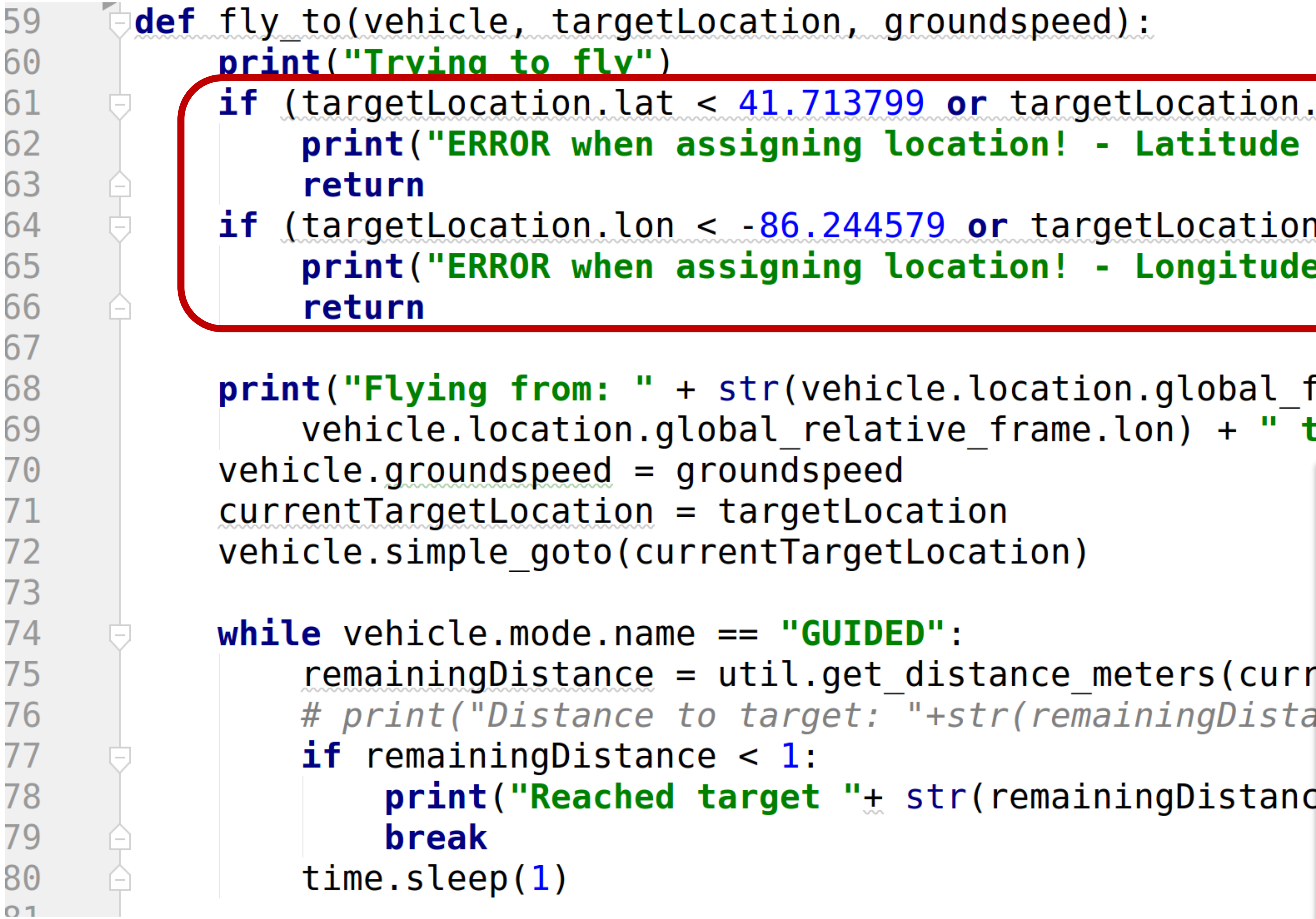


```python
@classmethod
def from_vehicle(cls, vehicle,g_id, v_id, p2sac='../cfg/sac.json'):
    battery = {
        'voltage': vehicle.battery.voltage,
        'current': vehicle.battery.current,
        'level': vehicle.battery.level,
    }

    lla = vehicle.location.global_relative_frame
    data = {
        'home': {'x': lla.lat,
                 'y': lla.lon,
                 'z': lla.alt},
        'safetycase': json.dumps({})}
    return cls(g_id,v_id, data)
```

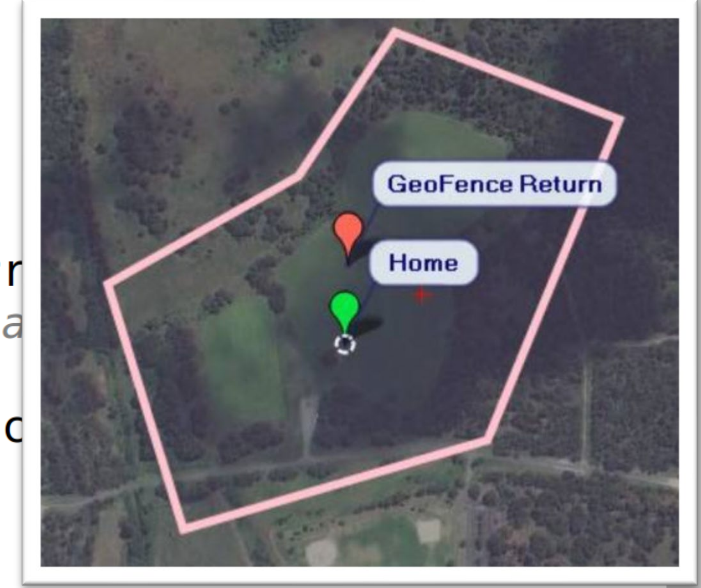
util.StateMessage

1. GCS creates a Connection instance which serves as a proxy for Dronology.
2. It starts up the thread that loops through all registered vehicles and sends their state to Dronology.

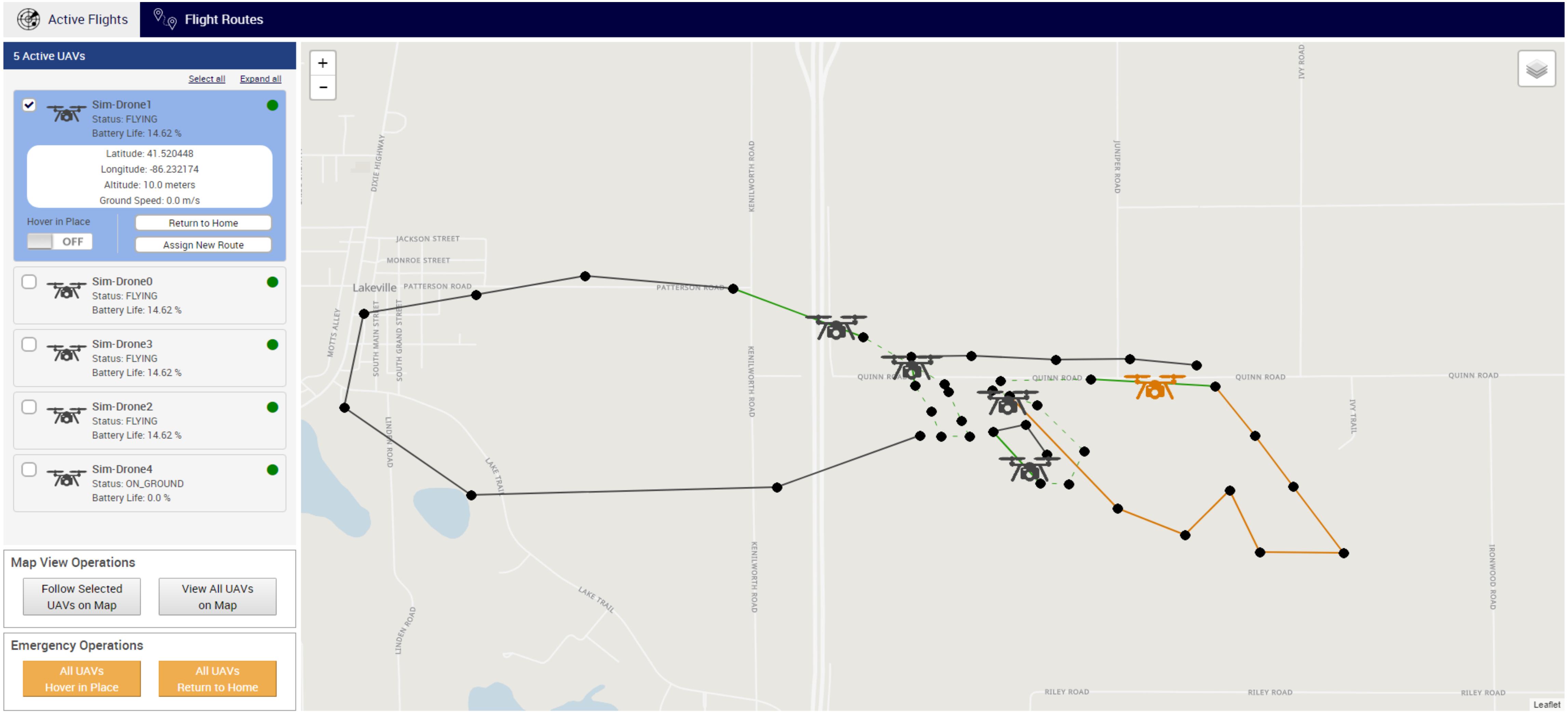
An Extra Safety Layer



```
def fly_to(vehicle, targetLocation, groundspeed):
    print("Trying to fly")
    if (targetLocation.lat < 41.713799 or targetLocation.lat >41.715593):
        print("ERROR when assigning location! - Latitude outside range!")
        return
    if (targetLocation.lon < -86.244579 or targetLocation.lon > -86.236527):
        print("ERROR when assigning location! - Longitude outside range!")
        return

    print("Flying from: " + str(vehicle.location.global_frame.lat) + "," + st
        vehicle.location.global_relative_frame.lon) + " to " + str(targetLoca
    vehicle.groundspeed = groundspeed
    currentTargetLocation = targetLocation
    vehicle.simple_goto(currentTargetLocation)

    while vehicle.mode.name == "GUIDED":
        remainingDistance = util.get_distance_meters(curr
        # print("Distance to target: "+str(remainingDista
        if remainingDistance < 1:
            print("Reached target "+ str(remainingDistanc
            break
        time.sleep(1)
```



Activity # 1: Execute main.py from multidrone3



5 Active UAVs

☑ Sim-Drone1 ●
Status: FLYING
Battery Life: 14.62 %

Latitude: 41.520448
Longitude: -86.232174
Altitude: 10.0 meters
Ground Speed: 0.0 m/s

Hover in Place: OFF

□ Sim-Drone0 ●
Status: FLYING
Battery Life: 14.62 %

□ Sim-Drone3 ●
Status: FLYING
Battery Life: 14.62 %

□ Sim-Drone2 ●
Status: FLYING
Battery Life: 14.62 %

□ Sim-Drone4 ●
Status: ON_GROUND
Battery Life: 0.0 %

Follow instructions to run Dronology. Start it on Notre Dame Campus and create a route to fly at least two UAVs over the stadium.

Activity #2: CRC Cards

CRC cards are typically created on index cars. Team members create one CRC card for each key class/object in their design.

The card is divided into three areas:

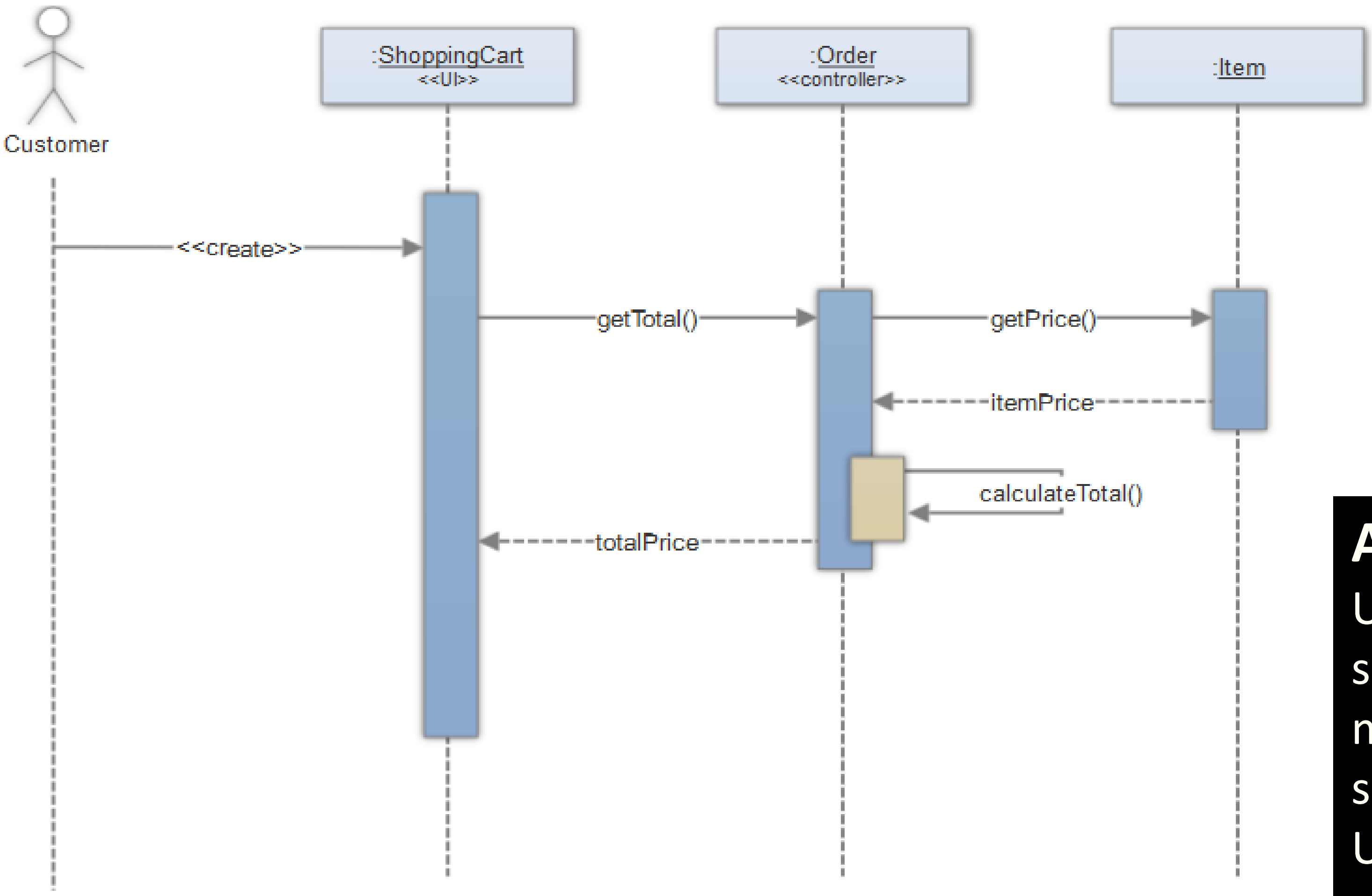
1. The **class** name at the top
2. **Responsibilities** of the class on the left
3. **Collaborators** (other classes) with which this class interacts to fulfill its responsibilities on the right.

As we look at ground control station code – we'll make **frequent stops to create CRC cards** for the key classes that we come across.

Class Name	
Responsibilities	**Collaborators**

ATM (Automatic Teller)	
Responsibility	**Collaborations**
Access and modify account balance	Account Balance Inquiry Deposit Transaction Funds Transfer Withdrawal Transaction

Activity # 3: Sequence Diagram (if time)



Shopping Cart example

Sequence diagrams:

- Capture behavioral aspects of a running program
- Swimlanes
- Messages
- Return items

Activity:
Using the CRC cards, source code, and slides, and starting with the main method (in main.py) create a sequence diagram that shows how a UAV is created, registered, and appears on the Dronology Map.

Before next week

<u>Install Eclipse on your computer.</u> We can use it for programming remotely on the Pis.

Next week:
Companion computers