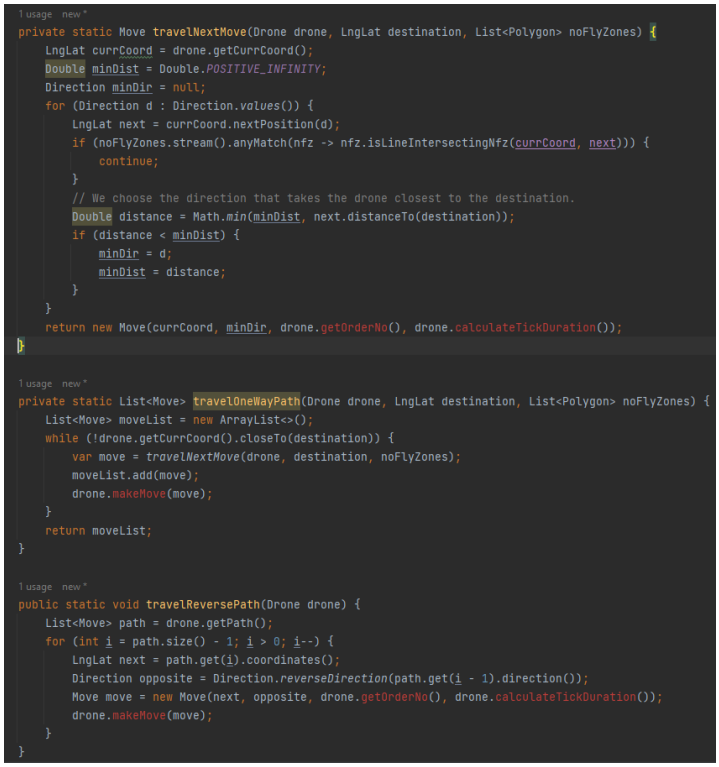
For the code review process, I took two main approaches to it: reviewing with an instructor and self-evaluating my code. An example of an instructor-led process is outlined below.

One piece of code that I was struggling with was the PathFinder class, which deals with finding the path with the minimum number of moves between one node to the next, with the main issue being that it was too tightly coupled with other classes and had redundant methods that could be refactored and removed. Due to the informal nature of drop-in sessions for ILP, some stages of the code review activities were irrelevant and/or not carried out.

Piece of code to be reviewed and refactored:



```
1 usage  new *
private static Move travelNextMove(Drone drone, LngLat destination, List<Polygon> noFlyZones) {
    LngLat currCoord = drone.getCurrCoord();
    Double minDist = Double.POSITIVE_INFINITY;
    Direction minDir = null;
    for (Direction d : Direction.values()) {
        LngLat next = currCoord.nextPosition(d);
        if (noFlyZones.stream().anyMatch(nfz -> nfz.isLineIntersectingNfz(currCoord, next))) {
            continue;
        }
        // We choose the direction that takes the drone closest to the destination.
        Double distance = Math.min(minDist, next.distanceTo(destination));
        if (distance < minDist) {
            minDir = d;
            minDist = distance;
        }
    }
    return new Move(currCoord, minDir, drone.getOrderNo(), drone.calculateTickDuration());
}

1 usage  new *
private static List<Move> travelOneWayPath(Drone drone, LngLat destination, List<Polygon> noFlyZones) {
    List<Move> moveList = new ArrayList<>();
    while (!drone.getCurrCoord().closeTo(destination)) {
        var move = travelNextMove(drone, destination, noFlyZones);
        moveList.add(move);
        drone.makeMove(move);
    }
    return moveList;
}

1 usage  new *
public static void travelReversePath(Drone drone) {
    List<Move> path = drone.getPath();
    for (int i = path.size() - 1; i > 0; i--) {
        LngLat next = path.get(i).coordinates();
        Direction opposite = Direction.reverseDirection(path.get(i - 1).direction());
        Move move = new Move(next, opposite, drone.getOrderNo(), drone.calculateTickDuration());
        drone.makeMove(move);
    }
}
```

The ‘Distribute code/tests’, and ‘Check code’ stages were all merged into one activity due to the drop-in session, where I was able to explain to the instructor what I wanted the code to do and what the current piece of code achieves.

For the ‘Write review report’ and ‘Discussion’ stages, they were conducted through constant communication through the Piazza forum platform. After the in-person code review, the instructor wrote additional comments and feedback regarding our session, as well as further information about how I could solve some of the problems and suggest some external reading material. This led to the creation of extensive Javadoc as well as in-line comments to explain the functionalities of each method and what some of the code achieves. I also ended up changing some methods’ visibilities to *private* such that only the most essential methods were public for encapsulation and avoid unnecessary methods being visible to non-developers. Some software development principles such as DRY (“Don’t Repeat Yourself) were considered at the recommendation of the instructor to ensure maintainability of the codebase for a long period of time. Furthermore, Intellij’s code formatter was used to make sure that my coding style was consistent with the other classes that I had reviewed and refactored.

Finally, the ‘Make to-do list’ part was done through the ‘Reminders’ app on my phone, as this was easier to both manage and update/add/complete tasks that were already due to be completed. This also meant that I did not necessarily need my laptop to think about some of these problems, and I could record potential solutions while I was not at home or when I did not have my laptop ready.

Here is the result after multiple review cycles:

```java
public class PathFinder {
    3 usages
    private long startingTick;
    10 usages
    private LngLat currCoord;

    /**
     * This method initialises the starting tick to a value on the first call (In nanoseconds).
     */
    1 usage  Daniel Kim
    private void setStartingTick() {
        if (startingTick == 0) {
            startingTick = Clock.tick(Clock.systemDefaultZone(), Duration.ofNanos(1)).instant().getNano();
        }
    }

    /**
     * This method returns the duration elapsed from the starting tick value to the current tick value.
     *
     * @return Elapsed time in nanoseconds since object initialisation.
     */
    4 usages  Daniel Kim
    private long getTiming() {
        return Clock.tick(Clock.systemDefaultZone(), Duration.ofNanos(1)).instant().getNano() - startingTick;
    }

    /**
     * This method finds the next move that will get this object's currCoord closest to the destination by avoiding
     * no-fly zones and travelling in one of 16 compass directions.
     *
     * @param destination LngLat coordinates of the destination.
     * @param orderNo     Order number of the current delivery
     * @param noFlyZones  List of no-fly zones to avoid.
     * @return A new Move object that contains the current coordinates, the next move, order number and elapsed duration
     * in milliseconds.
     */
    1 usage  Daniel Kim
    private Move travelNextMove(LngLat destination, String orderNo, List<Polygon> noFlyZones) {
        setStartingTick();
        Double minDist = Double.POSITIVE_INFINITY;
        Direction minDir = null;
        for (var d : Direction.values()) {
            var next = currCoord.nextPosition(d);
            if (noFlyZones != null && !noFlyZones.isEmpty()) {
                if (noFlyZones.stream().anyMatch(nfz -> nfz.isLineIntersectingNfz(currCoord, next))) {
                    continue;
                }
            }
            // We choose the direction that takes the drone closest to the destination.
            Double distance = Math.min(minDist, next.distanceTo(destination));
            if (distance < minDist) {
                minDir = d;
                minDist = distance;
            }
        }
        var coord = new LngLat(currCoord.lng(), currCoord.lat());
        currCoord = currCoord.nextPosition(minDir);
        return new Move(coord, minDir, orderNo, getTiming());
    }

    /**
     * This method repeatedly calls travelNextMove to travel from this object's currCoord to the destination by avoiding
     * no-fly zones.
     *
     * @param destination LngLat coordinates of the destination.
     * @param orderNo     Order number of the current delivery.
     * @param noFlyZones  List of no-fly zones to avoid.
     * @return A list of Move objects that represents the moves needed to get from currCoord to destination.
     */
    1 usage  Daniel Kim
    private List<Move> travelOneWayPath(LngLat destination, String orderNo,
                                        List<Polygon> noFlyZones) {
        List<Move> moves = new ArrayList<>();
        while (!currCoord.closeTo(destination)) {
            moves.add(travelNextMove(destination, orderNo, noFlyZones));
        }
        return moves;
    }

    /**
     * This method sets the return path of the drone going from the end of the path to the beginning.
     * Note that on the return path, the drone's respective direction will be matched to the drone's previous coordinates.
```

```java
    /**
     * This method sets the return path of the drone going from the end of the path to the beginning.
     * Note that on the return path, the drone's respective direction will be matched to the drone's previous coordinates.
     * E.g. [(0,0), E] -> [(1,0), null] -> [(1,0), W] -> [(0,0), null]
     * Here, the return direction for [(1,0), W] is the opposite to the previous coordinate's direction [(0,0), E].
     * This method does not return a NullPointerException error, since the list of moves when this method is called will
     * contain at least one hover move. Thus, the for-loop will terminate immediately and move onto the next method.
     *
     * @param moves      List of moves to reverse.
     * @param startCoord The coordinate to return to.
     * @param orderNo    Order number of the current delivery.
     * @return A list of Move objects that represents the moves needed to get from destination to the
     * starting coordinates.
     */
    1 usage   Daniel Kim
    private List<Move> travelReversePath(List<Move> moves, LngLat startCoord, String orderNo) {
        List<Move> newMoves = new ArrayList<>();
        for (int i = moves.size() - 1; i > 0; i--) {
            var prevCoord = moves.get(i).coordinates();
            var previousDir = moves.get(i - 1).direction();
            var oppositeDir = Direction.reverseDirection(previousDir);
            newMoves.add(new Move(prevCoord, oppositeDir, orderNo, getTiming()));
        }
        currCoord = startCoord;
        return newMoves;
    }

    /**
     * This method returns a hover move which has a null direction.
     *
     * @param orderNo Order number of the current delivery.
     * @return A Move object that contains the current coordinates, a null direction enum, order number and duration
     * elapsed since beginning of route calculation.
     */
    2 usages   Daniel Kim
    private Move hover(String orderNo) { return new Move(currCoord,  direction: null, orderNo, getTiming()); }

    /**
     * This method calculates the moves needed to travel from the startCoord to the destination and then back by
     * avoiding no-fly zones.
     *
     *
     * @param startCoord Starting coordinates.
     * @param nodePath   A list of LngLat coordinates that need to be visited for the shortest path.
     * @param orderNo    Order number of the current delivery.
     * @param noFlyZones List of no-fly zones to avoid.
     * @return A list of Move objects that represents the moves needed to travel from startCoord to the destination
     * and then back.
     */
    1 usage   Daniel Kim
    public List<Move> travel(LngLat startCoord, List<LngLat> nodePath, String orderNo, List<Polygon> noFlyZones) {
        List<Move> moves = new ArrayList<>();
        currCoord = startCoord;
        for (var coord : nodePath) {
            moves.addAll(travelOneWayPath(coord, orderNo, noFlyZones));
        }
        moves.add(hover(orderNo));
        moves.addAll(travelReversePath(moves, startCoord, orderNo));
        moves.add(hover(orderNo));
        return moves;
    }

    /**
     * This method returns a new list of Move objects updated with a new order number and time elapsed if there
     * already exists a path for a given LngLat coordinate.
     *
     * @param orderNo Order number of the current delivery.
     * @param moves   List of pre-calculated Move objects.
     * @return A list of Move objects that are the same as the input but with updated order number and time elapsed.
     */
    1 usage   Daniel Kim
    public List<Move> travelPresetRoute(String orderNo, List<Move> moves) {
        var newMoves = new ArrayList<Move>();
        for (var move : moves) {
            newMoves.add(new Move(move.coordinates(), move.direction(), orderNo, getTiming()));
        }
        return newMoves;
    }
}
```