CI Pipeline

5.1 Identify and Apply Review Criteria to Selected Parts of the Code and Identify Issues in the Code

To have high-quality software, a structured code review was conducted using established criteria. The key focus areas included:

Readability: Assessed the clarity of code, to make sure variable and method names were meaningful, comments were sufficient, and logic was straightforward to follow. Identified areas where comments were missing, particularly in complex algorithms like the AStarFlightPathSolver. Used JavaDoc comments for this.

Performance: Evaluated the efficiency of critical algorithms, including pathfinding. Issues identified included unoptimised loops and redundant calculations within the A* algorithm, particularly in handling visited neighbours. The open set that was used was adding unnecessary nodes into it. This was adjusted to imrove performance.

Error Handling: Verified robust error handling. While most areas managed exceptions well, certain untested pathways, such as those for unreachable nodes, were noted as gaps.

Compliance with Standards: Checked adherence to coding standards and best practices, such as avoiding deeply nested logic. Minor issues in the structure of conditional logic within isNextPositionValid were identified.

Test Coverage: Made sure the test coverage for all critical paths was good. Identified a gap in triggering certain error-handling branches.

These reviews led to actionable improvements, including refactoring parts of the A* algorithm and improve error messages. Code snippets were marked for refactoring where improvements could reduce complexity and improve maintainability

Before Applying Review

Weaknesses

- No comments for methods for clarity
- No break for specific edge cases
- The original implementation lacked a findNeighbour method, and redundant logic for finding neighbours was directly embedded in the main loop. This made the code harder to maintain and less efficient.

```
private final Map<LngLat, CoordinatePoint> openSetMap = new HashMap<>();
1 usage
HashSet<CoordinatePoint> closedSet = new HashSet<>();
4 usages
List<CoordinatePoint> path = new ArrayList<>();


4 usages
public List<CoordinatePoint> findShortestPath(CoordinatePoint coordinatePointForDelivery, NamedRegion[] noFlyZones,
                                              NamedRegion centralArea){
    LngLatHandler lngLatHandler = new LngLatHandler();
    openSet.add(APPLETON_TOWER_POINT);
    openSetMap.put(APPLETON_TOWER_POINT.lngLat, APPLETON_TOWER_POINT);

    while (!openSet.isEmpty()){
        CoordinatePoint current = openSet.poll();
        openSetMap.remove(APPLETON_TOWER_POINT.lngLat);
        closedSet.add(current);
```

After Applying Review

Strengths

- Added clear JavaDoc comments to all methods and complex logic blocks to explain their purpose, parameters, and return values, ensuring easier understanding for future developers
- Enhanced the early exit logic in findShortestPath by adding a break condition for specific edge cases, such as when no valid paths exist within a threshold of moves.
- Refactored the neighbour finding logic into the findNeighbour helper method, which significantly improved code readability and reduced duplication.
- **Scalability:**
    - The priority queue (openSet) and hash-based lookups (openSetMap) are good for handling large search spaces efficiently

```
/**
 * finds the optimal path from the start point (APPLETON_TOWER_POINT) to the end (delivery) point
 * @param coordinatePointForDelivery the end point for the search algorithm
 * @param noFlyZones the array of NamedRegion object no-fly zones
 * @param centralArea the NamedRegion specifying the central area
 * @return list of coordinatePoint objects that make up the optimal path, null if path was not found
 */
4 usages
public List<CoordinatePoint> findShortestPath(CoordinatePoint coordinatePointForDelivery, NamedRegion[] noFlyZones,
                                              NamedRegion centralArea){
    LngLatHandler lngLatHandler = new LngLatHandler();
    openSet.add(APPLETON_TOWER_POINT);
    openSetMap.put(APPLETON_TOWER_POINT.lngLat, APPLETON_TOWER_POINT);

    while (!openSet.isEmpty()){
        CoordinatePoint current = openSet.poll();
        openSetMap.remove(APPLETON_TOWER_POINT.lngLat);
        closedSet.add(current);
        //find the goal; early exit
        if (lngLatHandler.isCloseTo(current.lngLat, coordinatePointForDelivery.lngLat)) {
            path = new ArrayList<>();
            while (current != null) {
                path.add(current);
                current = current.parent;
```

```java
/**
 * retrieves the previously visited neighbouring cell, if it has been visited
 * @param position the position to check for a visited neighbouring coordinate
 * @return the CoordinatePoint representing the visited neighbouring coordinate, null if not visited
 */
1 usage
private CoordinatePoint findNeighbour(LngLat position){
    if(openSetMap.containsKey(position)){
        return openSetMap.get(position);
    }
    return null;
}
```

Example of isMoveWithinCentralArea Before and After Review

Code before:

- Redundant Logic - This version used deeply nested if, else statements to check conditions, which results in repeated redundant calls
- Reduced Readability - The nested structure increases cognitive load, making it more difficult to understand the logic at a glance
- Error-Prone - With so many nested conditions, it is easier to introduce errors during modifications, and testing was more complex

```java
private boolean isMoveWithinCentralArea(LngLat nextPosition, LngLatHandler lngLatHandler, LngLat current, NamedRegion centralArea) {
    if (lngLatHandler.isInCentralArea(current, centralArea)) {
        if (lngLatHandler.isInCentralArea(nextPosition, centralArea)) {
            return true;
        } else {
            return false;
        }
    } else {
        if (!lngLatHandler.isInCentralArea(nextPosition, centralArea)) {
            return true;
        } else {
            return false;
        }
    }
}
```

Code after:

- Simplified Logic - The conditions are written in a concise and straightforward manner, avoiding unnecessary nesting
- Improved Readability - By reducing nesting, the logic is cleaner and easier to follow, making the function maintainable and less error-prone
- Efficiency - The code minimises calls to LngLatHandler.isInCentralArea, performing only the necessary checks, optimising performance

```
private boolean isMoveWithinCentralArea(LngLat nextPosition, LngLatHandler lngLatHandler, LngLat current,
                                        NamedRegion centralArea) {
    if (!lngLatHandler.isInCentralArea(current, centralArea)) {
        return !lngLatHandler.isInCentralArea(nextPosition, centralArea);
    }
    return true;
}
```

5.2 Construct an Appropriate CI Pipeline for the Software

A Continuous Integration pipeline was implemented to streamline the development and testing process. The pipeline was structured:

Source Control Integration: Integrated with GitHub to automatically trigger the CI pipeline upon code commits and pull requests

Build Phase:

- Used Maven to compile the Java code, made sure all dependencies were resolved and the build was successful. Also made such the correct plugins were implemented

Test Phase (Unit, System and Integration Testing) :

- Configured the pipeline to run all test cases using JUnit.
- JUnit integration tests were configured to simulate workflows involving multiple components
  - Validating an order and ensuring it seamlessly passes through to pathfinding
  - Testing the interaction between the AStarFlightPathSolver and LngLatHandler with dynamically generated no-fly zones
  - Identified issues that might arise from the interaction of well-functioning individual components

This CI pipeline gives rapid feedback on code changes, improving the development cycle's efficiency and minimising the chances of introducing breaking changes. JUnit's detailed test reports provided actionable insights into failures, which helped streamline debugging. The combined testing strategy reduced the risk of undetected errors propagating to production and significantly improved overall software quality.

5.3 Automate Some Aspects of the Testing

Several aspects of testing were automated within the CI pipeline to reduce manual effort and improve reliability.

Unit Testing: Automated tests for all validation logic, including credit card information, order validation, and pathfinding functionality

Performance Testing: Integrated a custom script to measure the runtime of pathfinding algorithms under varying configurations of no-fly zones. Results were logged and plotted for analysis

Regression Testing: Configured the pipeline to run regression tests on each commit to make sure new changes did not break existing functionality

Dynamic Data Testing: Incorporated synthetic data generation for no-fly zones and orders to test edge cases automatically. This allowed for evaluating the system under varying conditions without manual setup.

Code Coverage Reports: Automatically generated after each test run and archived as part of the CI pipeline artefacts for further review.

Automating these testing components significantly reduced the time required for manual testing, whhile having consistent results and continuous deployment.

5.4 Demonstrate the CI Pipeline Functions as Expected

To validate the effectiveness of the CI pipeline, its functionality was demonstrated across several scenarios:

Code Commit and Build Trigger: The pipeline successfully detected changes pushed to the repository and triggered the build process without manual intervention

Test Execution: All unit, integration, and regression tests were executed automatically. Failures were appropriately flagged, with detailed reports generated for debugging

Coverage Analysis: The pipeline produced coverage reports showing detailed statistics for class, method, and line coverage, ensuring test quality met targets

Artefact Deployment: Successfully deployed build artefacts to a staging environment, demonstrating end-to-end functionality from source control to deployment

Failure Scenarios: Tested the pipeline with intentional code issues to ensure failures were accurately reported, and notifications were sent to the team. This confirmed the robustness of the error-handling mechanisms within the CI setup

These demonstrations provided confidence in the CI pipeline's ability to maintain software quality and streamline the development process. Future iterations could include additional features like stress testing and deployment to production environments for further enhancement.