5. Conduct Reviews, Inspections, and Design and Implement Automated Testing Processes

5.1 Identify and Apply Review Criteria to Selected Parts of the Code Review Approach:

1. **IntelliJ Built-In Inspections:**
    - Used IntelliJ's Code Analysis to detect potential issues like unused variables, overly complex methods, and redundancy in DeliveryController.java and OrderValidation.java.
    - Fixed warnings about boxing/unboxing and potential null pointer usage.
2. **Pair Programming:**
    - Engaged in real-time collaboration on the calculatePath method to ensure clarity of logic, especially handling no-fly zones.
    - Peer feedback uncovered hidden complexity in adjustDirection.
3. **Manual Code Inspection:**
    - Examined method lengths, conditional logic depth, and comment clarity in DeliveryController.java.
    - Ensured each CompassDirection in CompassDirection.java was tested**.**

High-Quality Code Indicators:

- Minimal Cyclomatic Complexity: Each method ideally had no deeply nested conditionals.
- Clear Naming Conventions: Methods like adjustDirectionToStayInside signpost intentions well.
- Consistent Formatting: IntelliJ auto-format ensured standard code style.

5.2 Construct an Appropriate CI Pipeline for the Software

- **CI Tool**: GitHub Actions
- **Workflow File**: .github/workflows/build-and-test.yml
    1. **Checkout & Set Up JDK**: Ensures code retrieval and Java 17 installation.
    2. **Cache Maven Dependencies**: Speeds up repeated builds.
    3. **Build & Test**: Runs mvn clean verify; fails build if any tests fail.
    4. **(Optional) Docker Build**: Creates the Docker image (ilp_submission_image) for consistent deployment.
- **Outcome**:
    - Automatic testing on every push or pull request.
    - Quick detection of issues like test failures, code coverage drops, or build breaks.

5.3 Automate Some Aspects of Testing

- **Unit Tests**:
    - **OrderValidationTest.java**: Ensures invalid orders (e.g., mismatched restaurant name) are quickly rejected.
    - **DeliveryControllerTest.java**: Checks correct path generation in normal and blocked routes.
- **Integration Tests**:

    - **Mocking** external data (like restaurants and no-fly zones) to confirm the system logic without real API calls.
- **System Tests**:
    - **Postman** or **curl** scripts to validate real endpoints, including /calcDeliveryPath and /calcDeliveryPathAsGeoJson.

5.4 Demonstrate the CI Pipeline Functions as Expected

- **On Each Commit**:
    1. **Lint & Build**: Confirm no compilation or style issues.
    2. **Tests**: JUnit coverage metrics confirm logic is stable (~80–90% coverage).
    3. **Docker Build**: Confirms container builds successfully under linux/amd64.
- **Result**:
    - A robust "red/green" feedback cycle ensures code merges are safe and performance regressions or no-fly zone logic issues are flagged quickly.
    - The pipeline also serves as a foundation for future expansions like security scans or performance tests.