Software Testing Portfolio

1 Outline of the Software Being tested

The project under testing is the PizzaDronz REST-based service, a Java- application designed for the drone-based delivery of pizzas to Appleton Tower's rooftop.This system includes the following core functionalities:

- Order validation: Ensures only valid pizza orders are processed by the system, detecting incorrect or missing data.
- Delivery path computation: Calculates optimized flight paths from the restaurant to the delivery point, avoiding designated no-fly zones and ensuring compliance with the central area's constraints. The path computation must execute within a runtime of under 60 seconds to meet performance requirements.
- GeoJSON path output: Generates delivery routes in GeoJSON format, enabling easy visualization of the drone's flight path using mapping applications.
- Integration with centralized data services: Relies on REST endpoints for access to information about no-fly zones, restaurant locations, and the coordinates of the university's central area.

Rigorous testing has been applied, including validation against error scenarios, performance benchmarking, and the use of Postman and other tools for endpoint verification. This ensures the system's capability to handle real-world operations effectively and maintain robust functionality under various conditions.
All source code, testing, and evidence is available in the following GitHub repository:
https://github.com/jamieossip1/SoftwareTesting

2 Learning Outcomes

1. **Analyze requirements to determine appropriate testing strategies**

   (a) Range of Requirements
   The functional requirements (F1-F5) included validating flight paths, and validating orders. Endpoint reachability and REST server data retrieval were also key functionalities. Non-functional requirements (N1-N3) focused on performance (path computation less than 60 seconds), maintainability through modular design, and robustness. These requirements were outlined in *LO1.pdf*.

   (b) Levels of Requirements
   Unit-level tests addressed path validity (F1) and order validation (F2). Integration tests focused on REST server interactions (F4) and ensuring valid orders led to path computations (F1). System-level tests validated endpoint reachability (F3), runtime performance (N1), and overall robustness (N3), as outlined in *LO1.pdf*.

   (c) Test Approaches
   Test strategies included equivalence partitioning and boundary value analysis for F1 and F2, schema validation for GeoJSON outputs (F5), and statistical testing for performance (N1). Maintainability (N2) was assessed through code reviews, all detailed in *LO1.pdf*.

   (d) Assessment
   The testing strategies ensured comprehensive coverage of requirements, balancing thoroughness with efficiency. The modular approach addressed functional requirements while meeting non-functional targets. Effectiveness will be assessed using line and branch coverage.

2. **Design and implement comprehensive test plans with instrumented code**

   (a) Test Plan Construction
   The test plan addressed all requirements. High-priority tests ensured critical functionalities were validated, while medium-priority tests covered maintainability and GeoJSON compliance. Detailed test cases are documented in *LO2.pdf*.

(b) Evaluation of the Test Plan Quality
The quality of the test plan was assessed based on coverage, and redundancy minimization. Edge cases like drone path anomalies and invalid input were simulated to capture unexpected behaviors. Redundant tests were consolidated to ensure efficiency. Refer to *LO2.pdf*.

(c) Code Instrumentation
Instrumentation included detailed logging to track flight path calculations and REST server interactions, assertions to validate runtime constraints (e.g., N1), and mocking to simulate REST server responses. GeoJSON schema validation ensured compliance with F5. Further details are outlined in *LO2.pdf*.

(d) Evaluation of Instrumentation
The instrumentation successfully identified faults in flight path validation (F1) and REST data integrity (F4). Logs and assertions provided reliable insights, while mocking reduced external dependencies. However, reliance on mock data limited real-world input variability. Additional assessments are included in *LO2.pdf*.

3. **Apply a wide variety of testing techniques and compute test coverage and yield according to a variety of criteria**

(a) Range of Techniques
A mix of functional, structural, and performance testing was applied to validate the system. Functional testing ensured flight path validity (F1) and order validation (F2), while structural testing assessed endpoint reachability (F3). Integration tests verified REST server retrievals (F4). Detailed methods are documented in *LO3.pdf*.

(b) Evaluation Criteria for Adequacy of Testing
Coverage metrics achieved 85% line and 78% branch coverage, ensuring most code paths were rigorously tested. Functional requirements like flight path validity (F1) and order validation (F2) were prioritized. Additional coverage details are in *LO3.pdf*.

(c) Results of Testing
All functional and non-functional requirements passed all tests. Performance tests confirmed compliance with runtime constraints. Test results are detailed in *LO3.pdf*.

(d) Evaluation of Results
Strengths included effective identification of key issues, with reliable testing across functional and performance requirements. Weaknesses included lower branch coverage, highlighting areas for improvement in robustness testing. Further evaluations are outlined in *LO3.pdf*.

4. **Evaluate the limitations of a given testing process, using statistical methods where appropriate, and summarise outcomes**

(a) Identifying gaps and omissions in the testing process
Gaps included the use of synthetic test data, which lacked real-world variability, and the absence of stress testing on REST server interactions. Resolutions for these gaps are detailed in *LO4.pdf*.

(b) Identifying target coverage/performance levels for different testing procedures
Targets included 100% functional test success, 90% line coverage, 80% branch coverage, and path calculations under 60 seconds. Outcomes are summarised in *LO4.pdf*.

(c) Discussing how the testing carried out compares with the target levels
Functional requirements achieved 100% test success, while line and branch coverage fell slightly short of targets (85% and 78%, respectively). Performance results were satisfactory but left room for improvement, as detailed in *LO4.pdf*.

(d) Discussion of what would be necessary to achieve the target level
Future work includes testing more edge cases, incorporating real-world variability into test data, and developing mock REST servers for high-load stress testing.

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

(a) Identify and Apply Review Criteria to Selected Parts of the Code
Code reviews were performed on modules critical to input validation, flight path calculations, and error handling. Issues such as incomplete validation checks for edge cases and insufficient error logging mechanisms were identified and resolved.

(b) Construct an appropriate CI pipeline for the software
A CI/CD pipeline was established to automate testing and deployment, streamlining the development workflow. This setup accelerated feedback and minimized manual testing, enhancing overall efficiency. Refer to *LO5.pdf* for the CI pipeline plan.

(c) Automate some aspects of the testing
Automated unit tests were implemented in Postman to ensure continuous validation of the system's components. This approach guarantees consistent checks after each code change, improving the testing process. See *LO5.pdf* for more details.

(d) Demonstrate the CI pipeline functions as expected
The CI pipeline successfully managed deployment scenarios, detecting failures early during build or test stages. This prevented faulty releases, ensuring only validated code was deployed. See *LO5.pdf* for the CI pipeline demo.