Online Auction

Testing Documentation

**Slippery Rock University of Pennsylvania**

Contributions by:

**Fall 2024**

Griffin Vasalani – glv@sru.edu

Grant Riley – [gwr1002@sru.edu](mailto:gwr1002@sru.edu)

Jayden Williams – jsw1015@sru.edu

**Core Testing Methodologies and strategies**

Our testing strategy uses methodologies from material learned during the course. We implement our strategies with unit testing utilizing JUnit and Mockito, proceed to integration testing to assess component interactions, and culminate with user acceptance testing for comprehensive evaluation of the application's end-to-end functionality.

**System Requirements**:

Windows OS

MySQL Workbench 8.x

JDK: 17

Tomcat 10.0.x

Spring Boot 4.26.0

**Black Box Testing**

Throughout the project lifecycle, our team conducted systematic black box testing to evaluate functionality from an end-user perspective. A notable example of this approach was our comprehensive testing of the offer system. Initially, users had unlimited offer-sending capabilities per listing. We implemented new restrictions allowing only one active offer per listing with a maximum of three total offers. Our testing protocol involved attempting to send offers on multiple listings simultaneously, typically five or more, as well as multiple offers per individual item. The system successfully enforced these restrictions by displaying appropriate notification messages when users attempted to exceed the established limits.

**White Box Testing**

White box testing focused on verifying the intricate relationships between frontend displays and backend database operations. One significant test case involved the car club membership system. When users joined a car club, we verified that the frontend HTML correctly displayed member information, chat functionality, and associated club data. Simultaneously, backend testing confirmed proper database population with user information, club membership details, role assignments, and all necessary metadata. This dual-focused approach ensured complete system integrity across all architectural layers.

**Regression Testing**

Regression testing formed a crucial part of our quality assurance process, focusing on core system functionality. We consistently verified essential features including the authentication system, offer management, messaging functionality, listing operations, payment processing, and navigation systems. This ongoing testing regime helped maintain system stability throughout the development cycle.

**Boundary Analysis**

Our boundary analysis testing concentrated on critical system limitations, particularly in the offer system. We implemented and verified strict boundaries including the three-offer maximum per item and single active offer requirement. Testing included edge cases in offer submission timing and validation of offer amounts within acceptable ranges.

**Volume Testing**

Volume testing evaluated system performance under high-load conditions. We conducted extensive tests with large numbers of simultaneous offers, high notification volumes, and multiple concurrent user sessions. This testing phase also included evaluation of bulk listing management and large-scale message thread handling capabilities.

**Resource Management**

Initial testing of the notification system revealed important performance considerations. We began with a 30-second notification check interval and systematically tested different polling frequencies. Through careful analysis, we identified performance degradation at sub-15-second intervals, where the system showed increased memory consumption and image processing errors. This led to establishing an optimal polling interval of 15 seconds or greater to maintain system stability while ensuring timely notifications.

**Missing Resource Handling**

We implemented robust error handling mechanisms, particularly for image processing failures. The system now automatically implements placeholder images when encoding issues occur, with automatic replacement upon page reload. Testing confirmed successful error recovery and proper image replacement functionality.

**Crash Recovery**

The system demonstrated exceptional crash recovery capabilities during our testing phase. In the event of unexpected termination or system crashes, the database-maintained data integrity and proper state restoration. The system successfully preserved session data, protected user information, and implemented transaction rollback procedures when necessary. Upon restart, all HTML elements rendered correctly, and the system restored to its pre-crash state with all critical data intact.

**Unit Testing: Test Environment Configuration**: We implemented our unit testing framework utilizing the Eclipse JUnit plugin, incorporating Mockito for mock object generation. This architecture enabled isolated component testing through effective dependency simulation.

**Test Suite Implementation**: Our comprehensive unit testing approach involved developing test cases that validated core functionality across domain entities, including property accessors and mutators. The testing scope encompassed both expected behavior and edge cases, extending across domain objects, form structures, and data transfer objects. We conducted thorough unit testing of service layer components to verify method behavior and outcomes aligned with specifications.

**Integration Testing**: Integration Strategy: Our integration testing framework focused on validating the seamless interaction between architectural layers - specifically the controller, service, and entity components - to ensure system cohesion under realistic operating conditions. The Test Cases we developed and executed integration test scenarios centered on core business functions, including product listing workflows, authentication processes, and transaction handling. These tests were designed to verify robust component interoperability and maintain system stability across integrated operations.

**Conclusion**:

During the initial project handover, we received a codebase containing 380 test cases. The initial test execution revealed 38 errors, 2 failures, and 340 successful tests. Through extensive development and enhancement efforts, we have significantly expanded the test coverage to 528 total test cases. Current test execution results show remarkable improvement with only 32 errors, 2 failures, and 494 successful tests.

The substantial increase in test coverage was necessary due to implementation of several key features and modifications to existing functionality. New test cases were added specifically for the preferences page, car clubs page, notification system, and modifications to the offer system. The test results demonstrate that the newly implemented features are successfully integrated with the existing codebase and are functioning according to specifications. The remaining errors are the result of the test base running out of memory during the initial boot phase and should be looked at closer. However the functionality of the program and the features that the tests are checking are in proper working order.