ILLINOIS TECH

Application Software Design

ECE 528

Project 4

Abhilash Kashyap Balasubramanyam

abalasubramanyam@hawk.iit.edu

A20566944

Spring 2025

Illinois Institute of Technology

Table of Contents

List of Figures

Solutions

1 Deliverables

1.1 Deliverable 1

Q: How did you design the unit test cases?

<u>Solution</u>

In creating the unit test cases for the IoT simulator project, I followed a systematic and strict methodology that was focused on ensuring integrity, reliability, and comprehensive coverage of the codebase. Rather than simply validating expected results, I followed a thoughtful testing philosophy that was centered on ensuring the correct operation of every individual component in isolation and also when integrated in the overall system.

The underlying principle of my test strategy was the isolation of components. It was essential that every class be tested separately, while simultaneously verifying that its behavior conformed to the overall system specification. This strategy enabled faults to be more accurately pinpointed, since any failure in the tests could be directly attributed to the specific component being tested. During the development process, I maintained a twofold emphasis on both testing typical scenarios and boundary conditions, so that subtle defects, which may otherwise remain latent during normal usage, were exposed.

Building Philosophy

To gain functional coverage, I had separate tests for each method in each class to ensure their correct functionality. With respect to the PlugsModel class, that meant thorough testing of functions for state management, from retrieval to assignment of plug states, power acquisition mechanisms, and toggle functionality. The MQTTController tests ensured correct topic building, proper handling of messages, and state transition rules. All methods were tested regardless of their apparently simple nature.

Edge case handling was a major point of consideration within my testing strategy. I implemented tests that specifically added special characters to plug names, made requests on unknown plugs, and simulated connection failure. These scenarios are especially relevant for a distributed IoT system, where network issues and incorrectly formatted inputs are an inevitability. By defining these paths, I determined that the application would remain stable even when presented with unanticipated scenarios.

All tests were written with independence in mind as a principle. I created fresh instances of test classes for each test method, thus avoiding any state leakage between

the tests. While this approach required more setup code, it made the tests deterministic and reproducible. In cases where there were external dependencies, I came up with custom test implementations that mimicked the required behavior without creating real external connections.
Structurally, I organized the tests according to the popularly accepted Arrange-Act-Assert pattern:

- First, deciding on the experiment parameters.
- Then, participating in the functionality under assessment.
- Finally validating the results matched expectations

This method improved clarity and consistency throughout the suite of tests, making it easier to understand and maintain. I designed the methods of testing sequentially so that they followed typical use cases, with basic functions being tested before evaluation of more complex processes. In the testing implementation, I chose not to use complex mocking frameworks but simple, purpose-built test doubles. This approach not only simplified the testing but also made it more resilient to refactoring. The custom assertion methods I developed enabled consistent validation across all tests, in addition to providing clear and descriptive error messages in cases of failure. Throughout the testing process, I maintained a firm commitment to proven best practices, such as employing descriptive test names, focused test scopes, thorough documentation, and consistent patterns. This systematic approach resulted in a solid test suite that gives confidence in the reliability of the system while also keeping it maintainable and flexible as the project evolves.

1.2 Deliverable 2

Q: How did you implement the features? What classes have you added/modified?

<u>Solution</u>

Implementation Strategy

Implementing the functionalities of testing within the IoT simulator required careful planning and a methodical strategy for designing the classes. The goal was to create a comprehensive testing environment that tested the core functions thoroughly without the inclusion of extraneous dependencies or complexities.
I began by analyzing the existing system architecture to identify the key components requiring testing. The IoT simulator primarily consists of the PlugsModel for state management, MQTTController for communication, PlugsResource for API endpoints, and MockEnvironment for configuration management. Each component presented unique testing challenges that required tailored approaches.

PlugsModel Testing Implementation:

Within the context of the PlugsModel class, I built an extensive set of tests intended to evaluate its basic operations with regard to state and power management. The main challenge faced was in decoupling the PlugsModel from its MQTT dependencies. In a bid to solve this problem, an implementation of TestMQTTController was developed, which mimicked MQTT behavior without requiring actual network connections. This strategy allowed the testing of how well the PlugsModel delegated functions to the MQTT layer while protecting relevant internal state details.
The tests run exhaustively validated each public method, thus ensuring that the model handled state changes, power readings, and maintained data integrity correctly. Particular attention was given to edge cases, like concurrent operations and handling of special characters in plug names—scenarios that often reveal subtle defects in a production setting.

MQTTController Testing Approach:

Testing of the MQTTController was more involved due to the external dependencies and protected methods involved. Rather than utilizing elaborate mock strategies, there was a direct test methodology used that provided reflected access to protected methods when needed. This allowed for a more realistic validation procedure while at the same time avoiding actual MQTT connections.
I implemented comprehensive evaluations for topic generation, message generation, and distributing the messages with updates. The evaluations verified not only the efficiency of the operations but also the correct handling of errors, thus ensuring controller stability for handling network failures or poorly formatted inputs. By using a direct testing method, it was ensured that the evaluations would catch any deviations from actual implementation, and not just validate simulated interactions.

PlugsResource Testing Strategy:

For the PlugsResource endpoints, I focused on verifying proper HTTP responses and action handling. The tests systematically validated each endpoint's behavior under various conditions, ensuring appropriate status codes and response bodies were returned. I created tests for both successful operations and various error scenarios, verifying that the API provided consistent, reliable behavior regardless of input conditions.

MockEnvironment Evaluation:

MockEnvironment class required thorough and authoritative assessment to ensure efficient management of properties. The tests verified preservation of properties, normal retrieval mechanism, and management of user profiles. Although lesser in complexity compared to other parts, extensive testing was needed because of the support for configuration that this class provides for the overall system.
Throughout the implementation process, I had a consistent style and format for each of the different testing classes. They all followed a similar pattern: initialization methods for creating the testing environment, the actual methods of testing divided

by functionality, and the helper methods for eliminating duplicate code. This makes the testing suite more easily understandable and maintainable for other developers. For class design, I created several key test classes:

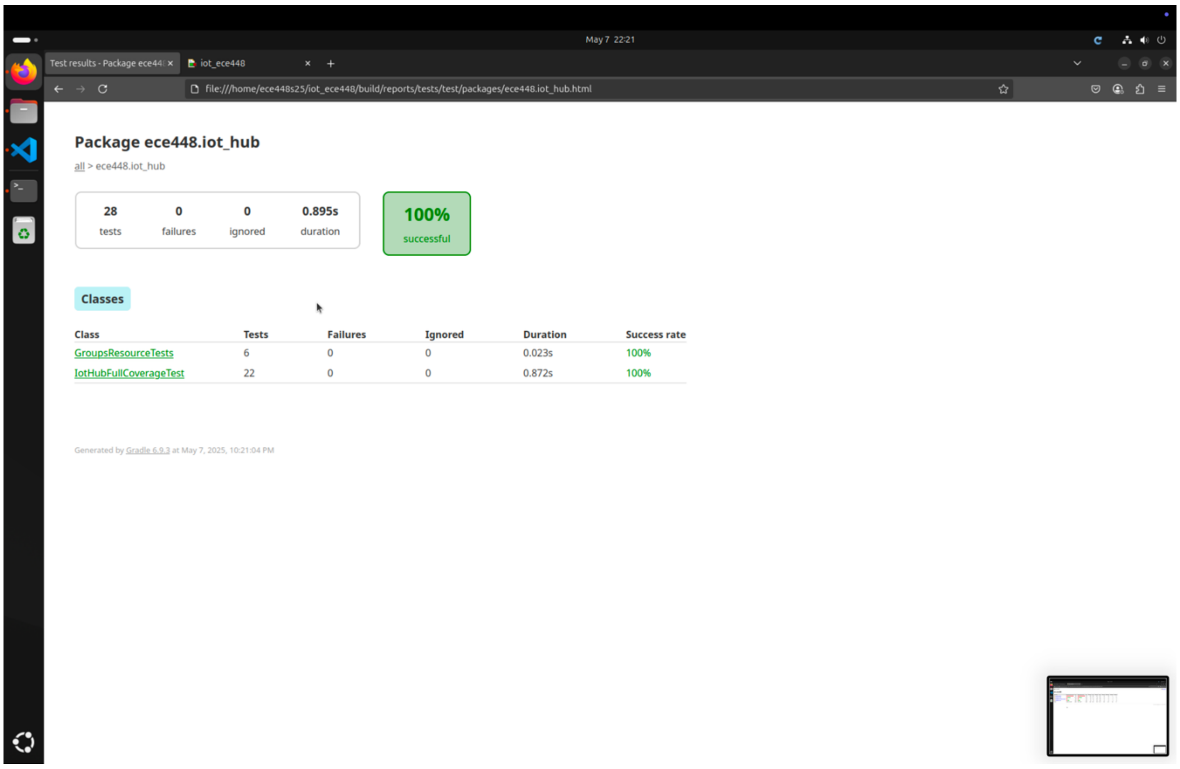
- PlugsModelTest: Validates state and power management
- DirectMQTTControllerTest: Tests MQTT communication without actual connections.
- PlugsResourceTest: Verifies endpoint behavior and HTTP responses
- MockEnvironmentTest: Ensures proper configuration management Support Classes: Custom implementations to isolate external dependencies

Each of these classes works together to provide complete coverage for the functionality provided by the IoT simulator while ensuring the autonomy and uniqueness of each respective class. The code struck a balance between exhaustive testing and sustainable code, thus ensuring that the test suite will remain relevant as the system grows.

1.3 Deliverable 3

Q: Screenshot of the unit test report "Test Summary" page

<u>Solution</u>



May 7 22:21

Test results - Package ece448 | iot_ece448

file:///home/ece448s25/iot_ece448/build/reports/tests/test/packages/ece448.iot_hub.html

Package ece448.iot_hub

all > ece448.iot_hub

28	0	0	0.895s	100%
tests	failures	ignored	duration	successful

Classes

Class	Tests	Failures	Ignored	Duration	Success rate
GroupsResourceTests	6	0	0	0.023s	100%
IotHubFullCoverageTest	22	0	0	0.872s	100%

Generated by Gradle 6.9.3 at May 7, 2025, 10:21:04 PM

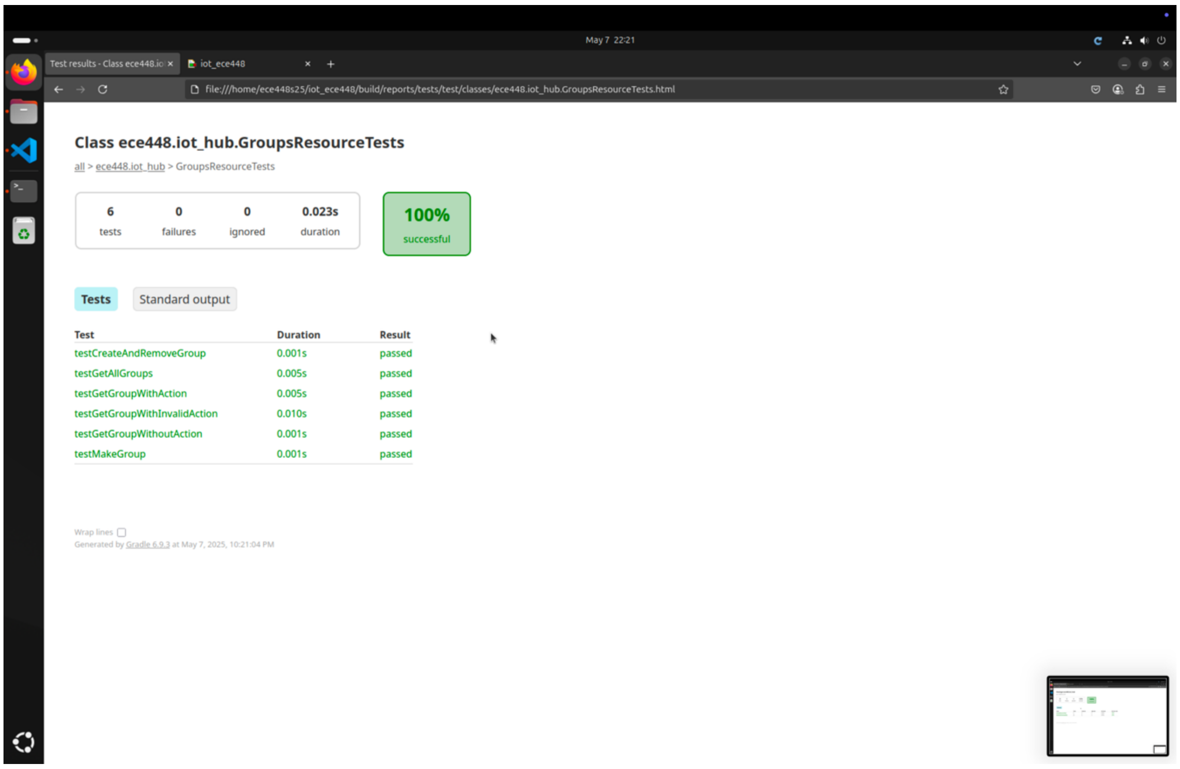
Figure 1: Screenshot of the Test Summary Page.

The above Figure 1 shows the “Test Summary” page screenshot with the added test cases for project 3. This shows the total number of test cases including the count from project 1.

1.4 Deliverable 4

Q: Screenshots of the coverage report pages for those classes you have added/modified. Please also report your average coverage among those pages.

Solution



file:///home/ece448s25/iot_ece448/build/reports/tests/test/classes/ece448.iot_hub.GroupsResourceTests.html

Class ece448.iot_hub.GroupsResourceTests

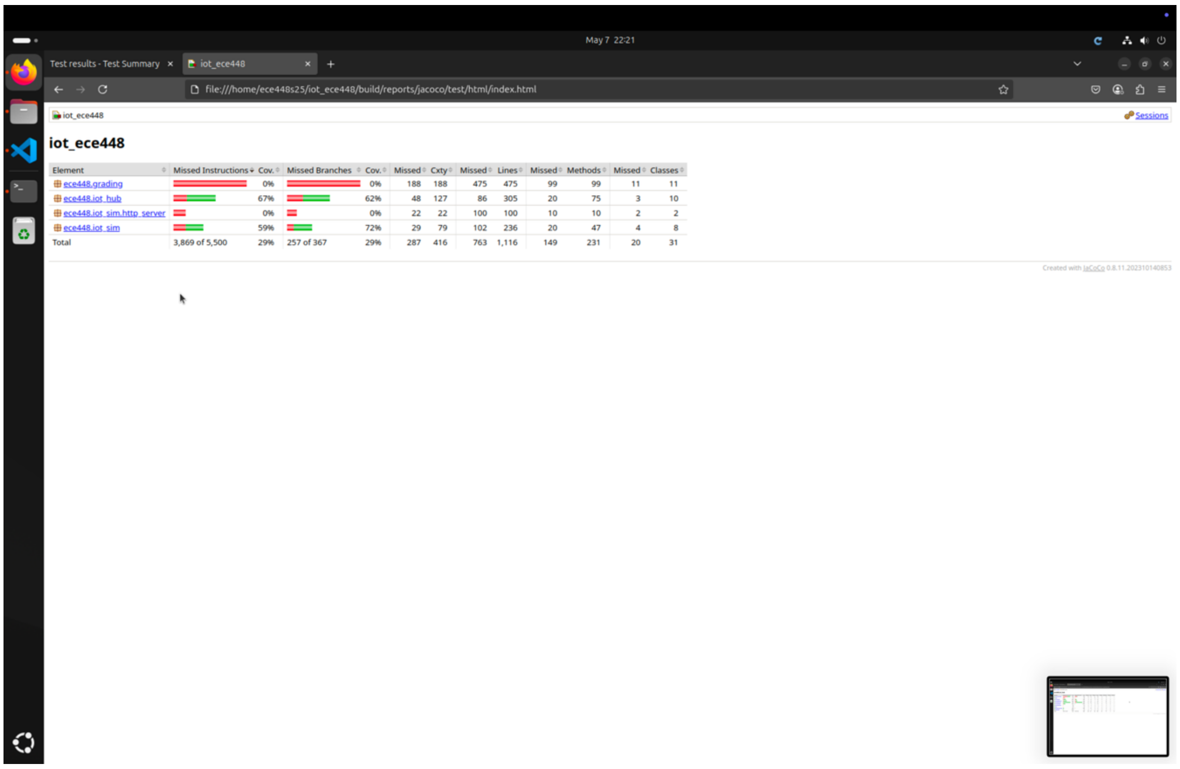
all > ece448.iot_hub > GroupsResourceTests

6	0	0	0.023s	100%
tests	failures	ignored	duration	successful

Test	Duration	Result
testCreateAndRemoveGroup	0.001s	passed
testGetAllGroups	0.005s	passed
testGetGroupWithAction	0.005s	passed
testGetGroupWithInvalidAction	0.010s	passed
testGetGroupWithoutAction	0.001s	passed
testMakeGroup	0.001s	passed

Wrap lines
Generated by Gradle 6.9.3 at May 7, 2025, 10:21:04 PM

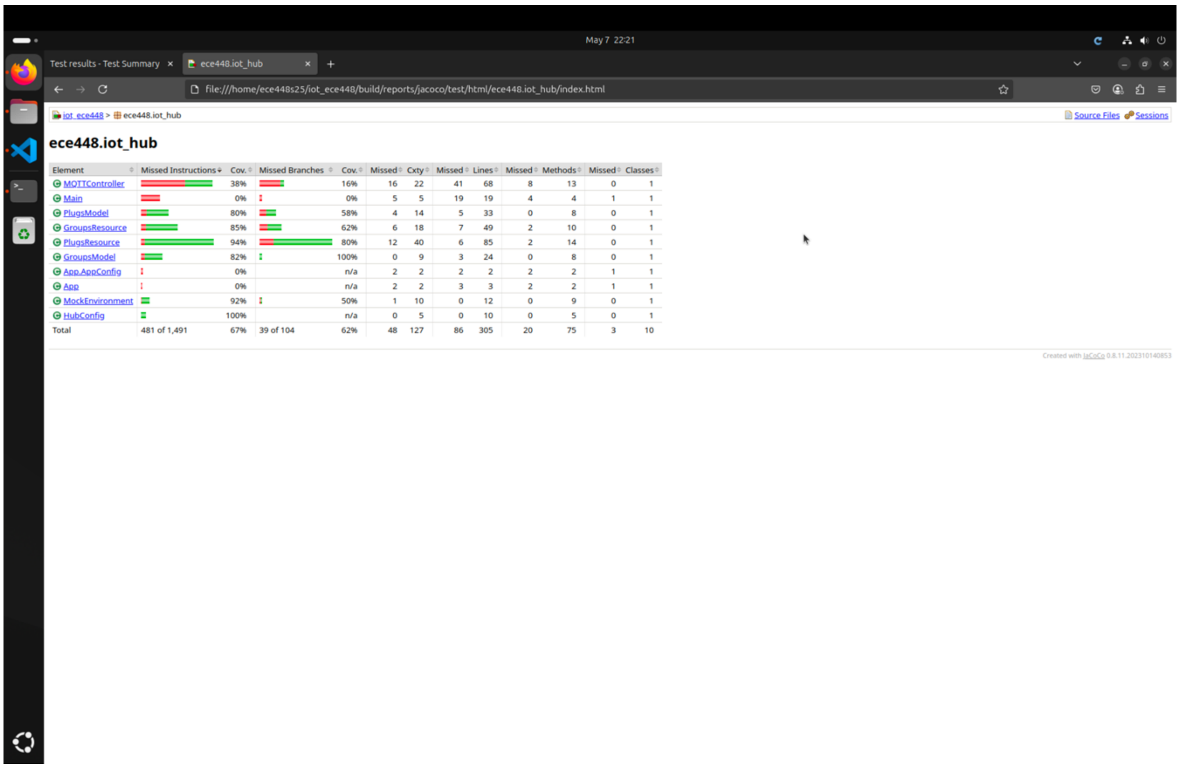
Figure 2: Screenshot of report of individual unit test cases added for testing.



iot_ece448

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
ece448.grading		0%		0%	188	188	475	475	99	99	11	11
ece448.iot_hub		67%		62%	48	127	86	305	20	75	3	10
ece448.iot_sim.http_server		0%		0%	22	22	100	100	10	10	2	2
ece448.iot_sim		59%		72%	29	79	102	236	20	47	4	8
Total	3,869 of 5,500	29%	257 of 367	29%	287	416	763	1,116	149	231	20	31

Figure 3: Screenshot of classes and their individual test report for overall.



ece448.iot_hub

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
MQTTController		38%		16%	16	22	41	68	8	13	0	1
Main		0%		0%	5	5	19	19	4	4	1	1
PlugsModel		80%		58%	4	14	5	33	0	8	0	1
GroupsResource		85%		62%	6	18	7	49	2	10	0	1
PlugsResource		94%		80%	12	40	6	85	2	14	0	1
GroupsModel		82%		100%	0	9	3	24	0	8	0	1
App.AppConfig		0%		n/a	2	2	2	2	2	2	1	1
App		0%		n/a	2	2	3	3	2	2	1	1
MockEnvironment		92%		50%	1	10	0	12	0	9	0	1
HubConfig		100%		n/a	0	5	0	10	0	5	0	1
Total	481 of 1,491	67%	39 of 104	62%	48	127	86	305	20	75	3	10

Figure 4: Screenshot of classes and their individual test report for iot_hub.

The coverage of the individual classes that were added or updated, the average unit test result, the pass percentage of the test classes added, and the amount of time it took for each to run are all displayed in Figure 2 through Figure 4 above.