Application Software Design

ECE 528

Project 3

Abhilash Kashyap Balasubramanyam

abalasubramanyam@hawk.iit.edu

A20566944

Spring 2025

Illinois Institute of Technology

Date: March 16, 2025

Table of Contents

List of Tables

List of Figures

Solutions

1 Deliverables

1.1 Deliverable 1

Q: How did you design the unit test cases?

<u>Solution</u>

To guarantee the durability, dependability, and accuracy of the software components being tested, the IoT simulator project's unit test cases were designed using an organized, methodical, and scholarly approach. Modularity, edge case research, and thorough feature coverage were all incorporated into the process. The design of the unit test cases, which reflects academic rigor and best practices, is described in full below.

Building Philosophy

The foundation of the unit test case design was the idea that separate components should be validated separately while making sure their behavior matched the intended functionality. Testing both common and unusual scenarios was a key component of the design process in order to find any potential defects or edge cases. This method is in line with industry requirements for unit testing, which call for tests that are designed to confirm error-handling procedures, edge circumstances, and regular activities. Making ensuring that every class and method in the IoT simulator project operated as planned in every scenario was the main objective. Each unit test targeted a particular function or method by emphasizing modularity, which prevented the introduction of dependence on external systems. Because of this isolation, the tests were guaranteed to stay reproducible and deterministic.

Essential Design Techniques

1. Coverage of Functions

All of the main features offered by the classes being tested were covered by the test cases. For example:

- To verify their accuracy, PlugSim's switchOn, switchOff, toggle, and measurePower techniques were tested.
- Tests were developed in MqttCommands to confirm how MQTT messages were handled for actions such as "on," "off," "toggle," and invalid circumstances.
- Tests in MqttUpdates concentrated on topic generation, MQTT message creation, and update publication.

This made guaranteed that during testing, every important path in the code was run at least once.

2. Handling Edge Cases
To guarantee robustness, extra consideration was paid to edge scenarios. Among the examples are:

- Testing plugs with names that contain special characters (test.123, for example).
- Verifying behavior when invalid actions or nonexistent plugs were mentioned.
- Concurrent operation simulation is used to confirm thread safety.

The tests sought to find hidden defects that might not show up during regular operations by foreseeing atypical inputs or conditions.

3. Self-reliance and adaptability
Every test case was made to function separately from the others. Isolating dependencies and resetting state prior to each test execution helped achieve this. For example:

- To avoid test-to-test interference, distinct instances of PlugSim were made for every test.
- For every test case, dependencies such as MqttClient were established from scratch.

This modular approach guarantees that errors may be unambiguously linked to particular units and is in line with best standards in software testing.

Implementation Architecture

A systematic strategy was used to implement the unit test cases in order to guarantee their robustness, thorough coverage, and clarity. The design was centered on the Arrange-Act-Assert (AAA) pattern. Objects and preconditions, like setting up MQTT clients or PlugSim instances, were initialized during the Arrange phase. During the Act phase, the code being tested was run, such as by calling handleMessage() or switchOn(). Lastly, the Assert step made sure the tests were deterministic and focused by using assertions (assertTrue, assertFalse, etc.) to validate the results.

Functionality was used to structure the test suites. PlugSim tests, for instance, verified power measurement and plug state management (on/off/toggle). Tests for MqttCommands concentrated on processing messages for actions such as "on," "off," and "toggle," while tests for MqttUpdates made sure that topic generation, message production, and publication ran well. This modular structure made sure that every important detail was covered.

To verify system behavior in the face of unforeseen circumstances, such as incorrect subjects, nonexistent plugs, or exceptions during message publication, error handling was thoroughly tested. The software's ability to manage edge cases smoothly without crashing or yielding inaccurate results was verified by these tests.

Throughout the design process, best practices were adhered to. To increase readability, test names were descriptive (e.g., testSwitchOn). To make debugging easier, each test focused on a specific functionality. Edge cases, such as concurrent operations and special characters in plug names, were extensively tested. All pathways were exercised thanks to code coverage tools like JaCoCo, and repetitive execution was made easier by automation using JUnit.

Getter methods for encapsulation were introduced to address issues like private field access in MqttCommands. Due to project limitations, genuine dependencies like MqttClient were used, however concurrency testing made use of tools like ExecutorService. These modifications preserved independence and modularity while guaranteeing reliable testing.

Test Case Explanations

1. PlugSim Tests
    a. testSwitchOn(): This test confirms that the PlugSim class's switchOn() method works as intended. DeclareTrue(plug.isOn()) asserts that a plug is "on" after it has been initialized and turned on.
    b. testSwitchOff(): This test verifies the proper operation of the switchOff() method. Using assertFalse(plug.isOn()), the plug's state is confirmed to be "off" after it has been turned on and off.
    c. testToggle(): This test assesses the functionality of the toggle. There are two toggles for the plug's state: from "off" to "on" and back to "off." Accurate state changes are confirmed by assertions.
    d. testMeasurePower(): When a plug is turned on, this test confirms that the power metering feature works. It uses assertNotEquals to confirm that the power value is non-zero after calling measurePower().
    e. testMeasurePowerWithDotInName(): This test determines whether plugs with unusual characters (such a dot) in their names are handled appropriately. An expected value based on the plug's name is used to validate the power measurement.
    f. testObserverNotificationOnSwitchOn(): This test makes sure that observers are informed when the plug becomes "on." The state update of a custom observer (TestObserver) is confirmed.
    g. testObserverNotificationOnPowerChange(): This test confirms that observers are notified when the power of a plug changes following the invocation of a measurePower().

2. MqttCommands Tests
    a. testHandleMessageOn(): This test confirms that a plug is turned on when a MQTT message containing the "on" action is sent. It is said that the plug is "on."

   b. testHandleMessageOff(): This test, like the one before it, makes sure that a plug is turned off when a MQTT message with the "off" action is sent.
   c. testHandleMessageToggle(): This test determines whether a plug's state is appropriately toggled by a MQTT message containing the "toggle" action.
   d. testHandleMessageInvalidTopic(): The system's handling of incorrect MQTT topics is verified by this test. It guarantees that the plug's state doesn't change and that no exceptions are raised.
   e. testHandleMessageUnknownAction(): This test looks at how MQTT messages handle unknown actions. When an invalid action is received, the plug should not modify its state.
   f. testHandleMessageNonExistentPlug(): This test makes sure that when a MQTT message refers to a plug that doesn't exist, there are no exceptions raised.
   g. testMqttCommandsConstructor(): This test adds various plugs and asserts their existence to confirm that the MqttCommands constructor initializes the internal map of plugs correctly.
   h. testGetTopic1(): This test confirms that the format of the topic string that getTopic() returns is correct.
   i. testHandleMessageExceptionHandling(): This test provides erroneous input (such as a null subject) to guarantee that exceptions are handled gracefully during message processing.

3. MqttUpdates Tests
   a. testGetTopic(): This test confirms that, given a prefix, plug name, and key, the topic generation method generates topics in the appropriate format.
   b. testGetTopicWithMultiLevelPrefix(): Like testGetTopic, this test verifies that topic creation appropriately handles multi-level prefixes.
   c. testGetMessage(): This test makes sure that the right payloads and retention flags are included in MQTT messages.
   d. testPublishUpdateSuccess(): This test verifies that updates are successfully published to a broker and that subscribers receive the appropriate messages using an actual MQTT client.
   e. testPublishUpdateExceptionHandling(): This test makes sure exceptions are handled politely and without crashing by simulating a publishing failure situation (such as a disconnected client).

4. Concurrency and Edge Case Tests
   a. testPowerRandomWalk(): This test makes sure that non-negative results are consistently provided by evaluating random power measurement over a number of rounds.

b. testPowerCalculationWithDottedName(): Like other tests, this one focuses on particular edge cases while validating power calculation for plugs with dotted names.
c. testConcurrentToggle(): This test verifies thread safety and consistent behavior under simultaneous operations by simulating multiple threads toggling a plug concurrently.

All essential PlugSim, MqttCommands, and MqttUpdates capabilities are well covered by the unit tests in MqttTests.java. Normal operations, edge cases, error handling, concurrency, and integration with third-party systems, such as MQTT brokers, are all validated. Every test case was meticulously created to follow best practices in software testing and guarantee modularity, maintainability, and robustness.

1.2 Deliverable 2

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

Solution

Implementation Strategy

Adding MQTT capability to the IoT simulator was the main goal of this project's feature implementation. In order to smoothly incorporate MQTT-related functions into the current architecture, two new classes, MqttCommands and MqttUpdates, had to be added. Additionally, the Main class had to be modified. A thorough description of the feature specifics, class modifications, and implementation strategy can be seen below.

The main objective was to make the simulator able to communicate with a MQTT broker so that updates could be published in real-time and smart plugs could be controlled using MQTT messages. To guarantee that new features were added without interfering with already-existing functionality, the implementation was done in a modular fashion. The design remained clear and extensible by adding classes specifically for MQTT commands and updates.

Features Implementation

1. MQTT Command Handling

To manage incoming MQTT messages that regulate plug actions (on, off, toggling), the MqttCommands class was created. Each message's topic is parsed by this class, which then retrieves the plug name and action and calls the appropriate methods on the designated plug.

- Topic Parsing: To determine the plug name and action, the handleMessage method divides the topic string into segments.

- Plug Actions: It invokes methods such as switchOn(), switchOff(), or toggle() on the corresponding plug based on the parsed action (on, off, or toggle).
- Error Handling: Unknown actions or invalid topics are recorded but do not interfere with execution.

2. MQTT Update Publishing

In order to publish updates anytime the power or condition of a plug changes, the MqttUpdates class was established. This guarantees that the simulator and external systems are synchronized in real time.

- Topic Generation: The method Update topics are generated by getTopic using a prefix, plug name, and key (such as power or status).
- Message Creation: Using the changed value as the payload, the getMessage function generates retained MQTT messages.
- Publishing Updates: The process publishUpdate uses the produced topic and message to publish these updates to the broker.

3. Integration in Main Class

To incorporate MQTT capabilities, the Main class underwent significant modification:

- MQTT Client Setup: The broker listed in the configuration file is linked to an instance of MqttClient.
- Command Handling: The MqttCommands class is used to process incoming messages by registering a callback with the client.
- Observer Pattern: Every plug has an observer set up that publishes changes in power or state using MqttUpdates.
- Subscription Management: MqttCommands generate topics that the client subscribes to.

Class Additions/Modifications

Class Name	**Type of Change**	**Methods Added/Modified**	**Implementation Details**
MqttCommands	New Class	Constructor, getTopics(), handleMessage(), addPlug()	Handles incoming MQTT messages for controlling plugs via actions (on, off, toggle).
MqttUpdates	New Class	Constructor, getTopics(), getMessages(), publishUpdate()	Applies MQTT topics and messages to publish real-time updates on plug statuses and power readings.
Main	Modified	Constructor, close(), messageArrived(), connectionLost()	Contains MQTT features, including posting updates, managing subscriptions, handling commands, and client setup.

Table 1: Additions/Modifications done to the class summarized.

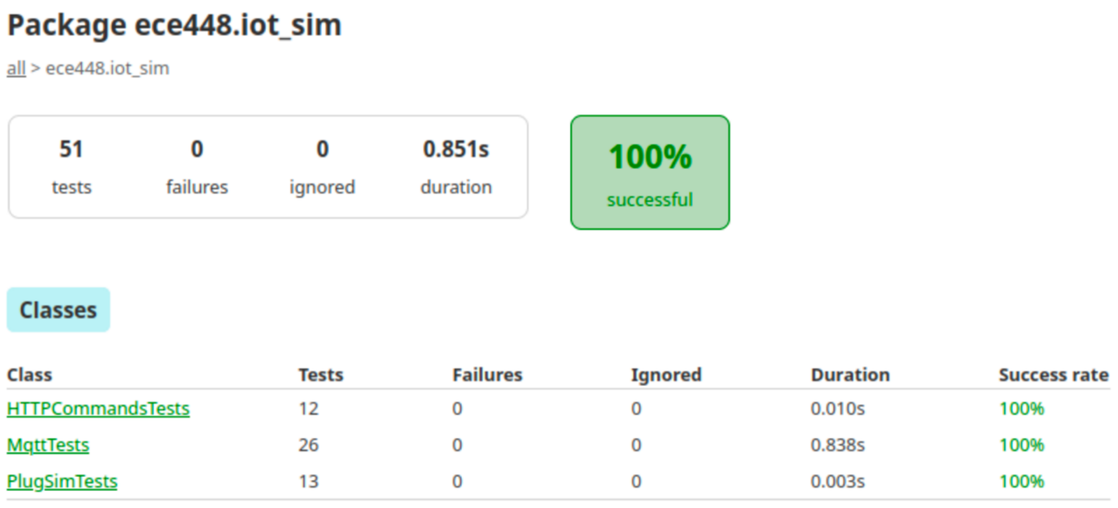
Table 1 provides a concise overview of the additions and modifications made to implement MQTT features in the IoT simulator project. It highlights how new classes (MqttCommands and MqttUpdates) were introduced to handle command processing and update publishing, while the existing Main class was adapted to integrate these functionalities seamlessly into the overall architecture.

The IoT simulator project gained strong MQTT capabilities while preserving flexibility and extensibility with the addition of the MqttCommands and MqttUpdates classes. The Main class was modified to provide smooth interaction with pre-existing elements such as HTTP commands (JHTTP) and smart plugs (PlugSim). This implementation shows a careful approach to design that strikes a compromise between adhering to established architectural principles and developing new features.

1.3 Deliverable 3

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

Solution



Package ece448.iot_sim

all > ece448.iot_sim

51	0	0	0.851s	100%
tests	failures	ignored	duration	successful

Classes

Class	Tests	Failures	Ignored	Duration	Success rate
HTTPCommandsTests	12	0	0	0.010s	100%
MqttTests	26	0	0	0.838s	100%
PlugSimTests	13	0	0	0.003s	100%

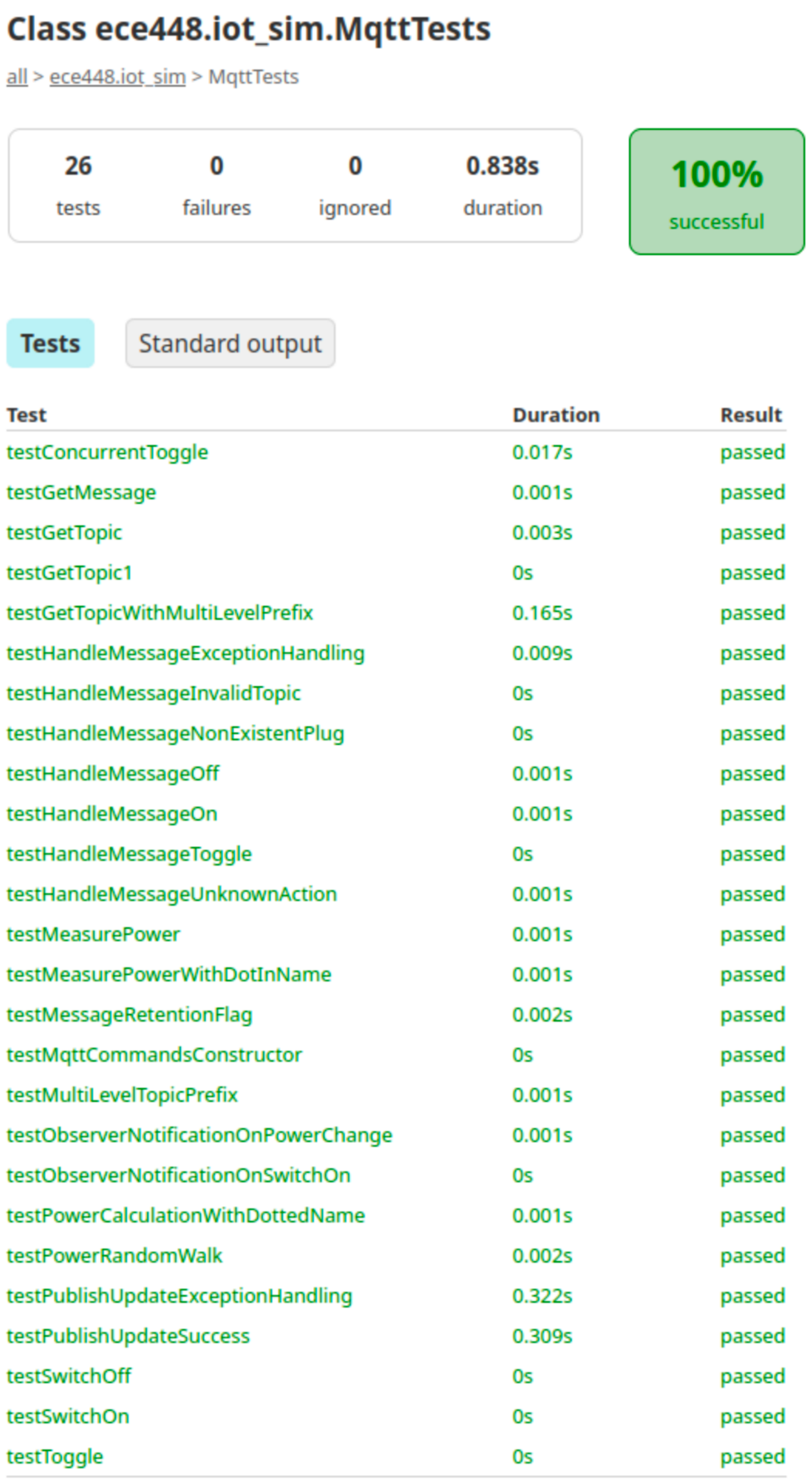
Figure 1: Screenshot of the Test Summary Page.

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


Class ece448.iot_sim.MqttTests

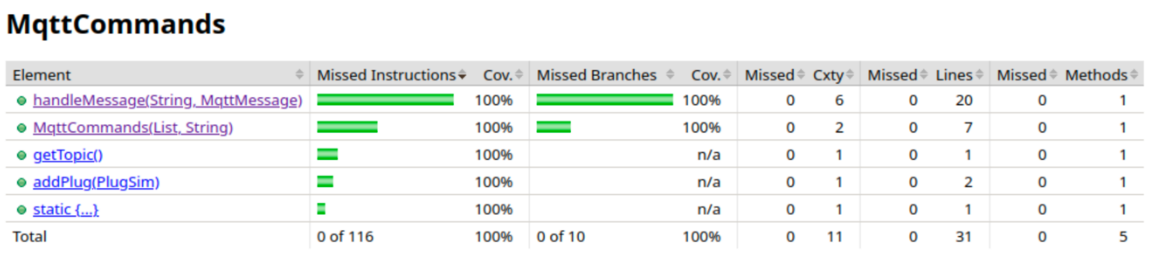
all > ece448.iot_sim > MqttTests

26	0	0	0.838s	100%
tests	failures	ignored	duration	successful

Tests | Standard output

Test	Duration	Result
testConcurrentToggle	0.017s	passed
testGetMessage	0.001s	passed
testGetTopic	0.003s	passed
testGetTopic1	0s	passed
testGetTopicWithMultiLevelPrefix	0.165s	passed
testHandleMessageExceptionHandling	0.009s	passed
testHandleMessageInvalidTopic	0s	passed
testHandleMessageNonExistentPlug	0s	passed
testHandleMessageOff	0.001s	passed
testHandleMessageOn	0.001s	passed
testHandleMessageToggle	0s	passed
testHandleMessageUnknownAction	0.001s	passed
testMeasurePower	0.001s	passed
testMeasurePowerWithDotInName	0.001s	passed
testMessageRetentionFlag	0.002s	passed
testMqttCommandsConstructor	0s	passed
testMultiLevelTopicPrefix	0.001s	passed
testObserverNotificationOnPowerChange	0.001s	passed
testObserverNotificationOnSwitchOn	0s	passed
testPowerCalculationWithDottedName	0.001s	passed
testPowerRandomWalk	0.002s	passed
testPublishUpdateExceptionHandling	0.322s	passed
testPublishUpdateSuccess	0.309s	passed
testSwitchOff	0s	passed
testSwitchOn	0s	passed
testToggle	0s	passed

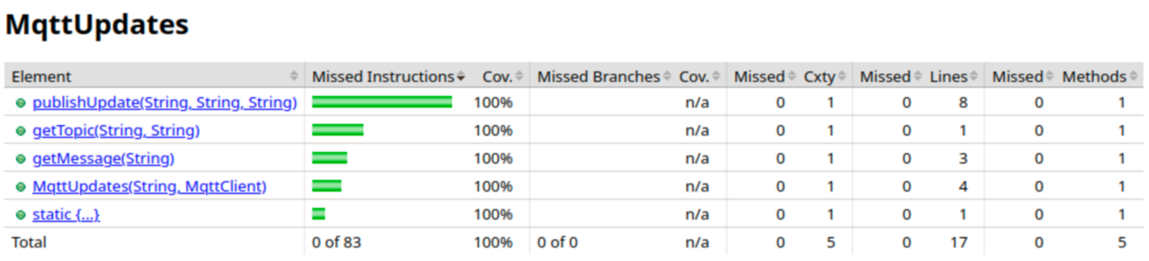
Figure 2: Screenshot of report of individual unit test cases added for testing.



MqttCommands

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
handleMessage(String, MqttMessage)	🟩	100%	🟩	100%	0	6	0	20	0	1
MqttCommands(List, String)	🟩	100%	🟩	100%	0	2	0	7	0	1
getTopic()	🟩	100%		n/a	0	1	0	1	0	1
addPlug(PlugSim)	🟩	100%		n/a	0	1	0	2	0	1
static {...}	🟩	100%		n/a	0	1	0	1	0	1
Total	0 of 116	100%	0 of 10	100%	0	11	0	31	0	5

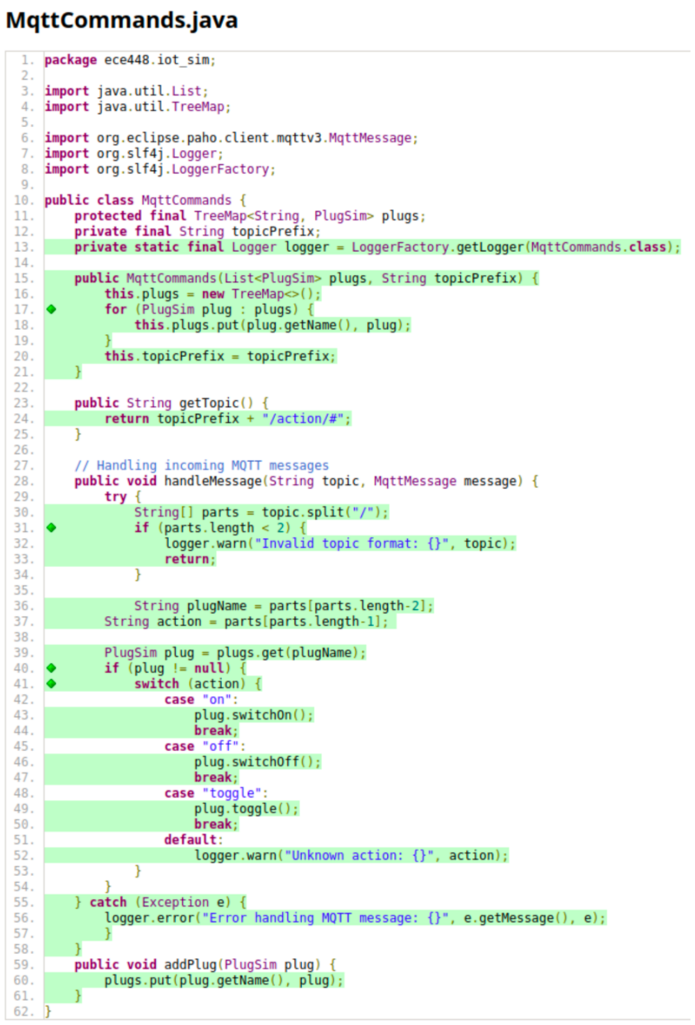
Figure 3: Screenshot of classes and their individual test report for MqttCommands.



MqttUpdates

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
publishUpdate(String, String, String)	🟩	100%		n/a	0	1	0	8	0	1
getTopic(String, String)	🟩	100%		n/a	0	1	0	1	0	1
getMessage(String)	🟩	100%		n/a	0	1	0	3	0	1
MqttUpdates(String, MqttClient)	🟩	100%		n/a	0	1	0	4	0	1
static {...}	🟩	100%		n/a	0	1	0	1	0	1
Total	0 of 83	100%	0 of 0	n/a	0	5	0	17	0	5

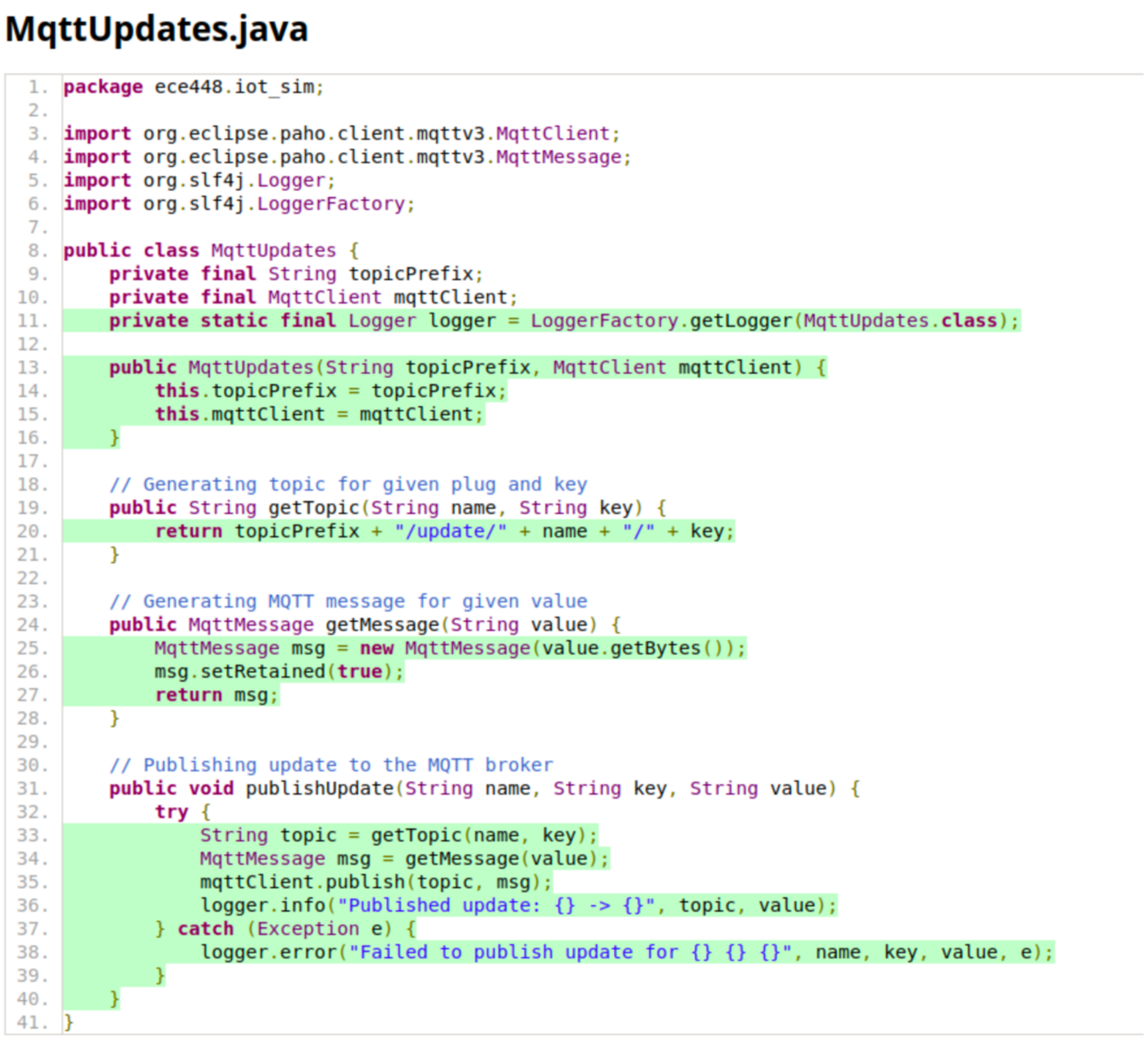
Figure 4: Screenshot of classes and their individual test report for MqttUpdates.



MqttCommands.java

```java
package ece448.iot_sim;

import java.util.List;
import java.util.TreeMap;

import org.eclipse.paho.client.mqttv3.MqttMessage;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class MqttCommands {
    protected final TreeMap<String, PlugSim> plugs;
    private final String topicPrefix;
    private static final Logger logger = LoggerFactory.getLogger(MqttCommands.class);

    public MqttCommands(List<PlugSim> plugs, String topicPrefix) {
        this.plugs = new TreeMap<>();
        for (PlugSim plug : plugs) {
            this.plugs.put(plug.getName(), plug);
        }
        this.topicPrefix = topicPrefix;
    }

    public String getTopic() {
        return topicPrefix + "/action/#";
    }

    // Handling incoming MQTT messages
    public void handleMessage(String topic, MqttMessage message) {
        try {
            String[] parts = topic.split("/");
            if (parts.length < 2) {
                logger.warn("Invalid topic format: {}", topic);
                return;
            }

            String plugName = parts[parts.length-2];
        String action = parts[parts.length-1];

        PlugSim plug = plugs.get(plugName);
        if (plug != null) {
            switch (action) {
                case "on":
                    plug.switchOn();
                    break;
                case "off":
                    plug.switchOff();
                    break;
                case "toggle":
                    plug.toggle();
                    break;
                default:
                    logger.warn("Unknown action: {}", action);
            }
        }
    } catch (Exception e) {
        logger.error("Error handling MQTT message: {}", e.getMessage(), e);
        }
    }
    public void addPlug(PlugSim plug) {
        plugs.put(plug.getName(), plug);
    }
}
```

Figure 5: Screenshot of MqttCommands.java 's coverage of individual classes that were added/modified.


MqttUpdates.java

```java
package ece448.iot_sim;

import org.eclipse.paho.client.mqttv3.MqttClient;
import org.eclipse.paho.client.mqttv3.MqttMessage;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class MqttUpdates {
    private final String topicPrefix;
    private final MqttClient mqttClient;
    private static final Logger logger = LoggerFactory.getLogger(MqttUpdates.class);

    public MqttUpdates(String topicPrefix, MqttClient mqttClient) {
        this.topicPrefix = topicPrefix;
        this.mqttClient = mqttClient;
    }

    // Generating topic for given plug and key
    public String getTopic(String name, String key) {
        return topicPrefix + "/update/" + name + "/" + key;
    }

    // Generating MQTT message for given value
    public MqttMessage getMessage(String value) {
        MqttMessage msg = new MqttMessage(value.getBytes());
        msg.setRetained(true);
        return msg;
    }

    // Publishing update to the MQTT broker
    public void publishUpdate(String name, String key, String value) {
        try {
            String topic = getTopic(name, key);
            MqttMessage msg = getMessage(value);
            mqttClient.publish(topic, msg);
            logger.info("Published update: {} -> {}", topic, value);
        } catch (Exception e) {
            logger.error("Failed to publish update for {} {} {}", name, key, value, e);
        }
    }
}
```

Figure 6: Screenshot of MqttUpdates.java 's coverage of individual classes that were added/modified.

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 6 above.