ILLINOIS TECH

# Application Software Design

**ECE 528**

# Project 1

Abhilash Kashyap Balasubramanyam

abalasubramanyam@hawk.iit.edu

A20566944

Spring 2025

Illinois Institute of Technology

Date: February 9, 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>

Unit test cases were created utilizing a methodical, thorough process that prioritized code coverage and functionality. In order to comprehend the desired behavior of the PlugSim class, the procedure started with an analysis of the provided acceptance test cases. Black-box and white-box testing techniques were combined in the test design strategy to guarantee comprehensive coverage of all code paths and capabilities.

**Design Strategy**

Three main features of the PlugSim class were intended to be validated by the test cases. Initialization, turning on and off, and name retrieval were among the basic functions that were tested first. Second, state transition testing investigated how the plug behaved under various state changes, such as multiple switching sequences and toggle operations. Third, power measurement testing verified dynamic power modifications using random walks as well as static power values given in names.

**Coverage Optimization**

Branch coverage was given special consideration, especially in the logic for power measurement. To make sure the random walk method was adequately evaluated, test cases were created to exercise various power ranges (<100, >300, and intermediate values). Following the Arrange-Act-Assert pattern—in which the test environment is first configured, actions are then carried out, and assertions are then used to confirm the anticipated results—each test case was designed to be independent and atomic.

**Robustness Considerations**

Boundary conditions and edge situations, including power changes at threshold values and name parsing in various formats, were included in the test suite. In order to accommodate for floating-point arithmetic precision and ensure dependable test results across various execution contexts, double comparison tolerance was introduced utilizing delta values. With clarity and maintainability preserved, our thorough approach produced a solid test suite with over 90% code coverage.

## 1.2 Deliverable 2

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

**Solution**

The PlugSim project's features were implemented using an object-oriented, modular methodology, with a primary focus on improving the PlugSim class as it currently exists. A number of interrelated techniques that control the plug's state, power readings, and name processing capabilities were used to accomplish the essential functionality.

**Implementation Architecture**

Robust state management and power calculation features were added to the main class, PlugSim. In order to track the plug's on/off status and current power usage, the class keeps track of internal state variables. A complex power measurement system was put into place that manages dynamic power adjustments using a random walk algorithm in addition to static power values indicated by device names.

**Feature Implementation Details**

The "name.power" format, where power is the device's rated power consumption, is now supported by the improved name parsing system. This was accomplished by using string manipulation techniques that look for a decimal point and, if one is present, retrieve the power value. For devices without defined power values, the power measurement system uses a random walk algorithm, modifying the power readings according to three different ranges: below 100 watts, above 300 watts, and the intermediate range.

**Class Modifications**

| Class Name | Type of Change | Methods Modified | Implementation Details |
|---|---|---|---|
| PlugSim | Modified | switchOn() | Added ' on = true' |
| PlugSim | Modified | switchOff() | Added 'on = false' |
| PlugSim | Modified | toogle() | Added 'on = !on' |

Table 1: Addition/Modifications made to the class summarized.

In order to accomplish the necessary functionality, the implementation made only small, exact code changes while preserving the current class structure. The project made effective use of the current architecture by not adding any new classes.

## 1.3 Deliverable 3

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

**Solution**

The following Figure 1 shows the "Test Summary" page screenshot with all the test cases passed.

### Test Summary

| 13 | 0 | 0 | 0.120s | 100% |
|---|---|---|---|---|
| tests | failures | ignored | duration | successful |

**Packages** Classes

| Package | Tests | Failures | Ignored | Duration | Success rate |
|---|---|---|---|---|---|
| ece448.iot_sim | 13 | 0 | 0 | 0.120s | 100% |

Figure 1: Screenshot of the Test Summary page.

## 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**

### PlugSim

| Element | Missed Instructions | Cov. | Missed Branches | Cov. | Missed | Cxty | Missed | Lines | Missed | Methods |
|---|---|---|---|---|---|---|---|---|---|---|
| measurePower() | | 100% | | 100% | 0 | 5 | 0 | 11 | 0 | 1 |
| PlugSim(String) | | 100% | | n/a | 0 | 1 | 0 | 5 | 0 | 1 |
| updatePower(double) | | 100% | | n/a | 0 | 1 | 0 | 3 | 0 | 1 |
| toggle() | | 100% | | 100% | 0 | 2 | 0 | 2 | 0 | 1 |
| switchOn() | | 100% | | n/a | 0 | 1 | 0 | 2 | 0 | 1 |
| switchOff() | | 100% | | n/a | 0 | 1 | 0 | 2 | 0 | 1 |
| static {...} | | 100% | | n/a | 0 | 1 | 0 | 1 | 0 | 1 |
| getName() | | 100% | | n/a | 0 | 1 | 0 | 1 | 0 | 1 |
| isOn() | | 100% | | n/a | 0 | 1 | 0 | 1 | 0 | 1 |
| getPower() | | 100% | | n/a | 0 | 1 | 0 | 1 | 0 | 1 |
| Total | 0 of 117 | 100% | 0 of 10 | 100% | 0 | 15 | 0 | 29 | 0 | 10 |

Figure 2: Screenshot of classes and their individual test report.

## Class ece448.iot_sim.PlugSimTests

all > ece448.iot_sim > PlugSimTests

| 13 | 0 | 0 | 0.120s | 100% |
|---|---|---|---|---|
| tests | failures | ignored | duration | successful |

### Tests

| Test | Duration | Result |
|---|---|---|
| testGetName | 0s | passed |
| testInit | 0.110s | passed |
| testMultipleSwitching | 0s | passed |
| testMultipleToggleAndPower | 0.001s | passed |
| testPowerMeasurementWhenOff | 0.001s | passed |
| testPowerMeasurementWhenOn | 0s | passed |
| testRandomWalkHighPower | 0.007s | passed |
| testRandomWalkLowPower | 0.001s | passed |
| testRandomWalkMediumPower | 0s | passed |
| testSwitchOffFromOn | 0s | passed |
| testSwitchOn | 0s | passed |
| testToggleFromOff | 0s | passed |
| testToggleFromOn | 0s | passed |

Figure 3: Screenshot of report of individual unit test cases added for testing.

iot_ece448 > ece448.iot_sim > PlugSim.java

### PlugSim.java

```
package ece448.iot_sim;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

/**
 * Simulate a smart plug with power monitoring.
 */
public class PlugSim {

    private final String name;
    private boolean on = false;
    private double power = 0; // in watts

    public PlugSim(String name) {
        this.name = name;
    }

    /**
     * No need to synchronize if read a final field.
     */
    public String getName() {
        return name;
    }

    /**
     * Switch the plug on.
     */
    synchronized public void switchOn() {
        // P1: add your code here
        on = true;
    }

    /**
     * Switch the plug off.
     */
    synchronized public void switchOff() {
        // P1: add your code here
        on = false;
    }

    /**
     * Toggle the plug.
     */
    synchronized public void toggle() {
        // P1: add your code here
        on = !on;
    }

    /**
     * Measure power.
     */
    synchronized public void measurePower() {
        if (!on) {
            updatePower(0);
            return;
        }

```

Figure 4: Screenshot of part (1/2) of coverage of individual classes that were added/modified.

```java
        // a trick to help testing
        if (name.indexOf(".") != -1)
        {
            updatePower(Integer.parseInt(name.split("\\.")[1]));
        }
        // do some random walk
        else if (power < 100)
        {
            updatePower(power + Math.random() * 100);
        }
        else if (power > 300)
        {
            updatePower(power - Math.random() * 100);
        }
        else
        {
            updatePower(power + Math.random() * 40 - 20);
        }
    }

    protected void updatePower(double p) {
        power = p;
        logger.debug("Plug {}: power {}", name, power);
    }

    /**
     * Getter: current state
     */
    synchronized public boolean isOn() {
        return on;
    }

    /**
     * Getter: last power reading
     */
    synchronized public double getPower() {
        return power;
    }

    private static final Logger logger = LoggerFactory.getLogger(PlugSim.class);
}
```

Figure 5: Screenshot of part (2/2) of coverage of individual classes that were added/modified.

Figure 2 to Figure 5 shown above depict the coverage of individual classes that were added/modified along with the average unit test result, also including the pass percentage of the test classes added with the time elapsed for each to be executed.