Software Testing Plan

March 25, 2025

Lumberjack Balancing

Project Sponsor: Dr. Scot Raab

Project Mentor: Paul Deasy

Team Members: Riley Burke, Cristian Marrufo, Sergio Rabadan, Braden Wendt

Table of Contents

Introduction

The Lumberjack Balancing application is a Python-based desktop tool developed to automate and streamline faculty workload calculations for Northern Arizona University (NAU). Designed for use by associate deans, the system ingests faculty data from Excel spreadsheets, applies customizable workload policies, identifies edge cases, and generates comprehensive, policy-aligned workload reports. Its core goals are to reduce the manual burden of data analysis, improve accuracy in workload assessments, and allow administrators to adapt quickly to evolving institutional policies.

Software testing is an essential phase of the development lifecycle, ensuring that the system functions as intended, performs reliably under expected conditions, and meets the defined requirements. At its core, software testing is about building confidence: in the code, in the user experience, and in the trust that the final product will support its users without failure. Testing helps uncover bugs, validates key functionality, and confirms that performance metrics and usability goals are being met. Especially for applications with critical decision-making roles, such as automating faculty workloads. Robust testing is vital to prevent miscalculations, data integrity issues, and user frustration.

Our testing plan for Lumberjack Balancing incorporates unit testing, integration testing, and user acceptance testing. Each will be executed in different ways to validate key modules and workflows.

- Unit testing will focus on the individual functions and classes within the data processing module.
- Integration testing will ensure that the main menu, data processing, and data visualization all work together.
- User acceptance testing will be conducted with our sponsor and will evaluate usability, report clarity, and overall user experience.

In the sections that follow, we will present each stage of our testing process in detail, outlining the strategies, tools, test cases, and expected outcomes for every type of testing performed. This testing regimen reflects our commitment to quality, accuracy, and usability in supporting NAU’s mission to improve faculty workload management.

Unit Testing

Unit testing is the process of testing individual components or “units” of a software application in isolation to verify that each part functions correctly. The primary goals of units testing are:

- Verify Correctness: Ensure that each method or procedure produces the expected output given specific inputs.
- Facilitate Maintenance: Detect changes or regressions in functionality early as code evolves.
- Document Behaviour: Provide a living specification of how the code is intended to work.

For this project, we will use the **unittest** library in Python as our primary testing framework. Additionally, we plan to use **pytest** for more complex test cases and **coverage.py** to measure test coverage. Our test process will involve writing test cases for both typical and edge cases, including valid inputs, boundary values, and erroneous inputs to ensure robustness and correctness.

Lumberjack Balancing is divided into several modules, the critical units we plan to test and their respective equivalence partitions and boundary values are detailed below:

1. **Row Validation Function**
    - Equivalence Partitions:
        - Valid Rows: Rows with all required fields present.
        - Invalid Rows: Rows missing any of the common fields or for certain courses, missing meeting details.
    - Boundary Cases:
        - A row where a numeric field is exactly 0.
        - A row with borderline valid date/times field.
    - Test Inputs/Examples:
        - Provide a row with complete and correct data to ensure it returns True.
        - Provide a row missing one of the common fields to ensure it returns False.
        - Provide a row for a research course that is missing meeting details (research courses do not require explicit meeting information) and ensure it still returns True.
    - Error Handling:

        - Test with null values and unexpected types to confirm the function handles them correctly.

2. **Course Class Methods**

For each method in the course class, we will define test cases as follows:

- **getGroupKey()**
    - Equivalence Partitions:
        - Courses with complete course identifiers (term, subject, catNbr, section).
        - Research courses, team-taught, and co-convened courses.
    - Boundary Cases:
        - Missing optional fields such as section or meeting details.
    - Test Input/Examples:
        - A research course row should return a key that includes the instructor ID, term, subject, and the research tag.
        - A team-taught course should group by meeting details.
        - A co-convened course should be grouped by instructor and course identifiers.
- **getBaseRate()**
    - Equivalence Partitions:
        - Courses that are independent study, research, or fieldwork.
        - Laboratory courses.
        - Lecture/recitations/seminar courses.
    - Test Inputs/Examples:
        - Verify that a course with “laboratory” in its category returns the laboratory rate.
        - Verify that a research course returns the independent study rate.
        - Verify default rate for a course that does not match any special keywords.
- **adjustForEnrollment(baseRate)**
    - Equivalence Partitions:
        - Enrollment below, at, and above threshold boundaries (90, 150, 200).
    - Boundary Cases:
        - Enrollment exactly at 90, 150, and 200.
    - Test Inputs/Examples:
        - For a lecture course with enrollment of 89, verify that the base rate remains unchanged.
        - For enrollment of 90, 150, and 200, verify that the rate is interpolated correctly by using the linear interpolation formula.

- **calculateLoad()**
    - Equivalence Partitions:
        - Courses with zero enrollment.
        - Courses with valid enrolment and max units.
        - Courses with extra load applied for special cases.
    - Boundary Cases:
        - Maximum load should not exceed the cap of 5 for research courses.
    - Test Inputs/Examples:
        - Verify that a course with enrollment of 0 returns a load of 0.
        - Verify that the load is computed as **maxUnits * adjustedRate** for a typical lecture course.
        - Verify that if the course qualifies for an extra load, the extra value is added.
- **adjustLoadDivision(count)**
    - Equivalence Partitions:
        - A course load is divided by the number of instructors (count > 1).
    - Test Inputs/Examples:
        - For a team-taught course with a computed load of 10 and count equal to 2, verify that the new load is 5.
- Error Handling:
    - Test methods with erroneous inputs such as non-numeric enrollment to ensure robust error management.

3. FacultyMember Class Methods

- **addCourse(course)**
    - Test Inputs/Examples:
        - Verify that adding a course with a unique group key stores it in the faculty member's courses.
        - Verify that adding a duplicate course (same group key) does not duplicate the entry.
- **calculateTotalLoad()**
    - Test Inputs/Examples:
        - Create a faculty member with several courses, and verify that the total load equals the sum of individual course loads.
        - Test with courses having load equal to 0 and non-zero loads.
- **calculatePercentage()**
    - Equivalence Partitions:
        - TT (Tenure Track) faculty: baseline 30 (60 for AY).
        - CT (Career Track) faculty: baseline 40 (80 for AY).
    - Boundary Cases:

        - Test when total load is exactly at the baseline and when it exceeds the baseline.
    - Test Inputs/Examples:
        - Verify that a TT faculty with a total load of 30 returns 100%.
        - Verify that a CT faculty with a total load of 40 returns 100%.
        - Verify that overload percentages are computed correctly.
        - Verify that underload percentages are computed correctly.

Integration Testing

Integration testing ensures that the interfaces between major modules work together correctly. Rather than testing individual functions in isolation, integration tests focus on the “plumbing” to ensure that data passed between modules, file interfaces, and inter-module procedures are maintained.

The primary integration testing goals for our application are the following:

- Confirm that data read from external sources such as Excel files, are correctly parsed and handed off to the appropriate processing modules.
- Validate that interfaces between modules, and specifically the loader functions, correctly exchange parameters and return values.
- Detect issues in the wiring that may appear during unit testing of isolated components.

- **Policy and Data Loading Functions**
    - Equivalence Partitions:
        - Valid File Input: Excel file with correct policy values.
        - Invalid File Path/Format: Missing or incorrectly formatted files.
        - Partial Data: Files with some missing or empty cells.
    - Boundary Cases:
        - A file that contains exactly the minimum required number of rows/columns.
        - Values that are exactly at the boundary, specifically pertaining to the threshold values indicating enrollment totals.
    - Test Inputs/Examples:
        - Verify that **loadWorkloadPolicy** returns the default policy when the file cannot be read.
        - Verify that **loadInstructorTrack** correctly builds a dictionary when provided a valid Excel file.
        - Verify that **loadSpecialCourses** returns an empty set when no valid courses are provided.
    - Error Handling:
        - Test that the functions log errors and do not crash when given an invalid path or file format.

Usability Testing

Usability testing is a method of evaluating how effectively and comfortably real users can interact with a software application. Unlike other forms of testing that focus on technical correctness or performance, usability testing is centered on the user experience. Its primary goal is to determine whether users can successfully complete tasks within the system, how efficiently they do so, and how satisfied they are with the overall interaction. This process typically involves selecting representative users—those who reflect the actual audience of the application—and asking them to perform realistic tasks while observers monitor their behavior, note points of confusion or frustration, and collect feedback. Usability testing helps uncover issues related to navigation, interface clarity, workflow logic, and general ease of use. For applications like Lumberjack Balancing, which are intended for non-technical administrative users, usability testing is especially critical. It ensures that the application is not only functional but also intuitive, accessible, and efficient to use. By addressing usability concerns early, a development team can improve adoption, reduce training needs, and create a smoother, more productive experience for its users.

Given the nature of our application and its intended users, usability testing is a critical component of our overall testing strategy. Lumberjack Balancing is designed for administrative personnel and associate deans at Northern Arizona University, many of whom may not have a technical background. Therefore, ensuring that the application is not only functional but also intuitive, accessible, and easy to navigate is essential for adoption and long-term success. Because this software will be used to support high-stakes, policy-driven decision-making, the interface must support accurate task completion without confusion or unnecessary complexity.

With this in mind, we've opted for a targeted and feedback-driven usability testing approach, focusing on realistic user studies and expert feedback rather than large-scale generalized testing. Our primary subject for this testing is our client, Dr. Scot Raab, who represents the primary user base and possesses deep familiarity with the current manual workload process. His feedback provides both a high-level administrative perspective and practical insights from daily use cases. We believe that working closely with a domain expert throughout development is more impactful than broad, shallow testing with unfamiliar users. However, to supplement this, we will also perform peer walkthroughs and short focus-group-style demonstrations with fellow students and mentors to identify any unexpected usability friction from fresh perspectives.

Our usability testing plan consists of the following key components:

Expert Review (1 session)

- Participant: Dr. Scot Raab (Client and Associate Dean)
- Format: A guided walkthrough of the current version of the application, highlighting all major workflows (uploading files, configuring policies, generating reports).
- Goals: Identify unclear interface elements, confusing terminology, and steps that may require further instruction or reordering.
- Timing: Early April
- Data Collection: Notes taken by team members and post-session discussion.
- Analysis: Feedback will be compiled into action items prioritized by severity and feasibility for immediate iteration.

Acceptance Testing Session (1 session)

- Participant: Dr. Raab
- Format: Final version of the application is presented and used in a fully self-directed way to complete key administrative tasks.
- Goal: Confirm that the software is usable and satisfactory in its final state and meets the original client expectations.
- Timing: Final week before delivery.
- Data Collection: Client feedback and final approval signature or revision notes.
- Analysis: Used to make final polishing adjustments before submission.

This usability testing process spans from early March through early April, overlapping with our general testing and polishing phases. By concentrating our efforts on qualitative data collection, real-world usage feedback, and one-on-one expert reviews, we're ensuring the interface is tailored specifically to NAU's context. The decision to keep the testing scope focused but highly representative stems from the fact that the application will be used by a small, well-defined user group with specific needs. Our limited but meaningful testing sessions will allow for deep insights, fast iteration, and a well-tuned user experience.

Conclusion

The Lumberjack Balancing software testing plan has been thoughtfully designed to ensure the delivery of a reliable, accurate, and highly usable application for Northern Arizona University. Through a comprehensive combination of **unit testing**, **integration testing**, and **usability testing**, we have addressed the system's core technical functionality, its inter-module cohesion, and the overall user experience. Unit testing will allow us to verify that each individual function performs as intended across a range of normal and edge-case scenarios, while integration testing ensures the application's core modules—data ingestion, processing, and visualization—work together seamlessly and robustly.

Our **usability testing strategy** reflects the practical reality of our target users: non-technical administrative staff who rely on this software to support high-stakes workload decisions. By working directly with our client, Dr. Scot Raab, and conducting real-world usage reviews and acceptance testing, we are able to gather meaningful qualitative feedback that directly informs design decisions. This targeted, context-aware approach is more effective than generic user testing in this case, as it prioritizes relevance, clarity, and task efficiency for the actual stakeholders.

Altogether, this testing plan provides layered coverage across the system's technical and experiential dimensions, ensuring that the final product is not only **functionally correct** but also **intuitive, resilient, and trustworthy**. By identifying potential issues early and resolving them through focused, iterative feedback, we are confident that Lumberjack Balancing will be a dependable tool for automating faculty workload assessments at NAU. The following sections of this document outline the detailed testing procedures, test cases, and metrics that will guide this process and ensure quality throughout the final stages of development.