Verification

Testing in the Small

White box testing: Currently we do not have the information to start our white box testing. However, we plan on doing the following types of coverage once we receive the necessary information.

Statement Coverage: We need to be able to hit every part of our code with our tests so that any possible errors can be addressed.

Edge Coverage: We will create a control flow diagram for our code when we have written it. It is important that we write tests that traverse each edge of the diagram at least once to catch as many errors as possible.

Condition Coverage: An enhancement of Edge coverage, we will make sure that our tests catch even more errors. To do this, we will create tests that result in all possible values of the constituents of compound conditionals at least once. This simply means that given a conditional such as "if A and B", we will create tests that address each combination of A and B where A and B can either be true or false.

Black box testing:

In black box testing, we are checking to see if a specific part of a program does what it is intended to do. This will be done through our cucumber tests in our project. By taking a test driven development approach, we can write tests that say how the features of our application should work and then we can actually code our features to make our tests pass. This type of testing is testing driven by logic specification.

When we have modules written, we will use other black-box techniques such as decision table-based testing and a cause-effect graph.

Testing in the Large:

There are three main aspects to testing in the large.

Modularity Testing: Based on how we decide to implement our code, we will test each module using white and black-box techniques. We will also identify which modules depend on one another to operate properly.

We currently have one feature with one scenario written as a cucumber test. The feature is logging in and the scenario is that the input is a valid username and password. The application will access the database for information, so it expects the database to be structured a certain way.

Integration Testing: As we continue to add modules, we will test to see if the new modules integrate well with the previously tested modules.

We will use a top-down approach for integration testing. This will allow us to view the application in a more abstract sense in terms of what features it has and then push down to subsets of those features.

System Testing: Once we have a finished prototype, we will test the entire system.

We will use overload and robustness testing to figure out the limits of our application. However we will not worry about testing for regression because this is just a semester-long project that will be, for the most part, out of our hands when we are finished. If we were required to be in charge of this application and adding new features as they were requested, we would be more concerned with testing for regression.

Analysis

Code Walk-Throughs: We will have several scheduled opportunities to go through our code and trace it by hand to find any logic errors that have occurred in the process of writing the code. When these errors are found they will be documented and dealt with appropriately.

Code Inspections: Similar to the code walk-throughs, we will have several code inspections to look through our code with the specific intent of looking for common errors in the code. If these are found then they will be documented and dealt with appropriately.

Correctness Proofs: We will not be using correctness proofs on our project and are instead focusing on testing using the Cucumber framework. We believe that this will be sufficient in providing feedback on the functionality of our system and feel that using correctness proofs would be more detrimental than helpful for the following reasons:

- would be longer and more complex than the programs it is intended to prove.
- overwhelms the designers with more details than could easily be dispensed with if they use an informal analysis.
- Even if mathematical certainty of the correctness of a program can be achieved, we cannot rely on it in an absolute way because there could be a failure in the implementation of the language (the compiler) or even in the hardware.

Symbolic Execution

Symbolic execution allows us to combine the techniques of analytical testing and experimental testing. We won't be needing to use symbolic execution in our testing.

Model Checking

For model checking, we already have a basic finite state machine. We will use it in our testing to see if our application does fulfill each property that we have specified. If not, we will have a counterexample, or a test case that cause the application to break.

Debugging

Debugging is technically not a verification activity, but it still goes hand in hand with our other testing approaches.
A failure is a behavior that does not match the product specifications. It is difficult to find certain failures in a project because the failure can be determined by multiple errors.
In order to debug our project, we have a few precautions in place. Using Sinatra, we can see if there is a certain action that breaks our system, we are immediately notified with the specific error on our web page. From there, we can go through our code and see what line caused the error. If we reach a failure that greatly affects the features of our application, then we will need to reevaluate the coding of those features and have our tests address that change.

Verifying Other Software Properties

Performance: For the scope of our project, we will not be using formal performance analysis technique such as worst-case runtime simply because taking into the intent of our application, unless the product performs notably in an unreasonable time, formal analysis is unnecessary.

Reliability: As with performance, it is unnecessary for our application to have extensive formal reliability testing that would be more extraneous than helpful and thus we will measure reliability with very basic level probability statistics such as how many times out of 10 tests does it return a correct result.

Subjective Qualities: Considering the scope and intent of our project, the measurements of subjective qualities in our application such as a program's complexity will be left to the developers internal discretion. It is not necessary for us to use complex theories to validate these qualities in our project because its small scope allows the developers to gauge these using human intuition and still be able to use accurate metrics for these qualities.