

Writing A Test Plan

Distributed: Monday, September 28th.

A Test Plan documents the testing performed to validate your implementation. It is an important part of the documentation, and this is reflected in its weighting in the grade distribution.

The Test Plan should convince the reader that your implementation has been carefully, thoughtfully, and thoroughly tested. This document is called a *plan* because it is more than a random collection of small test cases that were applied to parts of your implementation at some point. It should reflect a methodical approach to covering various categories of test input. Your test plan can be restricted to system tests only, but it should still cover all the error messages, boundary conditions, features and instructions, trivial cases, categories of input, as well as some typical inputs.

You do not have to write a great deal of prose, since this document will not be “read” from start to finish. It will, however, be referenced. A good test guide should allow us to quickly answer questions such as: “Was feature X tested?”, “How was feature X tested?”, and “Why was failure Y not caught in the test suite?”.

Your test plan does not have a primary author, but it should still be viewed as one of the four “primary documents” in your documentation bundle. (See the handout on “Instructions for Submitting Labs.”) Thus, it should be separated from the rest with a tabbed page and have its own title page, table of contents, and the tests that were run.

Table of Contents

The table of contents reflects the structure of the test plan. It should be clear from examining the table of contents whether the test suite is complete. Therefore, the table of contents should be laid out according to the methodical approach that was taken to generate the collection of test cases. It should include enough detail to allow the reader to determine which tests should be examined to answer the questions posed above (*e.g.*, was feature X tested?).

It should also contain page numbers or some other mechanism to allow quick indexing into the test plan document.

The Tests

Each test case should include input and actual output. The output mode used should generate enough diagnostic output to verify that the correct behavior was performed. There is a balance to be struck here: the most verbose output mode may make your test hard to follow, but the most terse mode may make your test hard to validate. Choose an output mode that is appropriate for each particular test case.

Automating the test cases, so that they can be executed with a single script and the output compared against expected output, is very useful. This will let you quickly verify that any changes (eg future revisions) made to the implementation preserve the original functionality. JUnit is an example of a testing framework that allows you to automate the testing process, at least for unit tests.