How to write a Test Plan

The Test Plan document describes how the software you will develop will be tested to ensure, or at least improve, correctness. There are several reasons why the Test Plan should be developed prior to, or at least concurrently with, the coding phase.

- In large teams, the Test Plan team can and should be a separate entity from the Coding team: both take their input information directly and only from the Design document, thus, if any discrepancies arise between a test and the code, this might point out ambiguities in the Design document, which must then be resolved before the test and the code can be reconciled.
- As an added advantage, having separate teams work on the Test Plan and the Code concurrently decreases the overall duration of the development schedule.
- If the Test Plan is developed after the code by the code developers themselves, it is much more likely that the tests will be written with the code in mind, regardless of what the Design document says or of what the User Manual specifies. While this is perhaps acceptable for very low-level unit testing (which might benefit from being aware of peculiar implementation details that should be stressed during testing), it is not a good idea due to the inherent coder's bias: humans tend to overlook their own mistakes.

A Test Plan should address the following testing activities

- Unit testing: to test individual data structures and algorithms (objects and methods).
- Integration testing: to test that multiple data structures and algorithms interface properly to deliver the overall required functionalities.
- User-oriented testing: to ensure that functionalities and modalities of interactions described in the Specification document or in the User Manual do indeed function as expected.
- Stress testing: to test whether the implementation is robust enough to support intense use (e.g., manages large inputs) while requiring acceptable resources (i.e., memory and time).

For each test case, a brief English description should be provided to motivate it. Then, the test should be specified (as a test driver, a textual input to the program, a sequence of user clicks in a graphical interface, etc.). Finally, the expected results should be clearly stated, so that a tester can know whether the test passed or failed.

Unit testing

The plan for Unit testing should specify, for each key class and each method in that class, the set of actual tests that will be run. For example, for an algorithm that sorts the elements of a set on which there is an imposed total order, one might want the following test cases:

- An empty set.
- A set with only one element.
- A "typical set" not already ordered (since the elements of a set are inherently presented in some order to the algorithm).
- A "typical set" already ordered.
- A "typical set" in reverse order.

- A set with contiguous duplicate elements (again, this is meaningful assuming that the set is specified by presenting its elements in some array or linked list).
- A set with non-contiguous duplicate elements.

In most cases, it also makes sense to test a class or a method for *erroneous inputs*. For example: "What happens if the input to a method should be a tree with dynamically-allocated nodes connected through pointers, but the method is instead called on an input which is a DAG or, worse yet, contains a cycle?". While perhaps not all erroneous inputs need to be managed gracefully (in the previous example, it might be acceptable for the method not to terminate if the input contains a cycle), it is still very valuable for the test plan to document what happens for the various classes of erroneous inputs.

Since Unit testing is by necessity performed on a portion of the code, it usually requires a *test harness* or *test driver* to call the function or methods under test in a controlled way, provide the required input, and, if possible, automatically check the expected output.

Integration testing

The plan for Integration testing can be incremental, that is, one might want to test increasingly large subsets of interacting data structures and algorithms, starting from a few related ones, up to the entire program. This way, while the targeted code base for Integration testing becomes increasingly larger, the required test harness might actually get simpler, as the environment required to test certain portions of the code does not have to be specified in a test harness, but it is actually the code itself (this is certainly the case in the final integration testing, where the only task required of the harness is to provide input cases to the overall program and to receive the expected outputs from it).

One of the aspects that should be thoroughly tested during Integration testing is that the interfaces between interacting modules are correctly used, or, in other words, that each function or method is invoked by its callers using parameters that respect the required assumptions.

User-oriented testing

This part of the Test Plan can also be merged with the final portion of the plan for Integration testing, as they both address the test of the overall software. However, the two aim at different goals, as their names suggest. Integration tests, even when addressing the overall software product, still target potential errors that arise when multiple software functionalities or groups of functionalities interact, so it is oriented toward discovering internal implementation errors. User-oriented testing, instead, aims at exercising the overall software product to help ensure that the user can interact with the software as specified (ideally, as specified in the User Manual, which should be already available at this stage, at least in some rough form).

Of course, when one such test fails, it might still point out an internal error, just as an Integration test does, but it might also, instead, discover that a required feature was misunderstood or even completely overlooked and not implemented at all.

Stress testing

This part of the Test Plan addresses the scalability and robustness of your software with respect to input or problem size. Some Stress testing can be already performed as part of Unit testing. For example, going back to the sorting algorithm, one might want to perform Stress testing using the following test cases:

- A very large set that should still fit in main memory.
- An even larger set that will not fit in main memory.

The expected output behavior for Stress testing should not just list the expected output, but also a rough estimate of the requirements on the resources being stressed (time, memory, or both). For the sorting algorithm, for example, one should be able to verify that the worst-case runtime is, say, $O(n^2)$, and the way to do so experimentally is to provide a sequence of worst-case inputs of large size (for timing accuracy) over a range of sizes (to be able to observe the quadratic trend).

For memory, it might be best to add to your software various memory monitors that automatically collect the current memory consumption and can be used to record the maximum memory consumption, perhaps for each key data structure. In addition, one might want to use an operating system memory tracking capability to double check that the memory consumption reported by your program is indeed correct. It should then be possible to turn off these monitors at will, so that, when running in "production mode", your software is not slowed down by their tracking overhead.

How to submit the Test plan

All test inputs should be given a meaningful name and be referenced in the *Test plan* document, so that the actual test input files can be retrieved by the TA from each team's `svn` repository.