# Developing Test Plans

## Introduction

Even though testing is rarely as scientific and comprehensive as we would like it to be; it always pays to plan testing *as if* it were going to be very scientific and comprehensive. The reason is that it helps to put any given test case into the fullest picture of the system and the testing risk being managed.

Using the guidelines below can help create such a plan, *but it only works if you write it all down!* Make sure you create a Test Plan *document*. It will look more rigorous than the actual process you follow during testing, but the results will be better if you take a little extra time to formalize the planning.

> *For example: The IRS does not look at every line of each tax return equally and in order. They focus immediately on the parts they know will break first; namely, missing 1099 income, charitable contributions as % of income, miscellaneous itemized deductions, and underreported interest/dividend income. They also have more specific scripts for some sub-function; such as tips for food service and hospitality employees, and home deductions for self-employed. Most tax returns fail these central tests, and the IRS has not wasted a lot of time testing every line on each return.*

We need to take the same approach in testing. We need to focus earliest on the parts of the system that are most likely to fail.

Create your test plan at multiple levels:

## Setting Test Objectives

- What are the overall testing objectives? Know why you're testing! Are you trying to prove the system works? Are you trying to find the defects and get them corrected? The latter option is the right one. If you are trying to prove the system works, you'll under-design your test cases. If the goal is to find the defects, you should emphasize key test cases early in the test cycle.
- What are the biggest risks of failure if something is wrong with the system? Options typically include security/privacy, service levels, financial controls, and lost data. Make sure that these risks are addressed in test planning, and that the critical portions of the system that present each risk are high on the priority list.

## Prioritizing Functional Testing

Within the context of the results of system level objective setting:

- List the major functions and sub-functions of the system. Use *business* functions, not technical components to do this. The result will be that key system components receive a lot of testing simply because they impact, or are used by, so many business functions. The different *ways* they are used will drive differences in the more detailed test cases later.
- What portion of the system risk is embodied in each sub-function? Allocate testing resources according to these proportions so that some functions don't get over-tested while others go untested. Testing should be proportional to *risk*, not *size*.
- Do any sub-functions have any particular risks associated with them? Ask yourself: What happens if it doesn't work? When will we know? If immediately, then testing is simpler; however, if errors might not be evident until much later, make special plans to emphasize these items in testing early on. Do not leave latent defects in the system for your customers to find.
- Are there dependencies among sub-functions that require them to be implemented or executed in any particular order? Use these dependencies to identify critical control points that need to be tested early. Don't over test the earlier dependencies unless they carry significant risk.

A successful test *finds* a defect!

## Planning Test Scripts

Within the context of the prioritized functions and sub-functions:

- List the various events to which the system must be able to respond correctly. Most such events will be things in the real-world that cause a user to use the system interactively. Others will be the system's reaction to the passage of time (e.g. release orders at end-of-day, send accounting data at end-of-week, or produce reports at end-of-month) Each of these situations constitutes a business process to be tested.
- Develop an initial test script for each process, making note of each step and which components of the system are needed to conduct the process effectively. Always consider where during the script's various steps the actual risks are encountered. Low risk portions should receive less testing than high risk portions.
- For each process script, identify several operating scenarios for which the process should be tested. Scenarios are the *flavors* in testing. For example; for order entry, scenarios might include big and little orders, simple and complex orders, and orders for in-stock vs. out-of-stock items. Scenarios could be built for each permutation of these factors to effectively demonstrate the system.
- Prioritize the scenarios according to their likelihood of encountering a defect. For example, you should test the big-complex-out-of-stock scenario before the little-simple-in-stock scenario. The latter scenario is actually a subset of the former, and some people would suggest testing it first. However, chances are that the earlier unit testing uncovered many of these related defects already, and time will have been wasted uncovering minor problems with the *big* problems still hidden.
- For each scenario, analyze the intended processing flow to identify all of the different test conditions that must be exercised. In the extreme, you should make sure that all of the conditional and operational logic of the system gets exercised. Realistically, you are trying to identify the key differences that might result in noticeable differences in defect patterns (e.g., do not test all U.S. addresses and find out too late that international addresses do not work).
- For each test condition, develop from one to several test cases that, when used, will cause the test script to meet its objectives. With good planning, most test conditions across many test scripts and scenarios can share test data, but it takes planning and coordination.

## Test Plan Structure & Execution

The test plan that results from this thought process will have the following structure:

- Prioritized system-level objectives and risks
  - Prioritized functional and sub-functional breakdown
    - Prioritized scripts for business processes
      - Prioritized scenarios within each script
        - Prioritized conditions within each scenario
          - Test cases and expected results for each condition

The key to execution is proper prioritization…..

**Always plan as though you only have an hour to test.**

**What would you test if you had to go into production with only one more hour of testing?**

**What is the key latent defect you would try to uncover?**

**After you test for *that* defect, repeat the process for another single hour…**

The *one-more-hour* approach prevents exhaustive testing of minor features. Your goal is to uncover defects, not exhaustively exercise the system.

***Focus on finding errors, not proving there are none!***