



Your Short Cut to Knowledge


UAT Defined

A Guide to Practical User Acceptance Testing

Rob Cimperman

CONTENTS AT A GLANCE

Chapter 1	Introduction	3
Chapter 2	Defining UAT— What It Is... and What It Is Not	6
Chapter 3	Test Planning— Setting the Stage for UAT Success	32
Chapter 4	Building the Team— Transforming Users into Testers	60
Chapter 5	Executing UAT— Tracking and Reporting ...	77
Chapter 6	Mitigating Risk— Your Primary Responsibility	103



Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this work, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this work, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

Visit us on the Web: www.awprofessional.com

Copyright © 2007 Pearson Education, Inc.

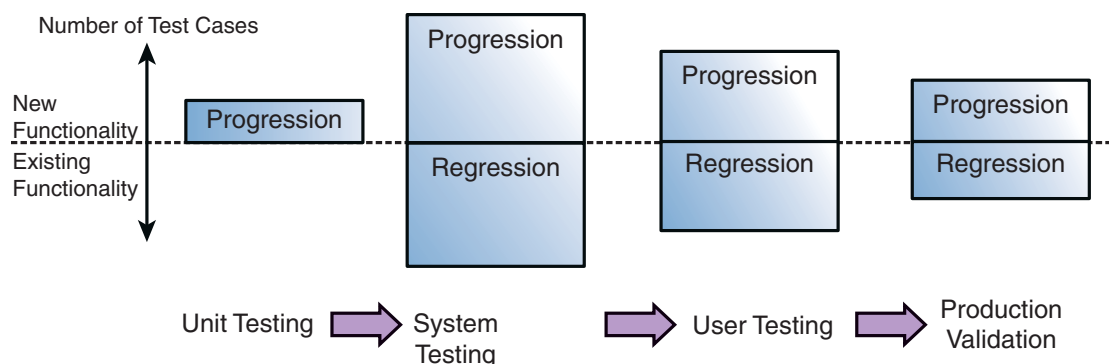
All rights reserved. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, write to:

Pearson Education, Inc.
Rights and Contracts Department
75 Arlington St., Suite 300
Boston, MA 02116
Fax: (617) 848-7047
ISBN: 0-321-49814-3
First release: November, 2006

cases looks like that shown in Figure 3-1. Notice that both system testing and UAT are performing a large number of regression tests. This is because the business users are implicitly saying that they do not trust the system tester's to validate that untouched functionality still works as it should. Also notice that there is a large number of production validation tests being performed. Again, this is because there is a low confidence by the business users that the IT team can move code that worked fine in UAT to production without breaking it.

FIGURE 3-1

Typical distribution of test cases across test phases with low confidence in system testing.

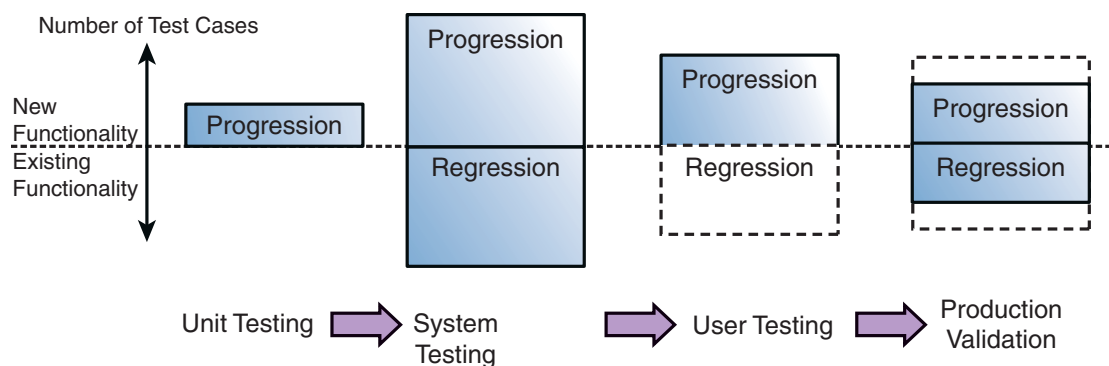


The way to minimize the size of the UAT testing phase is to restore confidence in the system testing process so the business users feel comfortable in validating only the new functionality. The first step to rebuilding business user confidence in IT testing is to open the communication channels between UAT testing and system testing. This requires getting the business subject matter experts involved much earlier, during the system test planning, to validate and assist in the development of the system test cases. The reciprocal information flow is beneficial in that system testers can learn a great deal about how to better represent the users' interest by seeing the test cases

planned and executed during UAT. At the same time, business testers can learn about the limitations of system testing to improve the coverage of the UAT tests to fill in the gaps.

Figure 3-2 shows the distribution of test cases in an environment where business users have high confidence in the system test capability of the IT team. This is described as the ideal situation because instead of wasting the time of functional users to double-check the regression tests or to validate that all code made it to production, those people are doing their *real* jobs (that is, making money for the company).

FIGURE 3-2
Ideal distribution of test cases across test phases with high confidence in system testing.



After the user community delegates responsibility for comprehensive testing to the IT test team, the question remains as to what is an appropriate quantity of test cases to have users perform. This can only be answered in context, but generally, the users should perform at least one positive and one negative end-to-end test case of each major type of task they perform.

- ▶ **The positive test case** has the goal of having the transaction move as smoothly and error free as possible.
- ▶ **The negative test case** has the goal of testing edits, validations, error codes, rejections, and failures of that transaction.

These should be sufficient to validate a release when it becomes the final step of user representation throughout the software development process.

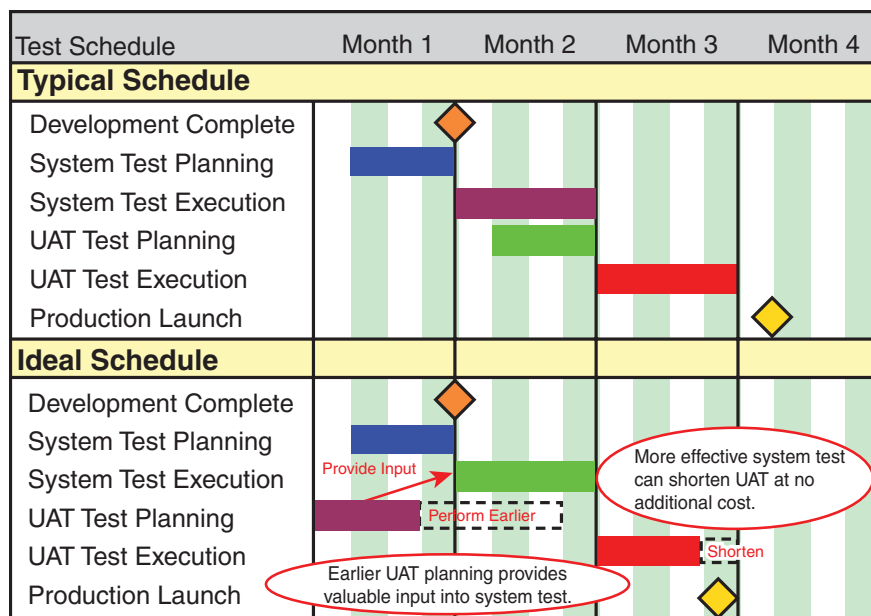
Common project management practices would put a start date for the UAT test planning activities a couple weeks before the UAT execution begins. Even if that is sufficient time to plan your test cases, it means you will be planning your test cases while the IT system test effort is wrapping up.

Instead, consider reversing the steps; plan UAT cases *before* the IT team performs its system test cases. As soon as you have planned your test cases, share those planned cases with the system test team. You may run into some office politics here, but imagine how much smoother your work will go if someone else has already validated your tests cases and worked directly with the IT team to resolve any issues you encounter.

Figure 3-3 represents the typical and the ideal UAT schedules. The red arrow represents contributions to a subsequent task in terms of resources, data, or other materials. Also note that the UAT test planning phase occurs earlier in the ideal schedule and, consequently, the UAT test execution phase is shortened in the ideal schedule.

Test Planning—Setting the Stage for UAT Success

FIGURE 3-3
Typical versus ideal schedule.



Notice in the preceding sample schedules that by planning the UAT test cases earlier, the system testing team can get a better understanding of what test cases best represent the most important real-world scenarios instead of choosing them arbitrarily. The other result is that the actual UAT execution period is shown to be shorter. That is simply because most issues that the UAT testers typically uncover have already been resolved by the development team during system testing. This is clearly a win-win situation: System testing is more effective, defects are caught sooner, UAT is shorter and more effective, and even the project manager will be glad that tasks are done ahead of schedule. Are you sold on this idea yet?

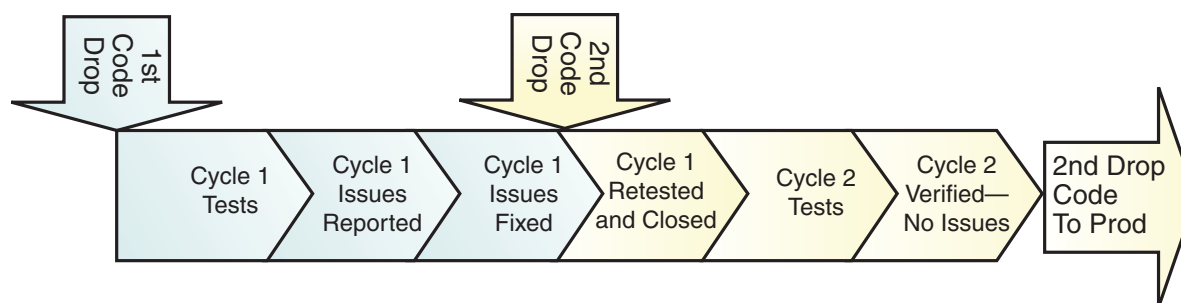
Test Cycles

Another important concept related to test planning is that of test cycles. Generally, you should plan for two cycles of testing. This means that you plan to execute each test case two times before signing off on the release. Why? Think about why you do regression testing in the first place: because code gets broken accidentally when new functionality is delivered. The same thing happens any time that code is changed, including when code changes are made to fix the issues uncovered during your testing, sometimes more so when the IT team is rushed.

As shown in Figure 3-4, cycle 1 completes when the fixes for issues uncovered during cycle 1 testing are delivered, retested, and verified. Cycle 2 begins when the testers believe that all issues have been resolved, so a clean test cycle should be expected. If there are no further issues uncovered that require a new code delivery, the final code is considered certified and can be moved to production. However, if additional faults are found that require new code to be delivered, it is not uncommon to require a third or even fourth test cycle.

FIGURE 3-4

No code should move from a test environment to production without an issue-free test cycle.



As you can imagine, such additional passes through the test cases play havoc with a testing schedule and the resources required to support testing. From a practical point of view, any cycle beyond

the second cycle most likely is limited to the areas of functionality that would reasonably be at risk of breakage as a result of the last round of issue fixes, in addition to a very limited set of core functionality. To define core functionality that should be included in every test cycle, think about what it is that the testing team would get in the most *trouble* for if it was later found to be broken in the production environment.

An example will help clarify this way of looking at test cycles:

1. Cycle 1 uncovers 25 issues scattered across the application, affecting install orders, change orders, and disconnect order types.
2. All 25 fixes are delivered and verified, allowing cycle 1 to complete.
3. Cycle 2 begins and finds that one new issue arose in the order cancellation functionality due to breakage from one of the 25 previous fixes.
4. The IT team gets the trouble ticket for the issue. It delivers the fix for the latest issue. You verify that cancellations now work, so you close out cycle 2.
5. Cycle 3 begins with a full test of all cases related to cancellation orders (every possible type of order is canceled, for example) as well as other typical order types to validate the very small chance that the single code fix broke something else. In this example, there should be a much smaller chance that anything else broke with the latest minor fix.

A practical suggestion is that you ask the developer who made a code fix what she would recommend that you look at to verify that the code fix did not break any other functionality. If anybody knows about the behind-the-scenes code dependencies, it is the developer. Unfortunately, most testers forget to ask the expert or are prevented to do so by hierarchical organizational boundaries.