14. The approach to integration has a significant impact. It must be carefully planned and evaluated.
15. A suggested integration test approach is to:
    - Select *small* functionally related groups of modules.
    - Get a basic or simple transaction running.
    - Stress-test the integrated modules to confirm a solid test platform.
16. Systems testing execution begins when a minimal skeleton is integrated.
17. Systems testing ends when we feel confident that the system will pass the acceptance test.
18. Systems testing depends on good planning
    - Requirements-based tests
    - Performance capability tests
    - Specialized tests (volume, documentation, etc.)
    - Design and implementation-based tests
19. Systems testing must produce a test data set for subsequent modification testing as a product.
20. Acceptance testing serves to *demonstrate* that a system is ready for operational use.
21. Acceptance testing must be planned and formal.
22. A vital part of the Master Test Plan is the selection and effective use of test tools for all phases of testing in the large.
23. Major tool categories of importance to testing in the large include
    - Instrumenters
    - Comparators
    - Test data generators
    - Simulators and test beds
    - Debugging aids
    - System charting and documentation aids
    - Capture playback systems
    - File aids
    - Test case management systems
24. The total cost of a software tool is many times the license fee or purchase price.

Chapter 9

Testing Software Changes

Testing Changes—An Introduction

In most organizations the majority of time is not spent in new development, but in making changes to existing systems. We know that testing such maintenance changes properly is fundamental if ongoing software quality is to be maintained.

Testing changes is also important to new development. We can seldom enjoy the luxury of testing a static product. Planning for testing in the large has to recognize that testing is performed *while* the system is changing. Fixes and corrections are applied as we move along, and the testing of changes and system retesting is a major concern.

Thus, in both the maintenance and the new development environments, special attention must be given to testing changes. Given that some changes are made, the basic question we must answer is, what has to be retested to provide proper confidence that the system will continue to perform as intended? What amount of testing is required? How should the testing be structured and who should perform it? Despite the importance of these questions, most organizations have given them little study. If testing itself is improvised and poorly structured, testing software changes hasn't even been considered!

Any change, no matter how trivial, involves its own minidevelopment cycle. First, we have to design the changes we are going to make. This requires

understanding the system or program logic; understanding the intended new result, and coming up with the changes that have to be made to achieve it. Second, we have to build the changes. This involves actually changing the code and modifying the system to conform to the new design. Finally, we have to implement the changes and install a new system version. This involves replacing the changed modules in the production library, ensuring that operators and users are familiar with any new impacts, updating documentation, and so on.

All these activities must be tested, just as every step of the normal development cycle is tested. The fact that the change might involve only a few statements does not take away the need to test the design (the means of achieving the change), test the changes in the small (the individual modules changed), and test the changes in the large (any overall systems impact). Experience has shown that most such changes are not tested sufficiently.

One set of studies found that the probability of making an incorrect change (one that introduces a new problem that subsequently has to be fixed or corrected) is little more than 50 percent—even for very small numbers (fewer than ten) of statements changed! Much of this is due to overconfidence and ineffective or nonexistent software change testing. We change just a couple of statements and *believe* we have not affected anything adversely. We execute one case that tests the path that was changed and tell ourselves that the change has been tested. Almost no design testing and very little testing in the large is performed. Is it, then, any wonder that we experience so many problems?

Developing careful design-based tests for even the tiny changes is more important than ever because original design concepts and subtleties have been forgotten. Systems testing is necessary to ensure that local changes do not have unanticipated side effects and to confirm that old faults are not causing any new problems. Quite simply, what we need is a mini (and sometimes not so mini) testing life cycle for each software change that is made! We may feel this is difficult and "too expensive" to justify, but that is what the testing practitioner must work to achieve. In this chapter I talk about techniques for achieving this goal and how the extra costs can be controlled or avoided.

Reviewing Change Designs

The most fertile area for improving software change quality is better design-based testing. This condition exists because, sadly, any form of design review on small changes is *nonexistent* in most organizations. Any steps toward more formal, or even more careful, analysis of planned changes is sure to pay off greatly.

All of the material on testing designs presented in Chapter 6 is applicable, but is probably an overkill. We don't need—and could not justify—a full design

review every time we are about to make changes to a system. We do need—and *must* find a way to provide—an effective measurement of proposed changes to give ourselves reasonable confidence that they don't mess up something they shouldn't have affected and really do achieve the desired change in a reasonable manner.

The critical questions for change design review become the following:

1. *Does the design for the change provide for all new features and desired requirements?*
   - Has anything been left out?
   - Has the change been applied to every affected part of the system?
2. *Does the design for the change ensure that there are no adverse side effects?*
   - Can anything else be affected?
   - How could this change create a problem?

We are still concerned with whether the design alternative is the best that might have been selected, but usually that is not nearly as important as it is in the early phases of new development. The key questions are the two listed above. Tracking the reason that changes fail shows two common change problems occurring over and over again. The first is unforeseen side affects. The change accomplishes what it was supposed to, but also affects something that was working before. The second problem is partial change completion—a change is applied to most parts of a system, but one or more parts are overlooked. It takes little design testing effort to detect the presence of either problem. The approach I recommend is the following:

Change Reviews

1. Description of change—have the person who is responsible for the change write down a description of it.
2. Modules affected—have the written description include how the programs that are affected were determined and which they are. Sort out:
   - modules requiring change
   - modules that are unchanged, but could be affected.
3. Change test plan—identify how the modification is to be tested.
4. Review the change description and change test plan (either by passing it on to an individual who can read it and sign off or by bringing it before a Change Review Team)

The key steps involve writing down what is being changed and then having the result reviewed by someone else. The time spent need not be extensive; it is the principle involved that makes the difference. Our old friends, planning (in this case a few minutes of thought) and independent evaluation, can once again bring about dramatic testing improvement.

We have found that evaluating changes works best when performed in groups or blocks of related changes. If the environment is such that you can group the proposed changes together, then the best strategy is a formal, simultaneous review by the entire group. Three or four qualified staff (including the end user) should be brought together and all change designs for a single system reviewed by the group.

All of the discussion of formal review techniques in Chapter 4 applies. If the change cannot be grouped easily, passing it by a single *qualified* reviewer works almost as well. In either case, the person preparing the change knows that some review will occur and is forced to spend a few minutes organizing his or her thinking and writing down what is to be changed and how it will be tested. That provides an effective feedback process and greatly reduces the number of errors that are made.

Preparing the Changes

Assuming we have a solid design for the change, the second step is to actually code the changes and check them out to be sure that they are ready to be installed. We must test both in the small (each program that is changed) and in the large (at the system level).

Let us assume a work environment that provides on-line terminal support for programmers and three independent working libraries (disk data sets) in addition to the production library, which contains the current production version of each system. The three work libraries might be named the personal library (the programmer's own work space), the integration library (where tested modules are brought together), and the acceptance library (which contains an exact copy of the production version of the system and is used for system level testing).

Developing changes and testing them effectively involves seven steps as follows:

Testing the Changes

1. *Move module copies to personal work space*—The programmer copies modules that need to be changed from the acceptance library into the personal library.
2. *Code changes made*—The modules are changed as required by the change design. Changes are applied to the module copies in the personal library.
3. *Testing in the small*—Testing in the small is carried out in the programmer's personal library. Whenever possible, this should involve executing existing cases or the test data sets that were used to develop the program.
4. *Move tested modules to the integration library*—Once satisfied that the modules have been changed as required, the programmer requests approval to move them into the integration library. This should require project leader/librarian approval and occur only after the change has been reviewed. Only the project or development leader has access to the integration library.
5. *Integration testing (if applicable)*—If this is a development effort with a separate stage of integration testing, the changed modules are reintegrated into the system and any specific integration tests are executed. In a maintenance environment this step will normally not occur.
6. *Testing in the large*—The changes are tested in the large by temporarily moving the modules from the integration library into the acceptance library. (No permanent change to this library occurs unless the production library is changed.) Test data sets are used to exercise the entire system or selected features as defined by the Change Test Plan. (Any defects found produce formal problem reports.)
7. *Installation*—Once all testing in the large is completed and sign-off for change installation is obtained, the new modules are moved into the production and acceptance libraries and deleted from the integration library. A copy of the production library before it is changed is maintained in the retired library in case the old production version has to be restored.

This methodology ensures that proper change control is maintained. Without strong controls over the code and documentation being changed, testing the changes properly becomes hopeless. It does not deal with the special situation of emergency maintenance, when a fix must be applied to the production

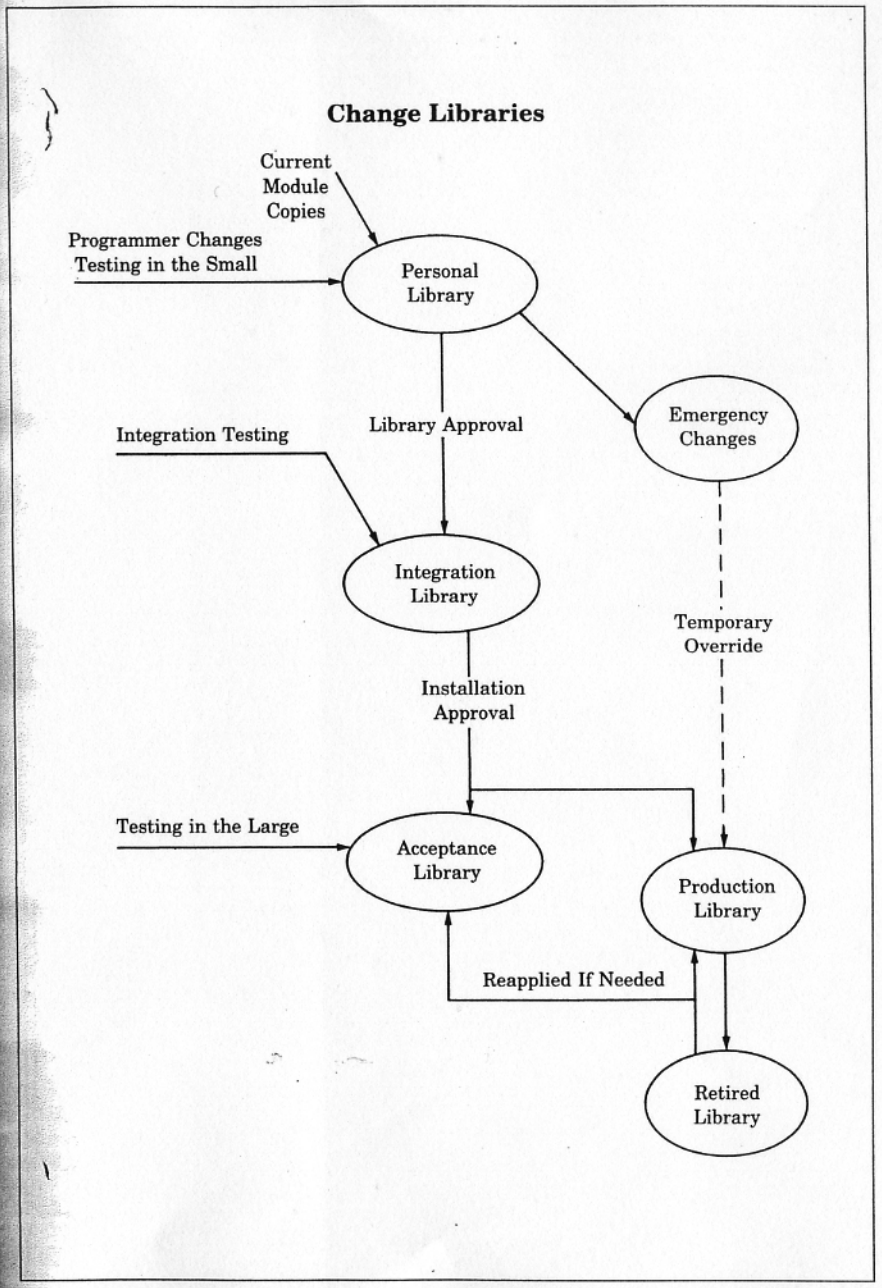
system as soon as possible. With emergency maintenance our primary goal must be to restore system operation—thorough change testing has to be deferred a little as a practical expedient.

Testing Emergency Maintenance

1. *Fix developed and tested in personal library*—As before, the proposed change is prepared and tested in the small in the personal library.
2. *Fix tested against acceptance library*—The fix is temporarily applied to the acceptance library (remember, the programmer has no way to change the library permanently) and *limited* testing in the large is performed. If a test data set is available it should be run.
3. *Fix applied to production on emergency basis*—Operations inserts the fix by operating from an auxiliary library.
4. *Testing the completed change the following day*—The next day the changed module is moved into the integration library and the change testing methodology is followed. Once it has passed acceptance testing and been moved into production, the auxiliary library may be deleted.

An auxiliary library is required for emergency maintenance to avoid changing the "real" production system without proper testing and sign-off. Operations continue using a JCL (Job Control Language) override to the auxiliary library until this has been done. If a fix is allowed to be applied directly to production, the acceptance and production libraries are no longer exact duplicates, and problems may occur with the retired library as well. This gets compounded and creates major change control problems when multiple emergency maintenance is going on at the same time.

The Change Libraries diagram on page 157 shows how the libraries described above may be used to control the testing of changes. Other approaches to organizing the change testing will work equally well. What is important is to have a *standard* procedure to ensure that effective testing is completed *before* any changes are moved into production.

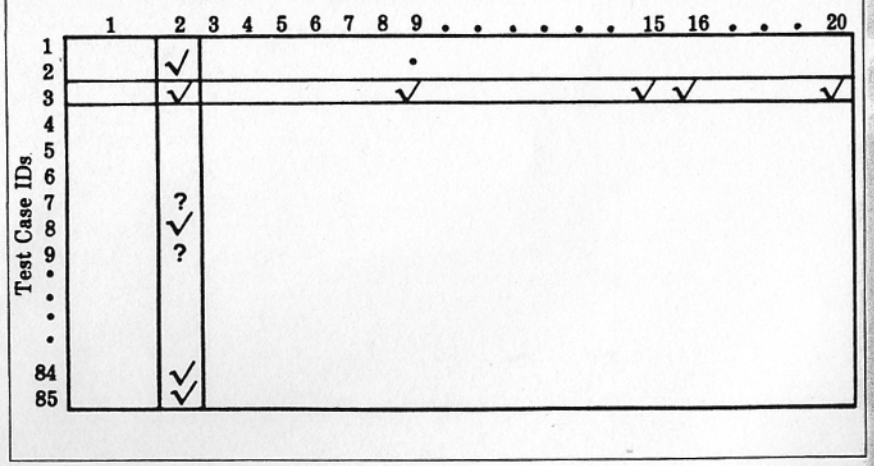


Testing the Changes in the Large

After changes have been tested in the programmers' personal libraries and released for testing in the large, the decision must be made as to how much systems and regression testing is required. If possible, the entire test data set should be rerun. If this is viewed as impractical or too expensive, the Retest Planning Matrix is one tool that may be useful for planning the change testing needed. I recommend preparing this matrix as a routine part of the testing in the large planning. Once completed, it is useful for guiding retesting during development and for determining the testing required for later maintenance changes.

Looking at the sample retest planning matrix, you see that the matrix relates test cases (folios, test procedures, or just individual cases) to system programs and modules. A check entry indicates that the case is to be retested when the module is changed. A question mark signals that retesting will be required in some cases and not in others—that is, we must examine the change to determine whether the test is necessary. No entry means that the test is not required. In the example, we have twenty modules and eighty-five test procedures. The completed row tells us that case number 3 is to be rerun whenever modules 2, 9, 15, 16, or 20 are changed. The completed column tells us that if we make a change in module 2, we have to rerun test cases 2, 3, 8, 84, and 85 and possibly tests 7 and 9.

Sample Retest Planning Matrix



Completing a retest planning matrix during initial development takes very little time. The planning matrix may be reviewed by system users and will help show the level of retesting expected whenever changes are made. Only minor continuing effort is required to keep the matrix current. Whenever modules are changed, the matrix should be reviewed to assure that it still indicates an appropriate retesting plant. If new cases or modules are added, then new rows or columns are simply added to the existing matrix. The matrix may also be generated automatically by instrumenting each module so as to invoke a utility that indicates which modules are executed by which test procedures.

The retest planning matrix helps us to *decide* what should be retested. The data needed to execute the selected cases will be available *if* we have maintained it since the initial development of the system. If not, the cost of recreating it quickly becomes prohibitive. It is impractical to have to start from scratch and prepare tests for small changes. Constructing new test data every time a change is made takes too long. To test changes effectively we *must save the test data for reuse*!

One Critical Key to Testing Changes Effectively

Save and maintain test data for reuse whenever changes are applied.

It should be emphasized that running the complete system test set (full regression run of all test cases) is necessary in high-risk situations to ensure that all functions and system capabilities continue to perform as required after a change has been made. Even assuming a "perfect" change, it is possible for systems to fail because of "old" faults already existing in the system. Short of retesting all functions, there is no way to prevent such faults from causing a major failure or unacceptable system behavior!

When the cost or time required for full regression is prohibitive, it is helpful to group or "batch" the changes to be tested. Testing in the large for a group of related changes takes no more effort than testing any one individually. Batching fixes and corrections minimizes retesting effort and justifies more comprehensive testing. Maintenance organized so that changes are batched and only applied at regular intervals is called *scheduled maintenance*. Schedules are established for each system (usually quarterly or semiannually). Changes are held and applied all at once at the designated interval. This permits a single design review of all the proposed changes, avoids having to change the same code multiple times, and allows a thorough systems and acceptance test to be carried out before the changes are made effective. It greatly improves change quality and saves a lot of money at the same time. Of course, if too many changes are grouped, then debugging becomes complicated by added difficulty in quickly isolating the source of the problem.

A Second Key to Testing Changes Effectively

Batch changes together for testing whenever possible!

Testing Installation Readiness

The final step in testing software changes involves accepting the change and being certain that the installation is ready for it. Collecting failure data shows that this is another highly defect-prone area. If we have tested the change design and code properly, we will have done the bulk of the work and satisfied ourselves that the change does what it is supposed to and will not have adverse side effects. Now we must determine that we are prepared operationally.

Installation Readiness Testing

1. *Completion of installation checklist*—A checklist form should be available to ensure that all required parties are cognizant of *any* planned changes. The form should describe the change to be made and specifically describe any functional or operational impact.
2. *Approval signatures*—The checklist form is routed to designated persons who are responsible for approving changes. Documentation completeness is verified, operations signs off that all operational procedures have been readied, and the user signs off that all user training is complete. Quality assurance should spot-check these steps and audit to ensure that the procedures are followed thoroughly.
3. *Change is installed*—The approved change is entered in the production system and announced to affected persons.

The key element is control over the process and consistent change testing and retesting procedures to make sure that every base is covered. It is easy for a correct change to cause a significant failure because people did not know about it or were improperly prepared for it. Effective testing must prevent this from happening.

SUMMARY

1. Changes must be designed, built, and implemented, and testing each step is necessary to have proper confidence that the change will work as expected.
2. Careful review of change design is crucial.
   - Is the design complete?
   - Does it upset anything that is working?
   - How will it be tested?
3. Testing each module in the small is crucial.
   - Every module changed must be carefully tested.
   - Techniques are the same as testing newly coded modules.
4. Testing in the large is crucial.
   - Careful planning is required to determine the tests needed.
   - A Retest Planning Matrix helps coordinate this planning.
   - Full regression retesting should be used whenever possible.
5. Control over any changes to the system is essential.
   - Changes applied to personal libraries
   - Moved to integration libraries when ready
   - Tested against production duplicate (acceptance library)
   - No changes occur to production until tested properly
   - Emergency maintenance done with auxiliary library
6. Maintain and test data sets and use them for retesting.
7. Batch changes together for testing and review.
8. Test installation readiness thoroughly before making the change effective.