



**Testing Software Rewrites
And
Re-Engineering**

**© 2003 Richard Bender
Bender RBT Inc.
17 Cardinale Lane
Queensbury, NY 12804
518-743-8755
rbender@BenderRBT.com**

There are two types of projects undertaken in replacing an existing system - rewrites and re-engineering. In a rewrite the new system has the same business rules; only the design changes. In a re-engineering project the business rules change in addition to the design. Actually some rules stay the same and others are revised to reflect improvements in the way of doing business.

Replacing large, complex, old existing systems is the riskiest project type. The failure rate is by far much higher than any other type of project. I have seen estimates in the 70% to 90% failure range. The primary issue is ensuring functional compatibility with the old system, even though the new design is totally different.

The old system could be batch and the new one on-line; the old flat files and the new on an RDBMS; the old centralized and the new distributed. The most common occurrence of this today is the migration of systems to the web. Yet, in a rewrite the business rules have to be the same. Unfortunately, user manuals, training material, and system documentation are never detailed enough and accurate enough to serve as the sole source for the current rules. What is in people's heads is also nowhere near complete. The combination of turnover, time, and imperfect memories ensures that. The only detailed definition of the business rules for most old systems is the source code.

Even in a re-engineering project, the rules for many functions have to be the same. Where they do change, they should change in a well defined manner. You should be able to tell existing users that the old rule was X and now it is Y. That implies that you knew what X was or even that you knew a rule X existed at all. Users get very annoyed when the new system does not act as they expect it to. This is especially true if there is a loss of functionality or the new function is flat out wrong after working correctly for years in the old system. This is not any user's definition of progress.

Testing Rewrites

Let us first discuss how to test a rewrite where the old and new system must be fully functionally compatible. Our recommended approach is to take your best shot at identifying what business functions you think are in the existing system from the available documentation and what is in people's heads. This creates a logical table of contents for the External Specification. You gather up all of the documentation and map it into the frame work you have created. This gives you your best guess as to what is currently in the system.

Once you have a first cut External Specification you design tests from these specifications. You run the tests against the existing code and do two things. You verify that you got the answer you expected and that you covered all of the statements and branches in the code (you use code coverage analyzers to do this). If the results do not match you determine which is correct – the code or the specification and make changes accordingly.

For the code that was not executed by the first round of tests you need to determine if the code represents application rules which must be understood or just design dependent code that will no longer be germane. For the code that represents application rules you need to extract those rules from the code via reverse engineering. The details of how this is done are beyond the scope of this paper. Needless to say it is a laborious process. The average productivity rate of the projects we have been on is about 100 lines of executable code per person day from start to final External Specification. This is actually a lot less time than normal analysis and has the added benefit of being very accurate.

You keep looping through this process of specifications, test cases, and test results until you have a set of specifications and test cases that are in sync with the current code. (See Figure 1)

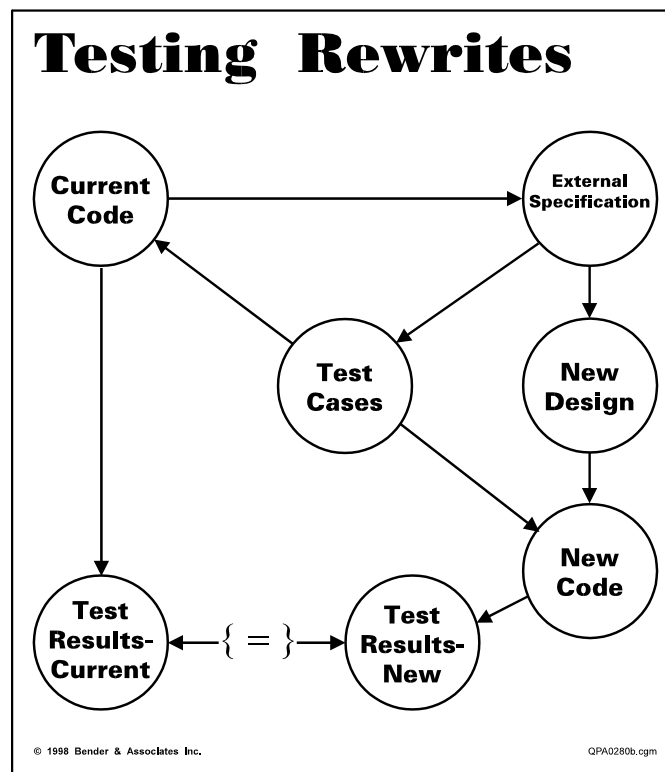


Figure 1

These tests are also walked through the new design. They are then run against the new code. This usually means implementing the test cases twice since the physical tests require that the inputs and data files be in the formats expected by their respective systems. However, the tests are still functionally equivalent. Once the tests have been run, the results from the old and new systems are compared. This compare process requires the use of data conversion utilities, etc., to do the mapping, of course, since again the data structures are different. Still, if the two systems are truly functionally compatible, the results will be identical. In running the tests you also turn on the code

coverage monitor for the new system. You should execute everything except for some design dependent functions. If more than that is unexecuted, you may have added some functions. Otherwise, you now know that the two systems are logically identical.

I should note at this point that most descriptions of the approach to rewrites tell you explicitly not to look at the existing code. The objective is to prevent you from being "tainted" by how things are done today. From a design standpoint this is easily overcome by having a concurrent architecture effort going on populated by staff not going through the existing code. From a business rules standpoint, not looking at the existing code tremendously increases the project risks and the failure rates. Look at the analogy in data modeling. How could you do a data model without looking at the existing files? How could you do data conversion if you did not know the details of how the current data structures are designed? Somehow people think it is OK to look at existing data, but not function. The reality is you must look at the existing function and it is embodied in the code.

Testing Re-Engineering

Testing re-engineering projects poses a major challenge. The old and new systems are the same only different. You still go through the same process as before to ensure that you fully understand what you have today. If a business rule is to be the same, it will be the same. If it is to change, it will be changed on purpose - not by accident. You will also know how the rule is different.

This is critical to the testing. Since the old and new systems are no longer fully compatible, comparing the test results takes a lot more planning. You still go through essentially the same testing process as for rewrites (refer to Figure 2 "Testing Re-Engineering"). However, the planning for the comparison of the results is much more intricate. Where they are the same, the compare is a simple yes/no as to whether you got the same answer in both. Where they are supposed to be different, you need to identify what the expected results are for each version and check them individually.

The complication sets in because in running a test or set of tests, there is rarely any way to cleanly organize the test libraries into tests whose results should be the same versus tests that will be different. A given test will invoke some functions with the same rules and others that have changed and still others that are totally new.

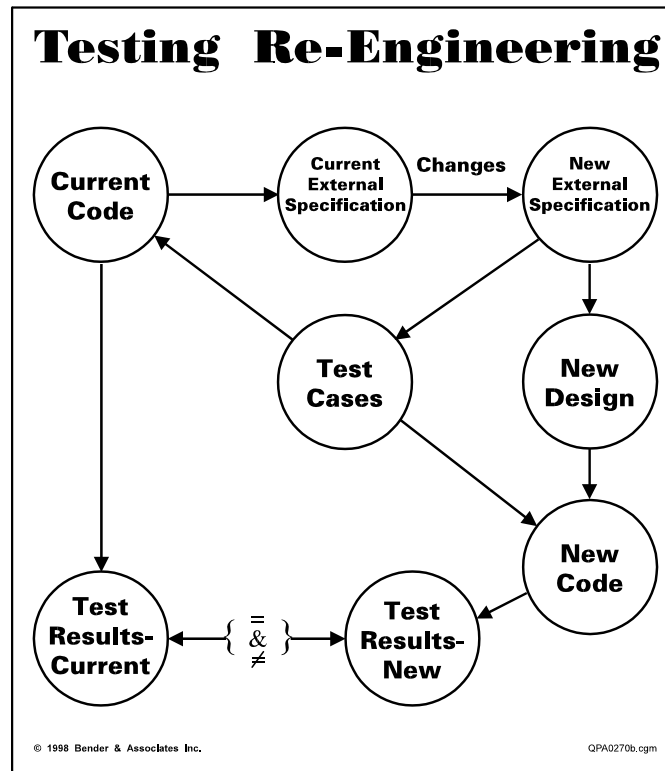


Figure 2

I always strongly recommend against transitioning via a re-engineering project directly from the existing system. Release 1 of the new system should be a “vanilla” (i.e., functionally identical) replacement for the old one. Release 2 then contains all of the new functionality. Using this sequence, it is much easier to validate that the existing business rules that you still want are all there and accurate. When project teams go into “dare to be great” mode and do this in one stride, they put the business at risk. The issue is that the value to the business of all of the existing functions that are to be retained is put at risk for the incremental business value of the enhancements. I have never seen a project where this equation favored the new functions.

If you cannot create a “vanilla” replacement Release 1 – this is generally due to political considerations – then you should create a “vanilla” replacement Release 0. This release never goes into production, but is fully verified to be functionally compatible with the existing system. You then add the enhancements to safely deliver Release 1. This two step approach is actually much faster than trying to do this in one great leap.