Testing Software Re-writes
And
Re-design/re-write

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

You have an existing system that has reached critical mass – i.e. the point at which every time you fix one defect, two more are created. There is also an ever growing backlog of service requests that the current architecture cannot accommodate. It is time to replace the system. You are about to embark on the riskiest type of project in the world of software. The failure rate for such projects is between 70% and 90%. In some cases the project is just killed off mid stream with nothing being delivered. In many cases something gets into production but significant problems occur for not just days, weeks, or months, but for about two years. The single biggest issue is ensuring functional compatibility. The existing system represents the cumulative experience and wisdom of years, even decades. However, this knowledge is really only fully defined in the code itself. The existing documentation is more of a verbal tradition and communicated by rumor.

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

Even in a re-design/re-write 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 Re-writes

Let us first discuss how to test a pure re-write 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 (i.e. the detailed Requirements Specification). You gather up all of the documentation and map it into the framework 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 legacy code with two mandatory goals: ensuring both accuracy and comprehensiveness.

You verify accuracy by recording the results expected from the specification. For the cases where the results do not match, you determine which is correct – the code or the specification – and make changes accordingly to both the legacy system and the tests.

You verify comprehensiveness by ensuring that your tests covered all of the statements and branches in the code (provided by code coverage analyzers). For the code that was not executed by the first round of tests you need to determine if the code represents business rules which must be understood and carried forward, or just implementation dependent code that will no longer be germane with the new architecture and can be safely ignored. For the code that represents business rules, you need to extract those rules from the code via reverse engineering.

The details of business rule extraction are beyond the scope of this paper. However, the average productivity rate on the projects my staff and I have assisted clients in performing this task is about 100 lines of executable code per person day from start to final External Specification. This rate is independent of the programming language. For a rough order of magnitude calculation on a given system of 1 million lines of code, let us assume that half the code is not executable and also assume that the initial tests covered half of the executable code, leaving 250,000 lines. If half of that is implementation dependent code, we still have to extract business rules from 125,000 lines. From the rule above, that corresponds to 1,250 days, or 5 person years. On the surface that might appear to be an untenable task. However, that is actually a faster productivity rate than the effort it took to create the original requirements from which the code was developed. You just need sufficient staff to ensure this task gets done in a timely manner.

You keep looping through the 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 – Test a Pure Rewrite). What you have done is created the baseline specification of the existing functionality with a matching full regression test library. The irony is that if the application team had kept the documentation and tests up to date all along you would not even have to do this task. You are paying for the past sins of all the prior projects that felt they did not have time to do this.

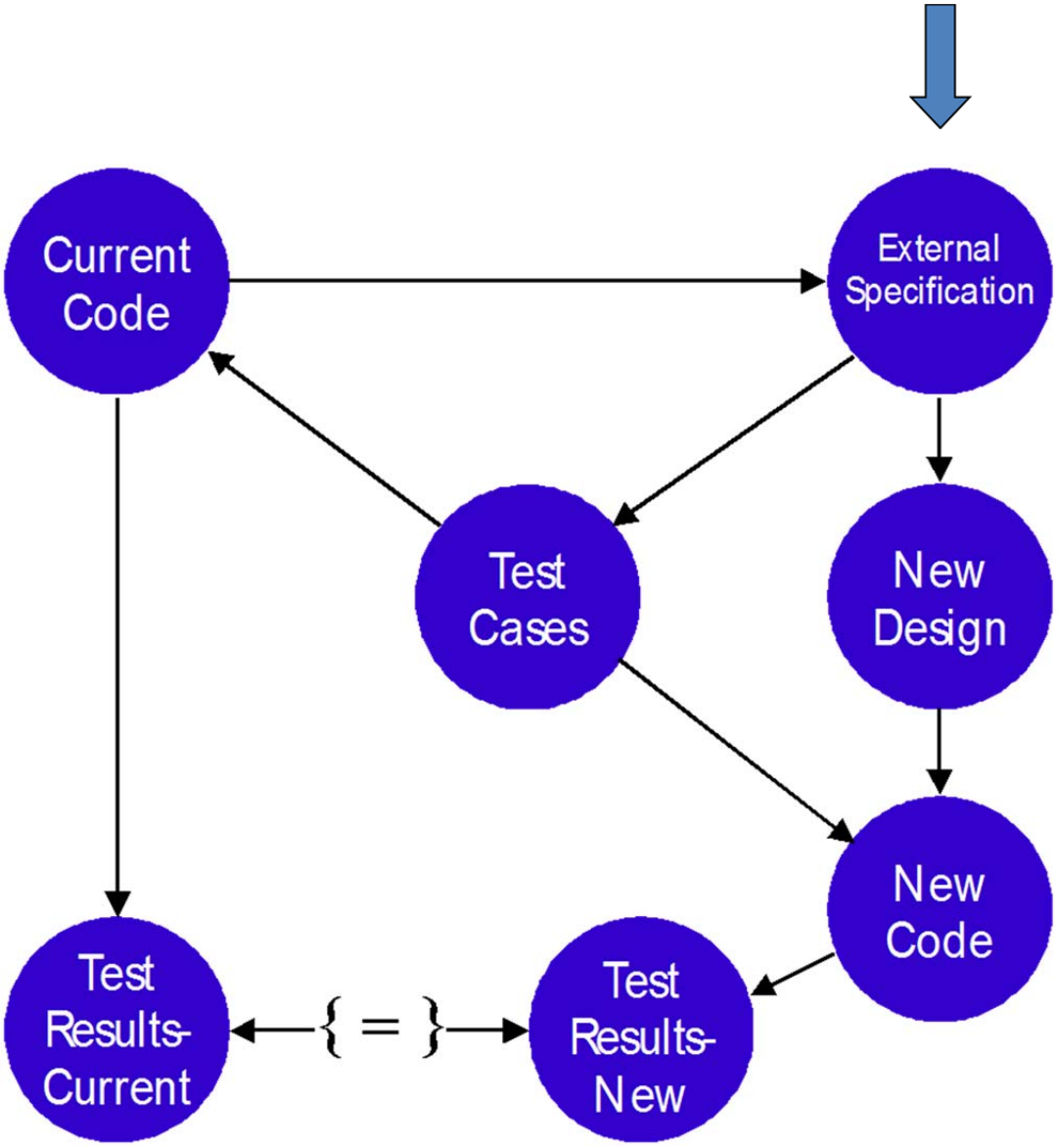

Figure 1 – Testing a Pure Rewrite

The tests designed for the current system are modified appropriately for the new implementation, and then run against it. This usually means creating 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 indeed functionally compatible, the functional results will be identical.

In running the tests you also turn on the code coverage monitor for the new system. You should have executed everything except for some implementation dependent functions. If more than that is still unexecuted, you may have added some functions that require separate testing. Otherwise, you now know that the two systems are logically identical.

Note that many descriptions of the recommended approach to re-writes 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 an implementation standpoint this “taint” 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, however, 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-design/Re-write

Testing re-design/re-write projects poses a major challenge. The old and new systems are similar but 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 in the replacement system. If it is to change, it will be changed on purpose – not by accident – and you will know what the new result should be. Similarly, for new business rules to be added, you must know what the new results should be.

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 pure re-writes (refer to Figure 2 – Testing a Re-design/Re-write). 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 when running a test or set of tests, there is rarely any way to cleanly organize the test libraries into groups whose results should be the same versus tests that should be different. A given test will invoke some functions with the legacy business rules, others that have changed, and still others that are totally new.

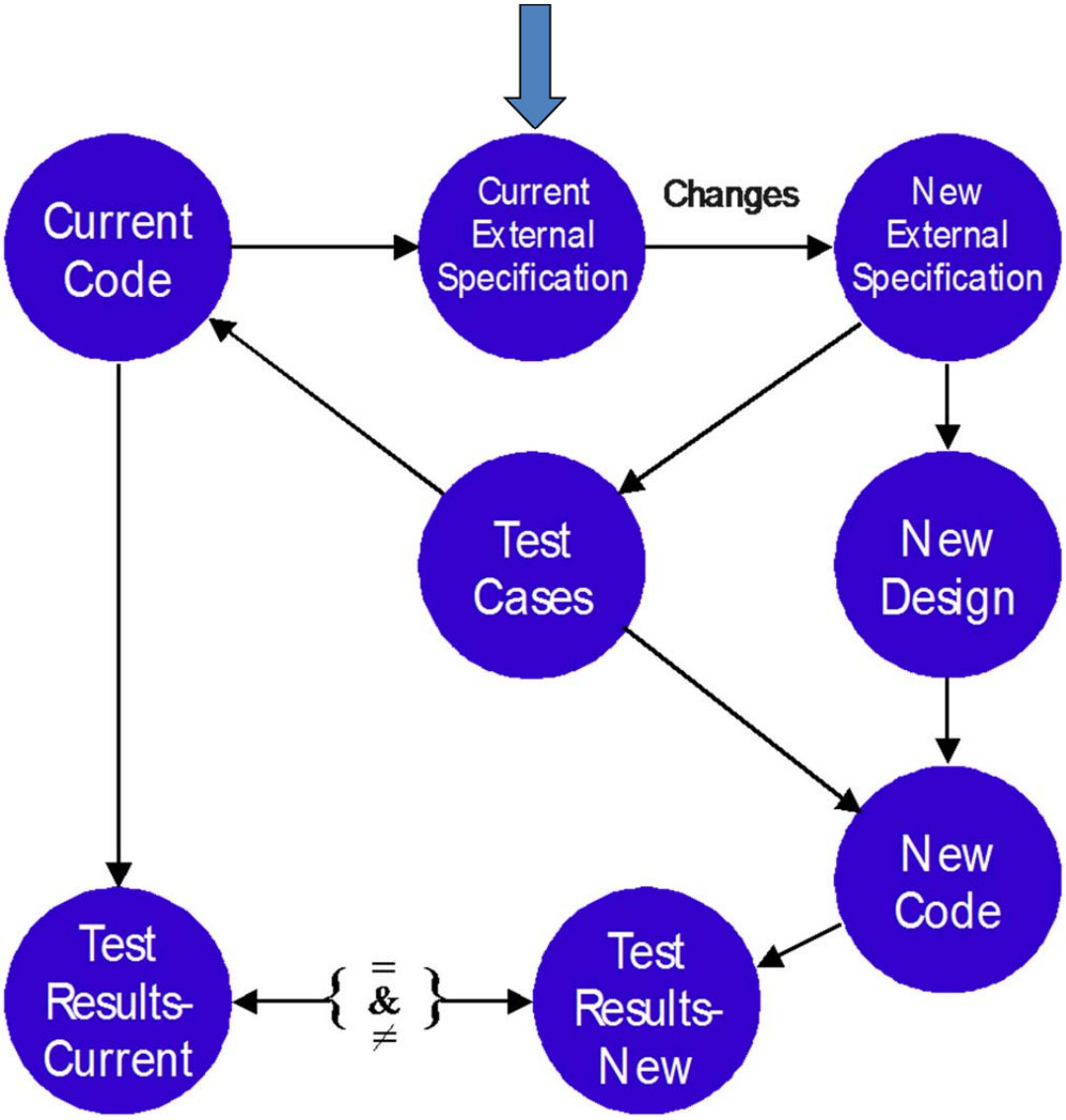

Figure 2 – Testing a Redesign/Re-write

Summary and Recommendations

You might ask why re-design/re-write projects have to go through all of the same tedious, expensive and time consuming test construction and validation against the legacy system as a pure re-write project. After all, we are replacing the legacy with a new system. Why bother with it at all?

The answer comes from the dual requirements that the results of testing be accurate and that testing be complete. Before the system goes into production, it must be both to avoid production disruptions. The lengthy and expensive periods of breaking in a new system can only be avoided in this way. It can cost 100 to 1,000 times as much to find and fix

problems in production as at the requirements stage, as shown time and time again by studies going back to the beginnings of IT.

For this reason, I always recommend avoiding, if possible, modernizing via a re-design/re-write 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 accurate and complete. What can seem wasteful to some is in fact saving a great deal of time and money, not to mention avoiding serious risks.

When project teams go into "dare to be great" mode and try to modernize all the way in one stride, they might put the business at risk. The business issue involved 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. If the enhancements are simply add-on functionality the risk is low. If the enhancements are deeply intertwined with existing functionality, that drives up the risk greatly. The problem is that it makes verifying functional compatibility far more difficult due to the differences. I have never seen a project where the value of the enhancements exceeded the value of the baseline functions.

That said, sometimes you cannot just get away with creating a "vanilla" replacement Release 1. There can be compelling business reasons for deploying the enhanced capability as quickly as possible. In such cases 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 in practice faster, less expensive, and greatly less risky than trying to do everything in one great leap.

Mr. Bender is the President of Bender RBT Inc, a company he founded in 1977. He has over forty-five years' experience in software with a primary focus on quality assurance and testing. He has consulted internationally to large and small corporations, government agencies, and the military. He has been involved in establishing industry standards for software quality, serving as the Technical Lead for the International Y2K Test Certification Standards and assisting the U.S. Food and Drug Administration in defining their Software Quality Guidelines. He was one of the first programmers ever awarded IBM's Outstanding Invention Award. This was for his breakthroughs on code based testing.