Requirements Based Testing

Process Overview

17 Cardinale Lane
Queensbury, NY 12804
518-743-8755
info@BenderRBT.com
www.BenderRBT.com

The requirements-based testing (RBT) process addresses two major issues: first, validating that the requirements are correct, complete, unambiguous, and logically consistent; and second, designing a necessary and sufficient (from a black box perspective) set of test cases from those requirements to ensure that the design and code fully meet those requirements. In designing tests two issues need to be overcome: reducing the immensely large number of potential tests down a reasonable size set and ensuring that the tests got the right answer for the right reason. The RBT process does not assume, going in, that we will see good Requirements Specifications. That is very rarely the case. The RBT process will drive out ambiguity and drive down the level of detail.

The overall RBT strategy is to integrate testing throughout the development life cycle and focus on the quality of the Requirements Specification. This leads to early defect detection which has been shown to be much less expensive than finding defects during integration testing or later. The RBT process also has a focus on defect prevention, not just defect detection.

To put the RBT process into perspective, testing can be divided into the following eight activities:

1. **Define Test Completion Criteria.** The test effort has specific, quantitative and qualitative goals. Testing is completed only when the goals have been reached (e.g., testing is complete when all functional variations, fully sensitized for the detection of defects, and 100% of all statements and branch vectors have executed successfully in single run or set of runs with no code changes in between).
2. **Design Test Cases.** Logical test cases are defined by five characteristics: the initial state of the system prior to executing the test, the data in the data base, the inputs, the expected outputs, and the final system state.
3. **Build Test Cases.** There are two parts needed to build test cases from logical test cases: creating the necessary data, and building the components to support testing (e.g., build the navigation to get to the portion of the program being tested).
4. **Execute Tests.** Execute the test-case steps against the system being tested and document the results.
5. **Verify Test Results.** Verify that the test results are as expected.
6. **Verify Test Coverage.** Track the amount of functional coverage and code coverage achieved by the successful execution of the set of tests.
7. **Manage and Track Defects.** Any defects detected during the testing process are tracked to resolution. Statistics are maintained concerning the overall defect trends and status.
8. **Manage the Test Library.** The test manager maintains the relationships between the test cases and the programs being tested. The test manager keeps track of what tests have or have not been executed, and whether the executed tests have passed or failed.

All of the test tools and test techniques fit into one or more of these categories – a convenient way of looking at the problem. The RBT process focuses on Steps 1, 2, and the black box portion of 6.

Typically, capture/playback tools address Steps 3, 4, and 5. The RBT process complements them very well, as will be discussed later.

The Business Case for RBT

There are numerous economic reasons to improve the quality of your software. These are addressed in some detail in the "Bender-Business Case For Software Quality" document. For this discussion, there are a few we will focus on: reducing the costs to detect and remediate defects, reducing the time it takes to deliver the software, and improving the probability of successfully installing the right solution.

Relative Cost to Fix an Error

The reason for integrating testing earlier into the life cycle is simple economics. Studies have shown two things. First, that the majority of defects have their root cause in poorly defined requirements (see Figure 1). And second that the cost of fixing an error is cheaper the earlier it is found. The issue is scrap and rework. If a defect was introduced while coding, you just fix the code and re-compile. However, if a defect has its roots in poor requirements and is not discovered until integration testing then you must re-do the requirements, re-do the design, re-do the code, re-do the tests, re-do the user documentation, and re-do the training materials. It is all this "re-do" work that sends projects over budget and over schedule. Let's say that a defect introduced during the requirements phase is found during the requirements phase. Define the cost of finding that defect as 1X. If that same defect is not found until integration testing or production it will cost hundreds or even thousands of times more (see Figure 2).

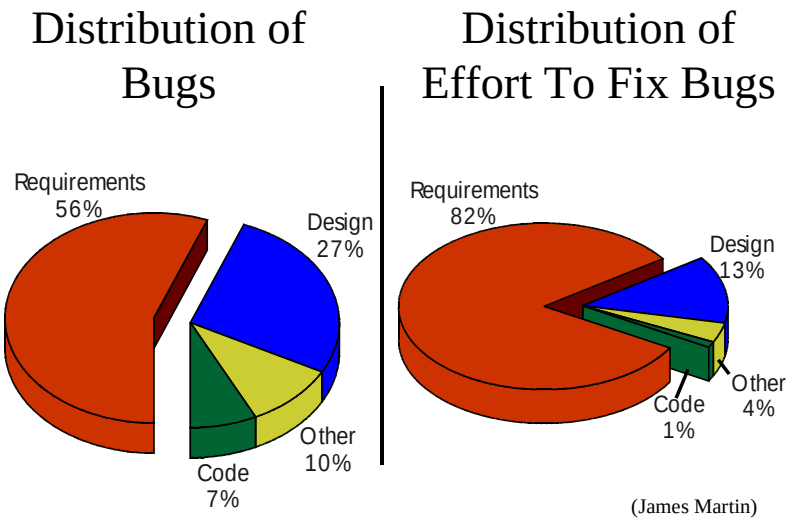


Figure 1: *Distribution of Defects – James Martin*

Phase in Which Found	Cost Ratio
Requirements	1
Design	3-6
Coding	10
Unit/Integration Testing	15-40
System/Acceptance Testing	30-70
Production	40-1000

(IBM, GTE, et. al)

Figure 2: Relative Cost to Fix an Error

Why Good Requirements Are Critical

A study by the Standish Group showed that American companies spend about $300 billion per year on software development and maintenance. Of that, $84 billion was spent for cancelled software projects which delivered nothing. Another $192 billion was spent on software projects that so exceeded their time and budget estimates as to reduce their ability to result in any positive return on investment. The Standish Group and other studies show there are three top reasons why software projects fail:

- Requirements and specifications are incomplete.
- Requirements and specifications change too often.
- There is a lack of user input (to requirements).

The RBT process addresses each of these issues:

- It begins at the first phase of software development where the correction of errors is the least costly.
- It begins at the requirements phase where the largest portion of defects have their root cause.
- It addresses improving the quality of requirements: Inadequate requirements often are the reason for failing projects.

A Good Test Process

The characteristics of a good test process are as follows:

- ***Testing must be timely.*** Any system without a tight feedback loop is a fatally flawed system. Testing is that feedback loop for the software development process. Therefore, it must begin in the initial phase of the project: when objectives are being defined and the scope of the requirements are first drafted. In this way, testing is not perceived as a bottleneck operation. Test early, test often.

- ***Testing must be effective.*** The approach to test-case design must have rigor to it. Testing should not rely solely on individual skills and experiences. Instead, it should be based on a repeatable test process that produces the same test cases for a given situation, regardless of the tester involved. The test-case design approach must provide high functional coverage of the requirements.
- ***Testing must be efficient.*** Testing activities must be heavily automated to allow them to be executed quickly. The test-case design approach should produce the minimum number of test cases to reduce the amount of time needed to execute tests, and to reduce the amount of time needed to manage the tests.
- ***Testing must be manageable.*** The test process must provide sufficient metrics to quantitatively identify the status of testing at any time. The results of the test effort must be predictable (i.e., the outcome each time a test is successfully executed must be the same).

Testing and the Software Development Life Cycle

In most software development life cycles, the bulk of testing occurs only when code becomes available (see Figure 3). With the RBT process, testing is integrated throughout the life cycle for all of the reasons stated above (see Figure 4). [Note: Actually, in our life cycle we use an iterative approach, not a "water fall" approach but it would have made the diagrams needlessly complicated to fully represent this.]

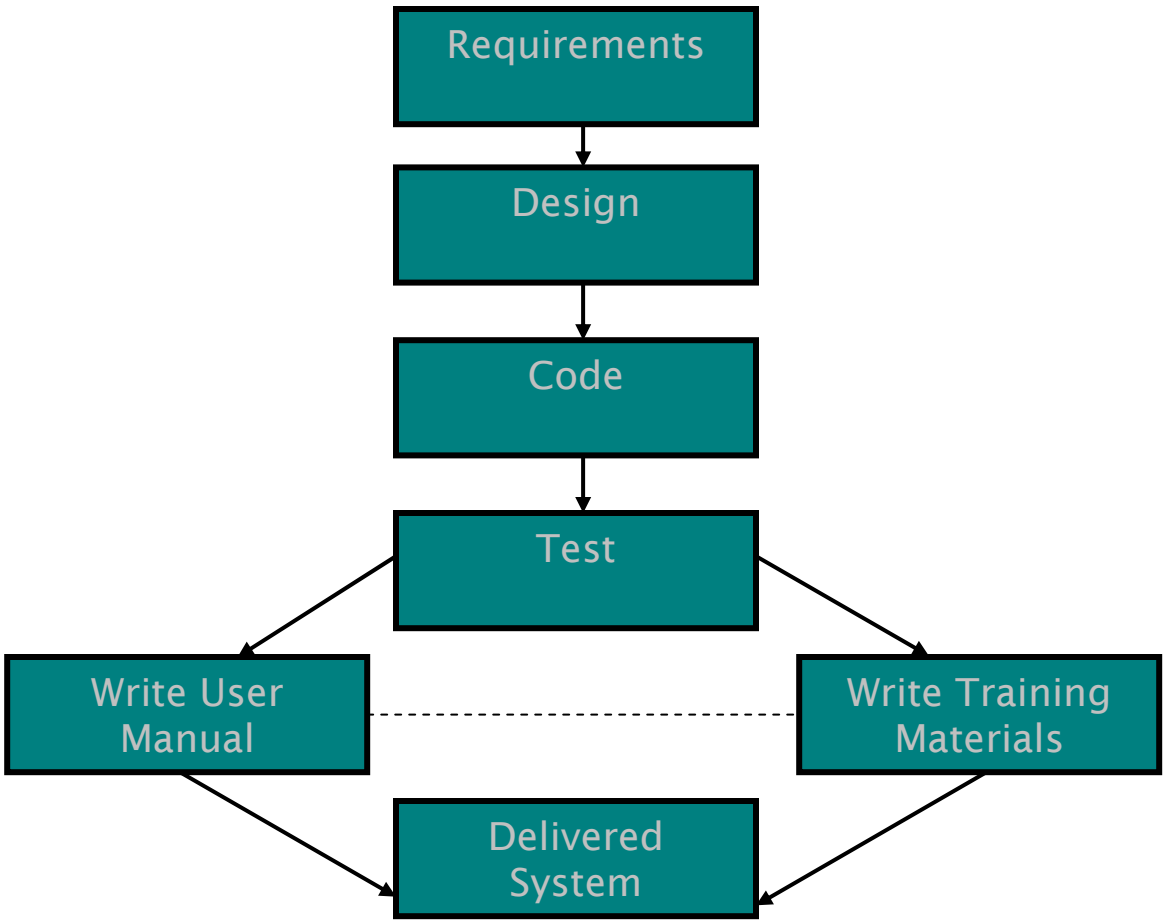


Figure 3 – Standard Development Life Cycle

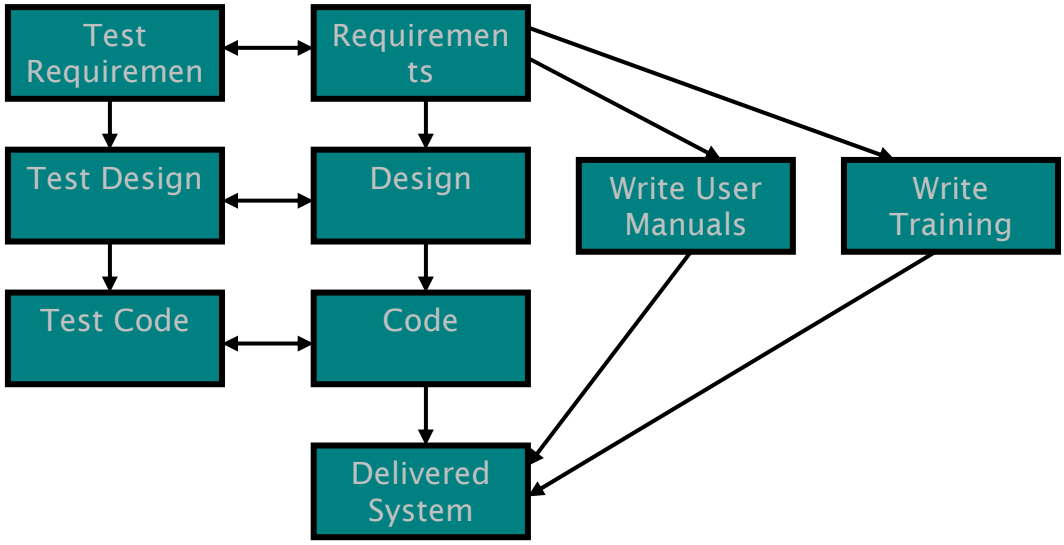


Figure 4 - Life Cycle With Integrated Testing

The RBT Process

The RBT methodology is a 12-step process. [That it is a "12 step program" is kind of appropriate for the dysfunctional world of software.] Each of these steps is described below.

1) Validate requirements against objectives
2) Apply scenarios against requirements
3) Perform initial ambiguity review
4) Perform domain expert reviews
5) Create cause-effect graph
6) Logical consistency check by BenderRBT
7) Review of test cases by specification writers
8) Review of test cases by users
9) Review of test cases by developers
10) Walk test cases through design
11) Walk test cases through code
12) Execute test cases against code

1. ***Validate requirements against objectives.*** Compare the objectives, which describe *why* the project is being initiated, to the requirements, which describe *what* is to be delivered. The objectives define the success criteria for the project. If the *what* does not match the *why*, then the objectives cannot be met, and the project will not succeed. If any of the requirements do not achieve the objectives, then they do not belong in the project scope.
2. ***Apply scenarios against requirements.*** **Apply scenarios against requirements.** A scenario is a task oriented users' view of the system. The individual requirements, taken together, must be capable of satisfying any scenario; otherwise the requirements are incomplete.
3. ***Perform an initial ambiguity review.*** An ambiguity review is a technique for identifying and eliminating ambiguous words, phrases, and constructs. It is not a review of the content of the requirements. The ambiguity review produces a higher-quality set of requirements for review by the rest of the project team.
4. ***Perform domain expert reviews.*** The domain experts review the requirements for correctness and completeness.
5. ***Create Cause-Effect Graph.*** The requirements are translated into a Cause-Effect Graph, which provides the following benefits:
    - It resolves any problems with aliases (i.e., using different terms for the same cause or effect).
    - It clarifies the precedence rules among the requirements (i.e., what causes are required to satisfy what effects).
    - It clarifies implicit information, making it explicit and understandable to all members of the project team.

- It begins the process of integration testing. The code modules eventually must integrate with each other. If the requirements that describe these modules cannot integrate, then the code modules cannot be expected to integrate. The cause-effect graph shows the integration of the causes and effects.
- Cause-Effect Graphs may look intimidating on the surface. They would appear to need someone with a formal logic background or electrical engineering background to create them. However, all you need to know to build them is the definition of "AND", "OR", and "NOT". If you are in the software profession and unsure of these three terms then it is time for a career change.

6. ***Logical consistency checks performed and test cases designed by BenderRBT tool.*** The tool identifies any logic errors in the cause-effect graph. The output from the tool is a set of test cases that are 100 percent equivalent to the functionality in the requirements.
7. ***Review of test cases by requirements authors.*** The designed test cases are reviewed by the requirements authors. If there is a problem with a test case, the requirements associated with the test case can be corrected and the test cases redesigned.
8. ***Validate test cases with the users/domain experts.*** If there is a problem with the test case, the requirements associated with it can be corrected and the test case redesigned. The users/domain experts obtain a better understanding of what the deliverable system will be like. From a Capability Maturity Model® Integration[SM] (CMMI[SM]) perspective, you are validating that you are *building the right system*.
9. ***Review of test cases by developers.*** The test cases are also reviewed by the developers. By doing so, the developers understand what they are going to be tested on, and obtain a better understanding of what they are to deliver so they can deliver for success.
10. ***Use test cases in design review.*** The test cases restate the requirements as a series of causes and effects. As a result, the test cases can be used to validate that the design is robust enough to satisfy the requirements. If the design cannot meet the requirements, then either the requirements are infeasible or the design needs rework.
11. ***Use test cases in code review.*** Each code module must deliver a portion of the requirements. The test cases can be used to validate that each code module delivers what is expected.
12. ***Verify code against the test cases derived from requirements.*** The final step is to build test cases from the logical test cases that have been designed by adding data and navigation to them, and executing them against the code to compare the actual behavior to the expected behavior. Once all of the test cases execute successfully against the code, then it can be said that 100 percent of the functionality has been verified and the code is ready to be delivered into production. From a CMMI perspective, you have verified that you are *building the system right*.

Characteristics of Testable Requirements

In order for the requirements to be considered testable, the requirements ideally should have all of the following characteristics:

1. ***Deterministic:*** *G*iven an initial system state and a set of inputs, you must be able to predict exactly what the outputs will be.
2. ***Unambiguous:*** All project members must get the same meaning from the requirements; otherwise they are ambiguous.
3. ***Correct:*** The relationships between causes and effects are described correctly.
4. ***Complete:*** All requirements are included. There are no omissions.
5. ***Non-redundant:*** Just as the data model provides a non-redundant set of data, the requirements should provide a non-redundant set of functions and events.
6. ***Lends itself to change control:*** Requirements, like all other deliverables of a project, should be placed under change control.
7. ***Traceable:*** Requirements must be traceable to each other, to the objectives, to the design, to the test cases and to the code.
8. ***Readable by all project team members:*** The project stakeholders, including the users, developers and testers, must each arrive at the same understanding of the requirements.
9. ***Written in a consistent style:*** Requirements should be written in a consistent style to make them easier to understand.
10. ***Processing rules reflect consistent standards:*** Processing rules should be written in a consistent manner to make them easier to understand.
11. ***Explicit:*** Requirements must never be implied.
12. ***Logically consistent:*** There should be no logic errors in the relationships between causes and effects.
13. ***Lends itself to reusability:*** Good requirements can be reused on future projects.
14. ***Terse:*** Requirements should be written in a brief manner, with as few words as possible.
15. ***Annotated for criticality:*** Not all requirements are critical. Each requirement should note the degree of impact a defect in it would have on production. In this way, the priority of each requirement can be determined, and the proper amount of emphasis placed on developing and testing each requirement. We do not need zero defect in most systems. We need good enough testing.
16. ***Feasible:*** If the software design is not capable of delivering the requirements, then the requirements are not feasible.

The key characteristics in the above list are Deterministic and Unambiguous. Testing, by definition, is comparing the expected results to the observed results. This is not unique to software. When you were in school and took a math test, the instructor did not just look to see if your answer was "reasonable". They compared your answer to their pre-calculated answer. If your answer was different it was wrong. Software testing is supposed to have that characteristic. Unfortunately, you almost never find Requirements Specifications written in sufficient detail to be able to determine the expected outcome. In the hundreds of projects that my staff and I have been involved in over the last twenty years, we have seen only two good specifications going into the project. As stated above, the RBT drives out ambiguity and drives down the level of detail in the specifications.

Ambiguity Review Checklist

The Ambiguity Review Checklist is made up of 15 classes of ambiguities that are commonly found in requirements. The Ambiguity Review Checklist is used to manually review requirements and identify all of the ambiguities so they can be eliminated. The Ambiguity Review Checklist includes the following types of ambiguities:

- The dangling Else
- Ambiguity of reference
- Scope of action
- Omissions
- Ambiguous logical operators
- Negation
- Ambiguous statements
- Random organization
- Built-in assumptions
- Ambiguous precedence relationships
- Implicit cases
- Etc.
- I.E. versus E.G.
- Temporal ambiguity
- Boundary Ambiguity

Note: For a more complete description of the ambiguity review process and checklist, see the "Bender-Ambiguity Review White Paper".

Savings Via Ambiguity Reviews

A study by Bender RBT Inc. showed that, if defects are found in the requirements phase using ambiguity reviews, the cost of finding each of these defect is only about $25 to $40 per severity 1 or severity 2 defect. This was calculated using a burden rate (employee salary, benefits, etc.) of $150K/year. Compare this figure to the results of a study by the Software Engineering Institute, which showed that if defects are found in integration/system test, the cost of finding each of these defects is between $750 to $3,000. Studies by Hewlett Packard and IBM showed that if defects are found in production, the cost of finding each of these defects is between $10,000 and $140,000. Our experience has shown that if these ambiguities are not dealt with in the Requirements Definition Phase, nearly 100% of them will show up as defects in the code.

The Test Case Design Challenge

One of the major challenges in test case design is this simple fact: any system, except the most trivial ones, has more possible combinations and permutations of the inputs/states than there are molecules in the universe (which is 10^{80} according to Stephen Hawking in

"A Brief History In Time"). The key challenge then is to select an infinitesimally small subset of tests which, if they run correctly, give you a very high degree of assurance that all of the other combinations/permutations will also run correctly.

The software industry has done a very poor job at meeting this challenge. The result is that software still averages worse than 5 defects / 1K executable lines of code. On the other hand those producing high end integrated circuits (e.g., Intel, Motorola) achieve defect rates of less than 1 defect / million gates. Even though high end chips now exceed 40 million gates you rarely hear about a logic error on the released products. There is no fundamental difference in testing rules in software versus rules in hardware - rules are rules are rules. The difference in quality is solely due to the difference in the development and testing approaches. Hardware engineers use disciplined, rigorous approaches while software is still essentially a "craftsman" endeavor. Software is now far too complex and critical to continue in this manner.

Having created the Cause-Effect Graph, which can be done from any style specification (e.g., use-cases, inheritance diagrams, finite state diagrams, narratives, decision tables), the logic rules are now in a form that looks like a circuit diagram. We can then apply the same type of rules to test design that the hardware industry has used for years. These are the path sensitizing algorithms for hardware logic testing. We have extended the rules in them to cover software specific issues. The basic issue is ensuring that you got the right answer for the right reason. If there is any defect in the code it must show up at an observable point. Given the long path lengths in most code today this is a non-trivial challenge.

Let’s look at what these algorithms do for us in designing tests using two examples.

Figure 5 shows a simple application rule that states that if you have A or B or C you should produce D. The test variations to test are shown in Figure 6. The “dash” just means that the variable is false. For example, the first variation is A true, B false, and C false, which should result in D true. Each type of logical operator – simple, and, or, nand, nor, xor and not – has a well defined set of variations to be tested. The number is always n+1 where n is the number of inputs to the relation statement. In the case of the “or” you take each variable true by itself with all the other inputs false and then take the all false case. You do not need the various two true at a time cases or three true at a time, etc. These turn out to be mathematically meaningless from a black box perspective. The test variations for each logical operator are then combined with those for other operators into test cases to test as much function in as small a number of tests as possible.

Let us assume that there are two defects in the code that implements our A or B or C gives us D rule. No matter what data you give it, it thinks A is always false and B is always true. There is no Geneva Convention for software that limits us to one defect per function.

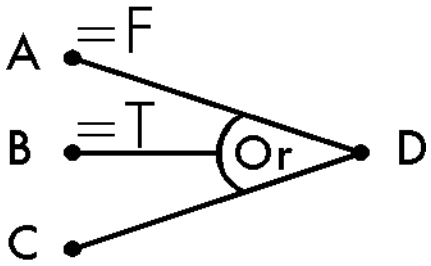


Figure 5 - Simple "OR" Function With Two Defects

1.	A	–	–	D
2.	–	B	–	D
3.	–	–	C	D
4.	–	–	–	–

Figure 6 - Required Test Cases For The "OR" Function

Figure 7 shows the results of running the tests. When we run test variation 1 the software says A is not true, it is false. However, is also says B is not false, it is true. The result is we get the right answer for the wrong reason. When we run the second test variation we enter B true, which the software always thinks is the case – we get the right answer. When we enter the third variation with just C true, the software thinks both B and C are true. Since this is an inclusive "or," we still get the right answer. We are now reporting to management that we are three quarters done with our testing and everything is looking great. Only one more test to run and we are ready for production. However, when we enter the fourth test with all inputs false and still get D true, then we know we have a problem.

1.	A	–	–	as	–	B	–		D
2.	–	B	–	as	–	B	–		D
3.	–	–	C	as	–	B	C		D
4.	–	–	–	as	–	B	–		(D)

Figure 7 - Variable "B" Stuck True Defect Found By Test Case 4

There are two key things about this example so far. The first is that software, even when it is riddled with defects, will still produce correct results for many of the tests. The second thing is that if you do not pre-calculate the answer you were expecting and compare it to the answer you got you are not really testing. Sadly, the majority of what purports to be testing in our industry does not meet these criteria. People look at the test results and just see if they look "reasonable". Part of the problem is that the specifications are not in sufficient detail to meet the most basic definition of testing.

When test variation four failed, it led to identifying the "B stuck true" defect. The code is fixed and test variation four, the only one that failed, is rerun. It now gives the correct results. This meets the common test completion criteria that every test has run correctly at least once and no severe defects are unresolved. The code is shipped into production. However, if you rerun test variation one, it now fails (see Figure 8). The "A stuck false" defect was not caused by fixing the B defect. When the B defect is fixed you can now see the A defect. When any defect is detected all of the related tests must be rerun.

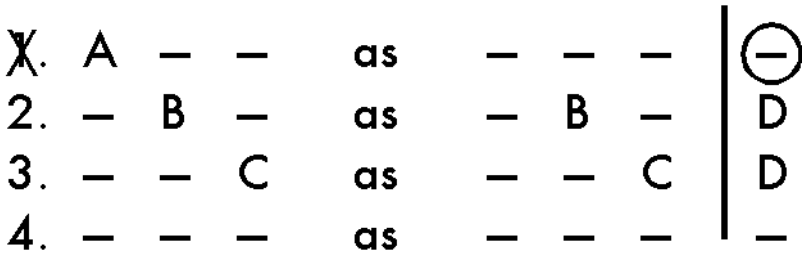


Figure 8 - Variable "A" Stuck False Defect Not Found Until Variable "B" Defect Fixed

The above example addresses the issue that two or more defects can sometimes cancel each other out giving the right answers for the wrong reasons. The problem is worse than that. The issue of observability must be taken into account. When you run a test how do you know it worked? You look at the outputs. For most systems these are updates to the databases, data on screens, data on reports, and data in communications packets. These are all externally observable.

In Figure 9 let us assume that node G is the observable output. C and F are not externally observable. We will indirectly deduce that the A, B, C function worked by looking at G. We will indirectly deduce that the D, E, F function worked by looking at G. Let us further assume there is a defect at A where the code always assumes that A is false no matter what the input is. A fairly obvious test case would be to have all of the inputs set to true. This should result in C, F, and G being set to true. When this test is entered the software says A is not true, it is false. Therefore, C is not set to the expected true value but is set to false. However, when we get to G it is still true as we expected because the D, E, F leg worked. In this case we did not see the defect at C because it was hidden by the F leg working correctly.

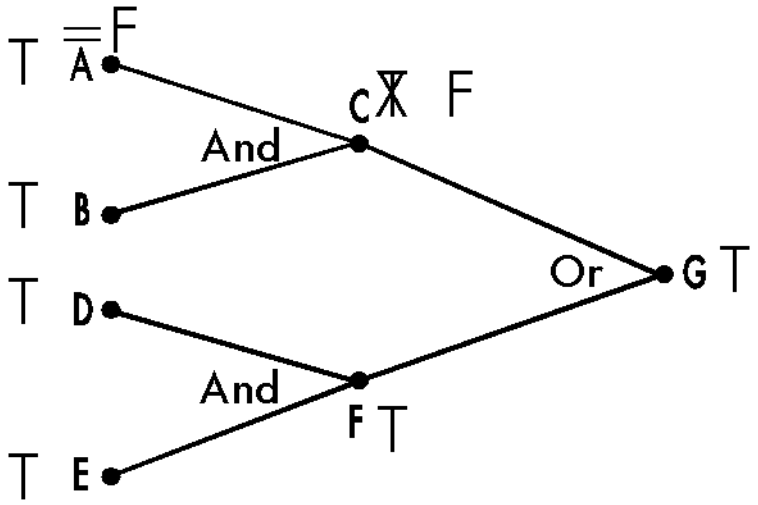


Figure 9 - Variable "A" Stuck False Defect Not Observable

Therefore, the test case design algorithms must factor in:

- The relations between the variables (e.g., and, or, not)
- The constraints between the data attributes (e.g., it is physically impossible for variables one and two to be true at the same time)
- The functional variations to test (i.e., the primitives to test for each logical relationship)
- Node observability

The design of the set of tests must be such that if one or more defects are present, you are mathematically guaranteed that at least one test case will fail at an observable point. When that defect is fixed, if any additional defects are present, then one or more tests will fail at an observable point.

The algorithms that the BenderRBT tool uses to design the tests are highly optimized. As an example, in the test of a GUI screen, the theoretical number of tests that could have been created was 137,438,953,472 possible test cases. RBT resolved the problem with 22 test cases that provided 100 percent functional coverage.

Consider the following thought experiment: Put 137,438,953,450 red balls in a giant barrel. Add 22 green balls to the barrel and mix well. Turn out the lights. Pull out 22 balls. What is the probability that you have selected all 22 of the green balls? If this does not seem likely to you, try again. Return the balls and pull out 1,000 balls. What is the probability that you now have selected all 22 of the green balls? If this still does not seem likely to you, try again. Return the balls and pull out 1,000,000 balls. What is the probability that you now have selected all 22 of the green balls? This is what ***gut-feel*** testing really is.

For most complex problems it is impossible to manually derive the right combination of test cases that covers 100 percent of the functionality. The right combination of test cases is made up of individual test cases, and each covers at least one type of error that none of the other test cases covers. Taken together, the test cases cover 100 percent of the functionality. Any more test cases would be redundant because they would not catch an error that is already covered by an existing test case.

Gut-feel testing often focuses only on the normal processing flow. Another name for this is the *go right path*. Gut-feel testing often creates too many (redundant) test cases for the go path. Gut-feel testing also often does not adequately cover all the combinations of error conditions and exceptions, i.e., the processing off the go path. As a result, gut-feel testing suffers when it comes to functional coverage.

Even reasonably methodical approaches, such as equivalence class testing with boundary analysis or orthogonal pairs testing, only address trying to reduce the total number of possible tests down to a reasonably small number. They do not address the other critical issue: did you observe the defect. This is why we are testing after all.

We have measured the coverage of existing test libraries, designed via traditional techniques, on numerous projects. They have generally been at only 35% to 40% of the coverage that would be achieved via Cause-Effect Graphing. The number of tests was also usually two to four times the number of graphing based tests.

Synergy With Other Testing Tools

The RBT process and the BenderRBT tool which supports it enhance your efforts with many of the other tools you may be using.

- Requirements Management tools (e.g. Borland's CaliberRM, Telelogic's DOORS, IBM/Rational's Requisite Pro) address the logistics of the managing the requirements objects. RBT addresses the quality of those objects.
- Capture/Playback Tools
  - The RBT process stabilizes the functional definition and the user interface earlier in the life cycle, allowing timely implementation of stable test scripts.
  - The BenderRBT tool minimizes the number of scripts required by calculating the minimum number of test cases.
- Code Coverage Analyzers
  - The industry average for code coverage at test completion is under 50%.
  - The RBT process results in between 70% and 90% code coverage by providing 100% functional coverage with tests designed before the first line of code was written. You then supplement the RBT tests with additional white box oriented test to get 100% code coverage. Another benefit is that, since the bulk of the tests are black box tests they are highly portable as the application is ported to new technologies.

Case Study Results

We have been applying the RBT process since the early 1970's. This has included just about every application type and technology including embedded to super-computer based applications and everything in between – e.g. financial, military, scientific, manufacturing, operating systems, communications, medical, etc. Over the years we have naturally refined it and improved the automated support. The following are brief discussions of the typical results achieved via this process from recent projects in the last year or so.

Manufacturing Support System

- System to manage design and production issues
- First iteration had >50% defect rate on initial test passes
- Second iteration switched to RBT
- Results:
  - Over 250 specification defects identified and resolved prior to development
  - < 10% initial defect rate
  - All testing completed in a single test phase
  - No functional defects escaped into User Acceptance Test
  - All tests automated immediately following first test phase
  - 20% reduction in rework
- Same results in iteration 3 except initial defect rate < 5%

Hardware/Software Vendor Configurator

This is a system that ensures the compatibility of any hardware and/or software being ordered with the other items being ordered and with what is already installed.

- 60,000 business rules
- 25% change each month
- New releases multiple times per week
- No room for error
- Prior to RBT – numerous severity one and two defects per release
- Business rules modeled in RBT
- Updates to Configurator reflected in models
- RBT generates revised tests descriptions
- DTT add-on generates executable scripts
- Defect rate drops to near zero
- No severity ones or twos in last 3 years
- 12 people maintain 20,000 executable test scripts

Note: DTT (Direct to Test) is an add-on to the BenderRBT Test Case Design Tool developed by one of our partners. It is a framework tool that generates executable test

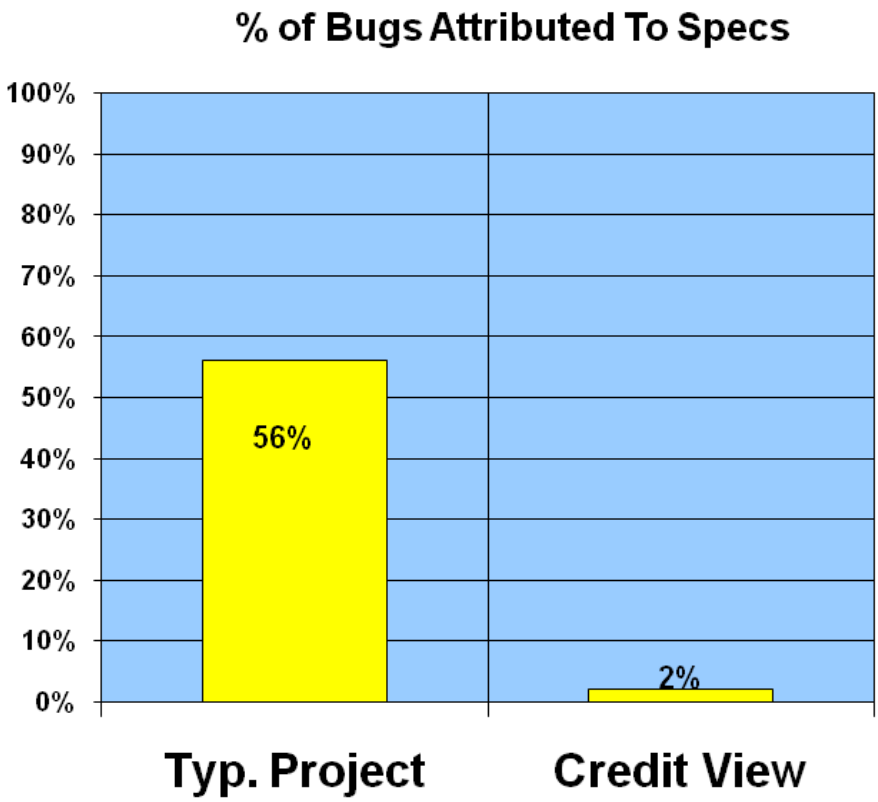
scripts from the RBT output. It can generate those scripts to run on any of the major playback tools.

Credit Card System

- Credit card processing system for a bank
- Major inventory of customizations, little documentation, limited SME (subject matter experts) availability
- 60 models and about 900 test cases
- No customization functional defects detected by the user
- Over 3000 bugs in the base technology detected – i.e. the existing functionality that had to be traversed to get to the new functionality.

Bank Credit System

As stated earlier, most projects find that about 56% of the defects have their root cause in poor requirements. On this project the integration test effort found that only 2% of the defects could be traced back to requirements errors. This resulted in a huge savings in the time and costs to deliver the system. Once in production, no defects were encountered.



Summary

In summary, the RBT methodology reduces the time to deliver by allowing testing to be performed concurrently with the rest of the development activities. It reduces the cost to deliver and time to deliver by minimizing scrap and rework. It delivers maximum coverage with the minimum number of test cases. RBT also provides quantitative test progress metrics within the 12 steps of the RBT methodology, ensuring that testing is being adequately performed and is no longer a bottleneck. Once you learn it you can apply it to any application written in any programming language, running on any machine in any operating system.