What Is Requirements-Based Testing?

Gary E. Mogyorodi
Bloodworth Integrated Technology, Inc.

This article provides an overview of the requirements-based testing (RBT) process. RBT is comprised of two phases: ambiguity reviews and cause-effect graphing. An ambiguity review is a technique for identifying ambiguities in functional [1] requirements to improve the quality of those requirements. Cause-effect graphing is a test-case design technique that derives the minimum number of test cases to cover 100 percent of the functional requirements. The intended audience for this article is project managers, development managers, developers, test managers, and test practitioners who are interested in understanding RBT and how it can be applied to their organization.

The requirements-based testing (RBT) process is comprised of two phases: ambiguity reviews and cause-effect graphing. An ambiguity review is a technique for identifying ambiguities in functional[1] requirements to improve the quality of those requirements. Cause-effect graphing is a test-case design technique that derives the minimum number of test cases to cover 100 percent of the functional requirements.

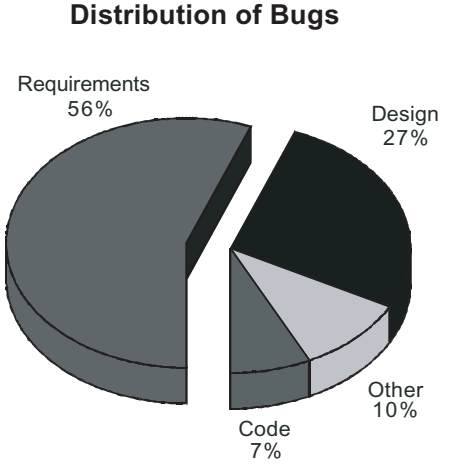
Testing can be divided into the following seven activities:

1. **Define Test Completion Criteria.** The test effort has specific, quantifiable goals. Testing is completed only when the goals have been reached (e.g., testing is complete when the tests that address 100 percent functional coverage of the system all have executed successfully).
2. **Design Test Cases.** Logical test cases are defined by four characteristics: the initial state of the system prior to executing the test, the data, the inputs, and the expected results.
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.** Testers are responsible for verifying two different types of test results: Are the results as expected? Do the test cases meet the test completion criteria?
6. **Verify Test Coverage.** Track the amount of functional coverage achieved by the successful execution of each test.
7. **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.

Activities one, two, and six are addressed by RBT. The remaining four activities are addressed by test management tools that track the status of test executions.

The RBT process stabilizes the application interface definition early because the requirements for the user interface become well defined and are written in an unambiguous and testable manner. This allows the use of capture/playback tools sooner in the software development life cycle.

Figure 1: *Distribution of Bugs*



Relative Cost to Fix an Error

The cost of fixing an error is lowest in the first phase of software development (i.e., requirements). This is because there are very few deliverables at the beginning of a project to correct if an error is found. As the project moves into subsequent phases of software development, the cost of fixing an error rises dramatically since there are more deliverables affected by the correction of each error. At the requirements phase the cost ratio to fix errors is one to one; at coding it is 10 to one; at production it is from 40 to 1,000 to one.

A study by James Martin showed that the root cause of 56 percent of all bugs identified in projects is errors introduced in the requirements phase. Of the bugs rooted in requirements, roughly half were due to poorly written, ambiguous, unclear, and incorrect requirements. The remaining half was due to requirements that were completely omitted (see Figure 1).

Why Good Requirements Are Critical

A study by the Standish Group in 2000 showed that American companies spent $84 billion for cancelled software projects. Another $192 billion was spent on software projects that significantly exceeded their time and budget estimates. 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 bugs 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**. Testing begins when requirements are first drafted; it must be integrated throughout the software development life cycle. 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 on individual skills and experiences. Instead, it should be based on a repeat-

able 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).

Standard Software Development Life Cycle

There are many software development methodologies. Each has its own characteristics and approaches, but most software development methodologies share the following six aspects:

- **Requirements.** There is a description of what has to be delivered.
- **Design.** There is a description of how the requirements will be delivered.
- **Code.** The system is constructed from the requirements and the design.
- **Test.** The behavior of the code is compared to the expected behavior described by the requirements.
- **Write user manuals/write training materials.** Documentation is created to support the delivered system.
- **International translations.** Code is often executed in different countries with different languages; the initial system must be translated into the native language of the target country.

In many software development methodologies, testing does not begin until after code is constructed. If a defect is found after coding, there is a good deal of scrap and rework to correct the code, and possibly the design, test cases, and requirements as well. Defects must be tested out of the system rather than being avoided in the first place. Testing often is a bottleneck activity. See Figure 2 for a graphical representation of a standard development life cycle.

Life Cycle With Testable Requirements

In a software development life cycle with testable requirements and integrated testing, the RBT process is integrated throughout the entire software development life cycle. As soon as requirements are complete, they are tested. As soon as the design is complete, the requirements are walked through the design to ensure that they can be met by the design. As soon as the code is constructed and reviewed, it is tested as usual. But because testing begins at the requirements phase, many defects are avoided instead of being tested out of the code.

This is a less costly and more timely approach. User manuals and training materials can be developed sooner. The entire software development life cycle is compressed. Testing is performed in parallel with development instead of all at the end, so testing is no longer a bottleneck. There are fewer surprises when the code is delivered (see Figure 3).

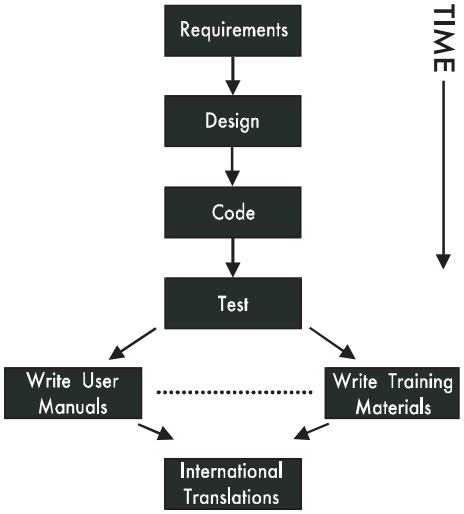


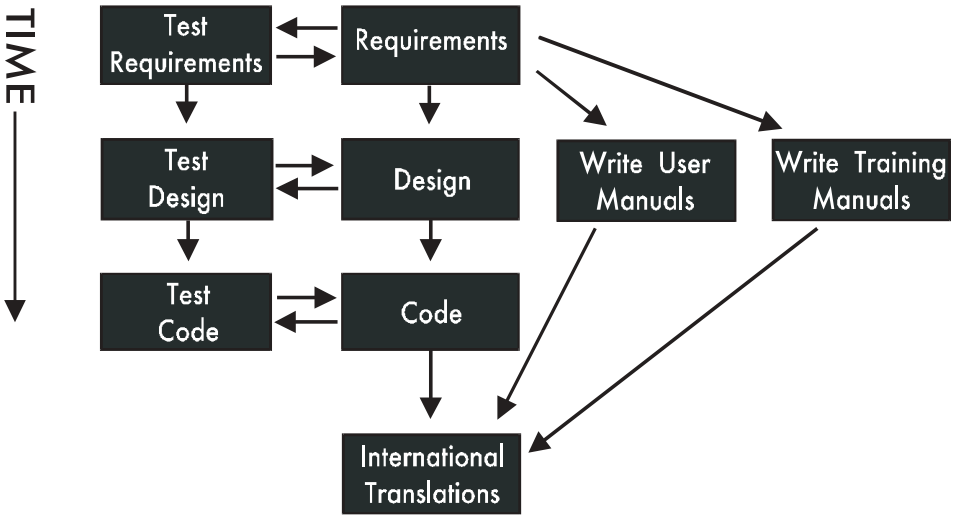
Figure 2: *Standard Development Life Cycle*

The RBT Methodology

The RBT methodology is a 12-step process. Each of these steps is described below.

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 use cases against requirements.** Some organizations document their requirements with use cases. A use case is a task-oriented users' view of the system. The individual requirements, taken together, must be capable of satisfying any use-case scenarios; 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,

Figure 3: *Life Cycle With Testable Requirements and Integrated Testing*



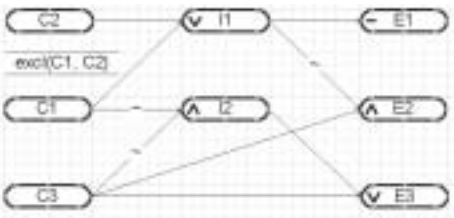


Figure 4: *Cause-Effect Graph*

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.

6. **Logical consistency checks performed and test cases designed.** A 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℠ (CMMI℠) 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*.

An Ambiguity Review

Here is a sample of a requirement written in first draft. It is not testable because it contains ambiguities.

> ATMs shall send an alert to the information technology (IT) department when the ATM has been tampered with. In the event that the ATM is opened without the key and security code, the ATM will alert the IT department immediately so the appropriate action can be taken.

After performing an ambiguity review of the requirements, the following ambiguities are identified:

- What type of alert does the ATM issue to the IT department?
- What is the definition of *tampered with*?
- Is tampered with the same as "in the event that the ATM is opened without the key and security code?"
- What happens if the key is used and an invalid security code is entered?
- What is the alert text?
- What is the *appropriate* action?

The requirements are revised so that the ambiguities are eliminated. The requirements are now testable.

> ATMs shall send a tamper alert to the IT department when the ATM has been tampered with, i.e., opened without the key and the valid security code.
>
> Case 1: (1) If the service operator enters the key into the ATM, then the following message displays on the ATM console: "Please enter the valid security code." (2) If the service operator enters the valid security code, then the ATM opens.
>
> Case 2: After entering the key in the ATM, if the service operator enters an incorrect security code, then (1) the following message displays on the ATM console: "Security Code invalid. Please re-enter." (2) The service operator now has three tries to enter the valid security code. If a valid security code is entered in less than or equal to three tries, then the ATM is opened. Each time an invalid security code is entered, the following message is displayed on the ATM console: "Security code invalid. Please re-enter."
>
> Case 3: If a valid security code has not been entered by the third try, then (1) the following message displays on the ATM console: "Security code invalid. The IT department will be notified." (2) The ATM alerts the IT department immediately.
>
> Case 4: In the event that the ATM is opened without the key and the valid security code, then the ATM sends a tamper alert to the IT department immediately.

A Cause-Effect Graphing Example

Consider a check-debit function whose inputs are *new balance* and *account type*, which is either postal or counter, and whose output is one of four possible values:

- Process debit and send out letter.
- Process debit only.
- Suspend account and send out letter.
- Send out letter only.

The function has the following requirements and is testable:

- If there are sufficient funds available in the account to be in credit, or the new balance would be within the authorized overdraft limit, then process the debit.
- If the new balance is below the authorized overdraft limit, then do not process the debit, and if the account type is postal, then suspend the account.
- If a) the transaction has an account type of postal or b) the account type is counter and there are insufficient funds available in the account to be in credit, then send out letter.

The causes for the function are as follows:

- C1 – New balance is in credit.
- C2 – New balance is in overdraft, but within the authorized overdraft limit.
- C3 – Account type is postal.

The effects for the function are as follows:

- E1 – Process the debit.
- E2 – Suspend the account.
- E3 – Send out letter.

A cause-effect graph shows the relationships between the conditions (causes) and the actions (effects) in a notation similar to that used by designers of hardware logic circuits. The check-debit requirements are modeled by the cause-effect graph shown in Figure 4. C1 and C2 cannot be true at the same time.

The cause-effect graph is converted into a decision table. Each column of the decision table is a rule. The table comprises two parts. In the top part, each rule is tabulated against the causes. A *T* indicates that the cause must be TRUE for the rule to apply and an *F* indicates that the condition must be FALSE for the rule to apply. In the bottom part, each rule is tabulated against the effects. A *T* indicates that the effect will be performed; an *F* indicates that the effect will not be performed; an asterisk (*) indicates that the combination of conditions is infeasible and so no effects are defined for the rule. The check-debit function has the decision table shown in Table 1.

Only test cases one through five in Table 1 are required to provide 100 percent functional coverage. Rule No. 6 does not provide any new functional coverage that has not already been provided by the other five rules, so a test case is not required for rule No. 6. No test cases are generated for rule Nos. 7 and 8 because they describe infeasible conditions since C1 and C2 cannot be true at the same time. The final set of test cases with sample-data values is described in Table 2.

Real-Life Problem Test Cases

With a real-life problem, there are usually far more than three inputs (causes). As an example, in one application where RBT was applied, there were 37 inputs. This allowed a maximum of 2**37, or 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 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.

Summary

In summary, the RBT methodology delivers maximum coverage with the minimum number of test cases. This translates into 100 percent functional coverage and approximately 70 percent to 90 percent code coverage. RBT also provides quantitative test progress metrics within the 12 steps of the RBT methodology, ensuring that testing is adequately provided and is no longer a bottleneck. Logical test cases are designed and become the basis for highly portable capture/playback test scripts.◆

Table 2: *Test Cases*

	CAUSES						EFFECTS
Test Case	*Current Balance*	*Debit Amount*	*Difference*	*Overdraft Limit*	*New Balance*	*Account Type*	*Action*
1	-$70	$50	-$120	-$100	-$70	Counter	**Send out letter.**
2	$420	$2,000	-$1,580	-$1,500	$420	Postal	**Suspend the account; send out letter.**
3	$650	$800	-$150	-$250	-$150	Counter	**Process the debit; send out letter.**
4	$2,100	$1,200	$900	-$1,000	$900	Counter	**Process the debit.**
5	$250	$150	$100	-$500	$100	Postal	**Process the debit; send out letter.**

Rules	1	2	3	4	5	6	7	8
C1: New balance is in credit.	F	F	F	T	T	F	T	T
C2: New balance is in overdraft, but within the authorized limit.	F	F	T	F	F	T	T	T
C3: Account is postal.	F	T	F	F	T	T	F	T
E1: Process the debit.	F	F	T	T	T	T	*	*
E2: Suspend the account.	F	T	F	F	F	F	*	*
E3: Send out letter.	T	T	T	F	T	T	*	*

Table 1: *Decision Table*

Note

1. A functional requirement specifies what the system must be able to do in terms that are meaningful to its users. A nonfunctional requirement specifies an aspect of the system other than its capacity to do things. Examples of nonfunctional requirements include those relating to performance, reliability, serviceability, availability, usability, portability, maintainability, and extendibility.

Did this article pique your interest?

You can hear more from Gary E. Mogyorodi at the Fifteenth Annual Software Technology Conference Apr. 28-May 1, 2003 in Salt Lake City, UT. He will be presenting in Track 1 on Monday, Apr. 28, at 3:10 p.m.

About the Author

Gary E. Mogyorodi is a senior consultant with Bloodworth Integrated Technology, Inc., consulting, training, and mentoring in software testing, and specializing in requirements-based testing. He has more than 29 years of experience in the computing industry and has presented at numerous conferences, including the Software Technology Conference, Software Quality Forum, Toronto SPIN, Starwest, and more. Mogyorodi has a bachelor's degree in mathematics from the University of Waterloo, and a master's degree in business administration from McMaster University.

Bloodworth Integrated Technology, Inc.
36A Mendota Road #8
Toronto, Ontario
Canada M8Y 1E8
Phone: (416) 521-7200
Fax: (419) 831-6407
E-mail: garym@bitspi.com