

Is my testing good enough?

S.Sharaniya
Honeywell
Madurai, Tamilnadu
sharaniya.srinivasan@honeywell.com

Abstract:

To imbibe agility in testing without sacrificing the quality, I have designed a tool called SAPTHA – is a excel workbook with seven sheets, with templates for orthogonal array testing, equivalence partitioning, defect prediction model, boundary value analysis, exit criteria, fault attack and bi-directional traceability matrix. Saptha helps tester whenever there is a schedule crunch allocated for testing, which usually happens in many real time projects, by selecting the appropriate combination of parameters that needs to be tested. In this paper I have explained about all the seven methodologies which are depicted in SAPTHA in detail along with their snapshots.

1. Introduction

In my project I have developed a simple excel sheet called “**Saptha**” which will ensure a tester that he/she has not left many/any surprises for the customer. It has seven template sheets for - vertical/horizontal traceability matrix, defect prediction graphs, equivalence partitioning, fault attack, orthogonal array, boundary value analysis, Exit criteria. SAPHTA – so let us start our journey in exploring the usage of this tool.

Saptha at present does not handle risk; the same is griped in our test plan. Saptha can be used across the projects irrespective of the domains since it encloses only the template which eases the assurance of complete testing

2. Orthogonal array testing

Testing all the modules with all the combination of variables with parameters more than three makes complete testing unfeasible. In order to achieve complete testing coverage, orthogonal array is used.

Benefits of orthogonal array is not limited to fewer test cases for complete testing, helps in creating pair-wise combinations of test, achieves complex combinations of variables, generate testcases which are simpler yet free from error compared to the one created manually, reduces test cycle time .

This method is extremely valuable for testing complex applications and e-comm. products. It does not guarantee the extensive coverage of test domain. Orthogonal arrays could be applied in user interface testing, system testing, regression testing, configuration testing and performance testing

The theory -*Orthogonal Array Testing* (OAT) can be used to reduce the number of combinations and provide maximum coverage with a minimum number of test cases. OAT is an array of values in which each column represents a variable - factor that can take a certain set of values called levels. Each row represents a test case. In OAT, the factors are combined pair-wise rather than representing all possible combinations of factors and levels

SAPTHA – focuses mainly on reducing the testing efforts and at the same time to imbibe quality into the product/project. It displays the possible set of combinations of levels and parameters. A tester after finalizing the number of parameters and the levels can just click on the particular cell and SAPTHA will display the corresponding orthogonal array.

Based upon the array values, the actual inputs can be substituted, and then user can get the matrix according to their labels. SAPTHA handles 2-5 levels and 2-31 parameters. Thus SAPTHA has captured around 72 combinations. The remaining combinations are not possible, so user can confine to the available combinations.

The underlying computation is hidden from user, this sheet will acts as guideline to the tester while designing the test cases for a project with piles of testcases, yet without missing the critical combinations. SAPTHA – does not allow user to customize the array with their own labels, this will be addressed in the next version, where tester has to key in the parameter names and the levels & corresponding matrix will be displayed with appropriate label.

Directions: Click on the square below according to the number of parameters and levels in your Project						
	Number of levels					
	2	3	4	5		
No of parameters	2	P=2,L=2	P=2,L=3	P=2,L=4	P=2,L=5	
	3	P=3,L=2	P=3,L=3	P=3,L=4	P=3,L=5	
	4	P=4,L=2	P=4,L=3	P=4,L=4	P=4,L=5	
	5	P=5,L=2	P=5,L=3	P=5,L=4	P=5,L=5	
	6	P=6,L=2	P=6,L=3	P=6,L=4	P=6,L=5	
	7	P=7,L=2	P=7,L=3	P=7,L=4	P=7,L=5	
	8	P=8,L=2	P=8,L=3	P=8,L=4	P=8,L=5	
	9	P=9,L=2	P=9,L=3	P=9,L=4	P=9,L=5	
	10	P=10,L=2	P=10,L=3	P=10,L=4	P=10,L=5	
	11	P=11,L=2	P=11,L=3		P=11,L=5	
	12	P=12,L=2	P=12,L=3		P=12,L=5	
	13	P=13,L=2	P=13,L=3			
	14	P=14,L=2	P=14,L=3			
	15	P=15,L=2	P=15,L=3			
	16	P=16,L=2	P=16,L=3			
	17	P=17,L=2	P=17,L=3			
	18	P=18,L=2	P=18,L=3			
	19	P=19,L=2	P=19,L=3			
	20	P=20,L=2	P=20,L=3			
	21	P=21,L=2	P=21,L=3			
	22	P=22,L=2	P=22,L=3			
	23	P=23,L=2	P=23,L=3			
	24	P=24,L=2				
	25	P=25,L=2				
	26	P=26,L=2				
	27	P=27,L=2				
	28	P=28,L=2				
	29	P=29,L=2				
	30	P=30,L=2				
	31	P=31,L=2				

These Combinations are not available

Figure 1. Snapshot of the orthogonal array

L4 - Orthogonal Array			
Three Factors & Two Levels			
EXP No	I	II	III
1	1	1	1
2	1	2	2
3	2	1	2
4	2	2	1

Figure 2. Snapshot of the orthogonal array which will be displayed if cell p=2 & l=2 is clicked

3. Equivalence Partitioning

Equivalence partitioning drastically cuts down the number of test cases required to test a system reasonably. It is an attempt to get a good 'hit rate', to find the most errors with the smallest number of test cases.

The testing theory related to equivalence partitioning says that only one test case of each partition is needed to evaluate the behavior of the program for the related partition. In other words it is sufficient to select one test case out of each partition to check the behavior of the program. To use more or even all test cases of a partition will not find new faults in the program. The values within one partition are considered to be "equivalent". Thus the number of test cases can be reduced considerably.

To use equivalence partitioning, we will need to perform two steps

- Identify the equivalence classes
- Design test cases

If any elements of an equivalence class will be handled differently than the others, divide the equivalence class to create an equivalence class with only these elements and an equivalence class with none of these elements.

The process of equivalence partitioning also applies to testing of values other than numbers, like date, group of IDs, time frames etc. Good equivalence partitioning cannot be considered as simply dividing the data set into valid and invalid partition classes and then picking some data from each of them, it's more of a context driven approach.

SAPTHA – provides template for the equivalence partitioning, so that tester can key in the fields to be tested and then they can identify the equivalence classes & to proceed with the test case design. It provides a set of guidelines which will help an invoice tester to write better testcases. This template will enable tester to minimize the number of testcases, and thus it imbibes the agility in testing.

SAPTHA – equivalence partitioning tab has a set of ready made testcases to ease tester in the preparation of test design. Until now only testcases for integers and characters has been enclosed. There are a humble set of general testcases that can be filled in. The test case designing technique differs from one tester to another, and also SAPTHA does not want to confine the thinking circle of a tester to within few classes. So based upon the exploratory skill of each tester, they can write their own test cases to satisfy the requirements.

Guidelines			
Equivalence Partitioning is more appropriate for Textbox, Editbox - where user can key in any values, unlike selecting an entry from the available options			
If the requirements state that a numeric value is input to the system and must be within a range of values, identify one valid class inputs which are within the valid range and two invalid equivalence classes inputs which are too low and inputs which are too high.			
If the requirements state that the number of items input by the system at some point must lie within a certain range, specify one valid class where the number of inputs is within the valid range, one invalid class where there are too few inputs and one invalid class where there are, too many inputs			
For a successful test execution test cases should pass the valid zone and fail in invalid zone			
	Input field to be tested	Valid Zone	Invalid Zone
Predefined testcases	Text box that accepts only characters	non-caps characters	integers,alphanumeric
		characters in CAPS	null
		charaters in both caps and non-caps form	float(decimal)
			special function keys
	Text box that accepts only characters with specific letters	all the specific letters	all other alphabets
	Text box that accepts only specific number of characters	string with expected length	string length less than allowed one
			string length greater than allowed one
			null string
	Text box that accepts only integers	integers	non-caps characters
			characters in CAPS
charaters in both caps and non-caps form			
null			
alphanumeric			
float(decimal)			
special function keys			
Text box that accepts only specific integers	all the specific integers	all other integers	
Text box that accepts only integers with specific range	integers within specific range	integers beyond specific range	
Enter your testcases from here			

Figure 3. Snapshot of the Equivalence Partitioning template

4. Defect prediction models

Many organizations want to predict the number of defects (faults) in software systems, before they are deployed, to gauge the likely delivered quality and maintenance effort. Organizations are still asking how they can predict the quality of their software *before* it is used despite the substantial research effort spent attempting to find an answer to this question over the last 30 years.

There are many papers advocating statistical models and metrics which purport to answer the quality question. Defects, like quality, can be defined in many different ways but are more commonly defined as deviations from specifications or expectations which might lead to failures in operation.

Generally, efforts have tended to concentrate on the following three problem perspectives

1. predicting the number of defects in the system;
2. Estimating the reliability of the system in terms of time to failure;
3. Understanding the impact of design and testing processes on defect counts and failure densities.

Tough various defect prediction models are available, SAPTHA – depicts only Rayleigh defect prediction model, as that suits many project in our organization. A very detailed instruction is provided to a tester who handles the sheet for the first time. A sample sheet is also provided to describe how the calculations are done and almost all the cells are read-only thus constraining the user to move on the right track.

Initially a tester keys in the KLOC in “Enter KLOC” provision, and then he can enter the defect per stage/cycle. Automatically based upon the entered data a graph is plotted below so that user can verify where they are and it helps to assess on the reliability of the project. Tester can enter the value for m , whose value will be decided based upon the data analysis done by the tester. Based upon the time scale, max defect density and cumulative defect rate the graph is plotted below.

But the adherence of the graph to the real time data completely depends upon the inputs collected by the user and the analysis he has done on the same. SAPTHA – does not perform any data validation. It just plots Rayleigh graph based upon the entered data. Since graphs are displayed as the datas are entered, user can get a insight about the quality then ad there.

Around nine instructions are provided to prevent the user from sticking at a particular step. User can modify the phase name according to their needs and they can a new field phase/cycle also, by carefully following the instructions.

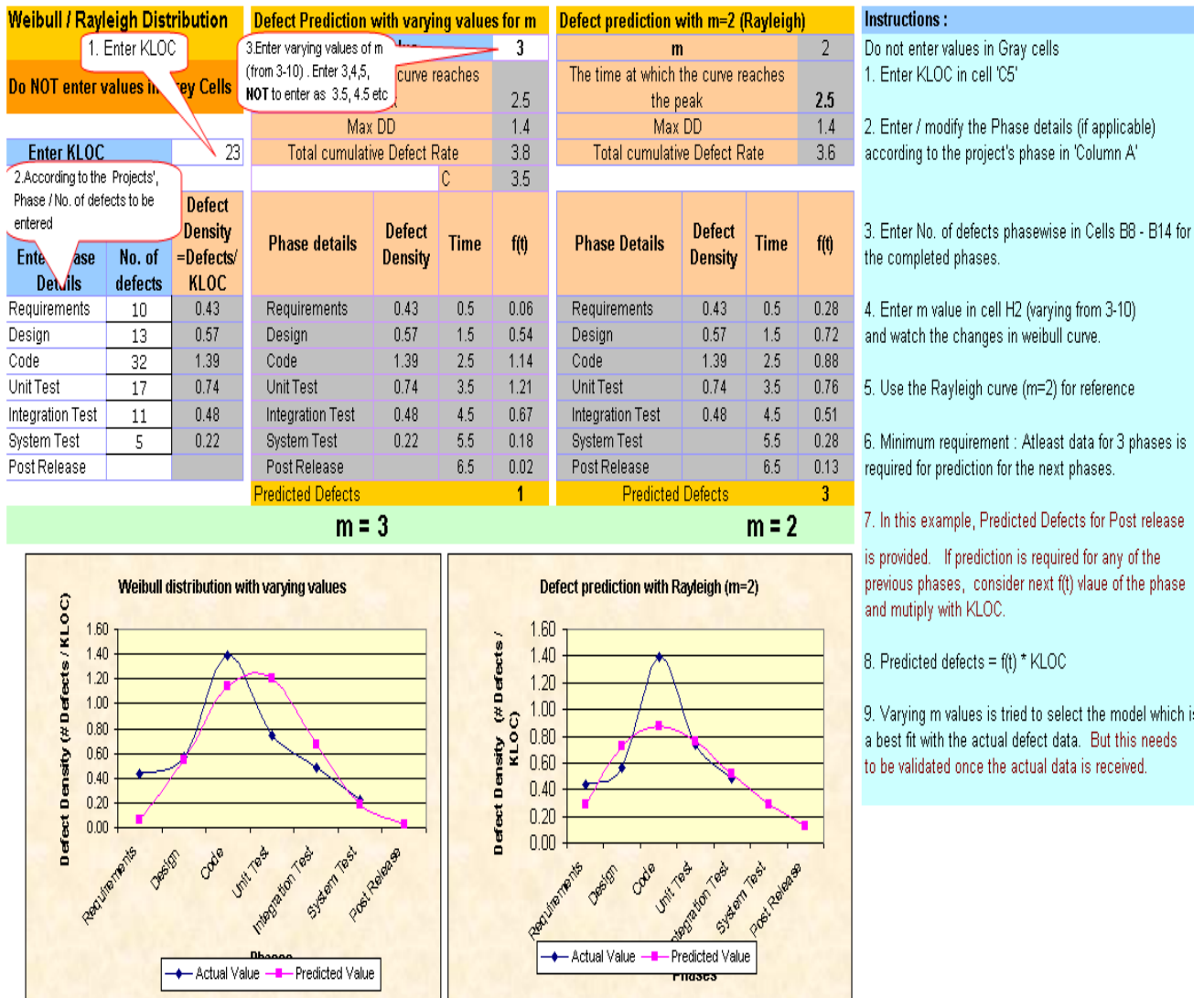


Figure 4. Snapshot of the Defect Prediction template – Rayleigh Model

5. Boundary Value Analysis

By applying boundary value analysis we can select a test case at each side of the boundary between two partitions. Boundary value analysis is a methodology for designing test cases that concentrates software testing effort on cases near the limits of valid ranges

Boundary value analysis is a method which refines equivalence partitioning. Boundary value analysis generates test cases that highlight errors better than equivalence partitioning. The trick is to concentrate software testing efforts at the extreme ends of the equivalence classes. At those points when input values change from valid to invalid errors are most likely to occur. As well, boundary value analysis broadens the portions of the business requirement document used to generate tests.

Unlike equivalence partitioning, it takes into account the output specifications when deriving test cases. Boundary conditions do not need to focus only on values or ranges, but can be identified for many other boundary situations as well. Boundary value analysis along with equivalence partitioning reduces the number of test cases.

Like equivalence partitioning, boundary value analysis we will need to perform two steps

- Identify the equivalence classes
- Design test cases only with valid & invalid boundaries

Thus Boundary value analysis is the technique of making sure that behavior of system is predictable for the input and output boundary conditions. Reason why boundary conditions are very important for testing is because defects could be introduced at the boundaries very easily.

SAPTHA – provides tester with boundary value analysis template where user can enter the input data which needs to be tested. It also provides a set of guidelines which will help to come up with efficient testcases to identify the errors on the boundaries. These comments refrains a tester from taking a wrong path while selecting the boundary values from the designed equivalence classes.

In addition to that it also provides a set of extreme test conditions which can assist tester during the test case design process. Comments are added to aid tester about the usage of the boundary value analysis process for both the valid and invalid zones.

Guideline		
# If the condition is a range of values, create valid test cases for each end of the range and invalid test cases just beyond each end of the range		
# Number of values must lie within a certain range select two valid test cases, one for each boundary of the range, and two invalid test cases, one just below and one just above the acceptable range		
# Design tests that highlight the first and last records in an input or output file.		
# Look for any other extreme input or output conditions, and generate a test for each of them.		
# Extreme Test Conditions		
1. zero or negative values, 2. zero or one transaction, 3. empty files, 4. missing files (file name not resolved or access denied), 5. multiple updates of one file, 6. full, empty, or missing tables, 7. widow headings (i.e., headings printed on pages with no details or totals), 8. table entries missing, 9. subscripts out of bounds, 10. sequencing errors, 11. missing or incorrect parameters or message formats, 12. concurrent access of a file, 13. file space overflow.		
Input to be validated	Valid	Invalid
one	one	two, minus one, plus one
		E356526:
		For single value:

		acceptable value + 1
		acceptable value -1
		For Range:

		right boundary value +1
		right boundary value - 1

Figure 5. Snapshot of the Boundary Value Analysis Template

6. Exit criteria

A set of decision-making guidelines used to determining whether a system under test is ready to exit a particular phase of testing. When exit criteria are met, either the system under test moves on to the next test phase or the test project is considered complete. Exit criteria tend to become more rigorous as the test phases progress. Exit criteria are the criteria or requirements which must be met to complete a specific process. Testing Exit criteria ensures that the testing of the application is completed and ready and delivered to the customer on time with the required quality and above all within the allocated budget.

An Efficient tester should know when to stop their testing, to ensure that right amount of testing is done, exit criteria is defined for each test cycle. Exit criteria must be updated as the test execution proceeds, to make sure that test cycle is on right track. It ensures that the project application has been satisfactorily completed before exiting the system test stage and clarifying the application as complete.

Exit criteria acts as a checklist to Tester, so that they do not leave any surprises for end-users or leak any post-release defects, thus it acts a Quality inducer for the product under test.

They must be defined before starting the testing process. So that criteria can be monitored, during the test execution process for completeness. And appropriate actions can be taken if any slippage is foreseen.

SAPTHA – helps tester by providing a predefined set of exit criteria, which are mandatory in every testing project. SAPTHA allows user to select the applicable criterias since we all know that Exit criterias are specific to user, Based upon the applicability to the project the tester can select the required criterias, if the applicability is set to NO, the SAPTHA will prompt user whether to make the particular criteria as invisible, if user selects “yes” then that particular option will be disabled. In this way a tester can customize the exit criteria tab

In future if any invisible criteria need to be made visible, then user has to click the View all button to display all the available options, and then user can select the appropriate options. Thus SAPTHA – is completely customizable according to user needs. For certain criterias user can define the maximum and minimum allowed values. In Status\Comment provision tester can update any modifications or comments.

SAPTHA – user friendliness eases tester to update the criteria status then and there and thus inturn gives a tester an insight about the testing process. It inturn acts as a score card. SAPTHA covers around 50 criterias, covering almost all the major criterias. The criterias specified in SAPTHA includes cost, time, quality, coverage etc. But it does not allow the user to enter new criterias. Adding new criterias will be added as an enhancement in the next version of SAPTHA

Project:	Calculator	Author:	Sharaniya	
Date of Last Review:	11/12/2007	Reviewed By:	QA Team	
Exit Criterias	Min	Max	Applicable	Status\Comments
No of bugs with prioiry level -1 after test execution	10	100	yes	done
No of bugs with prioiry level -2 after test execution	10	100	No	
No of bugs with prioiry level -3 after test execution	10	100	No	QA Team updated
No of bugs with prioiry level -4 after test execution	10	100	yes	
No of bugs with severity level -1 after test execution	10	100	No	done
No of bugs with severity level -2 after test execution	10	100	No	
No of bugs with severity level -3 after test execution	10	100	yes	
No of bugs with severity level -4 after test execution	10	100	yes	done
% of requirements covered			No	
% of test cases executed			No	
% of test execution time left with			No	
% of money/fund left with	\$	\$	yes	
Planned deliverables are ready ?	yes	yes	yes	
application documentation has been completed and is upto date	yes	yes	yes	
all types of test applied?	yes	yes	No	
Regression testing completed?	yes	yes	yes	
Business Requirement Document complete				
Program Required Documents complete				
Functional Specification				
Software Development Project Plan approved				
Software Verification Plan approved	yes	yes	yes	
Quality Assurance Plan approved				
Unit test has been passed				
The module resides in a baseline directory				
The project is code-complete. There are no missing features or media elements	No	No	No	
Internal documentation has been updated to reflect the current state of the product.	No	No	No	
The product satisfies the performance and memory requirements specified by the Functional Spec.	yes	yes	yes	
Software/Curriculum have run and passed a Sanity Test that has been provided by the QA group.	No	No	No	
The Integration engineer has tested for install ability	No	No	No	

Figure 6. Snapshot of the Exit Criteria Template

7. Fault Attack

Exploratory testing is extra suitable if requirements and specifications are incomplete, or if there is lack of time. The approach can also be used to verify that previous testing has found the most important defects. It is common to perform a combination of exploratory and scripted testing where the choice is based on risk.

The main advantage of exploratory testing is that less preparation is needed, important bugs are found fast, and is more intellectually stimulating than scripted testing. Another major benefit is that testers can use deductive reasoning based on the results of previous results to guide their future testing on the fly.

They do not have to complete a current series of scripted tests before focusing in on or moving on to exploring a more target rich environment. This also accelerates bug detection when used intelligently.

Another benefit is that, after initial testing, most bugs are discovered by some sort of exploratory testing. This can be demonstrated logically by stating, "Programs that pass certain tests tend to continue to pass the same tests and are more likely to fail other tests or scenarios that are yet to be explored." A structured approach to the error guessing technique is to enumerate a list of possible errors and to design tests that attack these errors. This systematic approach is called *fault attack*.

Fault attack technique is completely based upon the tester & project/product under testing. SAPTHA helps to capture the thinking of a tester, so that it can act as a guideline to some other tester. In this tab by default around two templates are available covering installation and user interface testing. Since this tab acts mostly like a diary to a tester, whenever a tester finds a new scenario, he can update the same over here. Thus it also acts as a lesson learned document, which is more valuable in present software testing industry.


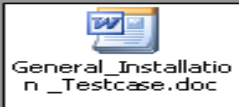
Project:	Calculator	Author:	Sharaniya
Date of Last Review:	10/12/2007	Reviewed By:	QA Team
Test Type	Document Source	Tester name	Description\Comments
User Interface	 checklist_UI.pdf	Sharaniya	
Installation	 General_Installation_Testcase.doc	Sharaniya	

Figure 7. Snapshot of the Fault Attack Template

8. Bi-Directional Traceability Matrix

Requirements traceability is the capacity to relate our requirements to one another and to aspects of other project artifacts. Its primary goal is to ensure that all of the requirements identified by our stakeholders have been met and validated. In my experience, traceability is a great idea in a small minority of situations, but for most project teams, it becomes a bureaucratic quagmire whose costs far outstrip its purported benefits.

Vertical traceability identifies the origin of items (for example, customer needs) and follows these items as they evolve through your project artifacts, typically from requirements to design, the source code that implements the design, and the tests that validate the requirements.

Horizontal traceability identifies relationships among similar items, such as between requirements or within our architecture. This enables our team to anticipate potential problems, such as two sub teams implementing the same requirement in two different components. Traceability is often maintained bidirectional: we should be able to trace from our requirements to our tests and from our tests back to our requirements.

As one can imagine, maintaining a traceability matrix is a lot of work; I've seen on traditional teams in which one or more people have been assigned to this task alone. These teams need one or more designated people for this work—a meager staff has little chance of keeping it up-to-date. An out-of-date matrix is worse than having no matrix at all because it contains erroneous information upon which invalid decisions will be based.

SAPTHA – was designed by taking care about the above concepts. SAPTHA concentrates mainly on agility in testing, i.e. within minimum time we should ensure that traceability is done for the requirements along with the corresponding testcases and also to do impact analysis if a corresponding testcase has been changed.

Whenever a tester key in a requirement, a corresponding *unique ID* has to be entered for the same, and then he can map the testcase (if any) to the added requirement. With unique ID we can easily trace the requirement to a particular testcase & vice versa and thus it eases the process of bi-directional traceability.

As we all know that for regression testing, based upon the changes the test cases will be selected, obviously a tester need to perform impact analysis to get the affected testcases. SAPTHA has a provision to mark the dependency, which user can update while he/she is logging in a testcase. For performing impact analysis, user has to enter unique ID of the requirement then SAPTHA will display the set of the testcases that need to be executed, which included the direct testcases and also the implicit testcases which shall be identified from the dependency provision

SAPTHA – does not handle version control for the requirements. Instead a column is provided to capture the same.

Project:	calculator	Author:	Sharaniya																		
Date of Last Review:	17/12/2007	Reviewed By:	QA Team																		
Unique Number	Requirement_Number	Other Tags covering similar Requirements (Ex: Functional Tag to Performance Requirements Tag or Safety Requirements Tag, etc)	Source of Requirement	Design Phase	Coding Phase	Testing Phase	Releases	Product Release Guide	Modification of Requirement	Status	Dependency										
					Source code	Test case/report document	User Document	Maintenance Document													
					Module/Function	Section No	Increment No	Section No													
Calc_001	Calc_Req_001		Calc_SRS sec 10.1	Calc_Add_number	Calc_Add_no_09	Calc_no_09_valid	Calc_Beta_01														
						Calc_no_09_invalid															
						Calc_no_09_extreme															
						Calc_no_09_bound															
						Calc_no_09_empty															
						Calc_Add_no_10	Calc_no_10_valid	Calc_Beta_01													
						Calc_no_10_invalid															
						Calc_no_10_extreme															
						Calc_no_10_bound															
						Calc_Add_no_11	Calc_no_11_bound	Calc_Beta_01													
						Calc_Add_no_12	Calc_no_12_bound	Calc_Beta_01													
							Calc_no_12_valid														
							Calc_no_12_invalid														
						Calc_Add_no_13	Calc_no_13_extreme	Calc_Beta_01													

Unique number for req identification

Mapping between req & testcases

Captures updates in req (if any)

Mapping between requirements

Figure 8. Snapshot of Bi-Directional Traceability Matrix

Unique Number	Requirement_Number	Other Tags covering similar Requirements (Ex: Functional Tag to Performance Requirements Tag or Safety Requirements Tag, etc)	Source of Requirement	Design Phase	Coding P																
					Source code	Test case/report document	User Document														
					Module/Function	Section No	Increment No														
Calc_001	Calc_Req_001		Calc_SRS sec 10.1	Calc_Add_number	Calc_Add_no_09	Calc_no_09_valid	Calc_Beta_01														
						Calc_no_09_invalid															
						Calc_no_09_extreme															
						Calc_no_09_bound															
						Calc_no_09_empty															
						Calc_Add_no_10	Calc_no_10_valid	Calc_Beta_01													
						Calc_no_10_invalid															
						Calc_no_10_extreme															
						Calc_no_10_bound															
						Calc_Add_no_11	Calc_no_11_bound	Calc_Beta_01													
						Calc_Add_no_12	Calc_no_12_bound	Calc_Beta_01													
							Calc_no_12_valid														
							Calc_no_12_invalid														
						Calc_Add_no_13	Calc_no_13_extreme	Calc_Beta_01													
						Id_decimal	Calc_de_09_extreme	Calc_Beta_01													
							Calc_de_09_bound														
Calc_002	Calc_Req_002																				
Calc_003	Calc_Req_003				Calc_Add_integer	Calc_in_09	Calc_in_09_invalid														
Calc_004	Calc_Req_004				Calc_Add_zero	Calc_Add_ze_09	Calc_ze_09_extreme														
							Calc_ze_09_bound														
							Calc_ze_09_empty														

Impact analysis – displays all the testcases affected

- a. Implicit – corresponding testcases
- b. Explicit – based upon dependency

Impact analysis for req – Calc_req_001 (affects calc_req_002 also)

Figure 9. Bi-Directional Traceability Matrix – Impact Analysis

Citations

- **Book**

Rick Craig and Stefan Jaskiel: 2002. *Systematic Software Testing*. Artech House Publishing.

Cem Kaner, James Bach, and Bret Pettichord: 2002. *Lessons Learned in Software Testing*. Wiley