Dynamic Test Case Design Scenario and analysis of Module Testing Using Manual vs. Automated Technique

[1]Er. RAJENDER KUMAR, [2]Dr. M.K.GUPTA

[1]PH.D RESEARCH SCHOLAR DEPTT OF COMPUTER SCIENCE, CCSU (INDIA)
[2]DEPTT. OF COMPUTER SCIENCE & MATHEMATICS CCSU (INDIA)

Abstract
Software can be tested either manually or automatically. The two approaches are complementary: automated testing can perform a huge number of tests in short time or period, whereas manual testing uses the knowledge of the testing engineer to target testing to the parts of the system that are assumed to be more error-prone. Despite this contemporary, tools for manual and automatic testing are usually different, leading to decreased productivity and reliability of the testing process. AutoTest is a testing tool that provides a "best of both worlds" strategy: it integrates developers' test cases into an automated process of systematic contract-driven testing. This allows it to combine the benefits of both approaches while keeping a simple interface, and to treat the two types of tests in a unified fashion: evaluation of results is the same, coverage measures are added up, and both types of tests can be saved in the same format. The objective of this paper is to discuss the Importance of Automation tool with associate to software testing techniques in software engineering. In this paper we provide introduction of software testing and describe the CASE tools. The solution of this problem leads to the new approach of software development known as software testing in the IT world. Software Test Automation is the process of automating the steps of manual test cases using an automation tool or utility to shorten the testing life cycle with respect to time.

Keywords Module testing, Test Case Design, Software testing of Manual and automated.

1. Introduction

Software testing is the process of executing a program with the intention of finding errors in the code. It is the process of exercising or evaluating a system or system component by manual automatic means to verify that it satisfies specified requirements or to identify differences between expected and actual results [1]

Software Testing should not be a distinct phase in System development but should be applicable throughout the design development and maintenance phases. 'Software Testing is often used in association with terms verification & validation 'Software testing is the process of executing software in a controlled manner, in order to answer the question: Does the software behave as specified. One way to ensure system's responsibility is to extensively test the system. Since software is a system component it requires a testing process also. The main contribution of this paper lies in the mechanisms that we provide to integrate the manual and automated testing strategies. This integration has the following advantages:

The overall testing process benefits from the strengths of both manual and automated testing;

Support for regression testing: any automatically generated tests that uncover bugs can be saved in the same format as manual tests and stored in a regression testing database;[2]

The measures of coverage (code, dataflow, specification) will be computed for the manual and automated tests as a whole;

association with terms verification & validation 'Software testing is the process of executing software in a controlled manner, in order to answer the question: Does the software behave as specified. One way to ensure system's responsibility is to extensively test the system. Since software is a system component it requires a testing process also. The main contribution of this paper lies in the mechanisms that we provide to integrate the manual and automated testing strategies. This integration has the following advantages:

The overall testing process benefits from the strengths of both manual and automated testing;

Support for regression testing: any automatically generated tests that uncover bugs can be saved in the same format as manual tests and stored in a regression testing database;[2][3]

The measures of coverage (code, dataflow, specification) will be computed for the manual and automated tests as a whole;

2 Testing strategies

In this section we introduce the two strategies unified by our tool, manual testing and automated testing, then an analysis of the advantages and disadvantages of each, and the rationale for integrating them.

2.1 Unit Testing

Unit testing is code-oriented testing. Individual components are tested to ensure that they operate correctly. Each component is tested independently, without other system components'

2.2 Module Testing

A module is a collection of dependent components such as an object, class, an abstract data type or some loser collection of procedures and functions. A module encapsulates related components so it can be tested or checked without other system modules.

2.3 Sub-system Testing

This phase involves testing collections of modules, which have been integrated in to sub systems. It is a design-oriented testing and is also known as integration testing.

Fig: Fi
prevents

2.4 System Testing

The sub-systems are integrated to make up the entire system. It is also concerned with validating that the System meets its functional and non-functional requirements. [4].

2.5 Acceptance testing

This is the final stage in the testing process before the system is accepted for operational use. Acceptance testing may also reveal requirement problems where the system facilities do not really meet the user's needs [5] "Let us see there are many problems if we test to the above mentioned software testing techniques using manual testing rather automated tools".

3 Proposed Module Testing

During unit testing of C programs, a single C-level function is tested rigorously and in isolation from the rest of the application. Often unit testing is also called module testing. *Rigorous* means that the test cases are specially made for the unit in question and that they comprise of input data that may be unexpected by the unit under test. *Isolated* means that the test result does not depend on the behavior of the other units in the application. It can be achieved by directly calling the unit under test and replacing calls to other units by stub functions. [6]

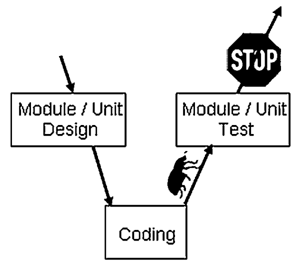


Fig: 1. Module Testing eliminates errors early on and them from showing up in later stages of the development process

3.1 What are the Benefits of Module Testing

3.1.1 Reduces Complexity of Test Case Specification

Instead of trying to create test cases that test the whole set of interacting units, the test cases for unit testing are specific to the unit under test (*divide-and-conquer*). Test cases can easily comprise of input data that is unexpected by the unit under test, something which may be hard to achieve during system testing.[5]

3.1.2 Easy Fault Isolation

If the unit under test is tested in isolation from the other units, detecting the cause of a failed test case is easy. The fault must be related to the unit under test, and not to a unit further down the calling hierarchy.[6]

3.1.3 Finds Errors Early

Unit testing can be conducted as soon as the unit to be tested compiles successfully. Therefore errors inside the unit can be detected very early.

3.1.4 Saves Money

It is generally accepted that errors detected late in a project are more expensive to correct than errors that are detected early. Hence unit testing saves money.

3.1.5 Gives Confidence

Unit testing gives confidence. After the unit testing, the application will be made up of single, fully tested units. A test for the whole application will then be more likely to pass. Module/Unit concentrates verification on the smallest element of the program – the module. Using the detailed design description important control paths are tested to establish errors within the bounds of the module. The tests that are performed as part of unit testing are shown in the figure below. The module interface is tested to ensure that information properly flows into and out of the program unit being tested. The local data structure is considered to ensure that data stored temporarily maintains its integrity for all stages in an algorithm's execution. Boundary conditions are tested to ensure that the modules perform correctly at boundaries created to limit or restrict processing. All independent paths through the control structure are exercised to ensure that all statements in been executed once. Finally, all error-handling paths are examined. [7] [9]

4 Module Testing Analysis

Module testing is code-oriented testing. Individual components are tested to ensure that they operate correctly. Each component is tested independently, without other system components. A unit test is a piece of code written by a developer that exercises a very small, specific area of functionality in the code being tested. Usually a unit test exercises some particular method in a particular context. For example, you might add a large value to a sorted list, then confirm that this value appears at the end of the list. [7][9]

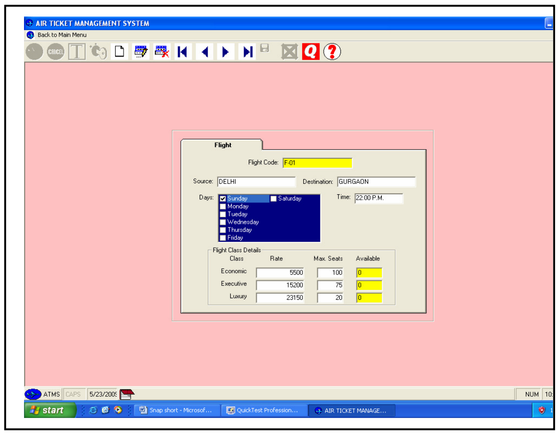
Module Testing = Unit Testing

- **Large programs cannot practically be tested all at once**
- **Break down programs into modules**
- **Test modules individually as first phase**

Fig .2 Module Test Structure

Fig: 2 Structure Module Testing

5. Example of Module Test for Airlines application System



AIR TICKET MANAGEMENT SYSTEM
Back to Main Menu

Flight

Flight Code: F-01

Source: DELHI Destination: GURGAON

Days: [✓ Sunday] Saturday Monday Tueday Wednesday Thursday Friday

Time: 22:00 P.M.

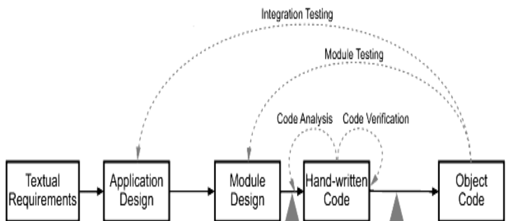
Flight Class Details

Class	Rate	Max. Seats	Available
Economic	5500	100	0
Executive	15200	75	0
Luxury	23150	20	0

Fig: 3. Air Ticket Management System.

5.1 Description: This is Airlines Ticket mgmt system i.e. complete module. In which researcher categorized to the module part e.g. Airlines flight Unit, Airlines Reservation Unit system. By this module system, no doubt testing is done easily rather test to complete system. Because module tests are performed to prove that a piece of code does what the developer thinks it should be done. These module is compared by manually or Automated tool i.e. QTP.

5.2. Description: This Module is show to the Airline Flight Categories System. In this unit each flight class details are mentioned e.g. economic class, executive class, luxury class etc.



5.3. Description: This Unit is show to the Airline Flight Categories System. In this Unit flight code is mentioned and validation and check point is given in the flight class details i.e. economic, executive, luxury e.g. economic class traveling rate under range 12000-18000, executive class rate is not less than 5000 or not more than 10000 rate, luxury rate 12000 to 18000 also.

Test Case Name	Test Case Describe	Test Steps			Test Case Status (p/f)
		STEPS	**EXPECTED Result**	**Actual Result**	
Economic Rate	Economic rate should be with in 5000-10000	1) <5000	Not Accepted	The input is accepted by the text box	Fail
		2)5000-6000	Accepted	The input is accepted by the text box	Pass
		3)6001-7000	Accepted	The input is accepted by the text box	Pass
		4)7001-8000	Accepted	The input is accepted by the text box	Pass
		5)8001-9000	Accepted	The input is accepted by the text box	Pass
		6)9001-10000	Accepted	The input is accepted by the text box	Pass
		7)>10000	Not Accepted	The input is accepted by the text box	Fail

Table: 1. Test Cases with approach of Equivalence Class Partitioning:

6. What is Test Case Design

A test case in software engineering is a set of conditions or variables under which a tester will determine whether an application or software system is working correctly or not. The mechanism for determining whether a software program or system has passed or failed such a test is known as a test oracle. In some settings, an oracle could be a requirement or use case, while in others it could be a heuristic. It may take many test cases to determine that a software program or system is functioning correctly. Test cases are often referred to as test scripts, particularly when written. Written test cases are usually collected into test suites[10]

Typical written test case format

A test case is usually a single step, or occasionally a sequence of steps, to test the correct behavior/functionalities, features of an application. An expected result or expected outcome is usually given.

Additional information that may be included:

- test case ID
- test case description
- test step or order of execution number
- related requirement(s)
- depth
- test category
- author
- Check boxes for whether the test is automatable and has been automated.

Additional fields that may be included and completed when the tests are executed:

- pass/fail

remarks

6.1 What are the types of Test case design Technique

There are two types of test case design techniques they are
1. Equivalence class partition.
2. Boundary value analysis
Equivalence class partition: here the test engineer writes the valid and invalid test cases i.e. positive test cases and negative test cases.
Boundary value analyses: if there is a range kind of input the technique used by the test engineer to develop the test Cases for that range are called as boundary value analyses.

6.1.1 Equivalence Class Partitioning:

Concepts: Equivalence partitioning is a method for deriving test cases. In this method, classes of input conditions called equivalence classes are identified such that each member of the class causes the same kind of processing and output to occur. In this method, the tester identifies various equivalence classes for partitioning. A class is a set of input conditions that are is likely to be handled the same way by the system. If the system were to handle one case in the class erroneously, it would handle all cases erroneously.[11]
Designing Test Cases Using Equivalence Partitioning

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

 Identify the equivalence classes
 Design test cases
Step 1: Identify Equivalence Classes

Take each input condition described in the specification and derive at least two equivalence classes for it. One class represents the set of cases which satisfy the condition (the valid class) and one represents cases which do not (the invalid class) Following are some general guidelines for identifying equivalence classes:
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. For example, if an item in inventory can have a quantity of - 9999 to + 9999, identify

The following examples of classes:

1. one valid class: (QTY is greater than or equal to -9999 and is less than or equal to 9999). This is written as (- 9999 < = QTY < = 9999)

2. the invalid class (QTY is less than -9999), also written as (QTY < -9999)
3. the invalid class (QTY is greater than 9999) , also written as (QTY >9999)

b) 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.

Test Case Name	Test Case Describe	Test Steps			Test Case Status (p/f)
		STEPS	**EXPECTED Result**	**Actual Result**	
Economic Rate	Economic rate should be with in 5000-10000	1) 4000	Not Accepted	The input is accepted by the text box	Fail
		2) 5000	Accepted	The input is accepted by the text box	Pass
		3) 10000	Accepted	The input is accepted by the text box	Pass
		4) 11000	Not Accepted	The input is accepted by the text box	Fail

Table: 2. Test Cases with approach of Boundary Value Analysis

6.1.2 Module for with Boundary Value Analysis

It is a software testing design technique in which tests are designed to include representatives of boundary values. The expected input and output values should be extracted from the component specification. The input and output values to the software component are then grouped into sets with identifiable boundaries. Each set, or partition, contains values that are expected to be processed by the component in the same way. Partitioning of test data ranges is explained in the equivalence partitioning test case design technique. It is important to consider both valid and invalid partitions when designing test cases.[12]

For an example where the input values were months of the year expressed as integers, the input parameter 'month' might have the following partitions:

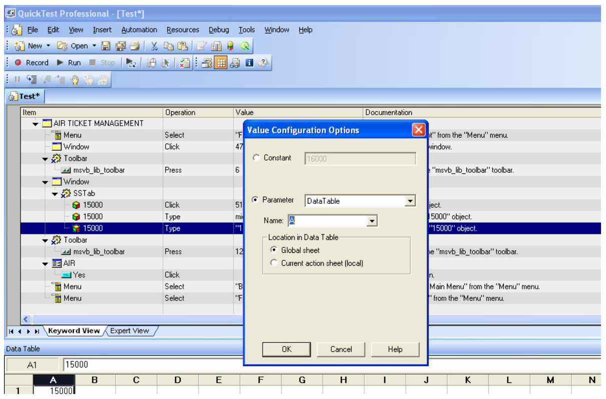
```
     ... -2 -1  0 1 .............. 12 13 14 15.....
  ---------------|------------------|---------------------
 invalid partition 1   valid partition    invalid partition
```

The boundaries are the values on and around the beginning and end of a partition. If possible test cases should be created to generate inputs or outputs that will fall on and to either side of each boundary. This would result in three cases per boundary. The test cases on each side of a boundary should be in the smallest increment possible for the component under test. In the example above there are boundary values at 0,1,2 and 11,12,13. If the input values were defined as decimal data type with 2 decimal places then the smallest increment would be the 0.01.

Where a boundary value falls within the invalid partition the test case is designed to ensure the software component handles the value in a controlled manner. Boundary value analysis can be used throughout the testing cycle and is equally applicable at all testing phases.[13][14]

After determining the necessary test cases with equivalence partitioning and subsequent boundary value analysis, it is necessary to define the combinations of the test cases when there are multiple inputs to a software component.

7. Airlines Module Tested using Automated Tool (QTP)



QuickTest Professional - [Test*]

Item	Operation	Value	Documentation
AIR TICKET MANAGEMENT			
Menu	Select	"F	t" from the "Menu" menu.
Window	Click	47	window.
Toolbar			
msvb_lib_toolbar	Press	6	e "msvb_lib_toolbar" toolbar.
Window			
SSTab			
15000	Click	51	ject.
15000	Type	mi	15000" object.
15000	Type	"1	"15000" object.
Toolbar			
msvb_lib_toolbar	Press	12	e "msvb_lib_toolbar" toolbar.
AIR			
Yes	Click		n.
Menu	Select	"B	Main Menu" from the "Menu" menu.
Menu	Select	"F	" from the "Menu" menu.

Value Configuration Options
Constant: 16000
Parameter: DataTable
Name: A
Location in Data Table: Global sheet [✓] / Current action sheet (local)

Data Table
A1: 15000

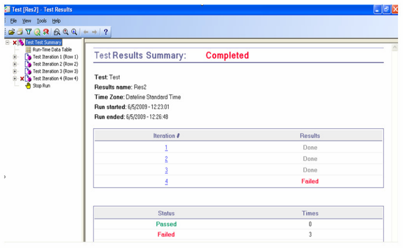
Fig: 4. Parameterized Testing for Airline module

7. Description: In which I have taken value in the parameter and test with Data Table that which showed to the conditions e.g. mentioned 15000,16000,18000,20000 as I had implemented validation on flight class unit. Suppose if I take <10000 and >18000 value then it would show the failed result in the last rate value and first three values will be done.

7.1 Airlines Module running using with QTP Testing Tool

Fig: 4. QTP Tool using on Airline Module

7.1 Description: This window is running the conditioned Data table as mentioned 15000,16000,18000,20000 as I had implemented validation on flight class unit. Suppose if I take <10000 and >18000 value then it would show the failed result in the last rate value and first three values will be done.



Test [Res2] - Test Results

Test Results Summary: Completed

Test: Test
Results name: Res2
Time Zone: Dateline Standard Time
Run started: 6/5/2009 - 12:23:01
Run ended: 6/5/2009 - 12:26:48

Iteration #	Results
1	Done
2	Done
3	Done
4	Failed

Status	Times
Passed	0
Failed	3

Fig: 5. Testing results of Airlines Module.

Description: This test results summary is showing the actual result that is first three values are right e.g. 15000, 18000, 12000 that have tested and done and the last value is wrong that has failed e.g. 20000

8. Comparative Graph of Manual Vs Automated Testing

Release	Manual Test	Auto Test	Manual Test Cumulative
1	10	10	10
2	10	0	20
3	10	0	30
4	10	0	40
5	10	0	50

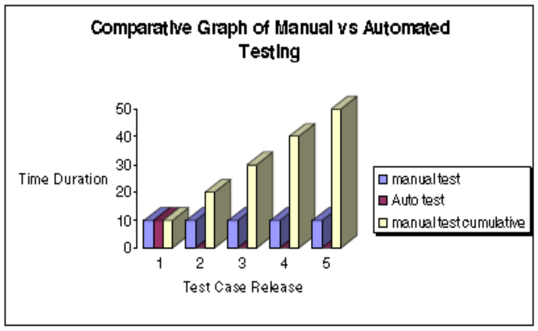


Fig: 6. Comparative Graph of Manual Vs Automated Testing

8.1 Description: This chart showing the comparative results of Manual Vs Automated Testing blue line is indicating to the manual testing and red line indicating to the automated testing and yellow line shows to the Manual Test Cumulative. The time duration is mentioned 0 to 50 and total test cases release is 1 to 5. by this chart we can understand if one test case has be released and time in manual testing assigned i.e 10 minutes and same assigned in Automated Testing Suppose if again test case is to be release the manual testing will assume time 10 minute but in the case of Automated testing time will assume second the zero minutes

9. Comparative Study of Manual vs Automated Testing

Manual Testing is time consuming.

a) There is nothing new to learn when one tests manually.
b) People tend to neglect running manual tests.
c) None maintains a list of the tests required to be run if they are manual tests.
d) Manual Testing is not reusable.
e) Tests have to be repeated by each stakeholder for e.g. Developer, Tech Lead, GM, and Management.
f) Manual Testing ends up being an Integration Test.
g) In a typical manual test it is very difficult to test a single unit.
h) Scripting facilities are not in manual testing.[1]

Automated testing with Quick Test addresses these problems by dramatically speeding up the testing process. You can create tests that check all aspects of your application or Web site, and then run these tests every time your site or application changes. [13]

Fast : Quick test runs tests significantly faster than human user.

Reliable: Tests perform precisely the same operations each time they are run, thereby eliminating human error.

Programmable: You can program sophisticated tests that bring out hidden information.

Comprehensive: you can build a suite of tests that covers every feature in your web site or application.

Reusable: You can build a suite of tests that covers every feature in your website or application.

9. A Cost Model Based Analysis

Building on the example from the previous section, we propose an alternative cost model drawing from linear optimization. The model uses the concept of opportunity cost to balance automated and manual testing. The opportunity cost incurred in automating a test case is estimated on basis of the lost benefit of not being able to run alternative manual test cases. Hence, in contrast to the simplified model presented in Section 2, which focuses on a single test case, our model takes all potential test cases of a project into consideration. Henceforth, it optimizes the investment in automated testing in a given project context by maximizing the benefit of testing rather than by minimizing the costs of testing.[7]

9.1 Fixed Budget

First of all, the restriction of a fixed budget has to be introduced to our model. This restriction corresponds to the production possibilities frontier described in the previous section. R1: $na * Va + nm * Dm \leq B$ *na := number of automated test cases nm := number of manual test executions Va := expenditure for test automation Dm := expenditure for a manual test execution B := fixed budget* Note that this restriction does not include any fixed expenditures (e.g., test case design and preparation) manual testing. Furthermore, with the intention of keeping the model simple, we assume that the effort for running an automated test case is zero or negligibly low for the present. This and other influence factors (e.g., the effort for maintaining and adapting automated tests) will be discussed in the next section. This simplification, however, reveals an important difference between automated and manual testing. While in automated testing the costs are mainly influenced by the number of test cases (*na*), manual testing costs are determined by the number of test executions (*nm*). Thus, in manual testing, it does not make a difference whether we execute the same test twice or whether we run two different tests. This is consistent with manual testing in practice – each manual test execution usually runs a variation of the same test case [6]

9.2 Benefits and Objectives of Automated and Manual Testing

Second, in order to compare two alternatives based on opportunity costs, we have to valuate the benefit of each alternative, i.e., automated test case or manual test execution. The benefit of executing a test case is usually determined by the information this test case provides. The typical information is the indication of a defect. Still, there are additional information objectives for a test case (e.g., to assess the conformance to the specification). All information objectives are relevant to support informed decisionmaking and risk mitigation. A comprehensive discussion about what factors constitute a good test case is given in [13].

9.3 Maximizing the Benefit

Third, to maximize the overall benefit yielded by testing, the following target function has to be added to the model. T: *Ra(na)* + *Rm(nm)* □ *max* Maximizing the target function ensures that the combination of automated and manual testing will result in an optimal point on the production possibilities frontier defined by restriction *R1*. Thus, it makes sure the available budget is entirely and optimally utilized.

9.4 Real Example

To illustrate our approach we extend the example used in Section 3. For this example the restriction *R1* is defined as follows. R1: $na * 1 + nm * 0.25 \leq 75$ To estimate benefit of automated testing based on the risk exposure of the tested object, we refer to the findings published by Boehm and Basili [5]: "Studies from different environments over many years have shown, with amazing consistency, that between 60 and 90 percent of the defects arise from 20 percent of the modules, with a median of about 80 percent. With equal consis- tency, nearly all defects cluster in about half the modules produced." Accordingly we categorize and prioritize the test cases into 20 percent highly beneficial, 30 percent medium beneficial, and 50 percent low beneficial and model following alternative restrictions to be used in alternative scenarios. R2.1: $na \geq 20$ R2.2: $na \geq 50$ To estimate the benefit of manual testing we propose, for this example, to maximize the test coverage. Thus, we assume an evenly distributed risk exposure over all test cases, but we calculate the benefit of manual testing based on the number of completely tested releases. Accordingly we categorize and prioritize the test executions into one and two or more completely tested releases. We model following alternative restrictions for alternative scenarios. R3.1: $nm \geq 100$ R3.2: $nm \geq 200$ Based on this example we illustrate three possible scenarios in balancing automated and manual testing. Figures 4a, 4b and 4c depict the example scenarios graphically.

• **Scenario A** – The testing objectives in this scenario are, on the one hand, to test at least one release completely and, on the other hand, to test the most critical 50 percent of the system for all releases. These objectives correspond to the restrictions *R3.1* and R2.2 in our example model. As shown in Figure 4a the optimal solution is point *S1* (*na* = 50, *nm* = 100) on the production possibilities frontier defined by *R1*. Thus, the 50 test cases referring to the most critical 50 percent of the system should be automated and all test cases should be run manually once.

• **Scenario B** – The testing objectives in this scenario are, on the one hand, to test at least one release completely and, on the other hand, to test the most critical 20 percent of the system for all releases. These objectives correspond to the restrictions *R3.1* and *R2.1* in our example model. As shown in Figure 4b any point within the shaded area fulfills these restrictions. The target function, however, will make sure that the optimal solution will be a point between *S1* (*na* = 50, *nm* = 100) and *S2* (*na* = 20, *nm* = 220) on the production possibilities frontier defined by *R1*. Note: While all points on *R1* between the *S1* and *S2* satisfy the objectives of this scenario, the point representing the optimal solution depends on the definition of the contribution to risk mitigation of automated and manual testing, *Ra(na)* and *Rm(nm)*.

• **Scenario C** – The testing objectives in this scenario are, on the one hand, to test at least two releases completely and, on the other hand, to test the most critical 50 percent of the system for all releases. These objectives correspond to the restrictions *R3.2* and *R2.2* in our example model. As shown in Figure 4c a solution that satisfies both restrictions cannot be found.

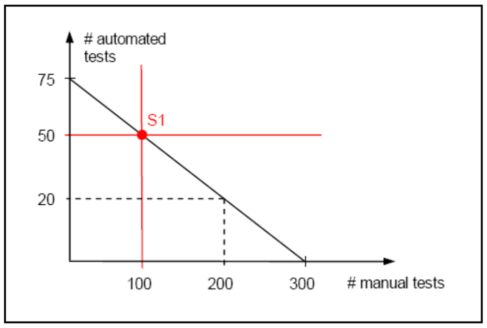


Figure 7: Scenario of Auto vs. Manual A.

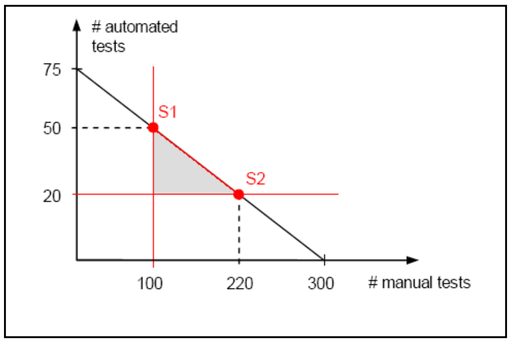


Figure 8: Scenario of Auto vs. Manual B.

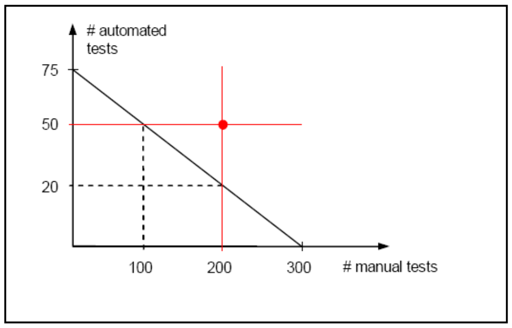


Figure9: Scenario of Auto vs. Manual C.

9. Conclusion

The Conclusion of this research and review paper is analyze to the manual testing drawback in software testing rather more benefits of automated software testing tools. The enlightened of this modern approaches leads to the new Methodologies of software test automation. The destination of software testing is considered to succeed when an error is detached. Effective Conclusions are given below. Software testing is an art. Most of the testing methods and practices are not very different from 20 years ago. In the current era there are many tools and techniques available to use. Good testing also requires a tester's creativity, experience and intuition, together with proper techniques. Testing is more than just debugging. Testing is not only used to locate defects and correct them. It is also used in validation, verification process, and reliability measurement. Although manual testing is not expensive but is no more effective rather automated testing because automation is a good way to cut down cost and time. Testing efficiency and effectiveness is the criteria for coverage-based testing techniques.

10. REFERENCES

[1] Leckraj Nagowah and Purmanand Roopnah, "AsT -A Simp le Automated System Testing Tool", IEEE, 978-1-4244- 5540-9/10, 2010.

[2] Alex Cerv antes, "Exploring the Use of a Test Automation Framework", IEEEAC p ap er #1477, version 2, up dated January 9, 2009.

[3] A. Ieshin, M. Gerenko, and V. Dmitriev, *"Test Automation- Flexible Way",* IEEE, 978-1-4244-5665-9, 2009.

[4] Boehm, B., *Value-Based Software Engineering: Overview and Agenda.* In: Biffl S. et al.: Value-Based Software Engineering. Springer, 2005.

[5] Schwaber, C., Gilpin, M., *Evaluating Automated Functional Testing Tools*, Forrester Research, February 2005.

[6] Ramler R., Biffl S., Grünbacher P., *Value-based Management of Software Testing.* In: Biffl S. et al. Value-Based Software Engineering. Springer, 2005.
[7] M.Grechanik, q. Xie, and Chen Fu, "*Maintaining and Evolving GUI- Directed Test Scripts*", IC SE'09, IEEE, Vancouver, Canada, 978-1-4244-3452-7, May 16-24, 2009.

[8] Khaled M.Mustafa, Rafa E. Al-Qutaish, Mohammad I. Muhairat, "*Cassification of Software testing Tools Based on the Software Testing Methods*", 2009 second International Conference on Computer and Electrical Engineering, 978-0-7695-3925-6, 2009.

[9] R.S.Pressman, " Software Engineering A Practitioner's Approach", Mcgraw-Hill International Edition, ISBN 007-124083-7.

[10] D. Marinov and S. Khurshid, "TestEra: A Novel Framework for Automated Testing of Java Programs," in *Proc.~16th IEEE International Conference on Automated Software Engineering (ASE)*, 2001, pp. 22-34

[11] P. Tonella, "Evolutionary testing of classes," in *International symposium on Software testing and analysis (ISSTA'04)*. Boston, Massachusetts, USA: ACM Press, 2004, pp. 119-128.

[12] N. K. Patrice Godefroid, Koushik Sen, "DART: directed automated random testing," presented at PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, 2005.

[13] Dustin, E. et. al., *Automated Software Testing*, Addison- Wesley, 1999.

[14] Fewster, M., Graham, D., *Software Test Automation: Effective Use of Text Execution Tool*, Addison- Wesley, 1999.f

FIRST AUTHOR

Er. Rajender Kumar Ph.D Research Scholar in the deptt of Computer Science & Mathematics from CCS University, India .His area of specialization in Software Testing. He did completed his master degree M.Tech from M.M.University

Mullana in 2009. Presently working as Asstt Prof in Computer Sc & Engg Deptt at HIET Kaithal. He has more than 40 research papers in reputed conferences and journals.

SECOND AUTHOR

Dr. M.K.Gupta is working as Professor in the Deptt of Mathematics & Computer science at CCS University. He did complete his doctorate degree in 1998 in the area of Mathematics Science . He has more than 50 research papers in a reputed journals. He got completed more than 36 candidates of M.Phil and more than 7 candidates of Ph.D degree under his supervision.