Impact Factor (SJIF): 3.632

International Journal of Advance Research in Engineering, Science & Technology

e-ISSN: 2393-9877, p-ISSN: 2394-2444

(Special Issue for ITECE 2016)

A Brief Comparison Between An Automated Testing And Manual Testing

Dikshita Viradia[1], Payal Bhatt[2], Payal Undhad[3]

1, Computer Department, B.H Gardi College Of Engineering And Technology
2, Computer Department, B.H Gardi College Of Engineering And Technology
3, Computer Department, B.H Gardi College Of Engineering And Technology

Abstract-- ***Software testing is a huge domain, but it can be broadly categorized into two areas which are manual testing and automated testing. Testing is an integral part of any successful software project. The type of testing (manual or automated) depends on various factors like including project requirements, budget, timeline, expertise, and suitability. Three vital factors of any project are of course time, cost, and quality – the goal of any successful project is to reduce the cost and time required to complete it successfully while maintaining quality output. When it comes to testing, one type may accomplish this goal better than the other. Both manual and automated testing offers benefits and disadvantages. It is worth knowing the difference and when to use one or the other for best results. This paper discusses the difference between both technique and how these techniques are implemented.***

Keywords: Software Testing, Manual Testing, Automated Testing, Difference between manual and automated testing, Software Quality

I. INTRODUCTION

In manual testing (as the name suggests), test cases are executed manually (by a human, that is) without any support from tools or scripts. But with automated testing, test cases are executed with the assistance of tools, scripts, and software. Manual Software Testing is the process of going in and running each individual program or series of tasks and comparing the results to the expectations in order to find the defects in the program. Automated software testing uses automated tools to run tests based on algorithms to compare the developing program's expected outcomes with the actual outcomes.

There are many manual testing types which are carried out manually as well as automatically. (1)Black Box Testing – It is a testing method to test functionalities and requirements of the system. It does not test the internal part of the system. (2)White Box Testing – It is a testing method based on information of the internal logic of an application's code and also known as Glass box Testing. It works on Internal working code of the system. Tests are based on coverage of code statements, branches, paths, conditions. (3)Integration Testing – Integrated modules testing method to verify joint functionality after integration. Modules are typically code modules, individual applications, client and server applications on a network, etc. This type of testing is especially applicable to client/server and distributed systems. (4)System Testing – It is a technique to test whole system. (5)Unit Testing – Testing method to test specific component of software or module. It is specially done by programmers and not by testers, because it needs thorough knowledge of the internal programming design and code.(6)Acceptance Testing –This type of testing verifies that the system meets the customer specified requirements or not. User or a customer does this testing to decide whether to accept application.

Manual Testing is a start of Testing, without this testing we can't start Automation Testing. The development of large software products involves many activities that need to be coordinated to meet the desired requirements. Among these tasks, we can distinguish activities that contribute mainly to the construction of the product, and activities that aim at checking the quality of the development process and of produced artifacts. This classification is not sharp, since most activities contribute to some extent to both advancing the development and checking the quality. In some cases, this characterization of activities is not precise enough, but helps identify an important thread of the development process that includes all quality-related activities and is often referred to as the quality process. The quality process is not a phase, but it spans through the whole development cycle: it starts with the feasibility study, and goes beyond product deployment through maintenance and post mortem analysis.

Automation Testing is a continuous part of Manual Testing. Advances in technology, such as networking and parallel processing, have enabled the development of distributed and concurrent systems. Most of these systems consist of subsystems, which can be independent or autonomous. In some cases, there is a need to allow a subsystem, who is sending a message, to continue with its tasks without waiting to determine what happened to the message. All these give the whole system an asynchronous characteristic. Testing and validation of asynchronous systems is challenging. In particular, automated testing tools need to deal with practical implementation challenges. For example, perfect communication channels (without losses or delays) used in the theory is not present in real systems. Therefore, testing tools need to handle systems with imperfect channels. In the same way, if some subsystems rely on external choices, tools for testing those subsystems need to handle non-determinism.

Our practical approach aims to make testing theory accessible to practitioners. To address this goal we bring the system under test (SUT) and the testing tool to work together in an on-line approach. Although this approach is present at some extent in available tools, we have extended capabilities already present in these tools to make this approach to work in black-box testing environments. This is we deal mostly with available interfaces rather than internal workings of the system. Second, we extend existing tools to implement theoretical knowledge on how to handle asynchronous communications. We deal not only with communication delays but also communication losses which are less common in the literature.

The rest of this paper is organised as follows. In section 2 we describe quality process of manual testing which is having functional testing, integration testing and module testing. Then, section 3 describes automated software testing process. Section 4 concludes the paper and identifies relevant problems of both these testing approaches.

II. QUALITY PROCESS OF MANUAL TESTING

The quality process involves many activities that can be grouped in five main classes: planning and monitoring, verification of specifications, test case generation, test case execution and software validation, and process improvement. Planning and monitoring activities aim at steering the quality activities towards a product that satisfies the initial quality requirements.

Test cases are usually generated from specifications, and are integrated with information from the application environment, development technology, and code coverage. The application environment and development technology can provide information on common faults and risks. Test cases might and should be generated as soon as the corresponding specifications are available. Early test case generation has the obvious advantage of alleviating scheduling problems: tests can be generated in parallel with development activities, and thus be excluded from critical paths. Early generation of test cases has also the important side effect of helping validate specifications. Experience shows that many specification errors are easier to detect early than during design or validation.

Testing can reveal many kinds of failures, but may not be adequate for others, and thus it should be complemented with alternative validation activities.

2.1 Functional Testing

Modern general-purpose functional testing approaches include category partition, combinatorial, and catalog-based testing. Category partition testing applies to specifications expressed in natural language and consists of three main steps. We start decomposing the specification into independently testable features: the test designer identifies specification items that can be tested independently, and identifies parameters and environment elements that determine the behavior of the considered feature (categories). For example, if we test a Web application, we may identify the catalog handler functionality as an independently testable feature. The following excerpt of a specification determines the production and the sale of items on the basis of the sales and the stock in the last sale period:

The production of an item is suspended if in the last sales period the number of orders falls below a given threshold t1 or if it falls below a threshold t2 >t 1 and the amount in stock is above a threshold s2. An item is removed from the catalog if it is not in production, the amount of orders in the previous period remains below t1 and the amount in stock falls below a threshold s1 <s 2. The production of an item in the catalog is resumed if the amount of orders in the former period is higher than t2 and the amount in stock is less than s1.

Items that are also sold in combination with other items are handled jointly with the assembled items, i.e., the production is not suspended if one of the assembled items is still in production, despite the sales and the stock of the considered item, and similarly it is kept in the catalog even if eligible for withdraw, if the assembled items are kept in the catalog. The amount in stock cannot exceed the maximum capacity for each item.

From the informal specification, we can assume the following categories that influence the behavior of the functionality: number of orders in the last period, amount in stock, type and status of the item, status of assembled items.

Then we identify relevant values: the test designer selects a set of representative classes of values (choices) for each parameter characteristic. Values are selected in isolation, independently of other parameter characteristics. For example, Figure 1 shows a possible set of choices for the categories extracted from the specification of the catalog handler. We notice that choices are not always individual values, but in general indicate classes of homogeneous values.

When selecting choices, test designers have to refer to normal values as well as boundary and error values, i.e., values on the borderline between different classes (e.g., zero, or the values of the thresholds for the number of orders or the amount in stock that are relevant for the considered case.) In the end, we generate test case specifications: test case specifications can straightforwardly be generated as combinations of the choices identified in the above steps.

Unfortunately, the mere combination of all possible choices produces extremely large test suites. For example, the simple set of choices of Table 1 produces more than 1,000 combinations. Moreover, many combinations may make little or no sense. For example, combining individual items with different status of assembled items makes no sense, or test designers may decide to test only once the boundary cases.

Table 1.Category partition testing

orders in the period	**amount in stock**	**status of the item**
0 [single]	0 [single]	in production
<t 1	<s 1	in catalog
t1 [single]	s1 [single]	not available [error]
<t 2	<s 2	status of assembled item
t2 [single]	s2 [single]	in production [if assemble]
>t 2	<s_max	in catalog [if assemble]
>> t2	s_max [single]	not available[if assemble][error]
type of item	>s_max [error]	individual assembled

Table 1 is category partition testing: A simple example of categories, choices and constraints for the example catalog handler. The choices for each category are listed in the corresponding columns. Constraints are shown in square brackets.

However, in other cases, choices are not naturally constrained, and the amount of test cases generated by considering all possible combinations of choices may exceed the budget allocated for testing.

2.2 Module Testing

Modules or components are first verified in isolation usually by the developers themselves who check that the single modules behave as expected (module testing). Thorough module testing is important to identify and remove faults that otherwise can be difficult to identify and expensive to remove in later development phases. Module testing includes both functional and structural testing. Functional test cases can be derived from module specifications using an appropriate functional or module-based testing technique, like the ones outlined in the previous section.

Structural testing complements functional testing by covering the structure of the code and thus coping with cases not included in functional testing.

Structural testing is often applied at two stages: first programmers measure the code coverage with simple coverage tools that indicate the amount of covered code and highlight the uncovered elements of the code, and then generate test cases that traverse uncovered elements. Coverage may be measured by taking into account different elements of the code. The simplest coverage criterion focuses on statements, and measures the percentage of statements executed by the test cases. Branch and condition coverage criteria measure the amount of branches or conditions exercised by the tests. Path coverage criteria measure the amount of paths covered by the tests. Different path coverage criteria can be identified based on the way paths are selected. Additional criteria refer to flow of data.

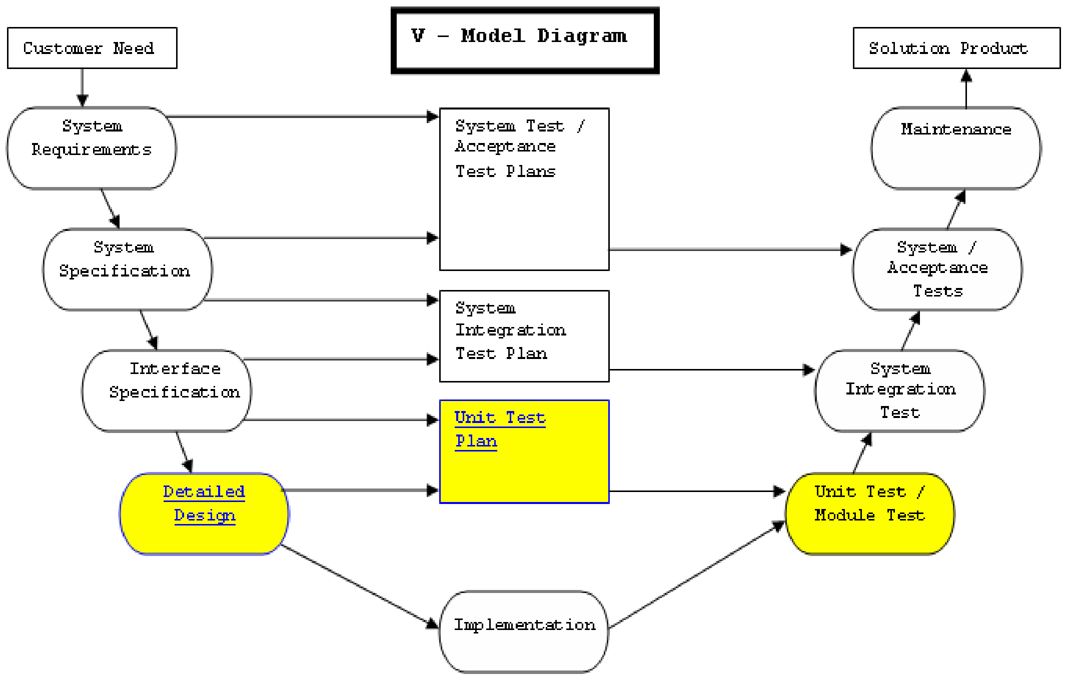

Figure 1. Development and testing phases in the V model

2.3 Integration and Component-based Testing

The quality of single modules is necessary but not sufficient enough to guarantee the quality of the final system. The failure of low quality modules fatally leads to system failures that are often difficult to diagnose, and hard and expensive to remove. Unfortunately, many subtle failures are caused by unexpected interactions among well-designed modules.

Integration faults are ultimately caused by incomplete specifications or faulty implementations of interfaces, resource usage, or required properties. Unfortunately, it may be difficult or cost-ineffective to specify all module interactions completely. Integration faults can come from many causes:

- Inconsistent interpretation of parameters or values.
- Violations of value domains or of capacity/size.
- Side effects on parameters or resources, as can happen when modules use resources not explicitly mentioned in their interfaces.
- Missing or misunderstood functionality which can happen when incomplete specifications are badly interpreted.
- Non-functional problems, which derive from under specified non-functional properties like performances.

Integration testing deals with many communicating modules. Big bang testing, which waits until all modules are integrated, is rarely effective, since integration faults may hide across different modules and remain uncaught, or may manifest in failures much later than when they occur, thus becoming difficult to localize and remove. Most integration testing strategies suggest testing integrated modules incrementally. Integration strategies can be classified as structural and feature-driven. Structural strategies define the order of integration according to the design structure and include bottom-up and top-down approaches, and their combination, which is sometimes referred to as sandwich or backbone strategy. They consist in integrating modules according to the use/include relation, starting from the top, the bottom or both sides, respectively.

Feature-driven strategies define the order of integration according to the dynamic collaboration patterns among modules, and include thread and critical module strategies. Thread testing suggests integrating modules according to threads of execution that correspond to system features. Critical module testing integrates modules according to the associated risk factor that describes the criticality of modules.

Feature-driven test strategies better match development strategies that produce early executable systems, and may thus benefit from early user feedback, but they usually require more complex planning and management than structural strategies. Thus, they are preferable only for large systems, where the advantages overcome the extra costs.

III. AUTOMATED TESTING PROCESS

The objective of automated testing is to simplify as much of the testing effort as possible with a minimum set of scripts. If unit testing consumes a large percentage of a quality assurance (QA) team's resources, for example, then this process might be a good candidate for automation. Automated testing tools are capable of executing tests, reporting outcomes and comparing results with earlier test runs. Tests carried out with these tools can be run repeatedly, at any time of day.

The method or process being used to implement automation is called a test automation framework. Several frameworks have been implemented over the years by commercial vendors and testing organizations. Automating tests with commercial off-the-shelf (COTS) or open source software can be complicated, however, because they almost always require customization.

Question is which Test Cases to Automate? Test cases to be automated can be selected using the following criteria:

- High Risk - Business Critical test cases
- Test cases that are executed repeatedly
- Test Cases that is very tedious or difficult to perform manually
- Test Cases which are time consuming

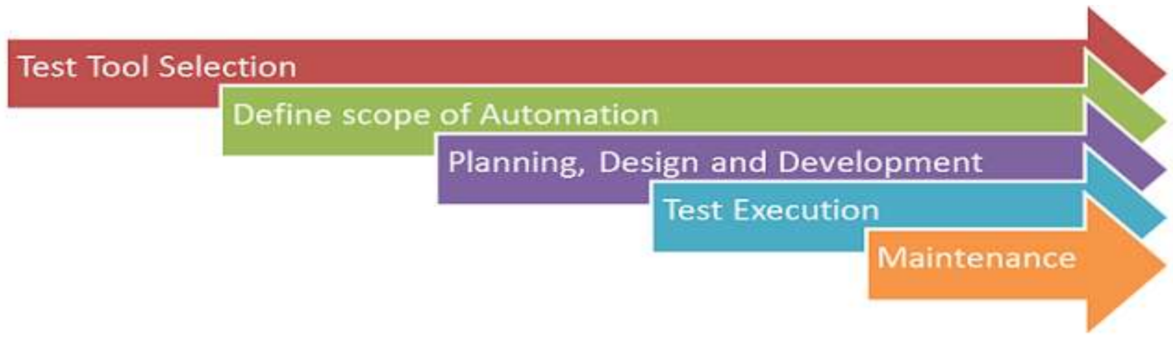

Figure 2.Phases of automated testing process

3.1 Test Tool Selection

Test Tool selection largely depends on the technology the Application Under Test is built on. For instance QTP does not support Informatica. So QTP cannot be used for testing Informatica applications.

Automation tool: QTP: HP’s Quick Test Professional (now known as HP Functional Test) is the market leader in Functional Testing Tool. The tool supports plethora of environments including SAP , Java , Delphi amongst others. QTP can be used in conjunction with Quality Center which is a comprehensive Test Management Tool. Know is light tool which can be recommended for web or client/server applications.

3.2 Define the Scope of Automation

Scope of automation is the area of your Application under Test which will be automated. Following points help determine scope:

- Feature that are important for the business
- Scenarios which have large amount of data
- Common functionalities across applications
- Technical feasibility
- Extent to which business components are reused
- Complexity of test cases

- Ability to use the same test cases for cross browser testing

3.3 Planning, Design and Development

During this phase you create Automation strategy & plan, which contains following details-

- Automation tools selected
- Framework design and its features
- In-Scope and Out-of-scope items of automation
- Automation test bed preparation
- Schedule and Timeline of scripting and execution
- Deliverables of automation testing

3.4 Test Execution

Automation Scripts are executed during this phase. The scripts need input test data before there are set to run. Once they are executed, they provide detailed test reports.

Execution can be performed using the automation tool directly or through the Test Management tool which will invoke the automation tool.

Example: Quality center is the Test Management tool which in turn it will invoke QTP for execution of automation scripts. Scripts can be executed in a single machine or a group of machines.

3.5 Maintenance

As new functionalities are added to the System Under Test with successive cycles, Automation Scripts need to be added, reviewed and maintained for each release cycle. Maintenance becomes necessary to improve effectiveness of Automation Scripts.

3.6 The model

SmartMBT takes as input a labelled transition system that specifies the behaviour of a software system. The testing process is driven by the model. The model provides the testing tool with a set of enabled actions calculated from its current state. Usually, every action in a LTS model will be available for execution sometime. There exist observable actions, which the SUT executes independently, usually in response to other actions. These actions should not be executed by the testing environment, therefore should not be included in any set of enabled actions. However, they should be part of the model because they actually update the state of the SUT (and of the model). We included a mechanism that allows the modeller to tag observable actions, and the tool to recognise them.

3.6.1 The Generator Module

The generator module represents most of the normal workings of the SmartMBT tool. It gets from the model the set of enabled actions in the current state, picks one (or let the tester pick one) randomly and simulate its execution over the SUT updating the state of the system in the model. If linked to the SUT, this module also sends the chosen action through the communication channel firing its execution into the SUT. In our implementation, this module corresponds to the generator algorithm and has access only to controllable actions. This is, in a given state of the system it can choose randomly (or in a user driven way) among enabled actions tagged as controllable. In general, this component stops its execution if no enabled actions are provided by the model. In asynchronous systems, it can happen that on a defined state, no controllable actions are enabled but observable actions are still pending of execution. Our generator module remains in a “waiting” state when this case arises. The module only stops its execution if no actions, nor controllable neither observable, are enabled on a defined state.

3.6.2 The Observer Module

This module is completely new in the testing tool. It has the ability to observe which (observable) actions have been executed in the SUT, usually in response of actions executed by the generator. This module implements our execution observer algorithm. It runs into a separate thread and is activated each time an action comes through the communication channel. Controllable actions are discarded because the generator retains the responsibility of updating the state of the model in this case. Observable actions are processed exactly as per algorithm.

IV. CONCLUSION

We have analysed that manual testing is time consuming and tedious since test cases are generated and executed by human resources, so it is very slow and tedious. It requires investment for human resources. It is less reliable as test case may not be performed with precision because of human error. This is random testing technique to find the bugs .It is not expensive but considered as low quality and gives low accurate result.

While automated testing runs test cases significantly faster than human resource. It requires investment for testing tool. It is reliable as it precisely performs same operation each time they are run. This technique performs testing through running scripts. It is expensive but gives high quality and more accurate result.

Software testing has been an active research area for many decades, and today quality engineers can benefit from many results, tools and techniques. So far, research on testing theory produced mostly negative results that indicate the limits of the discipline, but call for additional study. We still lack a convincing framework for comparing different criteria and approaches. Available testing techniques are certainly useful, but not completely satisfactory yet. We need more techniques to address new programming paradigms and application domains, but more important, we need better support for test automation.

References

[1] Luciano Baresi, Mauro Pezz`,An Introduction to Software Testing, Electronic Notes in Theoretical Computer Science 148 (2006) ,99-106,2005.

[2] Percy Pari Salas, Padmanabhan Krishnan, Automated Software Testing of Asynchronous Systems, Electronic Notes in Theoretical Computer Science 253 (2009),11-12,2009.

[3] I. Bhandari, M. J. Halliday, J. Chaar, K. Jones, J. Atkinson, C. Lepori-Costello, P. Y. Jasper, E. D. Tarver, C. C.Lewis, and M. Yonezawa. In-process improvement through defect data interpretation. The IBM System Journal, 33(1):182–214, 1994.

[4] B. W. Boehm. Software Engineering Economics. Prentice Hall, Englewood Cliffs, NJ, 1981.

[5] J.K. Chaar, M.J. Halliday, I.S. Bhandari, and R. Chillarege. In-process evaluation for software inspection and test. IEEE Transactions on Software Engineering, 19(11):1055–1070, November 1993.

[6] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton. The AETG system: An approach to testing based on combinatiorial design. IEEE Transactions on Software Engineering, 23(7):437–444, July 1997.

[7] P. G. Frankl and O. Iakounenko. Further empirical studies of test effectiveness. In Proceedings of the ACM SIGSOFT 6th International Symposium on the Foundations of Software Engineering (FSE-98), volume 23, 6 of Software Engineering Notes, pages 153–162, New York, Nov. 3–5 1998. ACM Press.

[8] T. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel. An empirical study of regression test selection techniques. In Proceedings of the 20th International Conference on Software Engineering, pages 188–197. IEEE Computer Society Press, April 1998.

[9] P. A. Hausler, R. C. Linger, and C. J. Trammell. Adopting cleanroom software engineering with a phased approach. IBM Systems Journal, March 1994.

[10] Independent Assesment Team. Mars program independent assessment team summary report. Technical report, 2000.

[11] A. Jaaksi. Assessing software projects: Tools for business owners. In Proceedings of the 9th European Software Engineering Conference held jointly with 10th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE 2003), pages 15–18. ACM Press, September 2003.

[12] Henniger, O., On test case generation from asynchronously communicating state machines, in: Proceedings of the 10th International Workshop on Testing of Communicating Systems, Cheju Island, South Korea, 1997.

[13] Herbreteau, F., G. Sutre and T. Q. Tran, Unfolding concurrent well-structured transition systems, in: O. Grumberg and M. Huth, editors, Proc. of the 13th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems

(TACAS07), Lecture Notes in Computer Science 4424, ETAPS (2007), pp. 706–720. URL http://www.labri.fr/publications/mef/2007/HST07

[14] Jard, C. and T. J´eron, Tgv: theory, principles and algorithms: A tool for the automatic synthesis of conformance test cases for non deterministic reactive systems, Int. J. Softw. Tools Technol. Transf. 7 (2005), pp. 297–315.

[15] Nakata, A., T. Higashino and K. Taniguchi, Protocol synthesis from context-free processes using event structures, in: Proc. of the 5th International Conference on Real-Time Computing Systems and Applications, Hiroshima, Japan, 1998, pp. 173–180.

[16] Nielsen, M., G. D. Plotkin and G. Winskel, Petri nets, event structures and domains, part I, Theoretical Computer Science 13 (1981), pp. 85–108.

[17] Stark, E. W., Connections between a concrete and an abstract model of concurrent systems, in: In Fifth Conference on the Mathematical Foundations of Programming Semantics, Springer-Verlag. Lecture Notes in Computer Science (1989), pp. 53–79.

[18] Tretmans, J., Model based testing with labelled transition systems.