Challenges of Cost Estimation for Software Testing

Bernard Stepien, Liam Peyton
School of Engineering and Computer Science
University of Ottawa
Ottawa, Canada
Email: {bstepien | lpeyton}@uottawa.ca

***Abstract*—Cost estimation for software testing is a complex process due to a great variety of testing strategies and factors to consider. In current practice, some of these are often overlooked. The subject has been well researched all the way back to the early stages of software development but always within the specific context of a single application. As a result, managers and researchers have created application-dependent solutions to the problem rather than general solutions. As a first step towards development of a general solution, we provide a summary of cost estimation for software testing using a taxonomy of testing strategies and factors.**

Keywords: software testing; cost estimation; taxomomy.

I. INTRODUCTION

Cost estimation for software testing is a complex process due to the difficulty in determining precisely the factors affecting costs. One of the most difficult tasks consists in separating adequately software development costs from software testing costs especially since they are inter-related. This is especially true when both tasks are performed by the same person, which is often the case.

Historically, cost estimation for software testing used macroeconomic models based on empirical studies and produced only approximate cost estimations. Unfortunately, one interesting characteristic of those models is that they only estimate the combined software development and testing cost without a clear indication on the cost share allocated specifically for testing. Boehm started it all with the Constructive Cost Model (COMOCO) [7] at a time when software development models themselves, such as structured programming, were being addressed. The concept of testing at that time was limited and based mostly on test plans for manual testing that would be approved by upper management without any quantitative evaluation of costs or benefits. They were based on a strong belief that there could be only a finite set of bugs in a piece of software and that most of the bugs could be caught in the early stages of testing. Later research based on automated test case generation proved the opposite mostly for state based software with unpredictable traversal of state transitions.

More recently, Holzmann [4] pointed out that no single system of metrics exists to measure costs or to measure benefits of testing. A major effort on understanding cost estimation for testing was achieved by the National Institute of Standards and Technology in an extensive report [3]. Hu et al. [2] came up with economic projection models to quantify testing results. Ellims et al. [8] studied the economics of unit testing and compared it to code reviews, which were a prevailing substitute to testing at the time. Tables showing the relative benefits of software testing and models for cost estimation were produced by Tawileh et al. in [5].

Application specific solutions came in the early ‘80s with the development of a test specification language for the telecommunication industry that was based on formal description techniques that was later extended by the European Telecommunications Standards Institute as Testing and Test Control Notation version 3 (TTCN-3) [11]. TTCN-3 is now used in many different domains in addition to telecommunications. It is particularly efficient when applied to testing applications that use parallelism intensively. Others addressed the testing of web application and started comparing the performance of the solutions in this domain [13].

Another interesting aspect of testing is the variety of metrics for testing without any clear preference or even understanding of their values. They include fault density, requirement compliance, test coverage and mean time to failure. As well, some, like in [3], differentiate between technical metrics and economic metrics. Also, the basic unit of measurement is traditionally the number of lines of code. This applies for both the actual piece of software being tested and the test software in case of test automation. For example, in one industry we worked in, there was a de facto standard that one would develop only 7 lines of codes per day including testing for a given project that comprised 3 million lines of code. In all the literature cited above, we have never found a concept of measuring the complexity of a piece of software that would make two different pieces of software with an equal number of lines radically different from a costing point of view both for development and testing. This results in biased comparisons and questionable precision of decisions.

Interesting is the fact that recent literature, Keshta in [9], attempts to summarize the research on software costing over a 30 years period but again without talking about testing.

The awareness of the difficulty of estimation for software testing costs is expressed by Wagner et al in [6]. They propose an approach that structures the factors of costs for an optimization of fault detection techniques. They also distinguish between minor and major faults and propose a priority system for handling them. In any case, the application of solution blurs the picture further due to the fact that faults detected and corrected in previous steps no longer exist in the subsequent stages. Ideally, one should start from scratch every time a new technique is applied. This is both unrealistic and no manager would approve a budget around such an approach. Finally, most studies are empirical rather than following a rigorous model as in [14].

II. THE REALITY OF TESTING STRATEGIES

Personal experience in industry has shown there is a great variety of testing strategies. A good collection of strategies, including the fundamental black box and white box testing strategy, can be found in [1] but with no associated costing theories other than the financial consequences of unsatisfied customers. Brill et al [12] decomposes the software development life cycle and proposes enhancements. A similar taxonomy can be found in [15].

A. Testing and test personnel configurations

1) Testing as a way to learn a software

Testing is performed by novices to learn how a piece of software functions. The tester is thus a temporary position towards the more glamorous position of software developer. One may discover that this approach increases testing costs, however, this is also a kind of training session for the same persons once they become developers, thus potentially reducing development costs afterwards, as shown on Figure 1. This is a typical case of tradeoff between the two complementary activities.

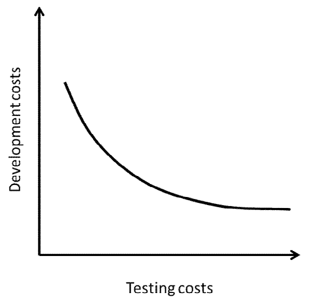

Figure 1. Relationship between development and testing costs

2) Developer is also tester

Testing is performed by the software developer. This approach has proven to be less efficient because the behavior of the developers involved has a tendency to be highly predictable. This results in always testing the same paths that they know and making all kinds of assumptions about their system. Whereas, an external tester, would try event sequences that the developer did not anticipate and uncover more bugs.

3) Independent team of testers approach

Testing is performed by an independent team devoted only to testing that develops test suites from the same requirements that the developers use. This approach has proven to be more efficient because it reveals the most unsuspected bugs. Testers are independent from developers and thus are not influenced by development activities and thus test unsuspected combinations of inputs. Also, from an economics perspective, testers were cheaper than developers since their activity was considered simpler than software development. However, this is not always true for testers who use sophisticated testing techniques, based on formal descriptions, with dedicated test specification languages, which require specialized knowledge and training.

4) Automated testing and test specification languages

When Object Oriented (OO) languages became common, software development languages were used as test specification languages to reduce testing costs mostly based on savings for personnel training costs and testing tools.

5) Reducing the number of programming languages

In the '80s, there was a proliferation of programming and test specification languages that required high training costs and recruitment difficulties that drove personnel costs up. This resulted in decisions to reduce considerably the number of programming languages for software development used in an organization especially in the light of the new object oriented approaches.

6) The software user is the tester

It is increasingly the fashion for widely publicly used software to use the end users as unsuspecting testers. By this, we mean the typical approach that consists upon a crash of the software when used by a user to ask the user if they want to report the problem. It is very economical for the software vendor but poses fundamental security problems since nobody knows nor controls the amount of user data that is sent to the software vendor. However, in a way, some of the testing cost is transferred to the end-user.

B. The use of Testing tools and frameworks

Some tools are application oriented while others are more general. However, the reality is that tools are often costly but reduce the cost of personnel because tools increase productivity as shown on Figure 2. Increased productivity results in a shallower total cost curve comprising cost of the tool and training the testers.

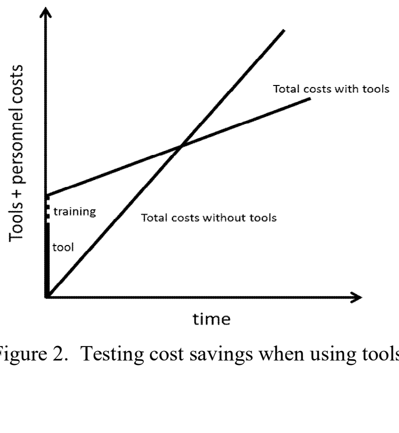

Figure 2. Testing cost savings when using tools

1) Manual testing

At the beginning, all tests were performed manually. This proved to be both expensive and difficult to reproduce especially in the case of personnel turn over. Thus, the concept of test automation using test software was introduced. These tests were either using General Programming Languages (GPL) or dedicated testing languages and later frameworks of all kinds.

2) Unit testing

Unit testing uses an open source framework (e.g. JUnit) that provides for ways of specifying the test as assertions on results from methods or functions. Unit testing consists mostly in testing applications functions or methods, one at a time and independently from each other. It verifies that given certain inputs, it correctly returns some specific output. It is an extension of traditional OO software testing languages and thus can be easily integrated in the software development process. Unit tests are developed first before software development itself. Initially all tests fail but as development progresses, tests pass. A special Graphical User Interface (GUI) that reports the failures or successes is provided by the framework, thus saving some test software reports development costs.

3) Application specificity of testing tools

Application specific tools are widely used for web application testing. Frameworks, such as Selenium, among others [13] feature the principle of record and replay based on the belief that the same input will always produce a unique result. In other words, there are no alternative software behavior considerations. This is mostly not the case and also test software is hard to maintain.

4) Automated test case generation

Test case generation derived from models, like finite state machines, can be very efficient if not running into state explosion problems. In this case, heuristics are used to choose a subset of test cases that would provide enough test coverage. Also, models allow testing the requirements themselves even before they are attempted to be implemented especially if they are specified in an executable language.

III. TAXONOMY OF TESTING COSTS

A. Categories of testing costs

There are many categories of testing cost:

- Tangible costs like the cost of manpower and equipment including testing tools to perform testing.
- Tangible costs of training personnel to a given testing technology.
- Intangible cost like traveling costs to the customer premises
- Intangible cost like loss of business due to lack of customer satisfaction.

However, it has been observed in industry that the cost of testing tools is considered as highly tangible while manpower to perform the tests is not in the sense that testing tools can be used only for a class of applications while manpower can be reassigned to other tasks or even projects, thus, not subject to a seemingly tangible loss.

B. Areas of difficulty in testing

1) Application making intensive use of parallelism

Applications making intensive use of parallel activities cannot be tested using unit testing alone because parallel test components need to be well coordinated. This is the case of most frameworks that are derivatives of unit testing. Also parallel testing is considered as very complex and most testers avoid it altogether. Testing tools, such as TTCN-3 are particularly efficient at testing parallel applications because of native features that enable to control several instances of parallel components easily and efficiently and provide methods to explore test logs efficiently. These cover applications based on Service Oriented Applications (SOA) and more recently Business Process Management (BPM). Both making extensive use of web applications as a front end that can be easily tested using TTCN-3.

2) Computation testing

Computation testing consists in testing software that depends solely on computation activities that upon an input are expected to return a predictable output. These are best tested using unit testing. This is particularly the case when the software is autonomous, i.e., it does not interact with other software components. It is, in a way, the benchmark in testing methodologies.

3) State testing

State testing consists in putting a system into different states and observing correctness of the transition to another state. A state can be represented by specific output values of a set of variables that the software manipulates. In this case, we do not only test the final value of the output but also any of the intermediary states to obtain it. This is often referred to as grey or white testing depending on the granularity.

C. Heterogeneous testing costing data

In theory, comparing testing approaches, strategies and supporting tools should be achieved on a specific application or groups of applications. The reality is that no one actually does that because having the same software developed several times using different approaches is highly counter-intuitive and no one would actually budget such a strategy. The solution that comes closes to that concept is found in [10] where an attempt to verify this theory is applied to past software development projects. This model also includes a technology factor. Consequently, comparisons are performed between different and heterogeneous applications on radically different domains and thus, results are not totally comparable. However, this has been achieved in a specific domain of GUI testing in [13] and in the transportation and financial sectors in [3].

D. *Industrial behavior on testing*

Most software is developed around a set budget. The same applies to testing with the difference that when budgets are overrun, it is the development budget that is allowed to do so while the testing budget is capped. Thus, we could say that testing is budget centric rather than application centric. Here, the most typical behavior consists in hiring testers without any performance consideration. The cases we have observed in industry include purchasing expensive testing tools and their related personnel training. Here, the costs are very tangible and require approval from upper management but these situations always resulted in the upper management requiring proof that there are benefits to do so. Again, decisions were guided by intuition and not by precise economic models.

As well, personnel turnover increases the cost of testing tools training costs because as people moved on, their expertise disappears with them. New personnel have to be re-trained. This has the result of reducing the benefits of dedicated testing tools as can be shown on Figure 3. Here the break-even point has moved up both in cost and time.

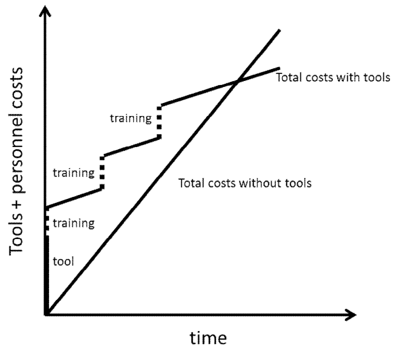

Figure 3. Incidence of personnel turnover on training costs

However, this is a false problem since most of the problem lies in the organization of testing tool usage. The most efficient way has been to create a pool of testing tool experts that train and mentor the testers along the testing life cycle. This is no different than the issues with the turnover of software developers that requires new developers to learn the software developed by their predecessors.

One case is particularly interesting: testing critical systems that require certification of tests. Here, changing test approaches regardless of the lack of efficiency of existing automated testing software is a major problem. Effectively, besides the cost of redeveloping test software using the new approaches, the certification process has to be redone from the start.

IV. CONCLUSION

Despite its long history, cost estimation of testing is mostly an ad hoc activity and still needs to explore new avenues. The great number of factors of cost makes it difficult to come up with an optimal solution to the problem. Also, a key factor, complexity of software, is missing completely from the literature. The available literature shows that tackling the problem is difficult and no final solution has been found as yet.

ACKNOWLEDGMENT

The authors would like to thank NSERC for funding this research.

REFERENCES

[1] G. J. Myers, C. Sandler and T. Badgett, The Art of Sotftware Testing, 3rd Edition,Wiley Pubishing, 2011, ISDN 978-1-118-03196-4

[2] Q. Hu, R. T. Plant and D. B. Hertz, "Software Cost Estimation Using Economic Production Models", Journal of Management Information Systems 15, no. 1, 1998. pp. 143-163.

[3] M. P. Gallaher and B. M. Kropp. "Economic impacts of inadequate infrastructure for software testing", Technical report 7007.011, RTI International, National Institute of Standards and Technology, 2002.

[4] G. J. Holzmann, "Economics of software verification", Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering (PASTE 01), pp. 80-89. ACM, 2001.

[5] A. Tawileh, S. McIntosh, B. Work and W. Ivins, "The Dynamics of Software Testing", proceedings of the 26th international Conference of the System Dynamics Society, 2007.

[6] S. Wagner. "Towards software quality economics for defect-detection techniques", 3rd Workshop on Software Quality, 29th Annual IEEE/NASA, pp. 265-274, 2005. pp. 1-6.

[7] B. Boehm. Software engineering economics. Englewood Cliffs, NJ:Prentice-Hall, 1981. ISBN 0-13-822122-7

[8] M. Ellims, J. Bridges, and D. C. Ince, "The economics of unit testing", Empirical Software Engineering 11, no. 1,2006: pp 5-31.

[9] I. M. Keshta, "Software Cost Estimation Approaches: A Survey." Journal of Software Engineering and Applications 10, no. 10, 2017.

[10] K. Pillai and V. S. S. Nair. "A model for software development effort and cost estimation." IEEE Transactions on Software Engineering 8 (1997): pp 485-497.

[11] ETSI ES 201 873-1, The Testing and Test Control Notation version 3 Part 1: TTCN-3 Core Language, May 2017. Accessed March 2018 at http://www.etsi.org/deliver/etsi_es/201800_201899/20187301/04.09.01_60/es_20187301v040901p.pdf

[12] C. Brill and A. Olmsted, "Security and Software Engineering: Analyzing Effort and Cost", SOFTENG 2017, pp110-114. Accessed December 6, 2018 at https://www.thinkmind.org/download_full.php?instance=SOFTENG+2017.

[13] P. Sabev and A. Kanchev, "A Comparative Study of GUI automated Tools for Software Testing", SOFTENG 2017, pp7-16. Accessed December 6, 2018 at https://www.thinkmind.org/download_full.php?instance=SOFTENG+2017.

[14] L. P. Kafle, "An Empirical Study On Software Test Effort Estimation", International Journal of Soft Computing and Artificial Intelligence, ISSN: 2321-404X, vol. 2, issue 2, Nov. 2014.

[15] K. R. Jayakumar and A. Abran, "A Survey of Software Test Estimation Techniques", Journal of Software Engineering and Applications 6, no. 10, 2013. pp. 47-52.