

# Improving Testing Efficiency: Agile Test Case Prioritization

## Introduction

A remaining challenging area in the field of software management is the release decision, deciding whether or not a software product can be transferred from its development phase to operational use. Many software manufacturers have difficulty in determining the 'right' moment to release their software products. It is a trade-off between an early release, to capture the benefits of an earlier market introduction, and the deferral of product release, to enhance functionality, or improve quality. Execution testing is currently the most common way of verifying whether a software system is sufficiently defect-free or not. With this approach, testing resources are absorbed not just in testing, but in identifying the causes of defects and aiding re-work of the software. The extent to which rectified software needs to be re-tested is not always known. The stop criterion used is either available testing time or a sufficiently stable product. Whatever criterion is taken, the test process should be as efficient as possible to remove as many defects as possible. In this paper, we describe a method called "Agile Test Case Prioritization" for optimizing testing efficiency.

## Model-based testing still lacking acceptance

New methods like model-based testing are slowly gaining market acceptance. Model-based testing requires that the full behavioural functionality of a software system is captured in its specifications. Mathematical verification guarantees that no behaviour is overlooked [4]. This in turn ensures that functionality and quality are analyzed and modelled in detail early in a project. Consequentially, this enables the automatic generation of thousands of correct test cases to automatically verify the correct implementation. Compared to the manual process of creating test cases, which is restricted by time and budget, model generated test cases can be used to confirm the correct operation of the product, knowing that the full extent of the functionality that has been covered [1]. It allows a test team to focus on validation rather than defect detection and removal. Software managers are still struggling with accepting such formal approaches for two main reasons. In the first place, they lack quantitative insight in their current performance that would reveal the clear benefits of such formal approaches. Secondly, they feel insecure whether the organization will be capable in successfully adopting such approaches. It requires a different mindset towards engineering and high professional skills for building and verifying such models. As a result, many organizations stick to conventional design approaches, and ambiguous and incomplete specifications may not be detected until the test stage. They are then faced with the challenge how testing efficiency can be optimized.

## Conventional approach

Conventional testing approaches result in costly redesign or fixes that merely propagate new problems through the system, adding to cost and time to market at a moment when the development budget has mostly been consumed. In addition, some common problems regarding test cases are [6]:

- Bad test cases. The error density of test cases themselves is sometimes higher than the error density of the code being tested.
- Duplicate and redundant test cases. About 30% of the test cases in large test libraries tend to be duplicates of others. This adds cost to testing but nothing valuable in terms of removal efficiency.

Many organizations choose to outsource the testing activities to hired professionals or external contractors. In this way, they can concentrate on value adding activities and avoid building knowledge outside their core expertise area. Whatever scenario is chosen, a remaining main concern is how to ensure a good level of testing efficiency. There are currently two main criteria used to determine when to stop the testing phase of a product:

1. Time-constraint: stop when a certain period of time has elapsed.
2. Quality-constraint: stop when a software product is 'good enough' for release. A study prepared for the National Institute of Standards & Technology, revealed a number of non-analytical methods that are commonly used to decide when a software product is 'good enough' for release [8]. The most common ones are based on thresholds on:
  - the percentage of test cases that complete successfully;
  - the percentage of code that executes without error;
  - the number of defects reported in a given interval.

With either of the two stop criteria from above, a low testing efficiency can have severe implications on the total cost of the software system. In time-constraint cases, a high number of defects may be still undetected at release time, resulting in expensive product recalls later on. The automotive industry is a good example in this respect. The total costs of auto recalls in USA exceeded \$6 billion in 2004 alone, not including the costs to consumers [9]. In quality-constraint cases, the testing phase can constantly postpone the release date, increasing development costs and decreasing revenues under time-to-market pressure. Consequently, reaching a high testing efficiency is an important challenge whatever stop criterion for testing is chosen.

Despite the rich evidence, both anecdotic and documented, about the influence of testing efficiency on the total cost of the software, there is little knowledge about how it can be measured and improved. Most of the existing efforts in this direction focus rather on the later aspect [5][6], and rely on the number of defects as prime measure for testing efficiency. In this article we address both challenges. We first describe a self-calibrating model for assessing testing efficiency based both on the number of faults discovered at a given moment and the actual code coverage. Next we describe a method to improve testing efficiency via context-directed test case prioritization. This method takes into account organizational, product and development aspects to increase the number of defects detected during a given testing interval.

**Step 1: Assessing testing efficiency**

The most common way of measuring testing efficiency used in the software industry at this moment is by counting the number of defects discovered in a given time interval. This method has one major drawback. While the testing phase progresses, the number of discovered defects decreases, possibly in a natural way. However, the method fails to recognize whether this decrease is due to a lower testing efficiency or simply due to the natural increase in the difficulty of finding defects (see Figure 1). A good method for measuring testing efficiency should consider also a factual indicator of how suitable the testing effort is for finding errors. A proven and commonly accepted way of assessing the probability of finding defects in software is the percentage of code coverage during testing: the higher the code coverage, the lower the probability of finding new defects [3][7]. We use code coverage to build a self-calibrating model for measuring testing efficiency that takes into account not only the output but also the quality of the testing effort.

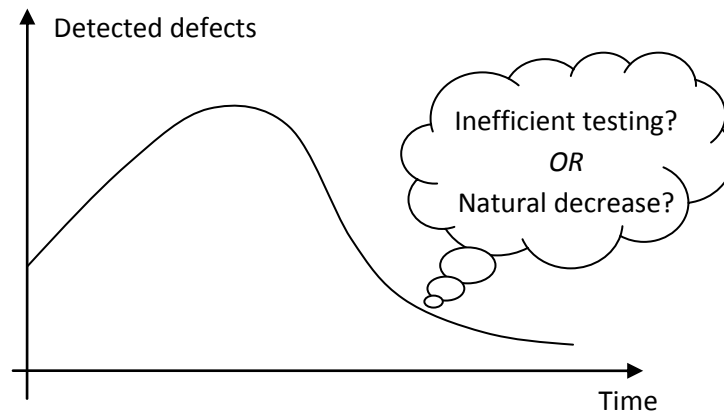


Figure 1: Typical defect profile during testing phase.

Figure 2 presents a graphical representation of the model, and how it can be used to assess testing efficiency. The grey area in the image depicts the region of inefficient testing specific to a given time interval  $t$ , in a two dimensional space described by number of detected defects and code coverage. Points outside the grey area depict values corresponding to efficient testing; points inside the area depict low-quality testing efforts. One should observe that a low number of defects can still be associated with high testing efficiency, provided that it is obtained from test cases with good code coverage.  $K_{low}(t)$  and  $K_{high}(t)$  are time dependent values computed using a moving average approach on the set of testing efficiency points obtained in the previous time intervals.  $C(t)$  is project specific growing function ranging in the interval  $[0, C_{max}]$ , where  $C_{max}$  is the chosen threshold for stopping testing or 100% in time-constraint testing situations.

At any moment during the testing phase, the number of detected defects and corresponding code coverage should define points outside the evolving gray area in order to reflect good testing efficiency. A low number of detected defects is always acceptable, as long as it corresponds to sufficient code coverage. Using such a measurement model, one can get a more accurate indication about testing

efficiency, based not only on the testing output but also on the quality of the testing efforts. Additionally, this model offers a better integration with the quality-constraint criterion for stopping testing, using a similar reasoning for building confidence.

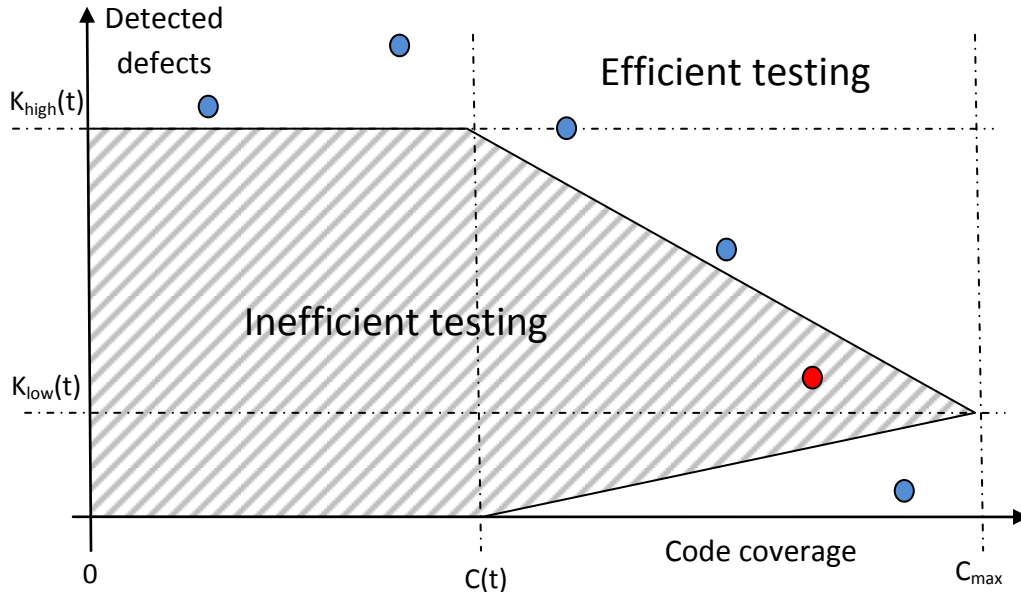


Figure 2: Model to assess testing efficiency.

### Step 2: Improving testing efficiency

A good measurement model for testing efficiency is essential to identify situations of underperformance that can influence the schedule and total cost of a software system. If timely detected, such situations can be corrected using organizational and process measures. Independent on these, additional methods can be used to boost testing efficiency by capitalizing on context and moment specific information about the defect occurrence patterns. These methods are collectively known under the name 'test case prioritization'. Their stated goal is to drive the testing process by selecting those test cases from the available pool that will discover the highest number of defects in a given time interval. Empirical studies on test case prioritization show a clear advantage of various methods over the random way of selection. However, the results are often contradictory. While some studies claim a given method substantially outperforms the random selection [6], others show there is little difference and propose alternative approaches [5]. The only aspect most researchers seems to agree upon is that a model exists for predicting the set of test cases that would lead to superior efficiency given a fixed organizational and process context.

We subscribe to this theory and recommend an agile-oriented way of performing test case prioritization. Our premise is that prioritization criteria are strongly dependent of the development context and on the previous defect occurrence patterns. No criterion has absolute advantage over the other, but at a given

moment one criterion can temporarily outperform the rest. The question that arises is how to select the best prioritization criterion at a given moment. To answer this question we use a multidimensional correlation approach to criterion selection. To this end, a number of software metrics is selected that can influence the number of defects in a given development and organizational context. Examples are: location in the code, code complexity, fan-out, code coverage, or code owner experience. Next, a visual driven approach is used to discover correlations between the discovered defects in a time interval and the associated metrics from the selected set. The method is based on TreeMap representations [2] of the software stack, with color and ordering of files given by various metrics. An example is given in Figure 3, showing the correlation between cyclomatic complexity and number of defects. The cyclomatic complexity determines the file position in a square (left upper corner = low complexity, right lower corner = high complexity). The color of each square indicates the number of defects (blue = few, red = many). From this figure, those areas that have both high cyclomatic complexity and a high number of defects can be derived.

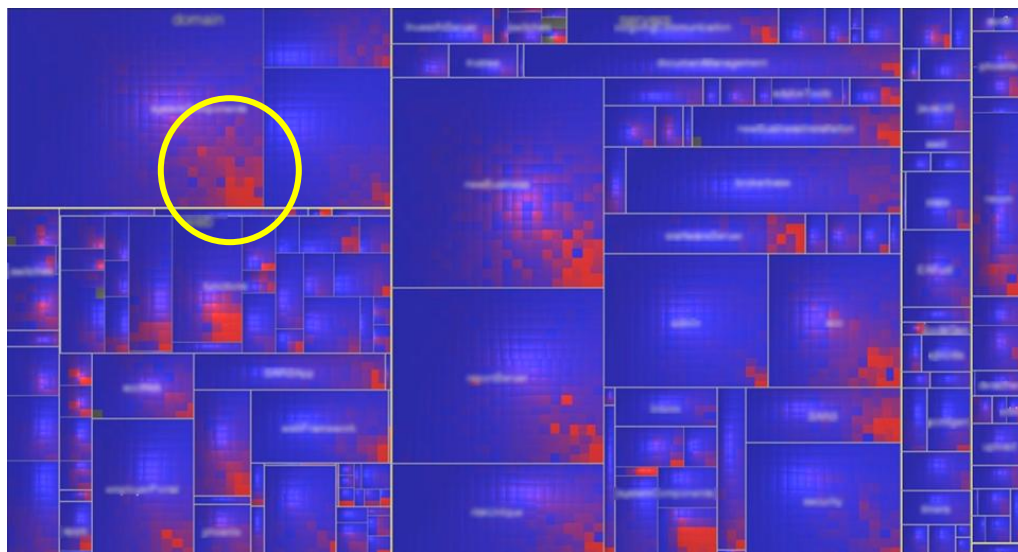


Figure 3: Example TreeMap representation of software.

By selecting different combinations for color and ordering of files in the TreeMap, one can discover correlations that support a specific criterion for test case prioritization at a given moment. Depending on the context, different criteria may be selected at different moments. This is the reason for calling this method *Agile Test Case Prioritization*. The presented method enables the selection of the most promising criterion, capitalizing on the defect occurrence patterns specific at a given moment during the testing phase. Using this selection approach on a regular base, one can constantly induce a boost in the testing efficiency level, achieving super testing efficiency, as in Figure 4.

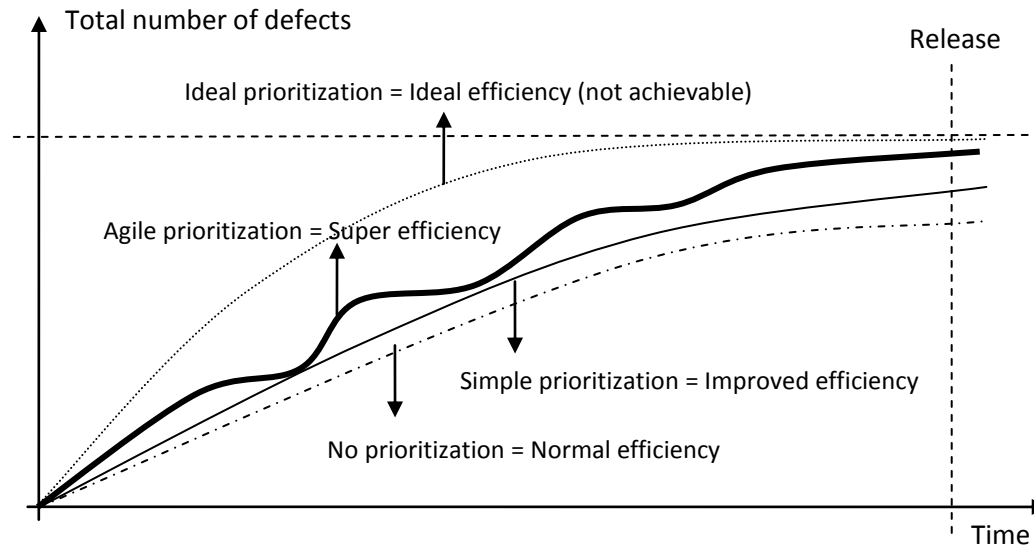


Figure 4: Test case prioritization methods.

### Conclusion

Testing efficiency is an important aspect of the testing phase with severe implication on the reliability of a software system at a given moment. In this article we presented an improved model for assessing testing efficiency that takes not only the test output into account but also the quality of the testing effort. This model can more accurately identify low testing efficiency situations that might be otherwise mistaken for a natural decrease in the testing output. We also presented an agile-oriented technique to test case prioritization. This technique uses a visual approach to discover correlations between defect occurrence patterns and various test case prioritization criteria. Using this technique on a regular basis, the most promising prioritization criteria may be selected at a given moment, enabling organizations to achieve a super testing efficiency.

### Authors

Dr. Lucian Voinea ([lucian.voinea@solidsourceit.com](mailto:lucian.voinea@solidsourceit.com)) is managing director of SolidSource BV ([www.solidsourceit.com](http://www.solidsourceit.com)) and specialized in code analysis and visualization.

Dr. Hans Sassenburg ([hsassenburg@se-cure.ch](mailto:hsassenburg@se-cure.ch)) is managing director of SE-CURE AG ([www.se-cure.ch](http://www.se-cure.ch)) and specialized in measurement and analysis (including benchmarking).

### References

1. Broadfoot, G., "ASD Case Notes: Costs and Benefits of Applying Formal Methods to Industrial Control Software", In: Lecture Notes in Computer Science, Springer Berlin, Heidelberg, 2005.

2. Bruls, D. M., Huizing, X., van Wijk, J.J., *"Squarified treemaps"*, In Proceedings of the joint Eurographics and IEEE TVCG Symposium on Visualization, pages 33–42, 2000.
3. Cai, X., Lyu, M.R., *"The Effect of Code Coverage on Fault Detection Under Different Testing Profiles"*, ICSE 2005 Workshop on Advances in Model-Based Software Testing (A-MOST), St. Louis, Missouri, May 2005.
4. Hu, A.J., *"Automatic formal verification of software: Fundamental concepts"*, International Conference on Communications, Circuits and Systems (ICCCAS 2009).
5. Jiang B., Zhang Z., Tse T.H., Chen T.Y., *"How Well Do Test Case Prioritization Techniques Support Statistical Fault Localization"*, In Proceedings of the 33<sup>rd</sup> Annual International Computer software and Application Conference (COMPSAC 2009).
6. Jones. C, *"Software Quality in 2008: A Survey of the State of the Art"*, white paper, 2008.
7. Rothermel G., Untch R.H., Chu C., Harrold M.J., *"Test Case Prioritization: An Empirical Study"*, In Proceedings of the International Conference on Software Maintenance (ICSM 1999).
8. RTI, *"The Economic Impacts of Inadequate Infrastructure for Software Testing"*, Planning Report 02-3, Prepared by RTI for National Institute of Standards and Technology, U.S. Department of Commerce, 2002.
9. Rupp N.G.: *"The Attributes of a Costly Recall - Evidence from the Automotive Industry"*. In: Review of Industrial Organization, vol. 25 (2004).