MIS Review Vol. 18, No. 2, March (2013), pp. 19-50

Optimal Software Release Policy Approach Using Test Point Analysis and Module Prioritization

Praveen Ranjan Srivastava[1], Subrahmanyan Sankaran[2], Pushkar Pandey[2]
[1]*Information Technology and System Group, Indian Institute of Management, Rohtak*
[2]*Department of Computer Science & Information Systems, Birla Institute of Technology and Science*

ABSTRACT: *When to stop testing and release the developed software is the one of the most important questions faced by the software industry today. Software testing is a crucial part of the Software Development Life Cycle. The number of faults found and fixed during the testing phase can considerably improve the quality of a software product, thereby increasing its probability of success in the market. Deciding the time of allocation for testing phase is an important activity of quality assurance. Extending or reducing this testing time, depending on the errors uncovered in the software components, can profoundly affect the overall project success. Since testing software incurs considerable project cost, over-testing the project can lead to higher expenditure, while inadequate testing can leave major bugs undetected, thereby risking the project quality. Hence prioritizing the components for testing is essential to achieve the optimal testing performance in the allotted test time. This paper presents, a Test Point Analysis based Module Priority approach to determine the optimal time to stop testing and release the software.*

KEYWORDS: *Module Priority, Expendability, Test Point Analysis (TPA), Normalization, Synchronization Factor, Maintenance Factor, Node Criticality, Reusability, Function Points, Technical Complexity.*

1. Introduction

Software testing is an expensive process. Many software organizations rely on the expertise of technical managers to decide the time to be allocated to test the software. The release time is thus mostly determined by prior experience. Research has shown that, as much as 45% of the total software development cost is spent in testing. In many projects, the time used in debugging can be around 50% of the total development effort (Myers, 1976). Releasing an under-tested software may lead to the risk of latent bugs in the software, leaving the customer dissatisfied and also incurring higher cost of removing the faults post-release. While testing the software beyond a certain limit may lead to an over-priced testing effort, resulting in loss of revenue. Also, a late release of the product in business may result in losing the market foothold. This tradeoff needs to be balanced by finding the optimal time of release and justifying the cost with the stop-test decision.

There are pertinent questions, which need to be addressed on scenarios where major bugs appear in a highly critical component of the software in a limited test time. After major code changes, if new bugs are introduced into the software, testing time may also include regression test.

However, exhaustive testing becomes too expensive and it is potentially impractical to test the software until all the bugs are recovered and removed (Goel & Okumoto, 1981). The management should be aware of the optimal testing time and cost required to test the modules. Thus, a tradeoff is sought to balance the overall cost and business goals that intends not to release a product with major bugs. It is essential to minimize the bugs after the release, as the cost of fixing such bugs could be much higher than in test phase. Hence prioritization of modules is pivotal to uncovering and fixing critical errors in allotted testing time before software release wherein, some faults are allowed in an accepted range instead of making the software product 100% error free (Mathur, 2008; Sommerville, 2011).

This paper presents a module prioritization technique to group the software components on priority. This is necessary to ensure the maximum testing of highly critical modules that impact the overall functionality of the project, thereby uncovering major errors in the project (Srivastava, 2010). The 'prioritization equation' of modules ought to stand pliable to both development and maintenance projects. This approach uses Test Point Analysis (TPA®) (Saaty &Vargas, 2011) to determine total testing time to be allotted for the software project. The prioritization of modules is done by Module Priority equation or by the Analytic Hierarchy Process (Saaty &Vargas).

Test Point Analysis (Dekkers & Veenendaal, 1999) is a robust, industry wide used technique to estimate the time of test phase in a project. This technique determines the time that must be allotted for the total testing process based on various software metrics, so that a delayed or premature release is avoided. After estimating test time, this paper proposes an algorithm to share the testing effort of lower priority modules to higher priority modules for achieving an optimal testing policy to release the software. Other than TPA (Dekkers & Veenendaal) technique, prior works of Goel and Okumoto (1981) have used non homogeneous poisson process to estimate the optimal time for release.

Next section gives the overview of the prior background work in this area. Section 3 discusses the proposed approach. Section 4 provides the merits of the proposed approach. A comprehensive case study and comparison with the existing work are done in the Sections 5 and 6 respectively. Finally, Section 7 presents the conclusion of the paper.

2. Literature review

Finding an optimal testing time for the Software Development Life Cycle (Pressman & Ince, 1992) lifetime has led to considerable work in the last few decades. Goel and Okumoto (1981) have suggested an approach dependent on time and Non homogenous poisson process (NHPP) (1981) to arrive at an optimal test policy. The stochastic process models the bugs that appear in software testing phase in a probability distribution which lays emphasis on number of faults detected during testing to decide the duration of the test phase.

Dalal and Mallows (1988) have used an economic and stochastic model to determine optimal testing time with a "stopping rule." Musa and Ackerman (1989) have presented software reliability model to predict when to stop testing. Software Reliability is defined as the probability of failure-free operation of a system for a specified time in a given environment (Mathur, 2008). Numerous software reliability models, better known as Software Reliability Growth Models (SRGMs) (Lyu, 2007) have been proposed in the past 3 decades in order to determine the optimal software release time. Huang, Lyu and Lin (Huang, 2005; Huang & Lin, 2006; Huang & Lyu, 2005) have proposed several SRGMs in their papers to determine the optimal software release policy. SRGM based software release policy is also used by Gokhale, Lyu and Trivedi (2006). Huang and Boehm (2006) rely on COCOMO II cost-estimation model and the COQUALMO quality-estimation model to determine when to stop testing the software. Fenton et al. (2007) propose Bayesian Networks based approach decide the software release time. Quah (2009) uses a predictive model based on Neural Networks and Genetic Algorithms for making software release decisions. Srivastava (2010; 2008) has done considerable work in this field with module prioritization. His paper describes a Component Prioritization Schema with stringency and fault tolerance approach.

Weighing the demerits of prior work, Dalal and Mallows' (1988) approach does not allocate a specific test time before deciding the time to stop testing by the number of faults encountered. Goel and Okomotu's (1981) paper determines the time to be allotted for testing, but uses a stochastic process to find the optimal testing time, depending on the faults uncovered and is considered the mother paper for finding approaches to solve the problem of when to stop testing. Thus, due to its seminal work three decades ago in the field, this paper has been referred for the comparison with the proposed approach. The above approaches do not prioritize the software components before applying a stopping rule to determine if the software can be released.

Srivastava's (2008) work uses the module prioritization approach, but the equation of Module Priority is not flexible to change as per development and maintenance projects.

Further, the Module Priority formula used in papers prior, mandate all parameters to be available. However, in practice not all parameters to calculate the Module Priority might be available easily. The above solution, does not consider the type of project i.e., software development project or maintenance project while calculating priority.

TPA® (Dekkers & Veenendaal, 1999) is a registered trademark and industry wide accepted method which gives accurate and robust results for total test time. The reliability of TPA results lies in the fact that it covers all the factors that affects the testing process directly or indirectly into its test time calculation process. The significant parameters such as size, complexity, quality characteristics, user-importance of function , function usage-intensity, productivity figures (knowledge and skills of testing-team) and environmental factors such as availability of test tools (automated tools for instance), test-team experience, development and test environment etc., are considered for the test-time calculation.

Each organization may have its own method other than TPA to determine the total testing time. The next section covers the proposed TPA based Module Priority approach in detail.

3. Proposed approach

The TPA (Dekkers & Veenendaal, 1999) based, weighted module prioritization is presented below. Figure 1 shows the architecture diagram of the proposed approach. The figure depicts the calculation of Module Priority from 'Optional' and 'Fixed' parameters.

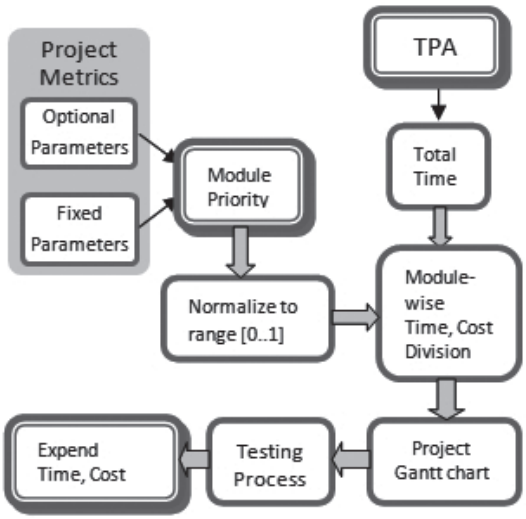


Figure 1 Solution Architecture

The project's metrics that are divided into Fixed and Optional parameters help the project in selectively choosing the metrics that suit the project. For example, a project with lot of Database (DB) calls will have the optional parameter of 'stress factor' (see Section 3.1) in Module Priority calculation, while all fixed parameters like function point will be mandatory to determine a module's priority.

The 'Fixed Parameters' (refer to Section 3.1) are those software metrics (Pressman & Ince, 1992) which is inherent in every software project and constitutes the major metrics determining the characteristics of the project. While the 'Optional Parameters' (see Section 3.1) are the ones which does play a significant role in determining a project's characteristics but it varies from project to project and have not yet been incorporated as a part of software project metrics yet.

After the module priorities are calculated, they are normalized (as seen in Figure 1) to the number range [0, 1] to ensure that all the software modules in the project proportionally divide the numeric range [0, 1] as per their importance in project functionality. Then the total test time for the project is calculated according to TPA (Dekkers & Veenendaal). This test time and budgetary cost of testing is divided in the same ratio as that of normalized Module Priorty values to award each module proportional time to test and allot the modules, the proportional cost from the total allocated cost for testing. Hence, if 3 modules in a project have module priorities of 0.2, 0.3, 0.5, they will be allocated 20%, 30%, 50% respectively of the total project time and cost.

Finally, the KLOC heuristic algorithm is applied after the testing process is started to determine the amount of cost and time a lower priority module can expend to a higher module in case the latter exceeds its allotted time or cost. The periodic check to see if modules exceed their allotted time, cost and the process of borrowing time and cost from lower priority modules helps us thus. It gives the important modules, maximum time and cost for testing, thereby helps form an optimal test release policy.

The Module Priority equation introduced here can be applied to both maintenance and development projects. Since maintenance projects can be considerable in size in a software firm's portfolio, the Module Priority equation factors in this consideration as well. TPA® (Dekkers & Veenendaal, 1999), an industry wide known approach is used to estimate and allot the total time of testing for the software project.

The basic steps of this papers' solution are:

- Software modules are prioritized on multifarious metrics (elaborated in Section 3.1) and the priority of each module is obtained in the range of [0, 1]. It is important to note that the sum of all module priorities is 1.
- Time units to be allotted for testing using TPA® are calculated to divide the cost

(budget of testing) and time allotted in the ratio of the calculated Module Priority. Thus, if 3 modules in a project have module priorities of 0.4, 0.3, 0.3, they will be allocated 40%, 30%, 30% respectively of the total project time and cost. This is the maximum 'Allowance' (time, cost). Division of time and cost as per priority helps the objective of deeper testing of highly critical modules. This division ensures the partition of test time as per the priority of modules.

- Testing is started and the KLOC bucketing algorithm (Section 3.4) is used to check for cost and time overrun so that the lower priority module can give cost and time units to higher priority modules.

The steps described above have been implemented in the Module prioritization tool application (see Appendix), developed to test and validate our approach. This is a project management tool that uses the approach presented in this paper to arrive at an optimal test release time. The module prioritization tool takes a project xml file with all the metrics mentioned in system architecture diagram of Figure 1 to calculate priority of all modules. The tool provides features to compute total test time using TPA and has options to borrow cost and time from lower priority modules and *expend* to higher priority modules. The updated Gantt chart after this process helps a project manage the test phase effectively.

The above solution is explained in depth below.

3.1 Prioritization of modules

The software modules are prioritized, so that the most crucial modules get maximum time in the testing phase. This step is pivotal to an optimized release time and caters to finding the critical errors that need to be uncovered in testing phase. The parameters in these classes are named as Optional and Fixed parameters.

These Optional and Fixed parameters become the prioritization criteria for maintenance and development projects. Two types of parameters are conceptualized in this paper to ease us with projects that may not have all the Optional parameters in determining Module Priority.

3.2 Rationale for optional and fixed parameters

While prioritizing modules in a project, there is a need to recognize that all projects have in common, parameters like LOC or function points etc., which are important and can be obtained or calculated. These parameters are important across all projects and play a significant role in determining module complexity and priority. Yet there can be other parameters that may determine priority like DB calls or threads and these may not be seen in all projects. Two types of parameters are conceived to ease us with projects that may not have all the Optional parameters in determining Module Priority. However,

fixed parameters are mandatory to determine a module's priority. These are indispensable as these serve as a minimum requisite to determine the priority of a module. Hence it is proposed to categorize parameters into Fixed and Optional Parameters.

3.2.1 Fixed parameters (ascertainable in every project)

Function Points/KLOC in Module (where KLOC is Kilo Lines of Code), Node Criticality, Technical Complexity, Coupling in the Module, Reusability (Object Oriented (OO) or Procedural).

LOC metric may not be suitable for every project such as in a web application and thus in such a case, function points could be used. However, a legacy system code could use either function points or the LOC metric.

These parameters do not disappear (unlike Optional parameters) with project type, and the above values can be calculated for all projects. A brief definition for the above fixed parameters is provided below.

Node Criticality: All those modules which are critical to the software project and lie on the critical path in the CPM (Critical Path Method) network (Hughes & Cotterell, 2002; Pressman & Ince, 1992).
Function Points: Software metric used as a means to measure the functionality delivered by the system (Albrecht, 1979; International Function Point Users Group, 1994a; 1994b).
Technical Complexity: It is the complexity of a module which is determined by counting the number of predicate nodes (Goel & Okumoto, 1981; Pressman & Ince, 1992).
Coupling: It is a measure of the relative interdependence among modules (Chidamber & Kemerer, 1994).
Reusability: This is the amount of external and inbuilt libraries a module uses to accomplish its functionality, thereby measuring the degree of reuse (Jacobson, Griss & Jonsson, 1997).

3.2.2 Optional parameters

Stress factor (calculated from the maximum number of DB calls and Data in a Project scope), Synchronization factor (obtained from the number of threading/ synchronization primitive calls in a program). Maintenance factor for maintenance projects (calculated from number of change requests CR's from a module), Frequency of occurrence in Primary Use cases/Flow.

The Risk of Failure of a module (a probability value) may be determined from the above defined optional parameters. Not every project will use threading, or would have database calls distributed across the code. Albeit these parameters increase complexity and chance of failures, they fall in Optional parameters.

Module Priority equation or the AHP Process (Vargas & Saaty, 2001) is used to calculate Module Priority.

3.3 Prioritization of modules by module priority equation

The Module Priority is calculated from the above parameters as

$$\text{Module Priority (un-normalized)} = \frac{(W_1^* P_1 + W_2^* P_2 + \ldots + W_n^* P_n)}{W_1 + W_2 + \ldots + W_n} \text{ or,}$$

$$MP_{ui} = \frac{\sum W_i^* P_i}{\sum W_i}, (0 < MP < \sum W_i) \quad (1)$$

Where W_i and P_i are the weight and parameter value for the ith parameter. It is important to note why we multiply the weight with the parameter as the weights are used to increase the influence or decrease the influence of a parameter in the Module Priority equation. This formula was devised by looking at the CGPA (Cumulative Grade point average) calculation in any university where important "subjects" receive higher "credits" and have a larger contribution to the CGPA. Thus, in the proposed approach, the "subjects" map to the "parameters" and the "credits" map to the weights in our equation. The weights for parameters can be determined by the Technical Manager in a project. This is done by taking into account factors like experience, capability of the manager and by also looking at similar projects in the same environment.

Module Priority formula above assigns weights from 5 through 1 to parameters. The management can decide to award the highest weight to a parameter if there are a lot of threading calls in the module and the module falls under the Critical path (Hughes & Cotterell, 2002). Thus this module could be assigned a 'Node criticality's weight to be 5 (see Table 1).

Table 1 Priority Weights Associated with Priority Levels

Highest	5
High	4
Medium	3
Low	2
Lowest	1

Thus a Maintenance project with lot of threading calls will have Module Priority calculated as:

MP_{ui} = $Weight_1$* Node Criticality + $Weight_2$* Function Points + $Weight_3$* Reusability + $Weight_4$* Synchronization Factor* $Weight_5$* Maintenance Factor / ($Weight_1$ + $Weight_2$ + … + $Weight_5$)

3.4 Selection of optional and fixed parameters

The selection of fixed parameters is based on judgment of the project manager. In the fixed parameters, function points over KLOC might be chosen if the programming language inherently takes more lines of code to achieve functionality, yet any one should be included in the prioritization formula.

A web application project could also use the function points as the fixed parameter as LOC metric is not suited to such a project. Similarly, optional parameters chosen or discarded are also based on judgment of the project manager by looking at factors like DB calls or threading calls that affect performance or have chances of program crashes or bugs. The manager may look at empirical evidence to decide the optional parameters by looking at number of bugs, crashes from previous projects with modules having extensive DB calls, threads.

For example, a development project with no threading calls will have all fixed parameters and no Synchronization factor in MP_{ui} calculation.

Module Priority (MP-normalized) is calculated as:

$$MP_i = \frac{MP_{ui}}{MP_{u1} + MP_{u2} + \ldots + MP_{un}} \qquad (2)$$

Development projects would use all the Fixed Parameters with Optional Parameters like Synchronization factor, if the project used considerable threading. A maintenance project will have maintenance factor (number of service requests) to prioritize modules.

The strength of the MP_i formula lies in the fact that it incorporates all major software metrics, which provide a robust assessment of the modules. Hence, depending upon the MP_i values, the module prioritization task is more efficient, effective and reliable.

3.5 An alternative prioritization technique for modules by analytic hierarchy process

The manager may also prioritize the modules by using Fixed and Optional parameters using the AHP Process (Saaty & Vargas, 2001). The management has to use its judgment to decide which parameter is weighted how much with respect to other

parameters. The process for determining Module Priority by AHP is described below:

Step 1: All the Optional and Fixed Parameters are written in matrix form and the manager decides the relative importance of one parameter over the other. This is done by considering factors like: experience, capability of the manager and also by looking at similar projects in the same environment (see Table 2). (Note: The matrix below shows only 5 criteria for ease of explanation. The real matrix would have all the parameters (Fixed and Optional, written in matrix form with relative importance to arrive at Module Priority.)

Table 2 Matrix to Decide the Relative Importance of Optional and Fixed Parameters

	Node Criticality	Function Point	Technical Complexity	Coupling	Reusability
Node Criticality	1/1	2/1	2/1	3/1	4/1
Function Point	1/2	1/1	1/2	1/2	3/1
Technical Complexity	1/2	2/1	1/1	1/2	2/1
Coupling	1/3	2/1	2/1	1/1	3/1
Reusability	1/4	1/3	2/1	1/3	1/1

Step 2: Express the matrix by replacing fractions by decimal. The matrix can be called **A** (see Table 3).

Table 3 Matrix (in decimal) to Decide the Relative Importance of Optional and Fixed Parameters.

	Node Criticality	Function Point	Technical Complexity	Coupling	Reusability
Node Criticality	1	2	2	3	4
Function Point	0.5	1	0.5	0.5	3
Technical Complexity	0.5	2	1	0.5	2
Coupling	0.333	2	2	1	3
Reusability	0.25	0.333	2	0.333	1

Step 3: The matrix is squared and the *eigen vectors* are found (see Table 4).

Table 4 Matrix (Squared) to Decide the Relative Importance of Parameters

	Node Criticality	Function Point	Technical Complexity	Coupling	Reusability
Node Criticality	4.99	15.33	19	9.33	27
Function Point	2.1665	5	9	3.75	9.5
Technical Complexity	2.665	6.66	8	4.16	13.5
Coupling	3.41	9.66	11.66	5	17.32
Reusability	1.7739	5.82	5.325	2.215	8

The row totals are added and normalized to get the eigen vector shown below:

[0.3598]
[0.1398]
[0.1664]
[0.2238]
[0.1100]

Step 4: The process is repeated by multiplying the matrix again with matrix **A** to normalize until the eigen vectors converge with previous step up to an accuracy of 4 decimal places. The vector determines the priority of each criterion and offers ranking of the criteria as shown below (see Figure 2):

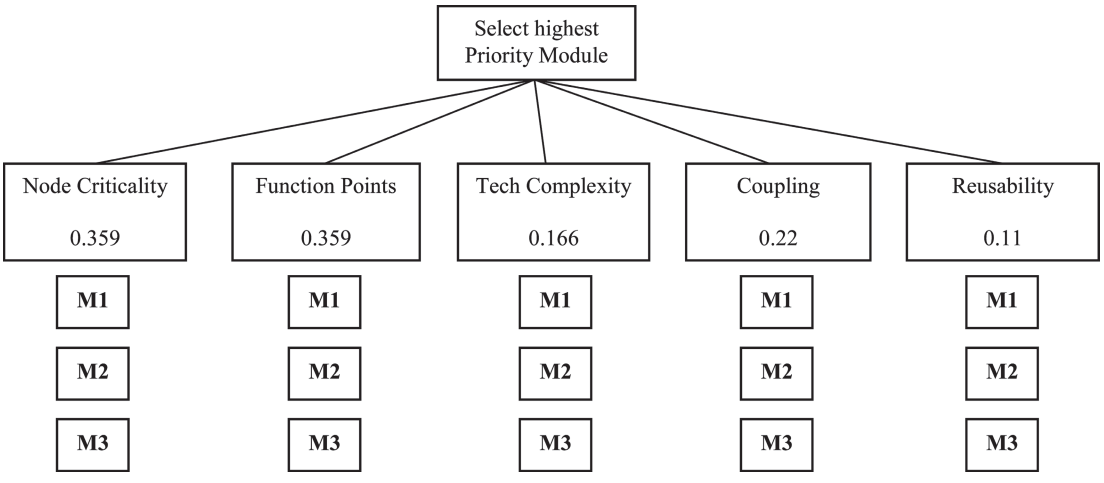


Figure 2 Ranking of Criteria is Determined

Step 5: On each criterion (parameter), the modules (Say M1, M2, and M3) are ranked by relative importance to draw up a matrix as shown below for one parameter (see Table 5).

Table 5 Matrix Showing Ranking of Modules

	Node Criticality		
	M1	M2	M3
M1	1/1	2/1	3/1
M2	1/2	1/1	4/1
M3	1/2	1/4	1/1

Step 6: The matrix for each criterion is defined and eigen vector is found for each matrix.

Step 7: Module Priority is obtained by multiplying [Module, Criteria] Matrix with the Initial Criteria Ranking Matrix (see Table 6)... to get the module priorities as shown in Table 7:

Table 6 [Module, Criteria] Matrix and Criteria Matrix are Multiplied to Get Module Priorities

	Node Criticality	Function Points	Technical Complexity	Coupling	Reusability	Criteria Rank
M1	4.99	15.33	19	9.33	27	0.35
M2	2.1665	5	9	3.75	9.5	0.139
M3	2.665	6.66	8	4.16	13.5	0.166

Table 7 Module Priorities are Determined

M1	0.5
M2	0.3
M3	0.2

(Note: The above matrix has imaginary values)

Once the priority of modules is found using either Module Priority equation or the AHP Process (Saaty & Vargas, 2001), the cost and time for modules are allocated based on priority.

3.6 Allocating time, cost based on module priorities

From the Module Priority values, the total time-units obtained by TPA is distributed among the various modules. The time and cost limit allotted to each module is called the 'Maximum Allowance' or Allowance (Time, Cost) of the module. As the name suggests, this is the maximum allotted time & cost for each individual modules.

$$Time_{(max)i} = MP_i^* \ Total\ time\ units \quad (3)$$

$$Time_{(max)i} = MP_i^* \ Total\ test\ budget \quad (4)$$

Each module has its Allowance (T_{max}, C_{max}) value. Allowance (max) used from here denotes the maximum cost or time allotted for the module.

Figure 3 shows the Gantt chart (Greene & Stellman, 2005) showing the individual allowance for each modules M1, M2 and M3 of a software project to undergo testing.

Figure 3 Project Gantt Chart after Awarding Cost, Time to Mules as per Module Priority

Clearly, Allowance (max) for the module M1 is greatest since its Module Priority value is the highest; M3's Allowance (max) is lowest as it is the module with least priority.

Since module M1 has the highest priority, it is one of the most important and critical modules in the software and it becomes imperative that such a high priority module must be tested as fully as possible. But what if the testing time for Module M1 (i.e., Allowance (max)) runs out and thus leaving the module M1 partially tested. Obviously, M1's testing can't be dropped to proceed to test the next modules if lower priority modules can be used to borrow test time and cost. To avoid a bug in M1 that may cripple the software with critical faults, one solution is to continue the testing process until module M1 is tested completely.

Next section describes an optimal approach called 'Expendability' that is used to borrow time and cost from lower priority modules and expend the same to higher priority modules.

3.7 Expendability

In order to ensure that the high priority modules are tested fully, the idea of

'Expendability' is introduced. It may happen that the testing time for high priority modules run out. In such a situation, a slice of time (or cost, depending upon time or cost the project considers to measure testing phase) is taken away from the lowest priority module's share of time-slot/cost in order to continue the testing process of modules with high priority. The amount of time a module can 'give up' or expend to a higher priority module is termed as *Expendability*. The individual time limit or time-slots or Allowance (max) for each module to undergo testing is already available.

Expendability is based on any of the following two parameters.

- Node Criticality : Found by Critical Path Method (CPM) (Dekkers & Veenendaal, 1999; Hughes & Cotterell, 2002)
- Number of requirements the component fulfills in the primary Use Case (if the user interacts with the system)

Expendability is calculated in the following manner.

Expendability equation:

$$\textbf{\textit{Expendability}} = \textbf{\textit{dev}}_i / \sum \textbf{\textit{dev}}_i \qquad (5)$$

Where, $\textbf{\textit{dev}}_i$ = deviations from next higher priority module.
$\sum \textbf{\textit{dev}}_i$ = sum of all deviations.

Deviation (dev_i) calculation:

Deviations are calculated based on the individual Node-criticality values of the modules.

$$\textbf{\textit{dev}}_i = N_{c_{i-1}} - N_{c_i} \qquad (6)$$

Where, N_c = Node Criticality.

The above Expendability equation has been explained more clearly with the help of an example in Section 5. Figure 4 below shows the Gantt chart with Expendability depictions for the respective modules. Amount of Expendability is shown with the white-shades.

Each lower priority module expends its share of time and cost to its higher priority module whenever the testing time or cost of the latter runs out and thus the higher priority

Figure 4 Project Gantt Chart after Expending Cost, Time from M2, M3 to M1

modules which are critical, get additional time, cost to continue their testing, hence reducing the risk of the critical modules not being tested completely. If the expendability of a module becomes negative, then it is set to zero. There is a rare chance when a lower priority module gets higher value of Node Criticality than a higher priority module.

After testing is started, it is essential to define a stopping rule for the test phase. It is also necessary to handle the scenario where a higher priority module requires more time/cost than its maximum Allowance (Cost, Time). This section explains the KLOC bucketing heuristic algorithm for an optimal test policy.

3.8 KLOC bucketing heuristic algorithm

The Allowance (maximum cost or time allotted to a module) is calculated by dividing the total time (found by TPA (Dekkers & Veenendaal, 1999)) and the total testing budget in the ratio of Module Priorities. Hence the T_{max} (maximum time to test a module) and C_{max} (maximum cost to test a module) are obtained and form the Allowance (Cost, Time) used in the algorithm below. If a project evaluates allowance by cost, Allowance (Cost, Time) returns cost. It returns time otherwise. The Algorithm is shown below.

(1) For Index = Highest to lowest MP,
If(Allowance(current) < Allowance (max))
Continue testing;

(2) While (Allowance(current) >= Allowance (max))
Find Min (List (MP_1, MP_2…, MP_n))

(3) Compute Expendability of MP (min)

(4) Use KLOC bucketing Heuristic to decide low priority module

(5) Update

T_{max} of MP (min) =
T_{max} of MP (min) - Expendable (time)

C_{max} of MP (min) =
C_{max} of MP (min) - Expendable (cost)

Update Allowance (max) for MP (min)

(6) Remove MP (min) from List
T_{max} of MP (max) =
T_{max} of MP (max) + Expendable (time)

C_{max} of MP (max) =
C_{max} of MP (max) + Expendable (cost)

(7) Update Allowance(max) for MP (min)

(8) End While

The above algorithm is used in the testing phase to check against cost or time over-run (Step 2). In such a scenario, the expendability for all modules lower in priority to the current module is computed to borrow time/cost units (Step 5) for the current module. This process continues for all the modules except the least priority module. Step 4 in the algorithm is explained in the next section.

3.9 Stopping rule

The above algorithm runs for two iterations before it is halted. It is upon the project manager to decide whether to go for the third iteration to borrow time and cost from lower priority modules. The project manager might want to go for the 3rd iteration in some cases if he feels that some more time should be given on testing the modules. So it is the manager's discretion to extend the time for testing if he feels to do so, depending upon the deadlines.

KLOC bucketing heuristic is used to group modules depending on code size. This heuristic groups modules with similar priority and near code size (KLOC) into the same buckets. Then the algorithm ensures that a larger module within a bucket of low priority modules (from the group of buckets) expends time and cost, than other modules with less code size within the same bucket, so that not all lower priority modules in a bucket expend time or cost.

4. Merits of this approach

This approach uses a Module priority calculation technique that is flexible to fit with maintenance and development projects. As it follows the TPA (Dekkers & Veenendaal, 1999) approach to allot time and cost for each module with calculated module priorities, it is more accurate and robust as compared to the stochastic and probabilistic based approaches. This is because former involves all the major software metrics and quality characteristics for the time estimation while the latter approaches are based mainly on

assumptions. The proposed approach also ensures that no critical modules are left untested completely. Our proposed algorithm iterates until the stopping rule is met and the management is left to decide when to stop testing after evaluating the extra time, cost each higher priority module has been awarded. A pivotal facet in contrast to Goel and Okumoto (1981) and Dalal and Mallows (1988) approach is these two models don't use the concept of *borrowing* from low priority modules to higher priority modules.

The next section presents a case study, where a well known open source project from Apache Software Foundation (International Function Point Users Group, 1994a) is chosen to calculate the priorities of modules as per the described metrics.

Then the KLOC heuristic algorithm is applied to analyze the amount of time and cost the higher priority modules receive in case of overrun, thereby illustrating the optimal test release policy of this paper.

5. Case study

An open-source project was taken from Apache Software Foundation (Logging Service, 2012) in order to analyze the proposed approach. This was then compared to Goel and Okumoto (1981), Dalal and Mallows's work (1988) and Srivastava's work (2008) and the results drawn as a graphical presentations shown in Section 6 (Figure 10, Figure 11, Table 10 and Table 11). The approach uses to calculate Module priorities by Module Priority Equation (1) and not the AHP process (Vargas & Saaty, 2001).

5.1 Apache Logger

Every software application includes logging/tracing APIs during its development phase as it plays an important role in tracing the entire build and execution process of an application code. It simplifies the job of debugging to a great extent. The Apache logging project - "log4xxx" logs statements in a file with efficiency and reliability (Greene & Stellman, 2005). The Apache logger project has a configuration file (an XML file) which could be changed according to the requirements. It has mainly three modules, namely Logger, Appender and Layout (Logging Service, 2012). Figure 5 depicts the basic architectural context diagram for the Logging project.

The main logger modules are explained as follows:

Logger- Creates a singleton class and initializes the Logger engine. It defines functions to log statements with TRACE, DEBUG, INFO, WARN, ERROR, FATAL levels.

Appender- Determines the output mode to write the logs and defines classes to write to Console, File and other output modes

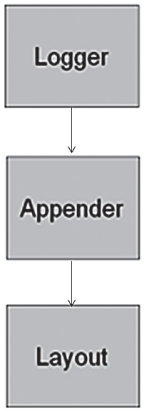


Figure 5 Apache Logger Modules in Program Flow

Layout- Defines classes to add functionality to have varied layout options to the log statement.

Calculating Module Priority (MP):
Module Priority (MP_{ui}) = $\sum W_i^* P_i / W_i$
Weighted priority for KLOC = 3
Weighted priority for Coupling (C_p) = 4
Weighted priority for TC = 5
Weighted priority for Reusability (R_e) = 4
Weighted priority for Node Criticality (N_c) = 4
The values are shown in the Table 8.

Table 8 Project Metrics Calculated in Apache Logger Project

Modules	KLOC	Coupling (Cp)	Technical Complexity (T.C)	Reusability	Nc
Logger	4.726	0.925	14	51	3
Appender	3.728	0.895	12	34	2
Layout	2.157	0.895	8	35	1

Step 1: Calculate Module Priority (MP)

Module Priority (MP) values for the respective modules in the Logger project are given below.
Logger Module:

$(3^*KLOC + 4^*C_p + 5^*T_c + 4^*R_e + 4^*N_c)/ \sum W_i$
$= (3^*4.726 + 4^*0.925 + 5^*15 + 4^*51 + 4^*3)/ 20 = 15.44.$

Similarly,
For Appender Module: MP = 10.938
Layout Module: MP = 9.657

Step 2: Normalization of Module Priorities:-

MP_1 (Logger) = $(MP_1/ (MP_1 + MP_2 + MP_3)) = 15.44/ (15.44 + 10.938 + 9.657) = 0.428$
MP_2 (Appender) = $(MP_2/ MP_1 + MP_2 + MP_3) = 0.303$
MP_3 (Layout) = $(MP_3/ MP_1 + MP_2 + MP_3) = 0.267$

Therefore, $MP_1 > MP_2 > MP_3$

Step 3: Determine total testing time using TPA (Vargas & Saaty, 2012):

Using the various steps, equations and formulae given in TPA method (Dekkers & Veenendaal, 1999) the total Test-Hours are calculated. A brief overview of the main steps and formula taken from the TPA method (Dekkers & Veenendaal) are given below. The appropriate values for TPA parameters of Logging project have been used in the calculation depicted below.

Step 3.1: Calculate the Dynamic Test Points (TP_f) (Dekkers & Veenendaal, 1999)

$$\boldsymbol{TP_f = FP_f * D_f * Q_d} \quad (7)$$

Where $\boldsymbol{TP_f}$ = number of dynamic test points assigned to each individual function.
$\boldsymbol{FP_f}$ = number of function points assigned to the function.
$\boldsymbol{D_f}$ = function dependent factors.
$\boldsymbol{Q_d}$ = dynamic quality characteristics.

IFPUG CPM 4.0, (International Function Point Users Group,1994a; 1994b), a widely accepted standard, is employed for computing the function points in this paper.

$\boldsymbol{TP_f}$ (Logger) = 87.3 * 1.2 * 1.005 = 105.805
$\boldsymbol{TP_f}$ (Appender) = 40.94 * 0.96 * 0.825 = 32.422
$\boldsymbol{TP_f}$ (Layout) = 19.36 * 0.75 * 0.825 = 11.979

Step 3.2: Calculate the total number of test points (TP)

$$\boldsymbol{TP} = \textbf{sum total } \boldsymbol{(TP_f)} + \boldsymbol{(FP * Q_s)}/500 \quad (8)$$

Where $\boldsymbol{TP_f}$ = the number of dynamic test points.
$\boldsymbol{F_p}$ = the Function Point count (FP) for the complete system to be tested.
$\boldsymbol{Q_s}$ = Static Test Points
$\sum \boldsymbol{TP_f}$ = 150.206

$$TP = \sum TP_f + (\sum FP_f * \sum Q_s)/ 500$$
$$= 150.989 + (\sum FP_f * \sum Q_s)/ 500 = 150.989$$

Step 3.3: Calculate the primary test hours (PT)

$$\boldsymbol{PT = TP * S * E} \quad (9)$$

Where $\boldsymbol{TP}$ = The total number of test points for the complete system under testing.
$\boldsymbol{S}$ = Skills factor
$\boldsymbol{E}$ = Environmental factor
$\boldsymbol{PT}$ = 140.12 * 1 * 1.5 = 210.18

Step 3.4: Calculate the total number of test hours (TH).

$$\boldsymbol{TM = PT * (T + C)}/ 100 \quad (10)$$

$$PT * (T + C)/ 100 = 200.01$$

$$\boldsymbol{TH = PT + TM} \quad (11)$$

Where $\boldsymbol{TM}$ = Team Management Allowance
$\boldsymbol{PT}$ = The primary test hours
$\boldsymbol{T}$ = The team size factor
$\boldsymbol{C}$ = The planning and control factor

TH = 210.18 + 200.01 = 410.19 hours

By applying above steps to the Apache Logger Module (Logging Service, 2012), the value for TH is calculated to be 410.19 hours.

The Gantt chart as shown in Figure 6 is drawn based on the above values of Module Priority (MP). Total time unit calculated by TPA is distributed among the modules depending on their MP values. E.g., if there are 410.19 total time units as calculated by TPA method, then module M1 gets MP1 * 410.19 = 0.426 * 410.19 = 174.47 time units. Similarly for M2 = 0.303 * 410.19 = 124.28 and M3 = 109.52.

Figure 6 Logger Project Gantt Chart Initially after Allotting Time, Cost to M1,M2 and M3

Step 4: Next, Expendability is calculated in the following manner:

Let the Node-criticality of the modules be:

MP1 = 5
MP2 = 3
MP3 = 1

Therefore $\boldsymbol{dev}_{M2}$ (from M1) is: - 5 – 3 = 2
Similarly $\boldsymbol{dev}_{M3}$ (from M1) is: - 5 – 1 = 4
$\sum \boldsymbol{dev}$ (sum of all deviations) = 2 + 4 = 6.

Therefore Exp_{M2} (Expendability of M2) = $\boldsymbol{dev}/\sum \boldsymbol{dev}$ = 2/6 = 1/3.
Similarly $Exp_{M3} = \boldsymbol{dev}/\sum \boldsymbol{dev}$ = 4/6 = 2/3.

Gantt chart with the Expendability calculation is shown in Figure 7.

White-strips in the Gantt chart are the Expendability values that could be expended to the higher priority modules. In case the testing time-slot for M1 get exhausted and M1 is still not completely tested, then 2/3rd of M3 time-slot will be expended to M1; i.e., 2/3 * 109.52 = 73.01. Therefore, the updated time-slot (Allowance (new)) of M1 = 174.47 + 73.01 = 247.48 units. And M3's time-slot reduces to 109.52 – 73.01 = 36.51 units. The iterations are shown in Figure 7 and Figure 8.

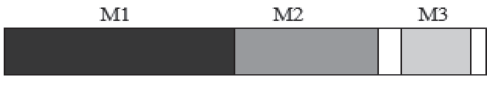


Figure 7 Gantt Chart after Calculating Expendability Cost, Time of M2, M3

The white space in the above Gantt chart shows the time/cost being awarded to M1

from M2 and M3. Figure 8 shows the expendability of M2, M3 merged to M1. No more time or cost will be expended to M1 if an overrun occurs. Figure 8 also depicts the next iteration of expendability that calculates the share of time, cost M3 will give to M2 while testing M2.

Figure 8 Gantt Chart with Expendability Space in M3 to be Given to M2

In the second iteration of the algorithm, the testing of M1 is complete. Figure 9 shows M2 being tested and borrowing time, cost from the least priority module M3. Hence the expendability white space of M3 being merged to M2 is depicted in the below figure.

Figure 9 Gantt Chart after Expending Cost, Time from M2, M3 to M1

The next section presents a comparison between this paper's approaches to prior approaches discussed in Section 2.

It is also important to note that we tested the proposed approach of module prioritization with other in-house projects like Student's Registration System, Library Management System and other open source applications, but chose the Apache Logger Project (Logging Service, 2012) to be included in the paper for ease of illustration.

5. Comparison

The proposed approach has been compared with the other standard approaches and the comparison is thus justified with graphical representation in (Figure 10, Figure 11, Table 10, and Table 11)

5.1 Goel & Okumoto's work

It follows a time-dependent & non-homogenous poisson based model to determine the optimal time for testing. Goel and Okumoto's model says:

If **m(t)** is the expected number of software failures detected in time t., then

$$m(t) = E(N(t)) \quad (12)$$

where, **N(t)** is the cumulative number of faults encountered in time t and

Goel and Okumoto have shown that **m(t)** can be described as:

$$m(t) = a(1 - e^{-bt}) \quad (13)$$

When $t \rightarrow \infty$, then $m(\infty) = a$.

Where **'a'** represents expected number of faults detected in the entire life time of the software and b represent the detection rate of the individual fault and its value depends upon quality of testing.

5.2 Comparison with the paper's proposed approach

The Goel & Okumoto model (1981) was applied to the Case Study "Apache Logging Application" (Logging Service, 2012) (an open source project) and the results are given below. Appropriate values for the time estimation are chosen.

a = 56860; b = 0.124; c1 = 10; c2 = 50; c3 = 1,000; t = 1,000 units

Then a*b = 7050.64 and Cr = 25
Since a*b > Cr, the optimal test policy will be
T = min (1/b ln (ab/Cr), t)
Therefore T* = min (45.5, 1000)
T* = 45.5

Therefore, the total estimated time-units come out to be 45.5 units. Time-units are mapped to staff-days.

With the TPA approach, the total time to test was 410 hrs i.e, 410.19/10 (if each staff-days = 10hrs) = 41.01units (in days).The comparison graph between these two models is shown below.

Thus, from graph in Figure 10, it can be seen that the TPA based approach results in less time to release the software for this case study.

5.3 Comparison with the Dalal & Mallow's Approach

Dalal and Mallow (1988) present a stochastic based model which assumes that there are N unknown faults in the system while this paper uses a TPA based approach instead of a stochastic model. This paper first calculates the total test time using TPA as opposed to a

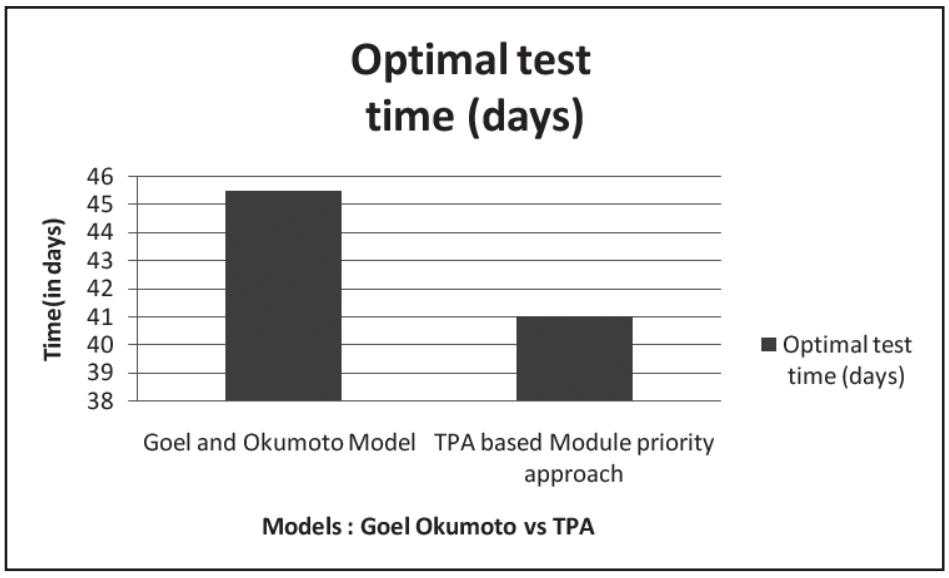


Figure 10 Optimal Test Days Calculated for Logger Project with Goel-Okumoto and TPA Approach

stopping rule based on number of faults. The algorithm normally stops (further iterations are possible if the management decides to expend more time, cost) with the second iteration of KLOC Bucketing Heuristic Algorithm (Section 3.4) where the lower priority modules expend their time, cost to higher priority modules than a faults based stopping rule.

The fundamental difference in this approach to Dalal's, is the estimation of total test time. While Dalal's work only used a stopping rule with a stochastic process with respect to the number of faults detected in the system, our approach estimates the time of test first.

5.4 Praveen R. Srivastava's Approach

This model prioritizes the modules on a priority scale from very high to very low depending upon the values obtained from the Cumulative Score (CS).

The formula for CS is presented as follows:

$$\boldsymbol{CS = e^{CR} * KLOC * CC * CP / KLOC_{avg}} \qquad (14)$$

Where ***CR***= Component Rank, ***CC*** = Cyclomatic Complexity. ***CP*** = customer priority (assigned a default value, if not specified by the customer), ***KLOC*** = Kilo lines of Code, = average of all the KLOCs.

This formula is applied in the Case Study (Apache Logger project) as below.

Component Rank (CR) (Musa &Ackerman, 1989):

First, all the modules get the equal CR. Since there are 3 modules in total, each module will get CR = 0.33. As M2 & M3 both are dependent on M1, therefore,

$$CR_{new}(M1) = C2(M2)/\ 2 + C3(M3)/\ 2 \quad (15)$$

Or CR_{new} (M1) = 0.33 + 0.33 = 0.66

Both M2 & M3 has 2 out-bound links,

CR_{new} (M2) = 0.33/2 = 0.165

Similarly, CR_{new} (M3) = 0.165

Calculating Cumulative Score (CS) (Musa &Ackerman, 1989):

$$CS = e^{CR} * KLOC * CC * CP/\ KLOC_{avg}$$

$CS(M_1)$ = e0.66 * 4.726 * 14 * 0.925/(4.726 + 3.728 + 2.157) = 11.16

$CS(M_2)$ = e0.165 * 3.728 * 12 * 0.895/(4.726 + 3.728 + 2.157) = 4.45

$CS(M_3)$ = e0.165 * 2.157 * 8 * 0.890/(4.726 + 3.728 + 2.157) = 1.716

Clearly,

$CS(M_1) > CS(M_2) > CS(M_3)$. Based on their Cumulative score each module is allotted a priority weight from high to low (see Table 9).

Table 9 Priority Levels with Weights in Praveen's Model

Modules	Priority Wts.	Priority Type
M1	1	High
M2	2	Medium
M3	3	Low

To compare the proposed approach and Praveen's Module Priority approach, a simple maneuver is done to convert Praveen's Module Priorities within a range of 0 to 1.

Thus M1 (new) = (sum of all weights) – (m1's weight) / [(sum of all weights)] = 6-1/6 = 5/6 = 0.833. Similarly, M2 (new) = 6-2/6 = 4/6 = 0.66, M3 (new) = 6-3/6 = 3/6 = 0.5

The comparison graph between TPA based Module Priorities and Praveen's set of

modules priorities is plotted in Figure 11.

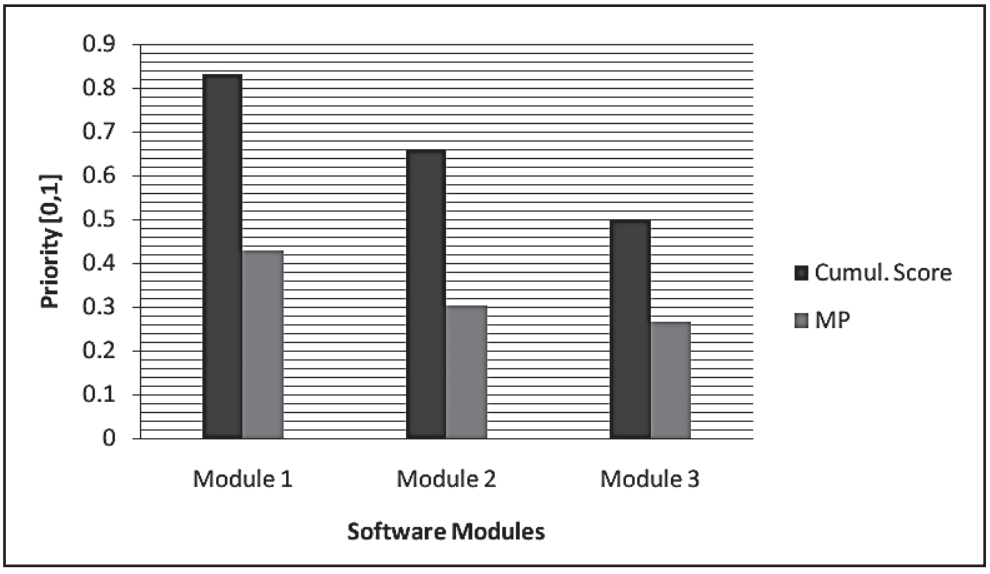


Figure 11 Module Priorities for M1, M2, and M3 Calculated from Praveen's Approach, TPA Based Model

Where, CS (in Table 10) stands for Cumulative Score (1989).

Table 10 Module Priority vs Cumulative Score

Modules	CS	MP
Module 1	0.833	0.428
Module 2	0.66	0.303
Module 3	0.5	0.267

Thus, it can be seen from the above graph that Module Priorities are in same order of priority from both the models. This graph illustrates that Module Priority calculated from optional and fixed parameters does not affect the order of priority in modules, yet there is more flexibility in choosing project metrics in this paper's Module Priority equation.

5.5 Advantage over Praveen's approach

This paper's approach allows the flexibility to choose Function points or KLOC and use Critical path or 'number of use cases' to determine the criticality. The equation for Module Priority is flexible to change as per the type of project. Table 11 below shows a comparison of the TPA based approach with these prior models.

Table 11 Comparison of Prior Model with TPA Based Module Priority Approach

Criteria/Model	Dalal	Goel and Okumoto	Praveen's Model	TPA's Model
Prioritization of modules with equation	×	×	✓	✓
Modified priority equation for development and maintenance projects	×	×	×	✓
Feasibility to add Fixed and Optional parameters for threading and high intense projects	×	×	×	✓
Faults based stopping rule	✓	✓	✓	×
Time of test estimation	×	✓	✓	✓
Lending of time, cost to higher priority module	×	×	✓	✓

6. Conclusion

The proposed Module Priority approach using TPA (Dekkers & Veenendaal, 1999) gives accurate and robust results (as per the industry standards) to determine Module Priority and total test time. Since it involves allotting a test time by TPA (Dekkers & Veenendaal) unlike prior models, it ensures a reasonably accurate time awarded for the test phase. Module Priority normalization ensures that higher priority modules get maximum test time than other less important modules. Further, the concept of Module Priority fits for maintenance and development projects and it can also be extended for re-engineering projects. This approach ensures that the critical modules are tested to the fullest in a given test time. This model also buckets the modules so that the higher priority modules can first borrow units from lower priority modules with greater KLOC, so that even the lower priority modules with least KLOC have a better chance of not being ignored in testing.

References

Albrecht, A.J. (1979), 'Measuring application development productivity', *Proceedings of the Joint SHARE, GUIDE, and IBM Application Development Symposium*, Monterey, CA, October 14-17, pp. 83-92.

Chidamber, S.R. and Kemerer C.F. (1994), 'A metric suite for object oriented design', *IEEE Transactions on Software Engineering*, Vol. 20, No. 6, pp. 476-493.

Dalal S.R. and Mallows C.L. (1988), 'When should one stop testing software?', *Journal of the American Statistical Association*, Vol. 83, No. 403, pp. 872-879.

Dekkers, T. and Veenendaal, E. (1999), *Test Point Analysis: A Method for Test Estimation*,

Shaker, Maastricht, the Netherlands.

Fenton, N., Neil, M., Marsh, W., Hearty, P., Marquez, D., Krause, P. and Mishra, R. (2007), 'Predicting software defects in varying development lifecycles using Bayesian nets', *Information and Software Technology*, Vol. 49, No. 1, pp. 32-43.

Goel, A.L. and Okumoto, K. (1981), 'When to stop testing and start using software?', *ACM SIGMETRICS Performance Evaluation Review*, Vol. 10, No. 1, pp. 131-138.

Gokhale, S.S., Lyu, M.R. and Trivedi, K.S. (2006), 'Incorporating fault debugging activities into software reliability models: A simulation approach', *IEEE Transactions on Reliability*, Vol. 55, No. 2, pp. 281-292.

Greene, J. and Stellman, A. (2005), *Applied Software Project Management*, O'Reilly Media, Sebastopol, CA.

Huang, C.-Y. (2005), 'Cost-reliability-optimal release policy for software reliability models incorporating improvements in testing efficiency', *Journal of Systems and Software*, Vol. 77, No. 2, pp. 139-155.

Huang, C.-Y. and Lin, C.T. (2006), 'Software reliability analysis by considering fault dependency and debugging time lag', *IEEE Transactions on Reliability*, Vol. 55, No. 3, pp. 436-450.

Huang, C.-Y. and Lyu, M.R. (2005), 'Optimal release time for software systems considering cost, testing-effort and test efficiency', *IEEE Transactions on Reliability*, Vol. 54, No. 4, pp. 583-591.

Huang, L.G. and Boehm, B. (2006), 'How much software quality investment is enough: a value-based approach', *IEEE Software*, Vol. 23, No. 5, pp. 88-95.

Hughes, B. and Cotterell, M. (2002), *Software Project Management*, McGraw-Hill, London, UK.

International Function Point Users Group (1994a), *International Function Point Users Group*, available at http://www.ifpug.org/ (accessed 5 November 2012).

International Function Point Users Group (1994b), *Function Point Counting Practices Manual, Release 4.1.1*, available at http://perun.pmf.uns.ac.rs/old/repository/research/se/functionpoints.pdf (accessed 5 November 2012).

Jacobson, I., Griss, M. and Jonsson, P. (1997), *Software Reuse: Architecture, Process and Organization for Business Success*, Addison-Wesley, New York, NY.

Logging Service (2012), 'Short introduction to log4j: Ceki Gülcü, March 2002', *Apache*,

available at http://logging.apache.org/ (accessed 5 November 2012).

Lyu, M.R. (2007), 'Software reliability engineering: a roadmap', *Proceedings of 2007 Future of Software Engineering*, Washington, DC, pp. 153-170.

Mathur, A.P. (2008), *Foundation of Software Testing*, China Machine Press, Zurich.

Musa, J.D. and Ackerman, A.F. (1989), 'Quantifying software validation: when to stop testing?', *IEEE Software*, Vol. 6, No. 3, pp. 19-27.

Myers, G.J. (1976), *Software Reliability Principles and Practices*, John Wiley & Sons, New York, NY.

Pressman, R.S. and Ince, D. (1992), *Software Engineering: A Practitioner's Approach*, McGraw-Hill, New York, NY.

Quah, T.S. (2009), 'Estimating software readiness using predictive models', *Information Sciences*, Vol. 179, No. 4, pp. 430-445.

Saaty, T. and Vargas, L.L.G. (Eds.) (2001), *Models, Methods, Concepts and Applications of the Analytic Hierarchy Process*, Kluwer Academic, Norwell, MA.

Sommerville, I. (2011), *Software Engineering*, Pearson Higher Education, London, UK.

Srivastava, P.R. (2008), 'Model for optimizing software testing period using non homogenous poisson process based on cumulative test case prioritization', *Proceedings of the TENCON 2008 - 2008 IEEE Region 10 Conference*, Hyderabad, India, November 19-21, pp. 1-6.

Srivastava, P.R. (2010), 'Non homogeneous Poisson process model for optimal software testing using fault tolerance', *MIS Review*, Vol.15, No. 2, pp 77-92.

About the authors

Praveen Ranjan Srivastava is working as an Assistant Professor in the Information Technology and Systems Group at Indian Institute of Management (IIM), Rohtak India. He is currently doing research in the area of software Engineering and Management using novel metaheuristic techniques. His research areas are software testing, quality management; quality attributes ranking, effort management, software release management, test data generation, agent modeling, expert system, decision science and advanced soft computing techniques. He has published more than 100 research papers in various leading international journals and conferences in the area of software engineering and management. His prime research area is software validation. He is **Editor in Chief** of *International*

Journal of Software Engineering Application and Technology (IJSEAT), published by Inderscience. He has received various funds from different agencies like Microsoft, IBM, Google, DST, CSIR etc. He is also member of Editorial Board of various leading journals. He has been actively involved in reviewing various research papers submitted in his field to different leading journals and various international level conferences. Contact him at praveenrsrivastava@gmail.com.

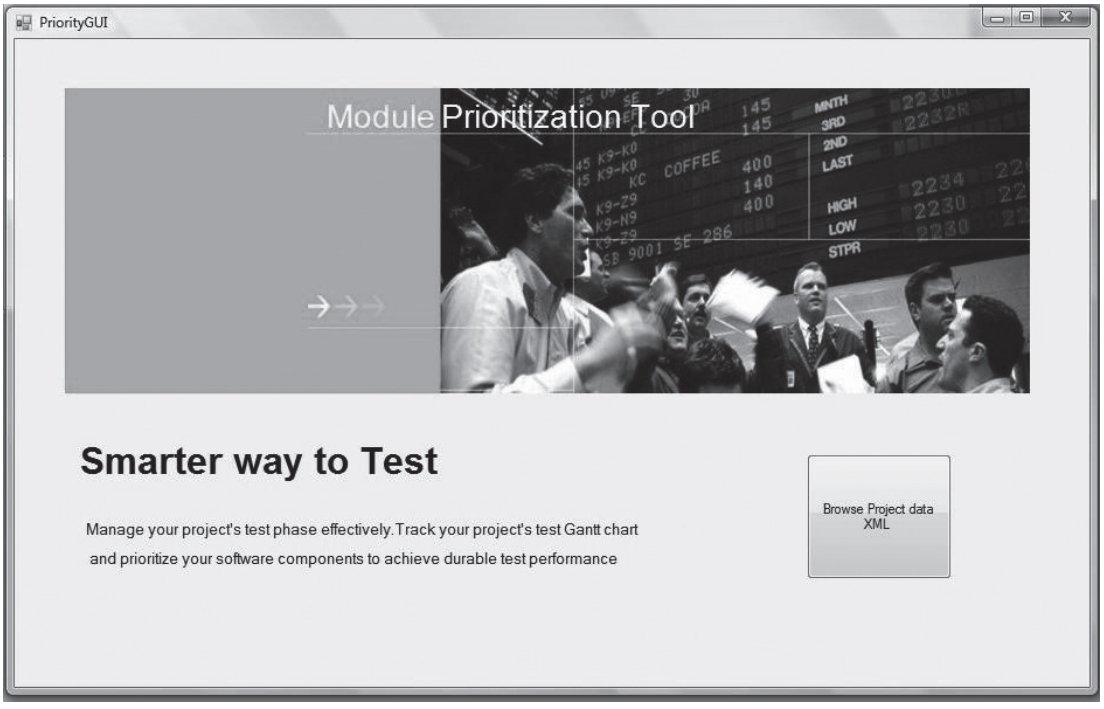
Subrahmanyan Sankaran has completed his M.E. (Software Systems) in Computer Science & Information Systems Group, from Birla Institute of Technology and Science, Pilani, India. His areas of interest are Data Structures and Algorithms, Software Engineering and Computer Networks. He is presently a Software Engineer at Cypress Semiconductor.

Pushkar Pandey has completed his M.E. (Software Systems) in Computer Science & Information Systems Group, from Birla Institute of Technology and Science, Pilani, India. His areas of interest are Software Development, Algorithms and Operating Systems & Networks.

Appendix

Façade Screen

The façade screen provides an interface to the user in a project where he could browse an XML file that contains the optional and fixed parameters for modules. An organization can have many projects running and this tool provides a way to analyze the projects and locate test time and cost as per Module Priority.



Gantt Chart Screen

As the user chooses the XML file with priorities in a project, the Module Prioritization tool (MPT) calculates the priorities of the modules and displays them in form of a Gantt chart. It draws the module wise important parameters (optional and fixed) in a tab control and also calculates the expendability of each module to be shown tab wise for modules. Two Gantt charts depict the initial time and cost allocation respectively.

"Compute expendability M2" button determines the cost and time we can expend from M2 to M1. Then this button changes to "Compute Expendable M3" and so on to show how much M3 can give from its share to M1. This process can continue. The "Extend M1" button will allot the calculated expendable chunks or tranches to M1 module. We can then utilize new time and

cost allocation for the testing of module M1 and move to M2 where we can again compute expendable chunks from M3 and other modules if we are running out of test time for M2. This process can be done till n-1 modules out of n modules, if we have time and cost allocation left.