Evaluation of Software Release Readiness Metric [0,1] across the software development life cycle

Piyush Ranjan Satapathy
Department of Computer Science & Engineering
University of California, Riverside
piyush@cs.ucr.edu

Abstract

Each day, software engineers and managers cope with the challenges of building complex systems and challenges which threaten the project cost, schedule, and technical performance. Software metrics are quantitative standards of measurement for various aspects of software projects. A well designed metrics program will support decision making by management and enhance return on the IT investment. There are many aspects of software projects that can be measured but not all aspects are worth measuring. In this paper I propose a formula for calculating a software release readiness metric, called the "ShipIt" coefficient, with values defined in the interval [0, 1], where the value 1 indicates complete readiness. I am including any and all measurements of the software development life cycle, the development team, and the owning organization and the release policies. The metrics I consider here are measurable in practice. And I hope it may help the software industry in the direction of decision making keeping the eye on business goals. Evaluation of the software release readiness factor at any point of time through out the software life cycle can be obtained by desired modification of our formula which may help in meeting the schedules and the control costs for the marketability and competitiveness of the software product. But as always the business decision has a greater role in software release time so my formula has certain limits.

1. Introduction

In the today's competitive commercial software market, software companies feel compelled to release software the moment it is ready. They have to always deal with treading the line between releasing poor quality software early and high quality software late. A well prepared answer to the question, "Is the software on the track for release maximizing the goals?" has not been found yet. It's always critical for a company's management to gauge this factor successfully. The answer is sometimes based on gut instinct, but several techniques can put this judgment on a firmer footing. We propose a method of gauging the software readiness factor by evaluating each and every stage of a software life cycle. We use various existing models for each stage and come out with a simple formula combining all those. Using this formula one can find out the relative completeness of the project at any point of the software lifecycle on a 0 to 1 scale basis.

The use of software metrics for controlling software projects throughout the development and testing processes is now well documented. Metrics for design evaluation [9], code complexity analysis [10], test allocation [11, 12], fault targeting [13], maintainability [14, 15], portability analysis [16], and general code health [17] have all been defined, implemented, and used to improve software quality in industrial systems. Also most of these metrics have been successfully applied in software decision making. Application of software metrics in project progress and estimation [8, 7], and in software system maintainability [5] gives a practical insight to the system.

The development of a software product is theoretically concluded by exhaustively testing its performance and quality. During testing it's expected that functional and requirement problems will be identified and corrected. At some point, a decision will be made that testing should be concluded and the product be released for customers use. The release decision is usually based on an evaluation of the software's expected quality balanced against its release date commitment. Based on this Author in [1] has given a

clear method for determining the release readiness of a software product. He has devised a method called Zero failure method to calculate the time to reach at an acceptable release time. However author in [3] has devised a method called the stopping rule based on the number of remaining software faults to determine the optimum software release time. Author in [4] has measured the software readiness with defect tracking by devising rules like Defect pooling and Defect Seeding.

We now turn our attention to the use of software metrics in evaluation of software release readiness decision. As the author in [6] says that starting a new metrics program or improving a current program consists of five steps, we have considered all those 5 steps. As the step1, we have identified our business goal. Our goal is to track the cost, quality and timeline of the software product at any point of time during the software life cycle. By evaluating a method for determining the software release readiness factor will give us proper insight to the project complexity during the development. Also it will help us managing the traceability and cost estimation of the software project through out its lifecycle. As step2 we have identified metrics for each stage of the life cycle. We have gathered some historical data to support our metrics as a step3. And then we automate our measurement procedures to reach a generalized formula which is result of step4. And as the step5 we use the formula and metrics value and do the decision regarding the software release.

We strictly follow the software life cycle processes defined in IEEE standard [18]. Also we refer [2,8] for the simulation of software development processes to get a historical data regarding the validation and sharing of each stage in the life cycle. Given the set of metrics for each stage of the software life cycle and given which stage of the software life cycle a software project is in, we can determine the software release readiness factor by a coefficient value called "ShipIT" on a scale of 0 to 1; where 0 indicates that the software life cycle has not been started yet and 1 indicates that the product is ready to ship. As per the software life cycle processes we consider the life cycle to be consisting of phases like Requirements and Definition stage, system and software design stage, implementation stage, testing stage, operation stage and maintenance stage.

Our major contribution in this paper is a top down approach for calculation of software release readiness factor which covers each and every stage of a software development life cycle. Starting from requirement analysis to the retirement process in customers' site we have considered all the major steps. The most important thing is that we have considered the sub processes of the major processes. We have devised methods to quantify all the significant metrics of each stage and sub stages with proper coefficients. Giving an experimental weight for each of these coefficients we calculate the weighted average and then finally we normalize it to a value between 0 and 1.

The rest of the paper is structured as follows. In section 2 we present the back ground and the relevant software metric models. Section 3 contains our main idea. Section 4 analyses our work. We have put some validations in section 5. And in section 6 we conclude the paper.

2. Background and Model

Delivery of a "system" from a supplier to a customer is called a release. The "system" consists of a set of authorized and integrated components. The supplier is usually the development organization. The customer is the internal or external recipient of the system. Software release management is a process by which a product is made ready for distribution to customers. The purpose of the release management is to ensure that the products are ready when promised. Software release management process is analogically similar to a baking process. The key practices of a software release would be as follows; Release Planning (Managing and publishing release schedule), Defining deliverables that included in a release, Building and Versioning the integrated release package, managing dependencies across components, Coordinating

activities across company, controlling changes that would impact the release schedule, establishing release criteria and assessing the release readiness. There are some significant components of a release. Amongst those important are the followings; Golden Build (software), Documentation and Help, Training materials, Marketing collateral, Internal documents, Customer Notification and Release Notes. Some of the common release problems seen now a days are (i) take too long which impacts revenue (ii) Unpredictable which impacts dependent plans (iii) Poor quality which impacts support (iv) Not useful which again impacts revenue (v) Contents unplanned which impacts integrity (vi) Contents uncontrolled which impacts reliability (vii) Overlooked distribution/installation which impacts support. There are so many works in the area of determining the release time. [1,3,4,7] but all these methods or models are on the basis of quality assurance and based on the testing phase of the software life cycle. So to handle the software release management, a proper method should be there to know the status of the release at any point of time of the software development process. The different models and methods that we have considered for our work are as bellows

2.1 COCOMO Prediction Model [19]

Effort = a (Size)^b

Effort = person months
Size = KSLOC (predicted Thousands of Source Lines of Code)
a, b are constants depending on type of system:

'Organic':	a = 2.4 b = 1.05
'semi-detached':	a = 3.0 b = 1.12
'Embedded':	a = 3.6 b = 1.2

Time = a (effort)^b

effort = person months
Time = development time (months)
a, b constants depending on type of system:

'Organic':	a = 2.5 b = 0.38
'semi-detached':	a = 2.5 b = 0.35
'Embedded':	a = 2.5 b = 0.32

2.2 Halstead's Metrics Model [21]

A program P is a collection of tokens, classified as either operators or operands.

n_1 = number of unique operators
n_2 = number of unique operands
N_1 = total occurrences of operators
N_2 = total occurrences of operands
So, Length of P is $N = N_1+N_2$
Vocabulary of P is $n = n_1+n_2$
Theory: Estimate of N is $N = n_1 \log n_1 + n_2 \log n_2$
Theory: Effort required to generate P is

$$E = \frac{n_1 N_2 N \log n}{2n_2}$$

(Elementary mental Discriminations)

Theory: Time required to program P is T = E/18 seconds.

2.3 Albrecht's Function Points Model [20]

ExtIP = Number of External inputs
ExtOP= Number of External outputs
Extenq= Number of External inquiries
Extfiles= Number of External files
Intfiles= Number of Internal files
W_i = Weighting factor of External inputs
W_o = Weighting factor of External outputs
W_e = Weighting factor of External enquiries
W_f = Weighting factor of External files
W_{if} = Weighting factor of Internal files

The Unadjusted Function Count (UFC) is the sum of all these weighted scores.
So,

$$UFC = \Sigma (ExtIP \times W_i) + \Sigma (ExtOP \times W_o) + \Sigma (Extenq \times W_e) + \Sigma (Extfiles \times W_f) + \Sigma (Intfiles \times W_{if})$$

DI = Degree of Influence = Σ [General Application characteristics [i]] (i = 1 to 14)

TCF = Technical Complexity Factor = 0.65 + (0.01 * DI)
FP = Adjusted Function Count; **FP = UFC x TCF**
Based on the programming language we can know the number of source statements/LOC for each FP. Let this number be defined by "SourceStatement". So the total number of lines of code is defined as;

LOC = SourceStatement x FP

2.4 Zero Failure Method [1]

The zero-failure method is a decision technique that specifies the number of test hours before software release in which no additional test failures are permitted to be found. It incorporates the idea that the longer testing proceeds without finding a failure, the greater the likelihood becomes that the number of remaining failures is very small. If no failures are seen in the target test time, the software is judged ready for customer release. If failures (even one) are detected, the request for release is rejected and conventional testing is continued. Like many quality models, the zero-failure method assumes that the rate of failure discovery decreases exponentially as testing progresses. This means that a high rate of failure detection early on becomes smaller **as** testing continues and fewer failures remain to be found. Also, like other models, it assumes that testing is representative of use, test intensity or stress is constant, and the probability of failure discovery is constant and equal for all failures. Given that the information submitted to the method is representative of the testing process, it is assumed that only two data points are needed to establish an acceptable reliability-model curve, one of which is at zero failures. To apply the method, it is assumed that 0.5 failures is equivalent to zero failures. This correction causes the method's projections to be slightly conservative. The method is derived from the exponential model Problem Rate

$$p(t) = a \times e^{-b(t)}$$

To calculate the zero-failure test hours requires three inputs: the target projected average number of customer failures, the total number of test failures detected so far, and the total test-execution hours up to the last failure. The calculation for zero-failure test hours is given by;

$$\text{Zero Failure Test Hour} = \frac{\ln[(\text{Customer problems}) / (0.5 + \text{Customer problems})]}{\ln[(0.5 + \text{Customer problems}) / (\text{test} + \text{Customer problems})]} \times (\text{Test hours to last problem})$$

2.5 Stopping Rules Method [22]

In practice it's not reasonable to test the software until all faults are removed. A possibility is to stop testing when the number of remaining faults is less than a prescribed number. Another possibility is to stop when the number of remaining faults is less than a prescribed portion of initial faults. Assuming the Jelinski-Moranda model [23] is valid and the parameters N_0 and Φ have been estimated using previously collected data. Denoted by M the number of acceptable faults and let T be the time needed to remove $n = N_0 - M$ faults. Then it can be shown that the expected value of T is given by

$$ET = \sum [1 / (\Phi (N_0 - i + 1))] \quad (i = 1 \text{ to } n)$$

Hence the test may be stopped at the time given above. Also there is a stopping rule method based on the software failure intensity requirements [3].

2.6 Maintainability Index Method [5]

The literature of at least the last ten years shows that there have been several efforts to characterize and quantify software maintainability. In the software industry, a program's maintainability is calculated using a combination of widely-used and commonly-available measures to form a Maintainability Index (MI). The basic MI of a set of programs is a polynomial of the following form (all are based on average-per-code-module measurement):

MI = 171 - 5.2 x ln(aveV) - 0.23 x aveV(g') - 16.2 x ln (aveLOC) + 50 x sin (sqrt(2.4 x perCM))

The coefficients are derived from actual usage. The terms are defined as follows:

aveV	=	Average Halstead Volume V per module s
aveV(g')	=	Average extended cyclomatic complexity per module
aveLOC	=	Average count of lines of code (LOC) per module
perCM	=	Average percent of lines of comments per module

Oman develops the MI equation forms and their rationale. He indicates that the above metrics are good and sufficient predictors of maintainability. Oman builds further on this work using a modification of the MI and describing how it was calibrated for a specific large suite of industrial-use operational code. Oman describes a prototype tool that was developed specifically to support capture and use of maintainability measures for Pascal and C. The aggregate strength of this work and the underlying simplicity of the concept make the MI technique potentially very useful for operational Department of Defense (DoD) systems.

3. Our Main Idea

As per IEEE standard for Developing Software Life cycle processes [18], we have identified the major steps of the software life cycle processes. Those are as follows: 1. Software Lifecycle Modeling process 2. Project Management Process 3. Pre Development Process, 4. Development Process 5. Post Development process and 6. Integral process. As the first step of the above processes we have chosen the waterfall model or the SPM model [24]. And then considering the waterfall cycle model and considering the rest 5 steps above, we have fine tuned the whole processes and we have come up with some significant ingredients of the software development processes. Those are as ;(1). Requirement gatherings (2). Requirements Analysis Process (3). System and software design process (5). Implementation Process (6). Testing process (7). Quality Assurance process (8). Manuals and Documentation Process (9). Supervision process (10). Support Process.

We have considered the above 10 steps for determining the software release readiness coefficient. Its rarely practical to include all the steps in the software development. So we are assuming some conditions

to make our method work properly. The first assumption is: while the software development is in the requirement gathering stage or in analysis stage or in design stages no coding or testing process is allowed. The second assumption is that we consider software ready to be released when the software product is fully deployed in the customers' site retiring the previous used methodologies or previous version of the software product. This assumption facilitates us to include the support factor in our calculation. The third assumption is that the requirements keep coming till the end of the detailed design process. After that all the new requirements are considered for next version of software.

Based on our 3 assumptions and 10 methodologies stated above, we came out with 7 granular components for which we can be able to collect the metrics during the software life cycle and using that we can calculate the software release readiness coefficient. Those 7 components are defined as 1. RequirementAnalysisDesign Stage (Includes first 3 processes) 2. Coding (Includes Implementation process) 3. Testing (Includes testing process) 4. Quality assurance 5. Manuals and Documentation 6. Supervision7. Support. Then we give a weight to each of these components as per the efforts (person per months) required for each stage. Bsasically this comes from experience. We have considered in terms of variables here. Let these weights be defined as W_{RAD}, W_{CODE}, W_{TEST}, W_{QA}, W_{MD}, W_{SV}, W_{SP} respectively on a scale of 0 to 100.

Then as the next step we consider the metrics from each step one by one (which will be our next section) and we come out with 7 significant factors from each step on a scale of 0 to 1. Let these factors be defined as RAD, CODE, TEST, QA, MD, SV, and SP. So the software release readiness metric (ShipIT) can be defined as bellows;

$$ShipIT = [(W_{RAD} x\ RAD) + (W_{CODE} x\ CODE) + (W_{TEST} x\ TEST) + (W_{QA} x\ QA) + (W_{MD} x\ MD) + (W_{SV} x\ SV) + (W_{SP} x\ SP)] / 100$$

Where, W_{RAD}, W_{CODE}, W_{TEST}, W_{QA}, W_{MD}, W_{SV}, $W_{SP} \in [0, 100]$ and RAD, CODE, TEST, QA, MD, SV, and $SP \in [0,1]$. Based on our assumption we can see that while the software development is in the RequirementAnalysisDesign stage, the factors CODE, TEST, QA, MD, SV and SP will all be '0'.And also we can note that $(W_{RAD} + W_{CODE} + W_{TEST} + W_{QA} + W_{MD} + W_{SV} + W_{SP}) = 100$.

4. Analysis

4.1 RequirementAnalysisDesign stage:

We divide this stage into three sub stages as (1). Defining and gathering Requirements (2).Analyzing the Requirements (3) Detail Designing of the requirements. We collect the metrics data for each of these stages. Let R_R be the till date number of required requirements from the customers and R_D be the defined and developed requirements by till now. So the factor by which the requirement gathering and defining is completed is given by R as bellows;

$$R = (R_D / R_R)$$

It's a point to note that R_R keeps increasing either from customers' side or from the market competitions. And R_D tries to catch up the R_R value. Similarly let A_{avail} be the no of available defined and developed requirements ready to be analyzed and let A_D be the number of requirements for which analysis is done. So from our generalization we can say that $A_{avail} = R_D$. And the factor by which the requirement analysis is completed is given by as bellows;

$$A = (A_D / A_{avail}) = (A_D / R_D)$$

For the design stage let D_{avail} be the number of available analyzed requirements ready to be designed and let D_D be the number of features or requirements for which the detail design has been done. So from our analysis we can follow that $D_{avail} = A_D$. And the factor by which the detail design is completed is given by as bellows;

$$D = (D_D / D_{avail}) = (D_D / A_D)$$

Assuming sufficient man powers to perform all these above three steps in a parallel processing way, we can calculate the RAD as bellows;

$$\boldsymbol{RAD = [(W_R \times R) + (W_A \times A) + (W_D \times D)] / 100}$$

Where W_R, W_A, W_D are the weights of the above mentioned three steps respectively on the basis of effort spent on each one. Its noted that W_R, W_A, $W_D \in [0, 100]$ and $(W_R + W_A + W_D) = 100$. And also its easily noted that the factors R, A and D all $\in [0, 1]$. From our analysis it's clear that when no requirements has been defined or developed then RAD = 0. That's because R = 0 and automatically all others should be 0. This is the case when our above method for calculating RAD doesn't hold true. So we exclusively say that in words.

4.2 Coding Stage:

We divide the coding stage into three categories. 1. Creating source 2. Creating Object 3. Building process. Now for each stage we have identified distinguished metrics. For the first stage, the metrics we consider are, no of system modules completed vs total system modules, the no of application modules completed vs total application modules and the no of GUI modules completed vs the total no of GUI modules. Let's define S_m, A_m, G_m as the fraction of completion of the system modules, application modules and graphical user interface modules respectively. So by definition I can represent these factors as below; S_m = (Completed no of system modules / Planned no of system modules); A_m =(Completed no of application modules/Planned no of application modules); G_m = (Completed no of GUI modules/Planned no of GUI modules). Lets assume that the weights of these three factors be W_{sm}, W_{am} and W_{gm} where W_{sm}, W_{am} and $W_{gm} \in [0,100]$ and $(W_{sm}) + (W_{am}) + (W_{gm}) = 100$. So we can express the factor by which the creating source contributes to the total coding phase as below;

$$Source = [(W_{sm} \times S_m) + (W_{am} \times A_m) + (W_{gm} \times G_m)] / 100$$

Now in the second category of coding stage we consider creating the objects. One can evaluate it by considering the number of modules as above but we give here a more practical approach. We consider the LOC (lines of code), Number of files and known anomalies. But however considering LOC only is not a good substitution for effort, complexity and functionality. LOC fails to take account of redundancy and reuse of the code. So we consider either the COCOMO prediction model combined with Albrecht's Functional point (section 2) or Halstead's software metrics model. If we use the 1st category, Albrecht's functional point method gives us the KSLOC value at the starting of the coding stage. And then using this KSLOC value we can calculate the time required to complete ($Time_{total}$) by COCOMO prediction model. And at any point of the coding stage let's say we have already spent "$Time_{spent}$ "amount of time. SO the fraction by which it contributes towards finishing of the coding stage is given by

$$Object = [(Time_{spent}) / (Time_{total})] \text{ which } \in [0,1]$$

Also we can calculate the ($Time_{total}$) by using the Halstead's software metrics model at the starting of the coding stage and we can use the above formula.

In the third category of the coding stage we consider the building process. This is somewhat equivalent to unit testing but not exactly unit testing. It considers the metrics like – Compiling time, -Handling warnings time, -Incremental build time as per platform dependency, and –Incremental build time as per compiler dependency. We assign some weights to each of these 4 metrics and let's say the weights are W_{CT}, W_{HT}, W_{BPT}, W_{BCT} respectively. And at any point of coding stage we can know the fraction of completion of all these processes. Lets say the completed percentage of each of these processes be defined as CT, HT, BPT and BCT respectively. So the fraction by which the building process contributes to the coding stage is as follows;

$$Build = [(W_{CT} \text{ x } CT) + (W_{HT} \text{ x } HT) + (W_{BPT} \text{ x } BPT) + (W_{BCT} \text{ x } BCT)]$$

Where CT, HT, BPT and BCT Є [0, 1] and W_{CT}, W_{HT}, $W_{BPT,}$ W_{BCT} Є [0,100] and $W_{CT} + W_{HT} + W_{BPT} + W_{BCT} = 100$. So finally we need to calculate the factor by which the coding stage contributes to the completion of the software development. Let this factor be called as "CODE". And let the weights of effort sharing amongst the three sub stages be defines by the variables, W_{source}, W_{object} and W_{Build}. So "CODE" will be defined as;

$$\boldsymbol{CODE = [(W_{source} \text{ x Source}) + (W_{object} \text{ x Object}) + (W_{Build} \text{ x Build})] / 100}$$

And its clear to note that $W_{source}+W_{object}+W_{Build} = 100$ and Source, Object and Build Є [0,1]. The basic assumption we have done here is that in any software life cycle in the coding stage, development of sources and objects and building the processes can happen simultaneously except a few lag time at the beginning which is required to start the building process. Because one can only compile and build once he has some raw code which is produced by the first two stages.

4.3 Testing

The testing process starts just after the coding stage and consists of testing and finding the bugs and debugging those. The testing usually consists of Unit testing (feature wise), Integration testing (Across the features) and System testing (across the platforms and compilers). We consider the metrics for this as follows; -no of total features vs. no of features already covered in unit testing, -No of possible relations amongst the features vs. the no of interactions already tested, and the no of integration tests required vs. no of completed integration tests. The metrics for debugging phase consists of -no of open issues and no of total issues at hand. These will take care of line coverages, purifying errors and profiler results. So let's assign weights to each of the three testing process as W_{utest}, W_{Itest} and $W_{stest.}$ Now let L1 be the factor by which the unit test has been completed. Its defined as the ratio of the completed no of unit testcases versus the planned no of unit testcases. Similarly we can know the L2 and L3 for integration test and system test respectively. So the factor by which the finding bugs contribute towards the testing is given as;

$$Bugfinding = [(W_{utest} \text{ xL1}) + (W_{Itest} \text{ xL2}) + (W_{stest} \text{ xL3})] /100$$

Where W_{utest}, W_{Itest} and W_{stest} Є [0,100] and $(W_{utest}+ W_{Itest} + W_{stest}) = 100$; L1, L2, and L3 Є [0, 1]. Now let's say the weight of finding bugs be $W_{Bugfinding}$ and that of debugging be $W_{Debugging}$. And let debugging stage contribute a factor "Debugging" towards the testing process which is defined as 1 minus the ratio of open issues to total issues counted from the beginning to that point. So the overall factor by which the testing contributes towards the software life cycle is as follows;

$$\boldsymbol{TEST = [(W_{Bugfinding} \text{ x Bugfinding}) + (W_{Debugging} \text{ x Debugging})] / 100}$$

And here $W_{Bugfinding} + W_{Debugging} = 100$; and Bugfinding, Debugging Є [0, 1].

4.4 Quality Assurance

Quality is the important factor of software product. But sometime marketability comes before that. So there is a proper trade off between these two. From the customers or from the market reviews we can find out the desired quality of the product and accordingly we can apply the Zero failure test hour method (section2) or the stopping rule method to know the exact remaining hour of the testing process to be carried on. We have considered the Regression testing as our metrics here. If we use Zero failure method we can find the Zero failure test hour and if we use stopping rule we can find ET (expected time to reach the desired quality). So then we can calculate the pseudo hours we completed by subtracting the zero

failure test hour or the ET from the required total test hour which is planned at the beginning of the stage. So the factor by which it contributes towards the completion of the software development is given by;

$$QA = [(Pseudo\ Test\ Hours\ Completed) / (Total\ test\ hours\ planned)]$$ on a scale [0, 1]

4.5 Manuals and Documentations

We divide this stage into number of stages as follows; -Requirement Documents, - Design Documents, -Implementation and Usability Documents, -Test plan documents, and -User guide documents. Usually documentation is a by default metrics for measurement at each stage of the development process. Let's distribute the weights from 0 to 100 amongst all these documentations. W_{RD}, W_{DD}, W_{ID}. W_{TD}, W_{UD} be the weights of all these documentation processes respectively. And at any point of time we can calculate the fraction of completed documentation at each stage. Let these fractions be RD, DD, ID, TD and UD respectively. So we can calculate the factor by which the Manuals and documentation contributes to wards the completion of the software development as follows;

$$MD = [(W_{RD} x\ RD) + (W_{DD} x\ DD) + (W_{ID} x\ ID) + (W_{TD} x\ TD) + (W_{UD} xUD)\ ] / 100$$

And here $W_{RD} + W_{DD} + W_{ID} + W_{TD} + W_{UD} = 100$; and RD,DD,ID,TD,UD $\in$ [0, 1].

4.6 Supervision

Supervision usually consists of two types of processes. 1. Installation process 2. Training process. The metrics involved in Installation process are, -distribution of software, -installation of software, -acceptance or package testing. We can collect the required and completed data and can find out the fraction of completion of each of these steps. Let the fraction of completions be DS, IS and AT respectively. Giving a particular weight to all these (lets say W_{DS}, W_{IS} and W_{AT}) we can calculate the contribution of installation process towards supervision by IP = [(W_{DS} x DS) + (W_{IS} x IS) + (W_{AT} x AT)] /100 on a scale [0, 1]. In the training process we can consider the metrics like, -Developing training materials, -Validating training program, and –Implementing training program. And if the fraction of completion of each of these stages are TDM, TVM and TIM respectively and if the weights of each of these metrics are W_{TDM}, W_{TVM}, W_{TIM} then we can calculate the contribution of training process toward the completion of supervision as;
TP = [(W_{TDM} x TDM) + (W_{TVM} x TVM) + (W_{TIM} x TIM)] /100 on a scale 0 to1. If the weight of the installation process and training process be W_{IP} and W_{TP} (W_{IP} + W_{TP} = 100) respectively then we can calculate the contribution of supervision towards the software development as follows;

$$SV = [(W_{IP} x\ IP) + (W_{TP} x\ TP)]/ 100$$ on a scale of [0, 1]

4.7 Support

We have considered the "Support" as the last step of a software release. In this step the main metric we consider is beta customer reported bugs and repeating the software cycle if any. Once we reach a proper maintainability factor as agreement with the customer we stop supporting and do the software version release. So as discussed in section 2.6, we calculate the Maintainability Index and divide it by the desired Maintainability Index to get the fraction of support which is completed. Let this be called SP.So by our assumption,

$$SP = [Maintainability\ Index\ reached / Maintainability\ Index\ desired]$$

5. Validation

To validate our result we have gathered some data from past research. For the weights W_{RAD}, W_{CODE}, W_{TEST}, W_{QA}, W_{MD}, W_{SV}, W_{SP} we have followed [8] which says W_{RAD} = 22% W_{CODE} = 19% W_{TEST} = 30% W_{QA}= 8%, W_{MD} = 7% W_{SV} =9% and W_{SP} = 5%. So using our proposed method we can calculate the release coefficient as; ShipIT = [(22 x RAD) + (19 x CODE) + (30 x TEST) + (8 x QA) + (7 x MD) + (9 x SV) + (5 x SP)] / 100. Similarly using past experiences and historical data we can calculate the RAD, CODE, TEST, QA, MD, SV, and SP as follows; RAD = [(30 x R) + (20 x A) + (50 x D)] / 100; CODE = [(40 x Source) + (20 x Object) + (40 x Build)] / 100; TEST = [(65 x Bugfinding) + (35 x Debugging)] / 100; QA = QA; MD = [(15 x RD) + (15 x DD) + (15 x ID) + (25 x TD) + (30 xUD)] / 100; SV = [(50 x IP) + (50 x TP)]/ 100; SP = SP. From paper [2], we can calculate that, Source = [(57 x S_m) + (28 x A_m) + (15 x G_m)] / 100; Build = [(40x CT) + (30 x HT) + (15 x BPT) + (15 x BCT)]; Bugfinding = [(35 xL1) + (35 xL2) + (30 xL3)] /100. Using all these weights from research and experiences and calculating the required metrics we can find out the ShipIT coefficient. However the weights we consider here may not be correct for all kinds of software developments. One can distribute the weights as per the conditions and apply the formula to calculate the coefficient.

6. Conclusion

In this paper we understood the requirement of determining the software release readiness coefficient from the factors of successful timely release, consistent release process, predictability, integration, completeness and desired quality. Considering the whole software life cycle and detecting the measurable metrics from each stage we came out with some quantization and normalization to calculate the desired coefficient.

As its rightly said that “no one tool or method should be relied on to arbitrarily make the final determination of whether a software product should be released”, we hope that our method would give a proper insight to all the complexities involved in the decision making of software releases and it may help in putting the software projects on track keeping the objective intact. As our method is a much generalized one, it can be used easily by knowing the mentioned metrics and weights. The weights can be achieved by experiences or by research but that varies across the software products and across the environments in which these products are developed. However we have some of our major limitations here. Through out the paper we have put our focus on the major releases rather than any minor, patch or emergency releases. And we have assumed perfect release plans before the Software development life cycle starts. Release plans like customer commitments, revenue recognition, resource availability, maximum changeability, organizational capability are all assumed to be in ideal condition.

References

[1]. Brettschneider, R., Motorola Inc., Phoenix, AZ, “Is your software ready for release?”, IEEE Software, July’1989, Volume6, Issue4, pp. 100,102,108

[2]. Gregory A. Hansen, GAPI, “Simulating Software Development Processes”, IEEE Software, January 1996 (Vol. 29, No. 1), pp. 73-77

[3]. Min Xie, “On the Determination of Optimum Software release Time”, IEEE Intl Symp. on Software Reliability Engineering, May1991. pp. 218-224

[4]. Steve McConnell, “Gauging Software Readiness with Defect Tracking”, IEEE Software, May/June 1997 (Vol. 14, No. 3), pp. 136-135

[5]. Don Coleman, Dan Ash, Bruce Lowther, Paul Oman, “Using Metrics to evaluate software system maintainability” IEEE Computer Society Press Los Alamitos, CA, USA, Volume 27, Issue 8, August 1994, pp. 44-49

[6]. Peter Kulik, “A practical approach to software metrics”, IEEE ITProfessional January/February 2000 (Vol. 2, No. 1), pp. 38-42

[7]. Troy Pearse, Tracy Freeman, Paul Oman, “Using Metrics to manage the End-game of a software project”, Sixth IEEE International Symposium on Software Metrics November 04 - 06, 1999 Boca Raton, Florida, pp. 207

[8]. Robert B. Grady, Hewlett-Packard, “Successfully Applying Software metrics”, IEEE Trans. Soft Engr., September 1994 (Vol. 27, No. 9), pp. 18-25

[9]. L. Briand, et al., “A Comprehensive Empirical Validation of Design Measures for Object-Oriented Systems,” Proc. Fifth Int’l Soft. Metrics Symp., CS Press, Los Alamitos, CA, Nov. 1998, pp. 246-257.

[10]. T. Koshgoftaar, J. Muson, & D. Lanning, “Alternative Approaches for the use of Metrics to Order Programs by Complexity,” J.sys. Software, V.24 (3), Mar. 1994, pp.211-221.

[11]. G. Atkinson, et al., “Directing Software Development Projects with Product Metrics,” *Proc. Fifth Int’l Soft. Metrics Symp*., CS Press, Los Alamitos, CA, Nov. 1998, pp. 193-204.

[12]. J. Zhuo, et al., “On the Validation of Relative Text Complexity for Object-Oriented Code,” *Proc. Fifth Int’l Soft. Metrics Symp*., CS Press, Los Alamitos, CA, Nov.1998, pp. 258-266.

[13]. J. Munson & T. Khoshgoftaar, “The Detection of Fault Prone Programs,” *IEEE Trans. Soft. Engr*., V.18 (5), May 1992, pp. 423-433.

[14]. D. Coleman, et al., "Using Metrics to Evaluate Software System Maintainability," *IEEE Computer*, Vol. 27(8), Aug. 1994, pp. 44-49.

[15]. T. Pearse & P. Oman, "Maintainability Measurements on Industrial Source Code Maintenance Activities,"*Proc. 1995 Int’l Conf. on Soft. Maint.*, CS Press, LosAlamitos, CA, 1995, pp. 295-303.

[16]. T. Pearse & P. Oman, "Experiences Developing and Maintaining Software in a Multi-Platform Environment," ," *Proc. 1997 Int’l Conf. on Soft. Maint.*,CS Press, Los Alamitos, CA, 1997, pp. 270-277.

[17]. D. Ash, et al., "Using Software Maintainability Models to Track Code Health," *Proc. 1994 Int’l Conf. on Soft.Maint.*, CS Press, Los Alamitos, CA, 1994, pp. 154-160.

[18]. “IEEE Standard for Developing Software Life Cycle Processes”, Software Engineering Standard Committee of the IEEE Computer Society, Approved September 21, 1995, IEEE Standards Board

[19]. “COCOMO Effort Model”, by Brad Clark, http://www.psmsc.com/UG1998/Presentations/cocomo2%201998.pdf

[20]. Paul Vickers, Northumbria University, “An introduction to Function Point Analysis”,

http://computing.unn.ac.uk/staff/cgpv1/downloadables/fpa.pdf

[21]. Joseph L.F. De, Kerf, "APL and Halstead's theory of software metrics", Proceedings of the international conference on APL, September 1981, Volume 12, Issue1, pp. 89-93

[22]. P.A. Caspi, E.F. Kouka, "Stopping Rules for a Debugging process based on Different Software Reliability Models", Proc. Int. Conf. on Fault – Tolerant Computing, pp. 114-119, 1984

[23]. M.C.J. van pul, " Simulations on the Jelinski-Moranda model of software reliability", 1991, BS-R9122, ISSN 0924-0659. http://ftp.cwi.nl/CWIreports/1991/BS-R9122.pdf

[24]. M.M.Lehman, "Process Models, Process programs, programming support", IEEE Computer Society, 1987, 9th Intl conf. on Software Engr., pp 14-16