Measuring Software Team Productivity

Author: *Debashis Chatterjee*

Table of Contents

1. Introduction

1.1. Problem statement

It is not uncommon to find that a certain SW development unit within a large organization has somehow mistakenly acquired the perception of being bloated, i.e. the feeling from outside is that their job function shouldn't be requiring so many headcounts. Part of the problem can be simply the fact that the group's name indicates they only do a smaller subset of tasks, but actually they do a lot more. If a group is named "Wifi Software", one immediately thinks that they merely develop device drivers for different OS platforms, but actually they could be doing a lot more; perhaps they develop not just drivers but also firmware, libraries, tools and even some application-level proof-of-concept work. But people from outside can look at their headcount and erroneously conclude that so many resources shouldn't be needed just for device driver development.

While this seems more like a messaging problem, the problem makes us want to know – how do we effectively measure the productivity of a SW development group, and compare that with productivities of similar groups? It is also useful to know about some standard practices that SW development organizations can adopt, which will showcase their efficiency and processes objectively and won't label them unfairly.

This report summarizes my investigation into this problem. Following areas were investigated and my findings have all been described in the report.

- In section titled "Individual Productivity", the industry-accepted metrics for SW developers' individual productivity was introduced. Effective lines of code per month is still the best available metric, but my analysis shows how to adjust the raw number for reused or modified code as opposed to new code. Data for average number of lines of code per month for various completed SW projects with varying degree of complexity is also presented, to provide SW managers realistic estimates (can be used for a post mortem analysis also) for their projects.
- In section titled "Organizational Productivity", software equations for measuring organizational productivity are presented, along with details on all the factors that contribute to productivity. Different ideas from the COCOMO, the SEER-SEM and the Sage productivity models are discussed in this section.
- One key aspect of whether a project is seen as being efficiently executed or not is the level of staffing as well as the rate of staffing. Mathematic models for

staffing, including the Rayleigh Development Model and Brook's Law, are introduced in section titled "Staffing".

- Rather than periodically collecting and reporting such data, it is much better for an organization to keep a running evidence of its execution efficiency in a dashboard. The numbers speak for themselves. In section titled "Software Quality Tracking (SQR)", we define a software quality measure, called Software Quality Ranking (SQR) and describe how every development unit can keep a dashboard of SQR rankings for their different projects and/or subunits visible to the whole organization.
- Section titled "Conclusion" is a summary, where the division head is given a recommendation on how to deal with the problem he is facing and what discreet steps are to be taken.

1.2. Main sources of information

Though software development isn't a very old industry, research on software productivity estimation has been ongoing for many years. On the Internet, as well as on scholarly research databases such as IEEE Explore, a lot of papers were found but many of them dated back to early 1980s and even early 1970s. The challenge was to extract information that is still relevant. Out of every ten papers downloaded and read, my success rate of finding relevant papers within that set was never greater than one or two out of ten. Even some of the relevant information had to be modified based on today's state of the art. I have even taken the liberty of slightly changing some of the model equations to nullify variables that are no longer relevant, such as response times of terminals and keyboards (probably a problem in the days of mainframes but certainly not a problem anymore) and the time it takes to collect a hard copy from a printer.

In my investigation I have primarily focused on the parametric estimation models for SW development. The three most widely quoted work in this were done by COCOMO, SEER-SEM and SLIM. COCOMO stands for "Constructive Cost Model" and was developed by Barry Boehm in the 1970s.[1] SEER-SEM is a project management application that started in the 1960s but the main work was done in 1980 and summarized in a paper written by Don Reifer and Dan Galorath.[2] SLIM is another software estimation model introduced in 1978 by Lawrence Putnam. Though not a good reference for SLIM itself, the interview of Putnam by CAI in 2006 serves as good source of information on how SLIM was built, as well as his current involvement in QSM (Quantitative Software Measurement), a website that provided many of the data cited in this report.[3]

[1] http://sunset.usc.edu/csse/research/cocomoii/cocomo_main.html
[2] http://galorath.com/company
[3] http://www.compaid.com/caiInternet/ezine/larryputnaminterview.pdf

Most areas of software work, whether on theory or practice, benefited from one or more iconic books that serve as the main source of information in those areas. Kernighan and Richie's book on C Language, or Aho and Ullman's book on Compiler Construction, are some of the examples. I was fortunate to find two such iconic books in this area that I benefitted immensely from. One is Barry Boehm's 1981 book titled *Software Engineering Economics.* The other is Randall Jensen's book titled *Software Development Productivity.*

My report doesn't have any original finding that hasn't been already published in the books or papers cited above. The main value is in summarizing information from many books and reports and ensuring the relevance of that information in modern context.

2. Individual productivity

2.1. Productivity definition

In the discipline of SW, productivity has been historically measured as

Productivity = ESLOC/PM, where
ESLOC = Effective or equivalent source lines of code
PM=person month

ESLOC must be greater than or equal to the number of source lines created or changed.

2.2. Adjusting for reused/modified code

Counting SLOC is straightforward when all the code is new. When modifications are accounted for, several other factors have to be considered –

- Modification must be preceded by a thorough understanding of the code to be modified, along with knowledge of associated system and architecture.
- If the existing documentation is insufficient, reverse engineering of the code has to be done
- Modifications can't break any of the existing interfaces, and thus they impose additional burden on both developers and testers
- If the new modification is to be done on a new language, or has to be compiled with newer compilers or linked with newer libraries than what the existing code used, the complexity increases

Keeping all this in mind, an adaptation adjustment factor (AAF) is applied to the modified code, as

$$AAF = 0.4F_{des} + 0.3F_{imp} + 0.3F_{test}$$

Where

F_{des} = the fraction or percentage of the reused software requiring redesign and reverse engineering,
F_{imp} = the fraction or percentage of the reused software that has to be physically modified (technically called "recoded"), and
F_{test} = the fraction or percentage of the reused software requiring regression testing

And $ESLOC = S_{new} + S_{mod} + S_{reused}*AAF$

Where
S_{new} = new lines of code,
S_{mod} = modified lines of code, and
S_{reused} = reused lines of code.

2.3. SLOC/PM data from industry

There isn't a lot of data available on what are acceptable numbers for ESLOC. Historical data shows that a Borland team was measured at delivering 1000 lines/week, the fastest on record so far, whereas a Microsoft Windows team, albeit on a different platform, tools and other variables, could only produce 1000 lines/year.

Another approach is to use the definition of function point, which is effectively the average number of lines of code for a typical function. Java has 53 lines of code per function point and the average Java team using Waterfall methodology does 2 function points per month per developer, or 106 SLOC. There are many studies that shows that Java teams using Scrum produces 15 function points per month per developer or 780 SLOC.

The best source of data on SLOC currently is the website *Quantitative Software Mesaurement* (QSM, at http://www.qsm.com). There we can find data collected on SLOC/PM for many different types of applications such as business systems, engineering systems and realtime systems

Business Systems (Source Lines of Code benchmarks)

Size: New & Modified SLOC	**Duration (Months)**	**Effort (PM)**	**Average Staff (FTE)**	**SLOC/PM**
2,500	5.9	11.3	1.9	430
10,000	7.1	26.1	3.5	650
25,000	8.5	45.8	5.2	876
50,000	9.5	70	7.2	1,088.00
100,000	10.4	102.7	9.3	1,300.00
Min: 600	*1.6*	*1.63*	*0.9*	*38.5*
Max: 7,920,000	*42.2*	*2,219.65*	*202.8*	*12,403.40*

The Business Systems group includes 450 Business (IT) Systems projects completed between 2008 and 2011.

Business Systems: Function Point Benchmarks

Size: FP	**Duration (Months)**	**Effort (PM)**	**Average Staff (FTE)**	**FP/PM**
50	5.7	11.9	2	7.1
100	6.4	17.9	2.7	8.6
250	7.6	31.6	4.1	10.8
500	8.5	46.4	5.6	13.1
1,000	9	71	7.4	15.5
Min: 10	*1.6*	*1.6*	*0.4*	*1.1*
Max: 5,000	*42.2*	*1,705.00*	*121.4*	*234*

The Business Systems: Function Point group includes approximately 250 Business (IT) Systems projects completed between 2008 and 2011.

Engineering Systems

Size: New & Modified SLOC	**Duration (Months)**	**Effort (PM)**	**Average Staff (FTE)**	**SLOC/PM**
2,500	6.7	22	3.2	192.2
10,000	9.7	53	5.4	294.5
25,000	12	92.4	7.1	394
50,000	14.2	143	9.3	497.3
100,000	16.8	225.7	12.2	621
300,000	23.8	453.4	19.3	887.7
Min: 32	*1.8*	*0.83*	*< 1*	*7.1*
Max: 2,573,612	*55*	*10,037.00*	*339.6*	*13,514.90*

The Engineering Systems group includes over 300 Command & Control, System Software, Telecommunications, Scientific, and Process Control projects completed on or after 2000.

Real Time Systems

Size: New & Modified Code	**Duration (Months)**	**Effort (PM)**	**Average Staff (FTE)**	**SLOC/ PM**
2,500	9.6	12.3	1.4	211.8
10,000	13.6	55.1	4	223
25,000	17.2	143	8.3	244.1
50,000	20.6	281	14.3	250.5
100,000	25.1	596.9	23.6	259.3
300,000	33.7	1,850.30	54.4	274.3
Min: 344	*4.5*	*2.04*	*< 1*	*21*
Max: 2,141,000	*94.1*	*43,221.28*	*760.9*	*4,598.70*

The Real Time Systems group includes approximately 145 Avionics, Real Time, and Microcode & Firmware projects completed after 1990.

The QSM data above is useful for analyzing individual productivities for a new project, either just completed or still being executed. If we can find the SLOC for a project after it completed, we can find out whether the productivity was worse than, near, or better than average SLOC/PM. Alternatively, if a certain project has to finish within a certain time, we can start with the SLOC/PM that will achieve that goal, and carefully monitor whether the progress is according to expectation or not.

2.4. ESLOC tools

It is not easy to count ESLOC for programmers and record them in a database. Modern repositories such as git allow customized scripts to be run on each commit, and that script can parse the difference generated by the repository, and then tag and record the finding. Today, the freeware static analysis tool named Sonar (http://www.sonarqube.org) calculates SLOC using internally developed tool. The freeware program SourceMonitor (http://www.campwoodsw.com/sourcemonitor.html) collects SLOC and many other useful metrics. There are many others freewares such as Refactorit, CodeFacts and OhLoh that provided this feature in the past but aren't in development anymore. Specific languages may offer specific plugins, such as JaCoCoverage is a NetBeans plugin for Java, but these aren't universal solutions. The website LocMetrics (http://www.locmetrics.com/alternatives.html) is a comprehensive source of information on current SLOC tools.

3. Organizational productivity

3.1. Overview

In this section, the main factors affecting organizational productivity are presented. Then some quantitative measures are introduced for some of the these factors. Finally, two independent software equations are presented to calculate organizational productivity from these quantitative measures.

3.2. Main factors affecting organizational productivity

From information collected from a variety of software development organizations, the following factors are often seen as exerting the most influence (positively or negatively) on organizational productivity –

- Motivation
- Proper use of team methods
    - Communication ability
    - Cooperation
- Working environment
    - Noise level
    - Individual working space
    - Proximity to team members
- Software engineering ability
- Problem-solving skills

3.3. Quantifying organizational effectiveness

3.3.1. Communication and management

Effectiveness of communication and management is measured using the following two key parameters – MCAP and PCAP.

3.3.2. Manager capability (MCAP) and Programmer capability (PCAP)

The capabilities are two numbers which are multiplied and the product gives an indication of the cost, hence lower numbers are better.

Organization description	MCAP value	PCAP value
Poorly motivated AND non-collaborating	1.46	1.42
Poorly motivated OR non-collaborating	1.19	1.17

Conventional or neutral organization	1.00	1.00
Highly motivated OR collaborating	0.86	0.86
Highly motivated AND collaborating	0.71	0.70

We can see that a cost of a highly motivated and collaborating team (0.71*0.7 = 0.497) is about 1/4th the cost of a poorly motivated and non-collaborating team (1.42*1.41 = 2.0022) and about ½ the cost of a conventional or neutral time (1.00*1.00 = 1.00).

The table also debunks the myth that the best team can be constructed by recruiting one top talent from each of the top 10 universities. Such a team may not be able to collaborate well, and thus end up with sub-optimal MCAP, PCAP ratings and higher cost.

3.3.3. Technology

The impact of technology on development is quantified using measures MODP, TOOL and AEXP. These measures are explained below.

3.3.4. Use of modern practices (MODP)

The MODP rating establishes the cultural level of modern developmental methods usage in an organization.

Category	**Practice usage level**	**Value**	**Description**
Expert	Routine use of ALL modern development practices	0.83	Standardized, integrated, structured methods across development. Well-established culture of inter-organization teaming.
Advanced	Reasonable experience in ALL modern development practices	0.91	Standardized, structured techniques across development. Well-established development teams. Inter-organization teaming. Mature development processes.
Basic	Reasonable experience in SOME modern development practices	1	Formal development standards and practices. Some use of structured design and programming. Limited use and stability of team approaches. Separate engineering, programming and test organizations
Novice	Beginning, EXPERIMENTAL use of modern development practices	1.1	High level language used. Some use of development standards and practices at individual level. Wide variation in test, documentation and QA practices

None	NO use of modern development practices	1.21	Assembly-level coding. Heavy reliance on super experts. Wide variation in software development practices

3.3.5. Use of modern tools (TOOL)

This parameter indicates the degree to which the software development practices have been automated, and will be used in the software development. Tools that are not part of the de facto standard tool set are not to be considered.

Category	**Practice usage level**	**Value**	**Description**
Very high	Fully integrated environment	0.83	Integrated application development environment. Integrated project support. Visual programming tools. Automated code structuring. Automated metric tools. GUI development and testing tools. Use of 4GLs. Code generators. Screen generators.
High	Moderately integrated environment	0.91	CASE tools. Basic graphical design aids. Word processor. Implementation standards enforcer. Static source code analyzer. Program flow and test case analyzer. Full program support library with configuration management aids. Full, integrated documentation system. Automated requirement specification and analysis. General-purpose system simulators. Extended design tools and graphics support. Automated verification system. Special-purpse design support tools.
Nominal	Extensive tools, little integration	1	Multi-user operating system. Interactive source code debugger. Database management system. Basic database management aids. Compound statement compiler. Extended overlay linker. Interactive text editor. Extended program design language. Source language debugger. Fault reporting system. Basic program support library. Source code control system. Virtual operating system

Low	Basic tools	1.1	Overlay linker. Batch source editor. Basic library aids. Basic database aids. Advanced batch debugging aids.
Very low	Primitive tools	1.24	Assembler. Basic linker. Basic batch debugging aids. High-level language compiler. Macro assembler.

3.3.6. Application domain experience (AEXP)

Application domain experience (AEXP) rates the project impact based on the effective average number of years of experience for the entire development team. The values are categorized in three levels of complexity. The low complexity applications require little or no interaction with the underlying operating system and usually doesn't have real time constraints for responses. An example would be a simple GUI for a Tic-Tac-Toe game. The medium complexity applications require nominal interaction with the underlying operating system and may have some timing constraints. An example would be a visual frontend for a web-based data base. The high complexity applications require complex interaction with the underlying operating system and will have strict real time constraints.

	Complexity		AEXP value
Low	Medium	High	
	Experience		
>4 years	>8 years	>12 years	0.82
1.2 years	3 years	6 years	0.91
10 months	2 years	3 years	1
4 months	8 months	1 year	1.13
<1 month	<3 months	<4 months	1.25

3.4. Calculation of organizational effectiveness

3.4.1. Organization capability index

When the parameters described above are available for an organization, the Organization Capability Rating (OCR) is expressed as

OCR = 0.23/(ACAP*PCAP*MODP*TOOL*RESP*TURN*AEXP)

The number 0.23 is a normalization constant based on the range of values of ACAP, PCAP, MODP etc. The RESP is the terminal response time and TURN is the hardcopy turnaround time (how much time it takes to pick up a printout from a printer). The variances in these numbers are no longer applicable in today's technology world, so we can apply the two most optimum numbers for RESP and TURN, which happen to be 0.96 and 0.87 respectively. Since 0.23/(0.96*0.87) = 0.28, we can rewrite the OCR equation as –

OCR = 0.28/(ACAP*PCAP*MODP*TOOL*AEXP).

This can be easily measured for the different developmental sub-units, and once the data is available, can be used to make some changes to normalize the sub-units. For instance, it doesn't seem rational that the team with the lowest OCR is assigned the next most important project.

3.4.2. Typical OCR values from industry

The table below is directly lifted from a software productivity study. The study describes seven different real projects, the OCRs and other parameters for those teams, and how the teams ranked (in percentile) within the SW industry. The value of the table below is that when a team calculates its OCR, it can use this table to get an idea of what that number means within the SW industry.

ACAP	PCAP	AEXP	MODP	TOOL	OCR	Percentile
0.81	0.86	0.83	0.97	0.97	0.4	75
0.86	0.86	0.89	1	1	0.36	70
1	1	0.89	1.04	1.03	0.26	55
1	1	1.1	1	1.05	0.2	35
1	1	0.97	0.95	1	0.25	54
0.91	0.9	0.97	0.92	0.95	0.32	64
1.46	1.42	1.1	1.1	1.1	0.06	1

3.4.3. Basic Technology Constant

Another approach to evaluating capability is a formula called Basic Technology Constant. It uses the same factors as OCR, but is mainly used to compute cost and schedule in a project's estimate through the use of Seer Software Equation presented in section 4.2. It has been found that the technology constant C_{tb} has a range of values typically from ~2000 for weak development teams to ~25000 for extremely competent development teams. Before Agile processes were introduced, the highest rating actually calculated was ~9000, and since then some Agile teams reported technology constants from 10000 to 12000.

The basic technology constant is derived as follows –

T = ACAP*AEXP*MODP*PCAP*TOOL
$V = -3.6427*\ln(T/4.11)$
$C_{tb} = 2000*\exp(V/4.35)$

It is to be noted that the actual value used in the Seer Software Equation is not the basi technology constant (C_{tb}) but the effective technology constant (C_{te}). They are related by the following relationship –

$C_{te} = C_{te} / \prod f_i$

Where f1, f2 etc are environment variables such as experience factors, volatility factors and management factors. In the SEER model, they all have associated values like the AEXP etc above.

The table below provides a way to compare the C_{tb} with OCR as well as SW industry percentile rank.

OCR	Ctb	Percentile
0.5	11454	86
0.4	8612	75
0.36	8023	70
0.32	7311	64
0.26	6416	55
0.25	6156	54
0.2	5129	35
0.06	2194	1

3.5. A note on the numbers above

To a development team manager, the method described above for calculating organizational effectiveness may not seem ideal. There could be a different set of factors that are more applicable for their organization. For instance, a networking SW team today may be more interested in measures such as Virtualization Software Experience (VEXP), Linux Kernel Experience (LEXP) etc. It is perfectly ok for a development team to come up with their own definition of OCR. However, it is expected that their adopted measures should be relevant to other development teams also who are part of the same organization. The main idea here is both to rank the development subunits and to normalize the capabilities across the entire organization, by distributing resources if necessary.

4. Staffing and schedule estimate

4.1. Rayleigh-Norden staffing pattern equation

Rayleigh-Norden staffing pattern equation came into existence after seeing many projects to fail simply because the project managers treated the project as a single monolithic activity done by a single team. In a monolithic project, one can estimate the total development time in man-months by first estimating the total number of man-months to complete the project and then dividing it by the available headcount. But a real project involves people from multiple teams, getting involved at various overlapping phases, with distinct team sizes and compositions.

Norden's study at IBM, summarized in his book *Management of Production,* contends that for successful projects, the staffing rate didn't exceed a maximum staffing rate that could be defined for that project based on its complexity. Exceeding this staffing rate often results in project failures. Norden took the Rayleigh distribution of probability and came up with Rayleigh-Norden staffing equation. The equation is

$p(t) = M.t.\exp(-t^2/2t_d^2)$

Where,

$p(t)$ = staffing rate at time t

t = time measured in years from the beginning of the project,

M = Maximum staffing rate

t_d = Full scale development time

M can be derived in successive steps as

E_d = Full-scale development effort in man-years

$K = E_d / 0.39345$

$M = K/t_d^2$

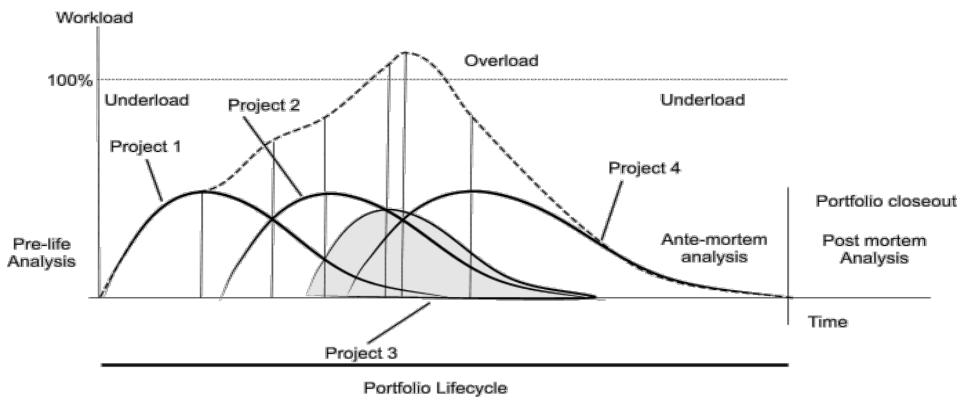

Figure 1 Examples of Rayleigh-Norden curves

Rayleigh-Norden equation gives us realistic staffing estimates for a new project. Often we worry that our reqs aren't getting approved quickly or we aren't hiring fast enough. But there are practical limits to hiring rate that we can calculate ourselves.

4.2. Seer Software equation

The Seer Software equation tells us that

$S_e = C_{te}. (K.t_d)^{0.5}$

Where,

S_e = ESLOC
C_{te} = Effective developer technology constant (calculated from the previous AEXP, MODP and other measures)
K = total software life-cycle effort = area under the Rayleigh-Norden curve
t_d = Full scale development time

So, we can either calculate the best case ESLOC within a given time for an organization of a certain size whose capabilities are known, or we can calculate the time it will take if the target ESLOC is given.

K and t_d are related to each other through the complexity of the project, D, according to the following equation –

$D = K/t_d^3$

The following numerical values are typically assigned as D, the project complexity, based on the project type. Higher numbers indicate lower complexity.

Project description	**Complexity**
Development primarily using microcode. Signal processing systems with extremely complex interfaces and control logic	4
New systems with significant interface and interaction requirements with larger system structure. Operating systems and real-time processing with significant logical code.	5
Applications with significant logical complexity. Some changes in the operating system but little or no real-time processing	12
New standalone systems built on stable operating systems. Minimal interface with underlying operating system or other system parts	15

Software with low logical complexity using straightforward I/O and primarily internal data storage	21
Extremely simple software containing primarily straight line code (minimal branching and control transfer) using only internal arrays for storage	28

4.3. "Paul Masson rule" for software staffing

The "Paul Masson rule" for software staffing provides us with a minimum development schedule given the complexity of the project. It comes from the observation that if we keep M, the maximum staffing rate, a constant, then the total cost K is related to the development time t_d in a parabolic relationship

$K = M.t_d^2$

The product development time t_d, according to the relationship above, will have a certain minimum value that can't be changed even if cost is increased. This minimum development time is called the Paul Masson point. If we attempt to reduce the schedule below the minimum time, the cost will increase and the schedule will also increase as described by Brooks Law, that says "Adding people to a late software project makes it later."

5. Software Quality Ranking (SQR)

5.1. Motivation

A matured, efficient, process-oriented SW development team should not just stop at running a detailed regression suite occasionally. They should actually run a regression suite several times a day, ideally before every commit, and they should have a running dashboard where the results of regression runs are immediately uploaded to be visible inside the whole organization. In the same way, a matured, efficient, process-oriented SW development team should not just stop at occasionally measuring their productivity and efficiency. They should create an index for their team quality, based on different aspects of SW development ranging from whether warnings are attended to during compilation, to whether adequate documentation is provided. One can call such an index the Software Quality Ranking index or SQR index.

5.2. Definition of SQR

The SQR index is a number and usually it is not an average of all the ranks but the lowest rank in any particular category. This discourages attempt to improve in specific areas of lesser importance that are low-hanging fruits, at the expense of ignoring improvement in areas that are important as well as hard to improve on.

Quality Area	Rank	Rank	Rank	Rank	Rank	Rank	Rank	Rank	Rank
	0	1	2	3	4	5	6	7	8
Compile			No errors	No warning is default			No warning, strict		
Static Code Analysis	N/A	Unresolved defects ≥ 30 days		Defects ≥ 30 days are resolved	Ignored defects are reviewed		All legacy defects fixed		
Module Documentation		None	Trouble-shooting Guide	Debug CLI	Arch. Document	1 add. doc if applies	2 add. docs if applies	3 add. docs if applies	4 add. docs if applies
Code Review		Not done	Desk check	Team		Lead	Walk-through		
Memory Analysis	N/A	Not done		Static		Dynamic & Fix			
Unit Test Coverage	N/A	None		All auto UT has code coverage ≥ 50% for new code	All auto UT has code coverage ≥ 60% for new code	All auto UT has code coverage ≥ 70% for new code		All auto UT has code coverage ≥ 80% for new code	
Code Coverage	N/A	None	< 50%	≥ 50%	≥ 60%	≥ 70%		≥80%	
Bug Backlog		Yes		At most 1 severity A,B > 30d,		At most 1 sev A,B > 28d	At most 1 sev A,B > 25d		At most 1 sev A,B >20d

				5 sev C > 50d		5 C > 42d	5 sev C > 45d		5 C > 30d

5.3. The recommended process for applying SQR

A SW development unit not using SQR index currently should proceed as follows. First all subunit managers should publish the SQR indices of their teams. Then the division head should mandate his/her goal for SQR number for the whole division, and the subunit managers must submit a plan of how to attend that SQR for their own group. For best effect, the SQR improvement should become an MBO goal for the year.

Once the results start coming in, the IT department can create an internal webpage where these numbers are displayed continuously.

6. Conclusion

6.1. Benefits of this report

Going beyond the original scope of the work, in its current form the report should help a SW development team manager in the following areas –

- Measuring individual productivity through ESLOC tools and using that data for the year end focal/calibration processes
- Measuring organizational effectiveness in a subjective way by calculating OCRs for the different subunits
- Estimating resource requirement for a new project
- Creating a feasible staffing and ramping up plan for a new project
- Creating a quality yardstick for all sub-units that can be easily measured, compared and shown in a dashboard.