QSM® Software Almanac

Development Research Series

Winter 2016 Edition

QSM® Software Almanac

Development Research Series

Winter 2016 Edition

Published by Quantitative Software Management, Inc.

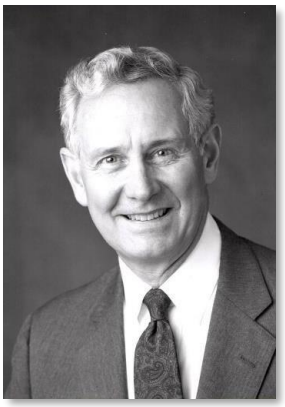
2000 Corporate Ridge, Ste 700
McLean, VA 22102
800.424.6755
info@qsm.com
http://www.qsm.com

Second Edition

TABLE OF CONTENTS

FOREWORD

Then and Now: My Journey of Discovery in the World of Software Application Estimation and Risk Management



Larry Putnam, Sr.,
Founder and President Emeritus

Then…

How Did I End Up in Software?

It was the mid-1970s, and I was a Lieutenant Colonel in the United States Army. After graduating from West Point in 1952, I had spent eight years serving in the Army as a tank officer, and then attended the Naval Postgraduate School to study physics and nuclear effects engineering. I was assigned to Sandia Base, New Mexico, where we attempted to figure out the best way to protect soldiers from the effects of nuclear explosions. I spent a great deal of time working on very tedious blast calculations using a slide rule. Sandia National Laboratory was next door to my office, and they had just received the biggest and best engineering computer then available. The Lab had spare capacity and offered computer time for anyone needing it, even offering to teach programming, so I decided to take a course in FORTRAN programming over the lunch hour so I could make my blast calculations a bit easier. I finished the course and received a certificate of completion, which I sent off to be placed in my personnel folder at the Pentagon. Little did I know then how important this twist of fate would become!

Fast forward—I was just completing a leadership assignment as a battalion commander at Fort Knox, Kentucky. In most military careers, one inevitably ends up at the Pentagon for a short stay, and it was now my turn. It turns out that there were not many jobs for nuclear guys, but my expertise in data processing — by virtue of my "vast" experience in FORTRAN programming — gave me the qualifications to be a budget director for the Army's computer

programs. Boy, was that a stretch! Back at that time, the Army was spending about $500 million per year on IBM mainframe computers (360/370 generation) and about 50 large administrative computer systems (payroll, personnel, logistics, etc.).

New to my duty station, we were getting ready to engage the DoD (Department of Defense) budget review authority (the Office of the Secretary of Defense, or OSD) and defend our budget. These can be difficult discussions, as everyone is posturing and protecting their own funding sources. There was one particular system that came into question: SIDPERS (the Army enterprise personnel system), which had been under development for five years, with its initial release just fielded. The staffing had already peaked at 110 people and we were now projecting 93 people for the next five years to "maintain" the system (Figure i.1).

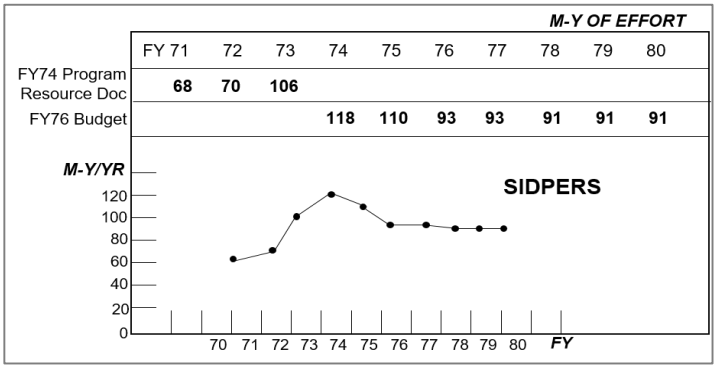


Figure i.1. SIDPERS man-year budget data.

The OSD analyst sitting across the table asked a simple question, "What are those people going to be doing?" to which I did not have a good answer. We adjourned for the day with the understanding that we would address this issue first thing in the morning. I scrambled back to my office and immediately called the development center, which had submitted that particular budget item. I asked them the question posed to me, but did not receive a satisfactory answer, only a lot of technical jargon and hemming and hawing. The next morning, by 10:00 a.m., I had lost 10 million dollars from the Army's budget! That got my boss's attention, and we had to spread the pain throughout the Army's IT (information technology) community. Looking back on it now, those 93 people would have been putting out the next release, fixing problems, providing help desk support, and evolving the infrastructure around which the system was built, but we were unfortunately unable to articulate that at the time.

As we walked away from the budget hearings, my boss, who was a major general in the Corps of Engineers, said "You know, Larry, when we have a big construction project, like building a dam or a new airfield, we have some high level parameters like the number of cubic yards of concrete it will take to do the job that would allow us to get in the ballpark for cost and schedule. Whenever I talk to the IT guys, I hear about bits and bytes, programming languages, and bandwidth, but nothing that relates to time, effort, and cost."

That became my mission! How long is it going take? How much will it cost? How many people will it take to build it? When will it be good enough to go into service? The management questions!

My Solution—The Rayleigh Equation & Software Production Equation

The Rayleigh Equation

Being an engineer, I approached this like any other engineering study: begin with a literature search and then start looking for meaningful relationships in the engineering data. Shortly thereafter, I stumbled upon some work by Peter Norden of the IBM Poughkeepsie Development Lab. He had discovered a time-varying function called the Rayleigh equation. Norden's observation was that the Rayleigh function seemed to be the ideal way to apply people to a design-intensive project. You start at zero, build up staff as the problem is decomposed, and reduce staffing as the release is constructed, integrated, and tested. Norden was applying this mostly to hardware and microprocessor chip designs, but from what I could tell, software followed a similar design process. There are some other really nice features of the Rayleigh function, whose dimensions are time and effort (see Figure i.2).

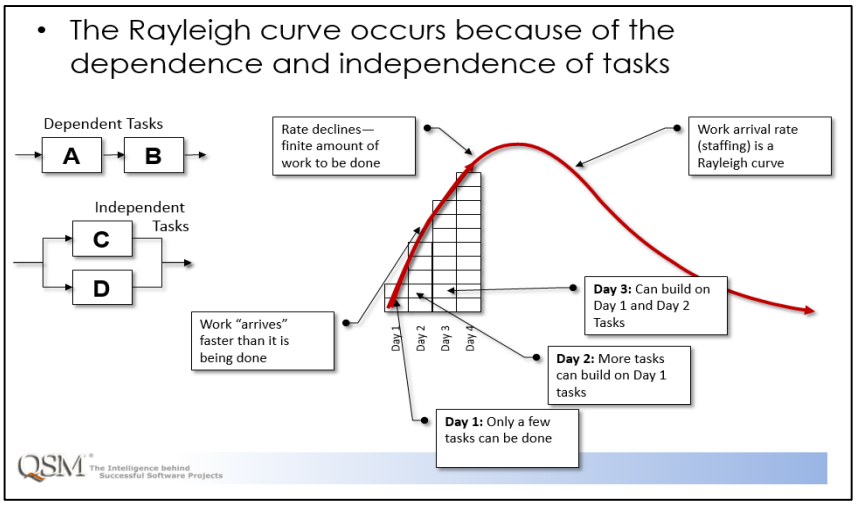


Figure i.2. The Rayleigh curve and work calculations.

The first questions asked by stakeholders are "How much is it going to cost?" and "When can I have it?" (wait, let me change that: "I need it by ***next week!***"). The Rayleigh curve predicts very practical management decision-making information, such as:

- Weekly/monthly staffing plan
- Cumulative hours of effort and the weekly/monthly effort rate
- Cumulative cost
- Weekly/monthly spending rate (cash flow)
- Defect discovery rate and cumulative defects
- Product construction rate and cumulative product complete

I began collecting some Army data to see if it followed the Rayleigh pattern. Just from comparing the budget data from a group of about 15 systems, I knew we had a match. Initially, we just used simple projections of the Rayleigh curves to get our 50 systems currently in development under financial control. The harder part was figuring out what the right Rayleigh curve should be when we needed to provide a new estimate.

The Software Production Equation

In engineering studies, one always looks for data that is relevant and substantial enough that you might actually be able to derive some meaningful relationships. So, I dug around and was able to come up with 19 product releases from the same organization (see Figure i.3). The design agency had the schedule, the man-years of effort, and several potential measures of size—namely, the numbers of files, reports, and application programs.

USACSC System Characteristics

System	Life Cycle Size K (MY)	Development Time t_d (Yrs)	Number of Files x_1	Reports x_2	Appl. Progs. x_3
MPMIS	73.6	2.28	94	45	52
MRM	84	1.48	36	44	31
ACS	33	1.67	11	74	39
SPBS	70	2.00	8	34	23
COMIS	27.5	1.44	14	41	35
AUDIT	10	2.00	11	5	5
CABS	7.74	1.95	22	14	12
MARDIS	91	2.50	6	10	27
MPAS	101	2.10	25	95	109
CARMOCS	153	2.64	13	109	229
SIDPERS	700	3.65	172	179	256
VTAADS	404	3.50	155	101	144
BASOPS-SUP	591	2.73	81	192	223
SAILS AB/C	1028	4.27	540	215	365
SAILS AB/X	1193	3.48	670	200	398
STARCIPS	344	3.48	151	59	75
STANFINS	741	3.30	270	228	241
SAAS	118	2.12	131	152	120
COSCOM	214	4.25	33	101	130

The original Army data set contained 19 good, homogeneous projects:

- Same organization
- Same application type
- Same tooling
- Complete data (size-time-effort)

"Man, was I lucky!"

Figure i.3. Original historical data from Army projects.

I began to work through some analysis. For my situation, I had two unknowns (time and effort) and four or five potential equations, so I needed to use matrix math to solve the simultaneous equations (Figure i.4). The final result was a number of equations which looked good. The statistical goodness of fit was decent, meaning that this was a very promising direction to go.

I graphed all of these equations in the Rayleigh curve dimensions of schedule and effort. I found that they all intersected in a very similar region, which told me we had a solution that could work. An early form of my ultimate software production equation is shown at Figure i.5, which is pretty close to the one we have used in the SLIM® tool for close to 40 years.

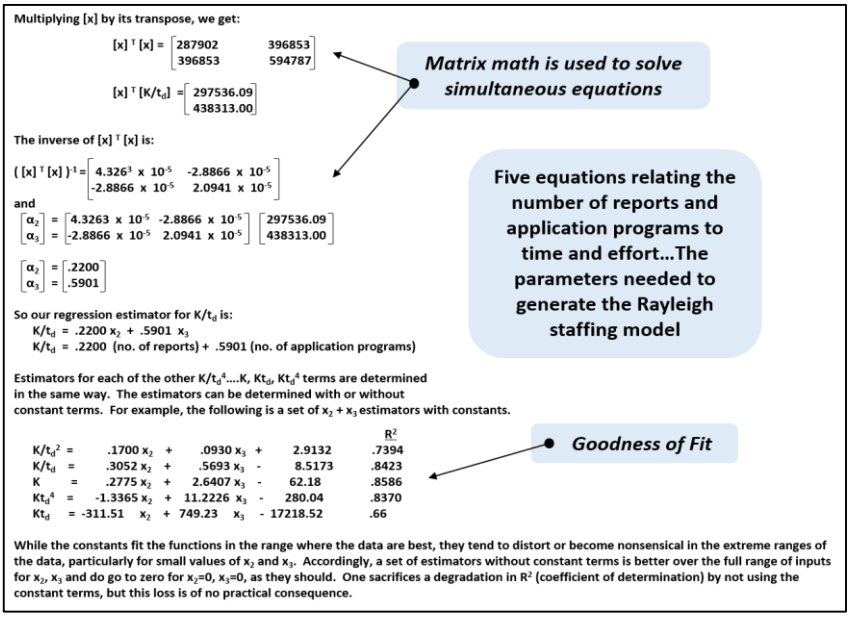


Figure i.4. Statistical equations used to determine the goodness of fit.

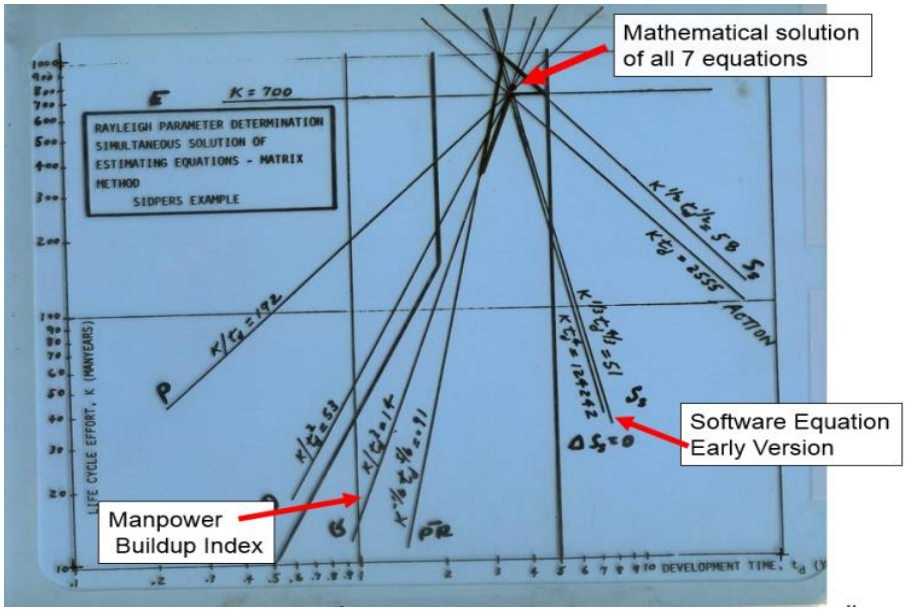


Figure i.5. Early form of the software production equation.

The **software production equation** is composed of four terms, namely: Size, Schedule, Effort and Productivity. The equation is described below:

$$\textbf{Size} = \textbf{Productivity} \times \textbf{Time}^{4/3} \times \textbf{Effort}^{1/3}$$

Notice the exponents attached to the time and effort variables. The schedule variable, or *time*, has been raised to the 4/3rd power, while the *effort* variable has been raised to the 1/3rd power. Without getting too technical, it means that there is a diminishing return of schedule compression as people are added to a software project (see Figure i.6). In the most extreme conditions, it means that spending a lot of money in labor costs produces little or no schedule compression.

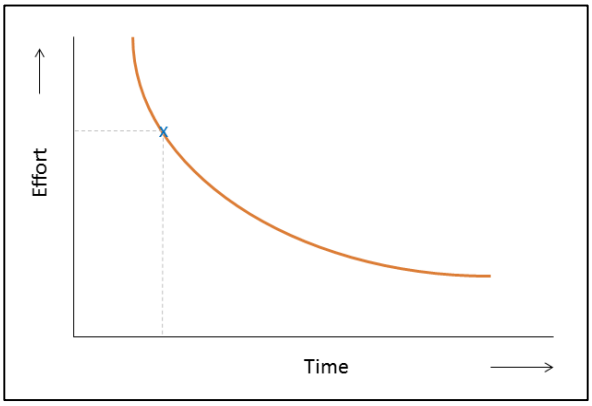


Figure i.6. Utilizing more people requires exponentially more effort.

However, what is nice is that this phenomenon also works in reverse. If a project can afford to lengthen the schedule, it can be accomplished significantly cheaper by using fewer people (see Figure i.7). In today's economic environment, this is a very attractive option.

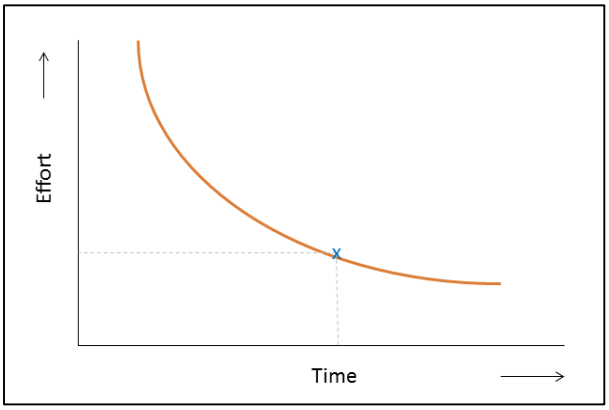


Figure i.7. Lengthening the schedule drastically reduces the effort/cost.

The software production equation is simply a quantification of "Brooks' Law," showing that the software production equation models the complexity of human communication, which manifests itself in defect creation and rework cycles. As more people are added to a project, human communication complexity grows exponentially, which in turn results in more defects being created and, thus, more rework cycles to fix them.

Putting It All Together

So how do we use the software production equation and the Rayleigh staffing model? For any new estimate, one would start with the software production equation. You will need to provide an input of size and productivity. To get at productivity, you will use a historical data

set of several similar projects and calculate the productivity parameter by simply rearranging the software equation and inputting actual size, schedule, and effort. Then you make a value judgement on whether to increase or decrease the productivity, based upon how similar or different the new project will be, relative to the historical calculations. The second input is the size of the new system. What we are really getting at is how big the system will be, and the sizing metric depends upon where we are in the software development lifecycle. Typical measures available early in the project would be the ***number of requirements***, ***use cases***, or ***agile epics***. Later, it might be the ***number of components*** or ***functional measures,*** such as function points. You can use range estimates of size and convert them to a common sizing unit equivalent, such as a single statement of expected code or a single configuration step (for package implementations), using a gearing factor.

Now you are able to begin the estimate. The software equation takes the assumed inputs and uses them to estimate a range of potential schedule and effort solutions. Generally, there are a finite but large number of schedule and effort combinations (every solution is a time-effort pair) for any set of inputs, so it is important to understand the available options. For example, you could use fewer people and lengthen the schedule, or you could shorten the schedule and use more people. If you move the solution far enough in either direction you will eventually hit a roadblock or an impossible situation. Shortening the schedule too much could put the project into the impossible zone, overstretch the budget constraint, or require more people than are available. Conversely, moving too far in the other direction could also cause roadblocks, as lengthening the schedule too much could cause the project to miss its deadline or go below the minimum staffing constraint (see Figure i.9).

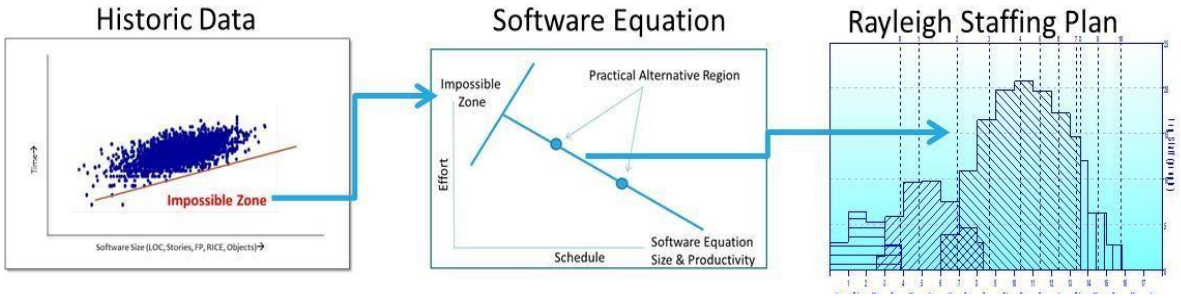


Figure i.9. The SLIM® estimation process.

Once project and organizational constraints are imposed, you can create a practical alternatives region, within which any combination of schedule and staffing would be a reasonable solution to select. The next step would be to identify the solution with the highest probability of success, and this optimum solution would be used as inputs to the Rayleigh equation to generate the staffing, effort, cash flow, and cumulative cost plans (Figure i.10).

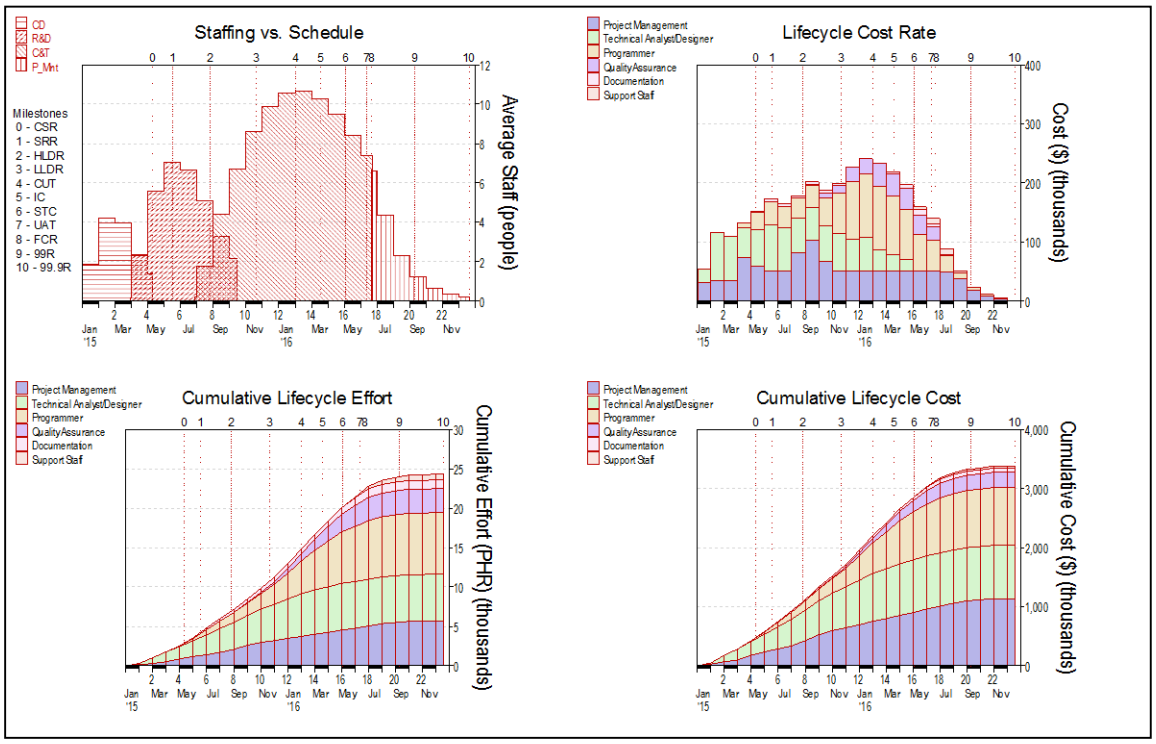


Figure i.10. SLIM-Estimate® outputs.

Well, that is my story of how I began and the basic methodology that I turned into the commercial product known as SLIM®. A lot has changed, however, in the last 40 years.

So that was then…*What about now? Are the algorithms still valid? Do things "hang together"?*

Now…

The world has changed a lot since I started on this journey. Waterfall development methodologies have morphed into Incremental and agile methods. Languages have totally changed. Platforms went from centralized time-sharing to desktops, then to servers, and have come full circle back to cloud-based platforms.

The funny thing is that people still have a common way of approaching problem solving. First, they decide ***what*** they are trying to accomplish. Then, they decide ***how*** they are going to solve the problem, before going on to develop the solution — the ***do*** step. Finally, they ***cleanup*** any residual defects that slipped through the net. I call it the ***what-how-do-cleanup cycle***. The degree of overlap between the phases and the nomenclature that people use to describe the work changes by methodology, but the basic discovery process and work content is there (see Figure i.11).

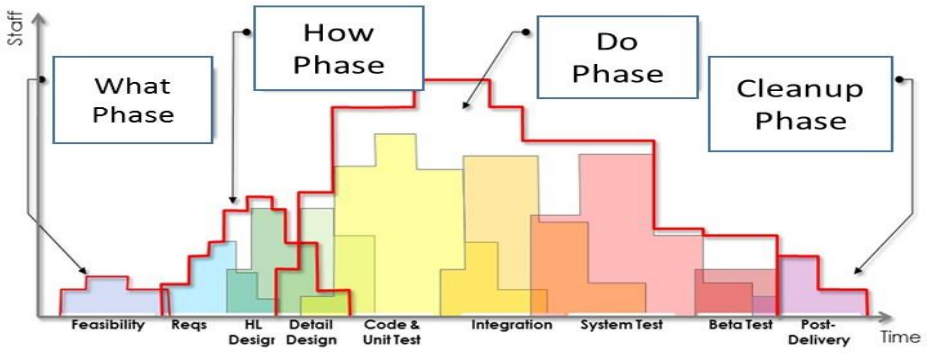


Figure i.11. High-level software development phases.

Table i.1 shows some of the more commonly used methods used in the past decade, along with the names that are used and how they relate to the general what, how, do, and cleanup phases.

Methodology	Phase 1: What?	Phase 2: How?	Phase 3: Build & Test	Phase 4: Deploy & Stabilize
Waterfall	Concept	Requirements & Design	Construct & Test	Deploy
RUP	Initiation	Elaboration	Construction	Transition
Agile	Initiation	Iteration Planning	Iteration Development	Production
SAP ASAP	Project Preparation	Business Blueprint	Realization & Final Prep	Go Live

Table i.1. Common software development methodologies over the last decade.

Below, we show two completely different development approaches, using the four-phase Rayleigh model. The first instance is agile development (Figure i.12), one of the more popular approaches today. You will notice that there are much higher degrees of overlap between the ***how*** and ***do*** phases, because the requirements/design/code/test activities are accomplished in short sprint cycles, which in turn requires more overlap in order to model this particular development strategy.

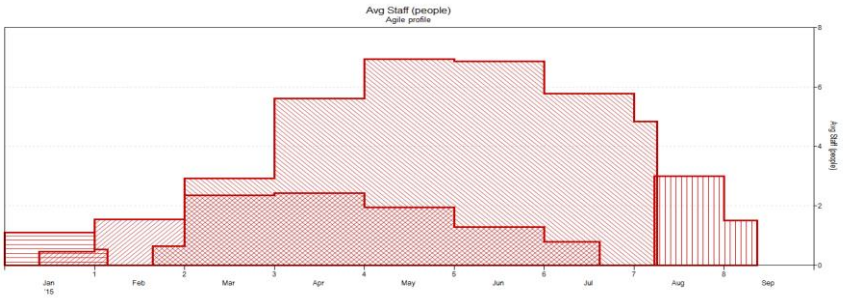


Figure i.12. Agile software development lifecycle.

The second case is the more traditional waterfall development method (Figure i.13), where requirements, design, code, and test are done in a more sequential fashion. Less overlap between phases, therefore, is more appropriate for this type of development approach.

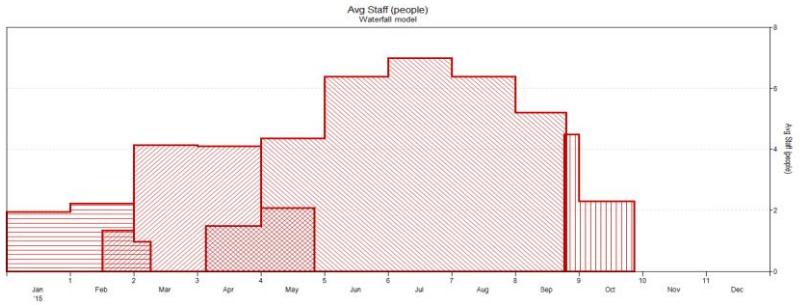


Figure i.13. Waterfall software development lifecycle.

Another observation I have is that the nature of code construction has changed. Today, people "configure packages" and develop in much more powerful languages. Developers are certainly better at reusing code. However, in the final analysis, we are still writing *some* type of code. Thus, the software equation that we started out with, close to 40 years ago, is still relevant today. Some people ask me if the tradeoff law between time and effort is still accurate. At QSM, we review that relationship on a three-year cycle, and I am happy to say that I feel more confident in that relationship today that I ever have. It works.

The real key to getting good results is capturing and using your own historical data and coming up with good methods for sizing the functionality that you have to build. Studying the historical data allows you to make accommodations for phase overlaps and changes in productivity based on actual experience, as new tools and methods are introduced. Having a sound sizing method is important because it relates back to the functionality that is being requested. In most cases, it is more than can be accomplished in the time or with the money that was initially allocated. Thus, functionality often becomes a big part of the negotiation. If you do not do a good job in this area, the negotiation often does not go your way.

Summary

So, I was lucky, and I believe that I uncovered something close to a fundamental law of nature in our software production equation. When matched up with the Rayleigh staffing model, we have a very powerful decision tool. Sophisticated enough to work, yet simple enough to use. Today this process is described as "predictive analytics." Back then, it was called systems analysis or industrial engineering.

I am just glad that I took this journey and have been able to help so many businesses and their stakeholders. Using my method, they are able to understand what is possible in software development and how to negotiate reasonable alternatives, so that they can all have successful outcomes.

It is always nice when there is some continuity between the "Then and the Now."

EXECUTIVE SUMMARY

"Some lessons can't be taught. They simply have to be learned."

– Jodi Picoult,
American author

"When you change the way you look at things, the things you look at change."

– Wayne Dyer,
American author and speaker

"I am still learning."

– Michelangelo, sculptor, painter, architect, poet, and engineer, aged 87

A Changing World: Applying Old Lessons to New Cases

Angela Maria Lungu, Editor

It has been nearly forty years since parametric estimation emerged as a compelling field of study, and since that time we have seen dramatic changes in how software is used in not only the larger environs, but in our daily lives. Changing social, cultural, and demographic patterns; spectacular technological achievements that transform how we do business; concerns about the physical environment and social responsibility; and a global marketplace that sends us competing worldwide while depositing competition at our doorsteps are only a few areas that affect how we view and use software today. Particularly germane to our discussion here is how these topics have influenced the process and development of software.

By itself, software is no longer a garage hobby or stand-alone, "nerdy" field of study. It is fundamental to and intrinsically linked with nearly every aspect of our daily "First World" lives. From simple communication to transportation, education, and our own residential tasks, software governs how we interact with each other, how we spend our leisure time, and even how we survive. Who would have envisioned Amazon's one-click ordering and same-day delivery, software updates pushed to our cars while we sleep, residential climate control and lighting that learns and automatically adjusts to our habits while home, or unmanned aircraft delivering our goods and groceries?

The impact of these changes, however, does not stop at the fringes of our personal lives. The actual production and development of software is also heavily influenced by these trends. Software is more complex and more interdependent than ever before, demanding a fresh look at how we manage its development. The prevalence of software has resulted in it becoming a far more attractive and viable career choice—everyone is developing apps and writing code. The management of global teams, across both diverse cultures and multiple time zones, requires a different type of focus and planning. Code writing and the proliferation of coding languages, advanced equipment, innovative office collaboration, new development methodologies, and a new generation of workers weaned on technology and a connected world all require a new perspective when addressing how to assemble a development team, design the software, manage the effort, and integrate all the pieces into one coherent final product. Additionally, but no less important, is the increased reliance on vendors as we see more dependence on specialized developers and a need to oversee contractors, often separated by geography and culture.

Finally, and most worrisome, is the increased criticality of software quality. Today, more than ever, software vulnerabilities and errors have instantaneous and compounding effects on a

global scale, similar to what we witnessed with the advent of global broadcast media such as television. Devices and systems, such as automobiles, power grids, banking and financial systems, transportation networks, hospital life support systems, and anything else that you can imagine, quite obviously have far less tolerance for sub-quality software than ever before, whilst relying more now than ever upon software. This is further exacerbated by the fact that distribution points, such as smart phone application stores or online vendors, dispense critical software across the world instantly, which only magnifies the potential impacts of errors in the software itself or unintended consequences if the software "does not play well with others." And, worrisomely, this does not even include mal-intentioned actors, focused on exploiting or injecting cyber vulnerabilities with which to leverage these areas of concern (we shall address those more specifically in a later edition).

So what does that mean to us? Will our current processes and methodologies be sufficient for managing and overseeing this dramatically complex development environment, or will we need to adapt just as significantly in the direction and scope of how we approach future research and techniques? Will the lessons learned over the last 35 years still be applicable, let alone relevant, for today's new challenges? How can they be adapted? This year's almanac covers a range of topics that explore these concepts.

We can frame our discussion by examining the impact on people, process, and tools. Beginning with people, Victor Fuster addresses critical personnel skills needed to successfully staff estimation centers of excellence, and Carol Dekkers focuses on the psychology of combining "soft" people skills with "hard" estimation tools. She notes that, despite a field of the brightest minds and whose jobs comprise four of the top ten *U.S. News and World Report*'s "Top 25 Best Jobs," there is a dismal 30% project success rate (on-time and on-budget delivery)—why this dichotomy? Taylor Putnam-Majarian looks closer at why these projects tend to fail, linking a lack of clear processes, definitions, and existing misperceptions, all of which may lead to this lack of success, along with the reasons brought up by Carol. Further addressing the larger management issue of overseeing vendors, Joe Madden, Jeff "JD" Ottenbreit, and Doug look at recommended practices for managing this aspect of outsourced software development.

Moving from people to process, there are several articles that re-examine foundational estimation processes and discuss their continued relevancy of and application in today's environment. Joe Madden provides a fabulous overview of automotive software and argues for the need to apply the same processes to automotive software. Paul Below and Don Beckett also make the case for continuing to apply the same basic processes of leveraging key metrics to better assess and manage projects. They look at whether the development of new and modified code requires large enough differences in effort and how to account for any differences in productivity indices. Looking at process proliferation, Phil Armour tackles the interesting dilemma of "big" versus "lightweight" process. He notes that as projects become more complex, so, too, do their respective processes, which in turn may become so overwhelming and cumbersome that they begin to slow down and disrupt the projects themselves. On the other hand, with too little processes, the complex projects of today can quickly go off track by assuming that the developers will have all the skills and experience to

intuitively understand the intricacies of the project, dooming it from the start (Taylor also addresses this in the discussion of why projects fail). Paul Below provides a detailed look at the importance of sample sizes and proper stratification to create trend lines, especially im portant in times of churning conditions. This is complemented nicely by Taylor's article on fact-based decisions, focusing on the importance of learning from historical data to better manage project expectations up-front (back to the "people" piece of the "people-process-tools" triad) and to establish a realistic, quantitatively-derived baseline.

Finally, we have several articles that focus on tools. James Heires presents a compelling argument for the continued relevancy of Source Lines of Code (SLOC), a sizing tool that has been with us from the start and evokes strong emotions from both sides. Andy Berner explains his perspective on using the level load staffing templates as a tool for agile projects, and Don Beckett gives us several useful rules for applying the right "tool" for our particular software project. He notes that "if the problem (or software project) we face today is fundamentally different from those with which we have dealt previously, past experience is not the proper framework" — we must adapt our previously learned lessons and apply them correctly. Hearkening back to Phil's article, Paul cautions on the proper balance of metrics analysis, noting that "bad statistical analysis, even on the right variable, is malpractice." And, again referring to the need for balancing, this time between resources and demand for IT development, Doug and Taylor tackle the struggle between capacity planning and demand management.

So what does all this mean? Must we disregard our old processes, tools, and lessons? I believe that the resounding answer is no, based on our authors' varied perspectives and points, although caveated with the directive to adapt to the new. The consensus appears to be a continued appreciation for and application of sound estimation principles and processes, validated repeatedly over the last 35 years. Although it may seem that old competencies no longer produce dramatic improvements, relying on proven methods *can* help us navigate clearly through the complex and changing conditions. Our sampling of articles demonstrates that these same lessons and principles can be adapted to the changing environment of today and, in fact, are even more relevant than ever.

As Larry Putnam, Sr., described in his "Foreword," there is a definite continuity between "Then and Now." It is now up to us, as practitioners and leaders, to ensure that we apply the same rigor and discipline to adapt and improve our techniques and perspectives along the way.

---

1. FIVE CORE METRICS

"Without metrics, you're just another guy with an opinion."

– Stephen Leschka,
Hewlett Packard

"Metrics are for doing, not for staring. Never measure just because you can. Measure to learn. Measure to fix."

– Stijn Debrouwere,
freelance data scientist

"If you torture the data long enough, it will confess to anything."

– Ronald Coase,
British economist and author

The Lowly Line of Code

James Heires

Source lines of code (SLOC) is a measure of software size, and has been in continuous use since the 1960s. This paper describes various uses of SLOC from the perspective of software measurement, and how relevant SLOC remains today for modern programming languages, mixed operational environments, and new development methodologies.

The purpose of using SLOC in a measurement practice is simply to capture software size. Period. SLOC has been used to size everything from Arduino to Zip code software. Despite its broad applicability (or perhaps because of it), SLOC seems to get the blame for many a failed project, process improvement, or software metrics initiative. There are even those who claim that "…in many situations usage of LOC metrics can be viewed as professional malpractice…" (Jones, 2008) However, SLOC may be more relevant than ever, when used intelligently.

To Count, or Not to Count

Early on, SLOC was pressed into service to help with myriad management challenges. Today, some of these uses of SLOC are considered bad practice, if not outright abuse. One often-cited example is establishing a reward system based on the amount of SLOC developed (or compiled, tested, documented, or released). The problem with this practice (especially if deployed on individual employees) is that it rewards the wrong behavior—or perhaps an incomplete view of the correct behavior. Without considering other factors (e.g., quality, maintainability, complexity, correctness, etc.), rewarding employees on SLOC alone can backfire rather quickly. This dysfunctional behavior (only related to defects) was aptly described by Scott Adams in his Dilbert cartoon strip from 1995 (Figure 1.1).

Figure 1.1. Dilbert wisdom.

Alternative "Sizing" Measures

Because of the early misuse of SLOC, all sorts of alternatives to SLOC (e.g., function points and more than 35 variants, story points, object points, use cases, and many, many others) were invented in an attempt to "fix" SLOC, but in the end these only complicated matters. As well-respected and widely used as function points are today, there are some serious considerations when comparing to SLOC as a software size measure. Measuring Function Points properly requires a certified Function Point Specialist (CFPS), preferably with decades of software experience. In addition, there are no automated tools with which to measure function points automatically and consistently. Finally, because function points are measured by people (not computers), ten CFPS-certified professionals analyzing the same system will very likely arrive at ten different results.

Most of the alternative sizing measures try to accomplish too much. Their value propositions include concepts such as functionality, complexity, user behavior, or elements of architectural design. As a result, only a few of these measures survive today, and none have the simplicity, understandability, or clarity of intent that SLOC enjoys. In fact, no other measure better answers the most important software sizing question: How much code?

Whether the need for software size involves historical actual size, an estimate of an upcoming project, or a forecast of an ongoing effort, SLOC is one of the most unambiguous measures of software size available. Other size measures do not enjoy the purity, flexibility, or direct observability that SLOC does.

Common Currency

Although it is certainly possible to "Frankenstein" together size measures from a number of disparate technologies (e.g., modules, reports, screens, queries, etc.), and then apply some kind of translation or gearing factor to get to a common size unit, it seems that collecting SLOC is not only more direct, but also more accurate. Starting with the end in mind, as Steven Covey states (Covey, 2004), it is best to use the same measure as is needed in the end, right from the start…and SLOC would be an excellent choice.

SLOC should always be carefully defined for operational use in an organization so that consistent results are obtained (Park, 1992). Subcategories of SLOC, such as new, modified and unmodified SLOC, are typically used, as each of these suggests differing amounts of effort. In other words, unmodified code (or reuse) is typically easier to implement than new code (although there are exceptions). Therefore, keeping SLOC counts separated helps later, when an estimate of a new project is needed (see sidebar: SLOC counting tools).

Modified SLOC, in particular, holds special value, but is more difficult to measure. The value of modified SLOC lies in its ability to communicate how much code has changed. Because a software program, over its operational lifetime, may undergo modification hundreds of times, is created new but just once. This modified SLOC count answers the next most important sizing question: Compared to what?

SLOC counting tools that provide these kinds of measures (Heires, 2014) serve as a solid size measurement basis across dozens of programming languages, platforms and methodologies.

But, What about Visual, Model-Based and Mixed-Language Environments?

Today's computing architectures are more complex than ever. Some of the more "visual" or model-based development environments leverage a pictorial lexicon as the developer interface. But it turns out (despite appearances) that almost all modern software development environments contain (sometimes cleverly hidden) source code that can be measured in terms of SLOC.

In the case of mixed-language environments (e.g., web-based code such as PHP, HTML, javascript, CSS, etc. in the same source file), one may need to get creative so that code is neither skipped nor double counted. That said, SLOC is still applicable when sizing these applications.

SLOC Counting Tools

A good SLOC tool will allow the user to make separate counts of important elements, such as source code, comments and blanks.

In addition, SLOC tools should be configurable to calculate new vs. modified vs. unmodified source code by comparing two versions of a system.

Finally, tools should support as many programming languages as possible, so that consistency in measurements is achieved.

See a list of SLOC counting tools at www.qsm.com

What about Agile?

Methodologies currently in vogue may seem different than other, more traditional development paradigms, but from a measurement perspective, they are very much like old-school approaches. Some agile/scrum teams shun estimation altogether, favoring instead the rapid-fire approach of code first; ask questions later. Although this is generally done to save time, this often causes rework and quality issues that, in the end, cost more to fix than to prevent. However, as a measurement consultant working with agile teams across the country, I have come to understand from a very detailed and quantifiable perspective that

this behavior is short-sighted and borders on irresponsible. Measurement, after all, is a form of communication that serves engineers and managers alike, and the more concise and comprehensible the measure, the better. It is not good enough to simply deliver whatever functionality the software team happens to have completed by the due date.

Use of SLOC in Estimation—Does Size Still Matter?

If the ability to directly measure software size were not enough, SLOC can also be combined with other measures and cost estimating relationships (CERs) to facilitate making effort, duration, or productivity estimates and forecasts.

In traditional software metrics practice, SLOC is typically used as the denominator of a metric, such as defects discovered per SLOC, to normalize the numerator. This is done to remove the (obvious, in this simple example) dependency of defects on SLOC. This practice also enables comparison between projects of different sizes (although caveats are in order, as you will see).

The Caveats

The use of linear productivity metrics, such as SLOC per person-hour, is troubling for a number of reasons. These metrics ignore the well-understood fact that productivity varies widely with size, duration, and quality (Figure 1.2). If these dependencies are not accounted for, one can quickly get into trouble.

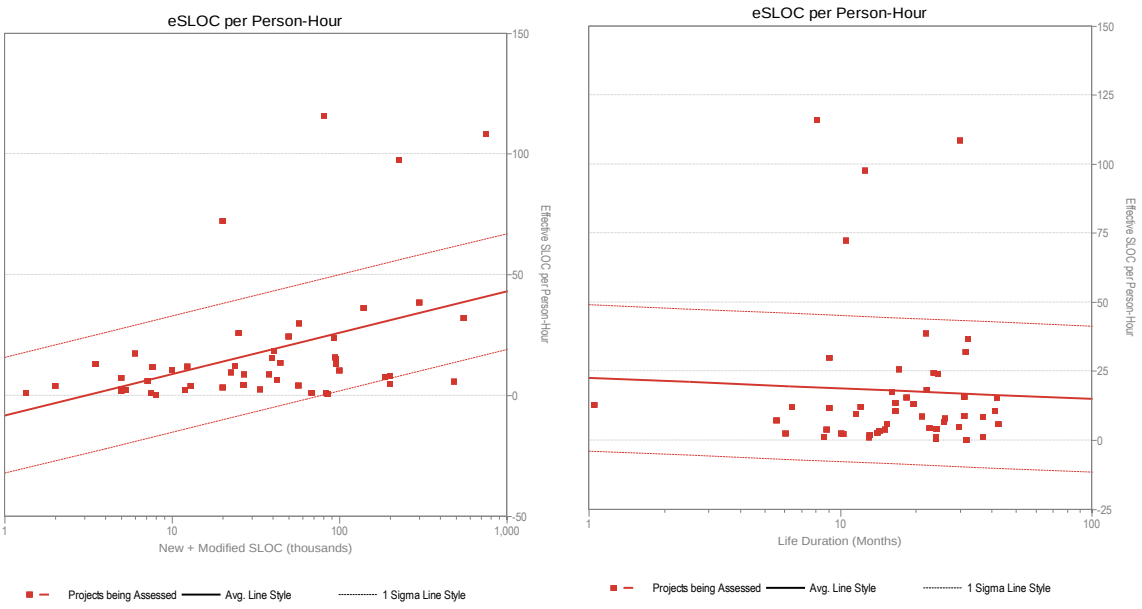


Figure 1.2. SLOC and duration graphed against SLOC/person-hour.

The QSM approach, honed over many decades, understands these relationships. The Putnam model, built into QSM's SLIM® family of tools, makes use of the Putnam software equation (Putnam, *Measures For Excellence*, 1992), decades of research, validated historical

data, and an understanding of modern software engineering methods, and brings these concepts and relationships into practical focus. Without getting into undue detail, QSM's historical data demonstrates that effort, defects, and productivity all increase (in general) with size. However, as you can see from Figure 1.2, SLOC per person-hour tends to decrease with size, and all of these relationships tend to be non-linear (please note the log-log scales in Figure 1.2). So, it is time to do away with old-school linear relationships such as SLOC per person-month.

Summary

Although unglamorous, SLOC remains the most universally applicable of all software sizing measures. Across programming languages, development paradigms, application domains, and methodologies, SLOC is the common currency of size. As one of the five core software measures (Putnam L. H., 2003), size matters, after all…

Works Cited

Covey, S. R. *The 7 Habits of Highly Effective People: Restoring the Character Ethic*. New York: Free Press, 2004. Print.

Heires, James T. P. *EZ-Metrix* 4.1.0.3 (1 Dec 2014). Web.

Jones, C. *A Short History of Lines of Code (LOC) Metrics.* 10 May 2008. Web.

Park, R. E. *Software Size Measurement: A Framework for Counting Source Statements*. Pittsburgh: Software Engineering Institute - Carnegie Mellon University, 1992. Print.

Putnam, Lawrence. H. *Five Core Metrics: The Intelligence Behind Successful Software Management*. New York: Dorset House Publishing, 2003. Print.

Putnam, Lawrence H., and Ware Myers. *Measures for Excellence: Reliable Software On Time, Within Budget*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1992. Print.

Productivity versus Size and Staff: A Paradox Explained

Paul Below

*An article based on this research appeared in the January 26, 2015, edition of the **BrightHub PM** online journal*, and is reprinted here with permission.

Large projects have higher productivity. And large projects have higher staff. But higher staff results in lower productivity. How can this be? We need to examine all three variables at once and use transformation to clearly see that all three statements can be, and are, true.

Size and Effort

In software development, larger software applications take more effort to develop than smaller ones. The bigger the software size, the more effort. That seems reasonable, and is what we would expect. What is not so obvious is that the relationships between the core metrics are all exponential. For example, the relationship between size and effort is logarithmic.

This relationship causes some surprises. For example, software productivity (size produced per unit of effort) rises as software size rises. QSM's data (>10,000 software projects) definitely shows an upward trend in productivity as application size increases. This is true whether we use measures like QSM's PI (Productivity Index) or ratio-based productivity measures (e.g., SLOC or function point per person-month of effort).

I took another look at productivity data, and addressed the follow-up question, "Does productivity (measured as SLOC/PM) always increase with system size, or could the size-productivity relationship actually behave differently in certain regions of the size spectrum?" To answer this, I used standardized residuals to evaluate the size/productivity regression trend.

Simply put, residuals measure the difference between predicted values (the value of the regression trend at a particular size) and actual metric values. If the regression line provides a poor "fit" in certain size regimes, the residual values will reflect the gap between the values predicted by the trend and actual productivity values for that size regime.

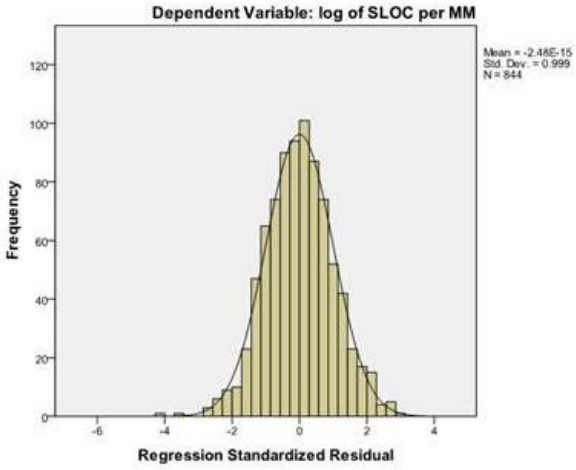


Figure 2.1. Regression histogram.

As can be seen in Figure 2.1, the residuals form an almost perfect normal distribution. This implies that there was no unexplained skew in the data and no systemic variation not explained by the prediction. .

Productivity and Staff Paradox

Effort, productivity and staff size all tend to be higher on larger software size projects. This can be seen in Figure 2.2, which uses over 4,000 projects completed between 2001 and 2011.

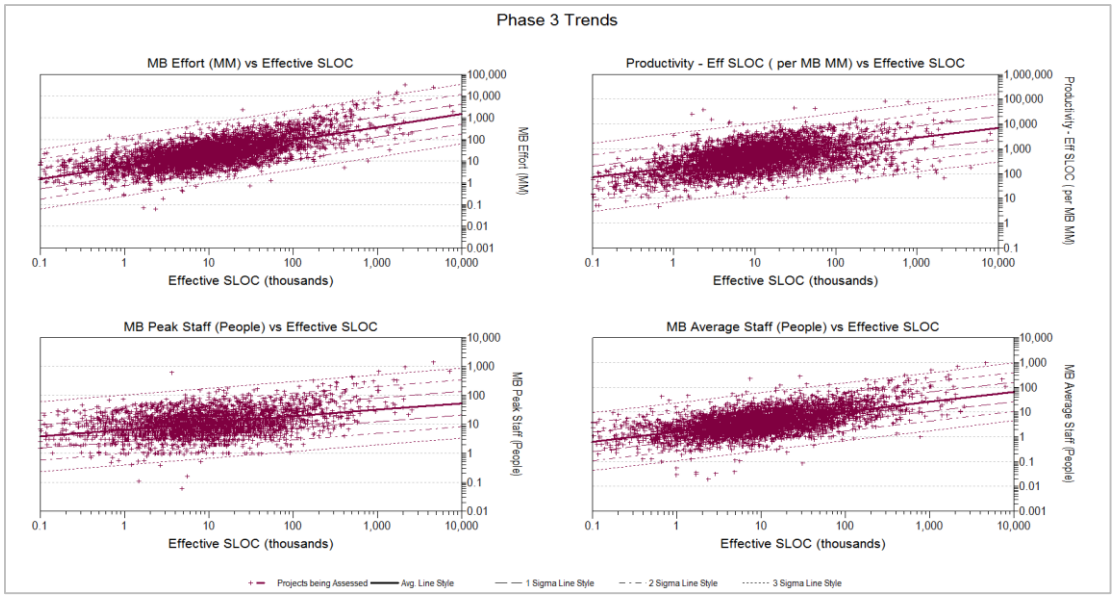


Figure 2.2. Scatter plot of effort, productivity, and staff size against size.

Previous research (e.g., see Armel in "Works Cited") has shown that large team sizes (higher staff) results in lower productivity. Large projects have higher productivity, and large projects have higher staff. But higher staff results in lower productivity. How can this be?

To understand the underlying relationships, we need a way to visually examine three variables at once: size, productivity, and staff. Clustered boxplots are one way to do so, and they provide a nice visual of these trends. A box in a boxplot represents the interquartile range of the data. The bottom of the box is at the first quartile (25th percentile), the dark line inside the box is the median (50th percentile), and the top of the box is the third quartile (75th percentile). The "whiskers" extending out from the box represent the range of the values. Individual outliers (if any) show up as circles, and extreme values are asterisks.

To create the following plot (Figure 2.3), the projects were first divided into quartiles for size and also in quartiles for peak staff. Quartile 1 has the smallest 25% of projects and Quartile 4 has the largest 25%. Productivity on the vertical axis is expressed on a log scale to further improve the readability. In this plot, you can see that productivity decreases as peak staff increases within each quartile of size. To see this, pick any of the size quartiles, and compare the position of the four adjacent boxes.

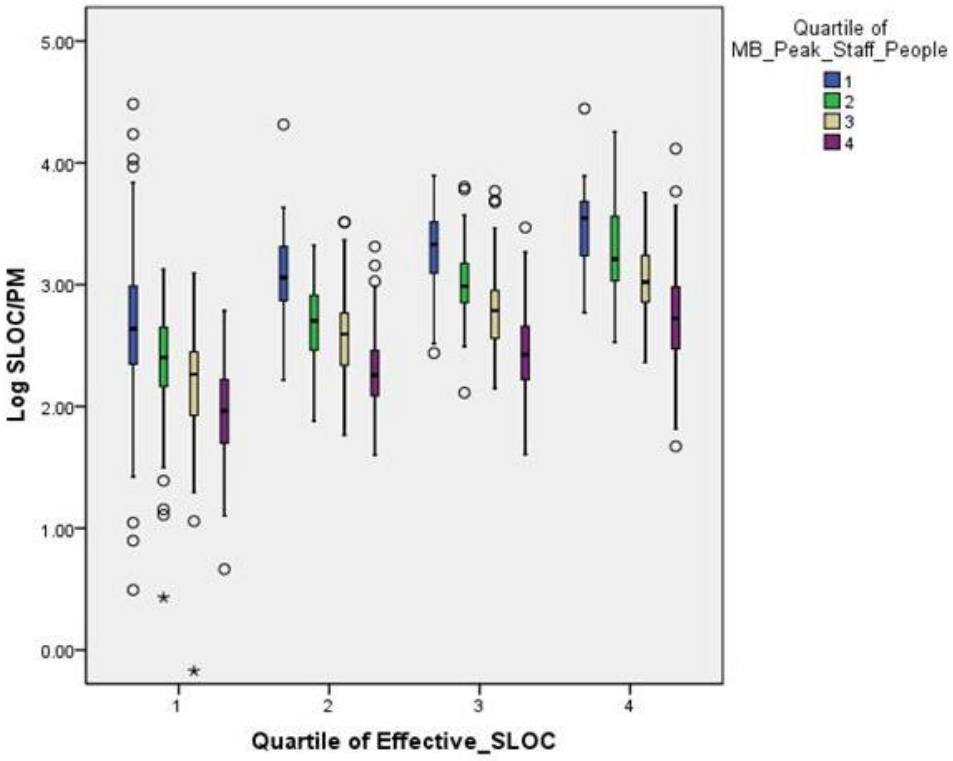


Figure 2.3. Box plot of productivity against size regime.

In the next graph (Figure 2.4), this has been done, with an oval drawn around the second quartile of size. Productivity drops as staff increases, for any given size range.

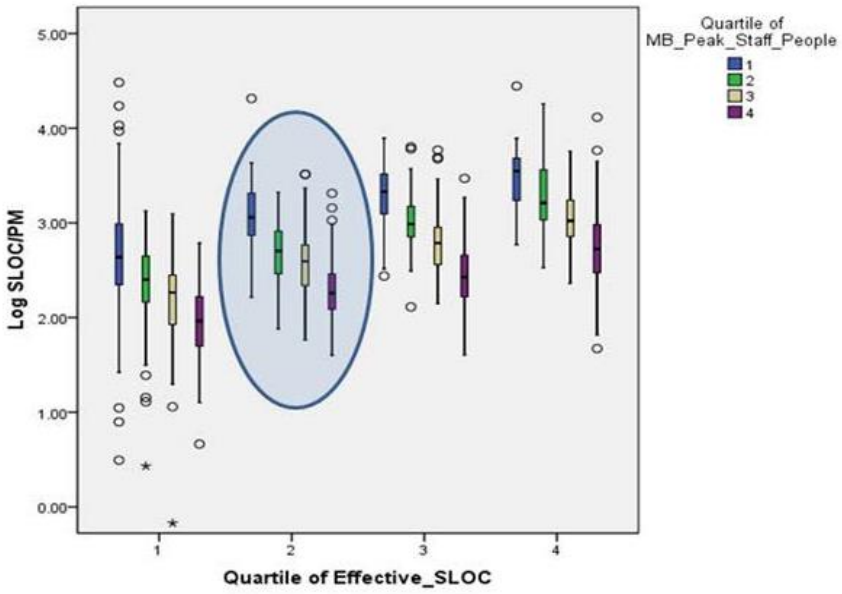


Figure 2.4. Box plot highlighting the drop in productivity as staff increases.

Next, we can see that within each quartile of peak staff, productivity increases as size increases. Pick any color of box, and compare the position of the four boxes with the same color (one from each quartile of size). For a given staff size, productivity is higher on larger projects. In Figure 2.5, the largest staff sizes are identified with arrows.

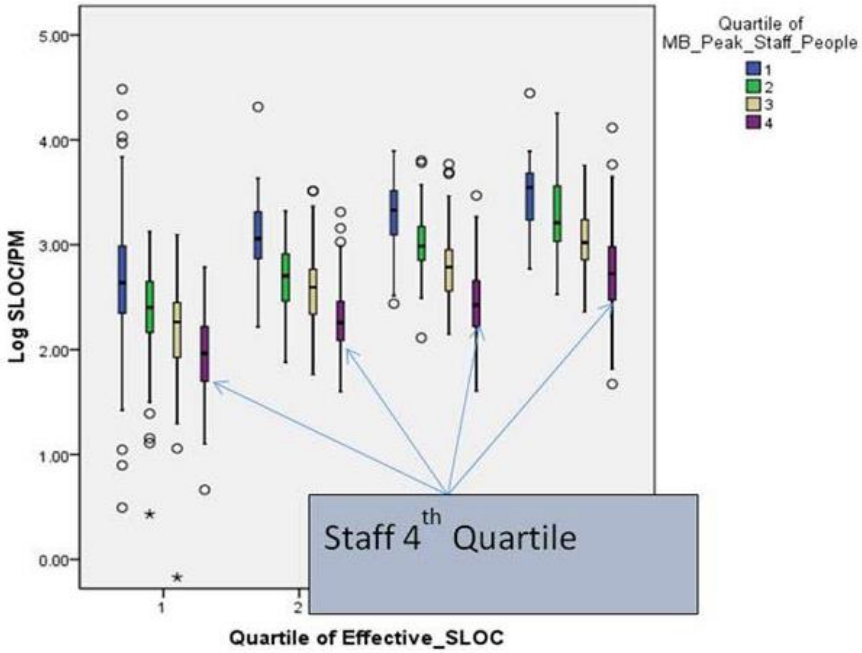


Figure 2.5. Box plot highlighting highest productivity within each quartile.

Simple productivity is higher on larger software development projects. Smaller team sizes tend to have higher productivity. With this data set, we have shown that these two statements are not mutually exclusive. Larger teams become more productive as project size increases, but productivity increases *even further* as team size decreases.

Glossary

MB: Main build, which includes design, code, and test. In SLIM®, this is also known as Phase 3.

MM: Man-month. One MM is one person working full-time for one month.

N: Sample size, or the number of projects used in the analysis

PM: Programmer-month or person-month (similar to MM)

SLOC: Lines of source code (source lines of code)

Works Cited

Armel, Kate. "Small Teams Deliver Lower Cost, Higher Quality." *QSM Software Almanac: Application Development Series, 2014 Research Edition*. McLean, VA: QSM, Inc., 2014, 73-75. Print and Web.

Below, Paul. "Maximizing Value: Understanding Metrics Paradoxes through Use of Transformation." *The IFPUG Guide to IT and Software Measurement: A Comprehensive International Guide*. Boca Raton, FL: CRC Press (Auerbach Publications), 2012: 319-333. Print.

Below, Paul. "Data Mining for Process Improvement." *Crosstalk: The Journal of Defense Software Engineering*. Hill AFB, UT: Software Technology Support Center (Jan/Feb 2011): 10-14. Print and Web.

QSM Performance Benchmark Tables. Web. <http://www.qsm.com/resources/performance-benchmark-tables>

Fact-Based Decisions: Starting with Data and Establishing a Baseline

Douglas T. Putnam and Taylor Putnam-Majarian

*This article originally appeared in the May 19, 2015, edition of the **Project Times** online journal,* and is reprinted here with permission.

If we do not know our history, we are doomed to repeat it. Knowing an organization's past failures and successes is key to smooth project management in the future. Unrealistic expectations are avoidable, colossal barriers to project success. So, how do we manage these expectations from the very beginning of a project before it is too late?

One of the best ways to determine realistic expectations for any project is to examine historical data. Where have you done something similar? How long did it take? Where did the spikes in activity occur? Obtaining these answers will aid in establishing baselines for how the organization can feasibly complete new and future projects. If you have not been collecting this data all along, it may sound like an overwhelming journey on which to embark. However, it does not have to be so.

To get started, there are actually only a few pieces of information that you need to collect:

1. **Size:** A measure of the amount of functionality or value delivered in a system. This can be measured in Source Lines of Code (SLOC), Function Points, User Stories, Business or Technical Requirements, or any other quantifiable artifact of product size.
2. **Time:** The duration required to complete the system (i.e., schedule). This is typically measured in months, weeks, or days.
3. **Effort:** A measure of the resources expended. This is typically measured in person-hours or person-months of work, and is related to staffing as well as cost. The total effort multiplied by the labor rate is typically used to determine the project cost.
4. **Quality:** The reliability of the system is typically measured in the number of errors injected into a system, or the Mean Time to Defect (MTTD), which measures the amount of elapsed time that the system can run between discoveries of new errors.

Once the size, schedule, effort, and quality data have been collected, a fifth metric—the Productivity Index (PI) —can be calculated. The PI is a measure that determines the capability and complexity of a system. It is measured in Index Points ranging from 0.1 to 40, and takes

into account a variety of factors, including personnel, skills and methods, process factors, and reuse.

What is an ideal PI for which to aim in your projects? That depends on the size and type of software that you are building. Typically, Business systems, such as billing systems or online banking portals, have the highest productivities, averaging around 20 Index Points. These systems are relatively simple and straight-forward to build because they have lower reliability requirements than other software applications. Engineering systems and Real Time embedded systems have lower productivities, ranging from 15-18 Index Points and 10-12 Index points respectively (see Figure 3.1). This does not mean that software engineers working on Real Time systems are less productive—the lower PI is caused by the greater complexity of work and the need for more thorough testing compared with Business systems. The size of the application also impacts the productivity rating. Smaller projects, or those with less functionality, will have lower productivities than larger projects. In short, a lot of factors need to be taken into account to assess the productivity of a project, but we will get more into that later.

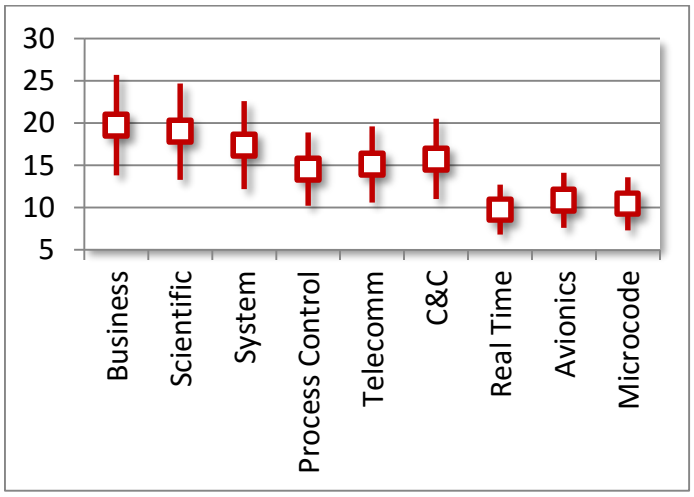


Figure 3.1. Typical PI ranges for different application types.

Together, these five metrics give a complete view of the project, which can be used to assess its performance. In order to establish a true baseline; a broad reaching sample of historic performances is preferred (see Figure 3.2). However, it is better to start with something rather than nothing at all, so begin by creating your baseline with whatever is practical and then build on that as you realize success from newly completed projects.

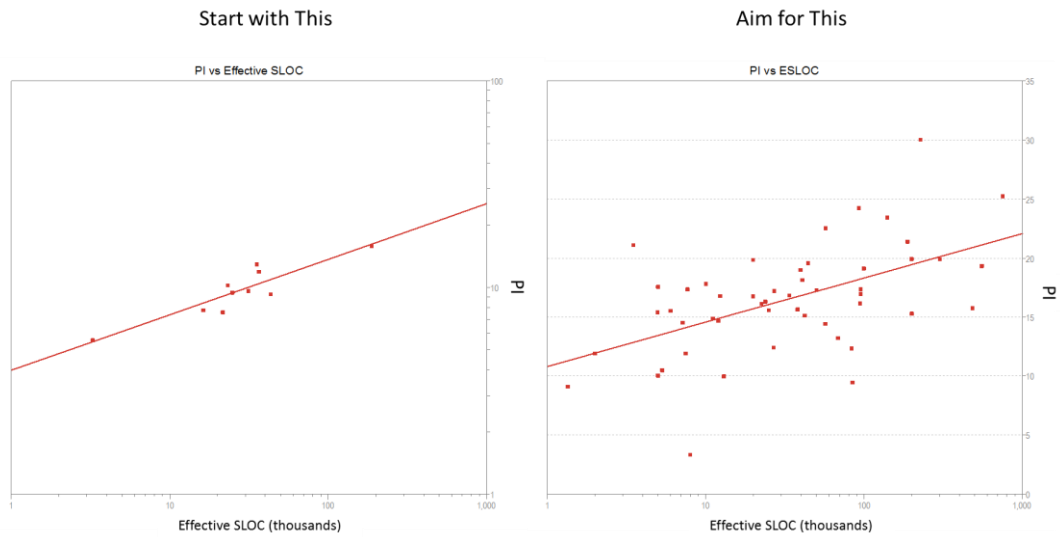


Figure 3.2. Build datasets to include a sample of historical data at a variety of project sizes.

The Internal Baseline

Once a repository of an organization's completed projects has been established, custom trend lines can be calculated to use for creating the baseline. These trend lines serve as a reference point, which can be used for comparing projects within your organization. Where your projects fall relative to the trend line will indicate better or worse performance than the average. This will give insight into your organization's current capabilities. Understanding the baseline for your current development operation can help set reasonable expectations for future projects by showing what has been accomplished in the past. If the desired project parameters push the estimate into uncharted territory, you can use the historical baseline to negotiate for something more reasonable. This baseline can also be used for contract negotiation, evaluating bids and vendor performance, and navigating customer constraints, thus allowing you to achieve your cost reduction and process improvement goals.

We can learn a lot about what we do well and upon what we can improve from looking at our projects relative to the baseline. Examining how far a project deviates from the various trends can help isolate best- or worst-in-class performances. Project outliers can also provide great insight into this. Figure 3.3 displays the company project portfolio against its baseline average productivity. Two of the projects stand out, falling outside two standard deviations, one above the average and one below, respectively. Further examining the factors that influenced these projects (i.e., new technology, tools and methods, personnel, or project complexity) will help shed some light on why these projects performed so well or so poorly. Replicating what went well for best-in-class projects and avoiding what did not for the worst-in-class projects can help improve the performance of future projects.

While the internal baseline provides valuable insight into the practices of an individual organization, we have the market to think about, as well. The internal project standings within an organization may not matter if they are not competitive with the industry. Therefore, it is important to have an external comparison for your project data.

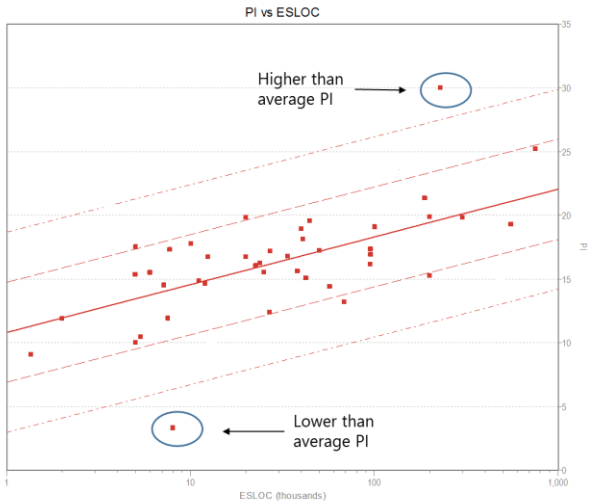


Figure 3.3. Project baseline and outliers.

When using these trend lines to determine performance, it is important to examine the project holistically. Judging a project's performance based upon one metric can be very misleading because of the tradeoffs that occur in software development. For instance, a project may have been viewed favorably because it was delivered quickly. However, looking beyond only the schedule, a project may not have performed well overall.

Figure 3.4 shows a project that was delivered 4.7 months ahead of the industry average, an accomplishment that is often viewed favorably by management because it provides an advantage over market competition. While speed of delivery may be desirable in some circumstances, compressing the schedule unnecessarily can have tradeoffs, such as higher costs and lower quality.

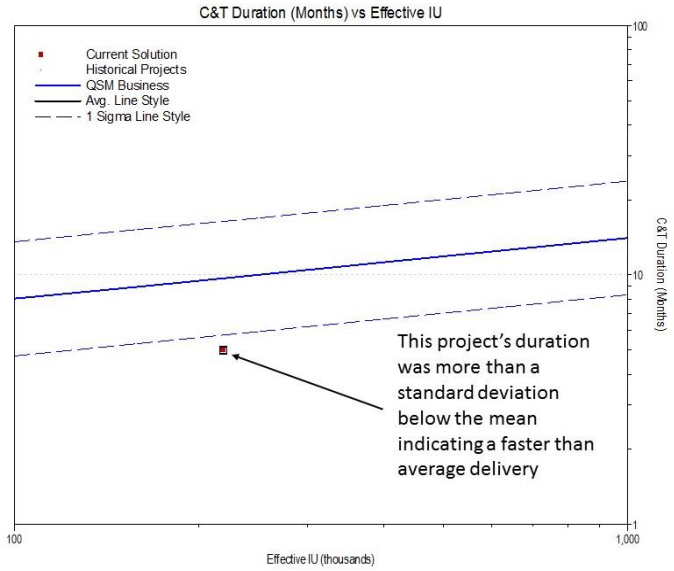


Figure 3.4. Project delivered 4.7 months earlier than average.

Figure 3.5 shows how the project performed in other areas, including productivity, staffing, effort expended, and the number of defects present during testing. These graphs tell a very different story.

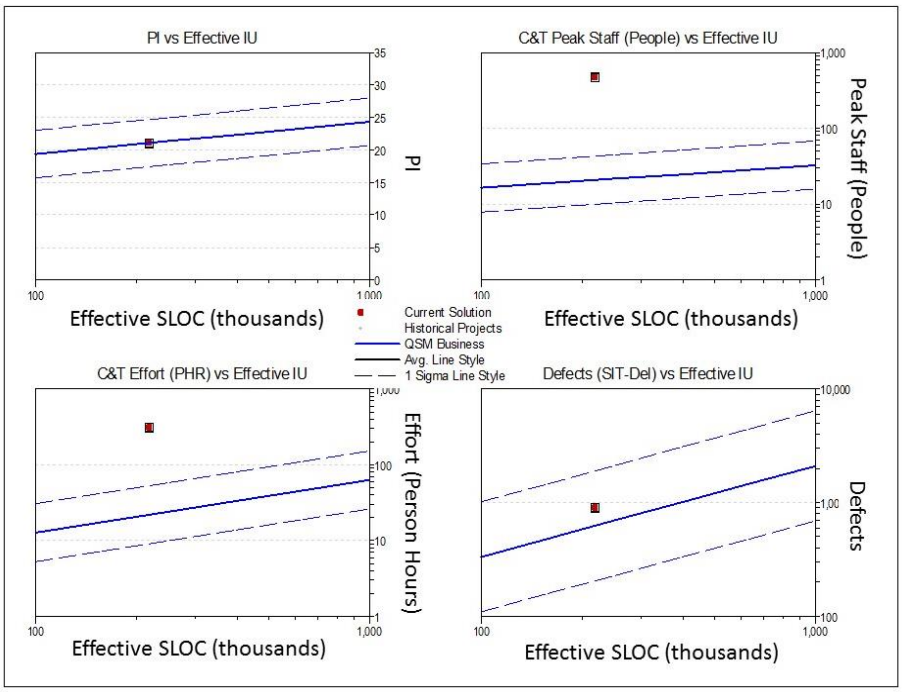


Figure 3.5. Holistic view of project shows that effort, staffing, and defects are higher than industry average.

While the productivity of this project was comparable with the industry average, the peak staffing and effort expended were drastically higher than what is typical for other projects of similar scopes. This directly translates into higher project costs to pay for the additional labor hours.

Additionally, the defects present in the system were considerably higher than what is typical for the industry (see Figure 3.5), and is likely a result of the higher staffing. When more people work on a project, there is a greater chance that miscommunication between team members will lead to errors being injected into the system. Also, utilizing more people further divides the code base, which can result in more defects during integration.

Another way to look at overall historic performance is with Five Star Report views, similar to the one shown in Figure 3.6. These can be used to rate the overall performance of a project or group of projects. For each metric, the project is given a rating of 1-5 stars, with one star being the worst performance and five stars being the best. The Composite Project Star Rating column on the right side gives an overall grade to the project by factoring in the performance of all the individual metric scores. This value helps determine what effects adjusting the staffing or schedule will have on the overall project rating. Here it is easy to see when a project staffs up in order to decrease their schedule. In such a situation, the project would have a duration rating of 4 or 5 stars but with a staffing and effort rating around 1 or 2 stars. The opposite can also occur, thus indicating a project that used fewer than the

average number of staff and lengthened the schedule duration. Ideally, project managers should aim for an average of 3 or more stars for their projects.

Summary

Five Star Report

Data Set: Binary Systems Projects

Deviations Calculated From: QSM All Systems

Projects	PI	MB Duration (Months)	MB Effort (MHR)	MB Average Staff (People)	Defects (SIT-Del)	MTTD 1st Month (Days)	Composite Project Star Rating
ACC Maintenance	★★★★	★★★★	★★	★★	★★★★	★★★★	★★★
Benefits Package Assessment	★★★★	★★★	★★★★★	★★★★	★★★★★	★★★★★	★★★★
Database Entry Handler	★★★	★★★	★★★★	★★★★	★★★★	★★★★	★★★★
Database Support	★★	★★	★★★	★★★	★★★	★★	★★★
File/Font Control System	★★★★	★★★★	★★★	★★	★★		★★
Graphics Engine	★★★★★	★★★★★	★★	★★	★★		★★★
HES/SYN System	★★	★★	★★★	★★★★	★★★★★	★★★★	★★★
Multi-Tasking Overseer	★★★	★★★	★★★★	★★★★	★	★	★★★
PGA	★★★	★★	★★★★	★★★★	★★★★	★★★	★★★
USTA	★★★	★★	★★★★	★★★★★	★★★★★	★★★★★	★★★★
Aggregate	★★★	★★★	★★★	★★★	★★★★	★★★★	★★★

Star Mappings: 1 star (bottom 20%), 2 stars (bottom 20% to 45%), 3 stars (bottom 45% to top 30%), 4 stars (top 30% to top 10%), 5 stars (top 10%)

Establishing a baseline will eliminate much of the up-front ambiguity and will provide detailed recommendations based on quantifiable data. As various organizations strive for improvement, knowing where they stand relative to competition is important. A company with a lower than average productivity will have different goals and implement different process improvement measures than one that is average or better than average. Knowing where you stand as an organization can help you determine the most appropriate measures to take and decide the best method for moving forward. With this data, you will empower the organization to move toward fact-based decision making, thereby maximizing the possibility of having successful project outcomes.

Relative Impact of Modified Code versus New Code

Donald Beckett and Paul Below

This article investigates the relative impact of modified versus new code on software development effort and duration. Note that reused code is outside the scope of this discussion.

Why is this topic important to estimators? It is because the question of how to deal with a mix of new and modified code when estimating is one that comes up frequently, and in our industry, there are different solutions to the situation.

For example, in the COCOMO II model, the determination of ESLOC (equivalent lines of code) is a complex interaction of several factors:

> *…Equation 2 is used to determine an equivalent number of new instructions, equivalent source lines of code (ESLOC). ESLOC is divided by one thousand to derive KESLOC which is used as the COCOMO size parameter. The calculation of ESLOC is based on an intermediate quantity, the Adaptation Adjustment Factor (AAF). The adaptation quantities, DM, CM, IM are used to calculate AAF where:*
>
> - *DM: Percent Design Modified. The percentage of the adapted software's design which is modified in order to adapt it to the new objectives and environment. (This is necessarily a subjective quantity.)*
> - *CM: Percent Code Modified. The percentage of the adapted software's code which is modified in order to adapt it to the new objectives and environment.*
> - *IM: Percent of Integration Required for Modified Software.*
>
> *The percentage of effort required to integrate the adapted software into an overall product and to test the resulting product as compared to the normal amount of integration and test effort for software of comparable size.* [1]

In addition, International Function Point Users Group (IFPUG) rules include procedures for counting new, modified, and deleted function points. Projects that counted Function Points, along with those that collected new, modified, and deleted SLOC, have successfully collected the data that we can analyze.

To credibly investigate the impact of new versus modified code question, we need several things:

- A sufficiently large sample size (refer to the next article in this Almanac on sample size), which we obtained from the QSM database.
- Normalization and transformation of the project metrics, which is a prerequisite to using basic statistical tools such as linear regression, standard deviation, and sample mean. [2]
- Use of median instead of mean as a measure of center, when analyzing data sets with skewed distributions. For example, software metrics datasets almost always have many small projects and a few very large ones.

Effort and Duration Exploratory Analysis

For this analysis, we used a sample of over 4,000 completed software projects from the QSM database. For these projects, Table 4.1 shows medians for development (SLIM® Phase 3) effort and duration, broken out by quartile of size as well as by the percent of modified code (modified code as a percent of new plus modified code).

With a few exceptions, the amount of effort or duration is fairly consistent across the percentage columns. There is noise in the data, of course, and it is not clear whether the largest or smallest values are due to some underlying factor or just random variation. For example, effort in the smallest size quartile is lower when the code is all new. This is also true for the second size quartile, but it is not true for the two largest quartiles. In fact, the largest quartile has the highest effort for all new code.

Modified Code in Smallest Size Quartile						
	No Modified	**<20% modified**	**20 - 40%**	**40 - 60**	**60 - 80**	**80 - 100**
MB effort	5.29	8.16	11.56	9.02	11.66	9.56
MB duration	3.07	4.02	5.00	4.61	5.77	5.60
Modified Code in 2nd Size Quartile						
MB effort	12.00	13.29	18.79	12.68	19.33	18.30
MB duration	4.50	4.81	5.93	5.00	5.94	5.13
Modified Code in 3rd Size Quartile						
MB effort	22.2	20.1	21.6	23.8	40.3	34.93
MB duration	5.5	4.8	5.6	6.0	7.5	6.94
Modified Code in Largest Size Quartile						
MB effort	89.9	70.4	64.4	73.9	44.5	42.5
MB duration	9.0	8.9	9.0	6.7	7.5	6.7

Table 4.1. Development metrics median values.

Effort and Duration Multivariate Regression

Regression was conducted using the log(new SLOC) and the log(modified SLOC) as predictors. The first dependent variable was log of main build effort, and the second was log of main build duration. The log transformation was used to normalize and linearize the metrics. [2]

The form of the regression formulas is:

dependent = a(log new SLOC) + b(log modified SLOC) + constant

If new and modified code had different impacts on either effort or duration, then we would expect to see a statistically significant difference between factors a and b.

For effort, the logs of modified and new SLOC explain 43% of the variation in log effort (blue circle on table 4.2). The coefficients are essentially identical (red circle on Table 4.3) with their 95% confidence levels almost perfectly overlapping. We have no reason to assume that there is a general difference between new and modified code impact on project effort.

Model Summary[b]

Model	R	R Square	Adjusted R Square	Std. Error of the Estimate
1	.654[a]	.428	.427	.41718

a. Predictors: (Constant), Log Modified SLOC, Log New SLOC

b. Dependent Variable: Log MB Effort

Figure 4.2. Model summary.

Coefficients[a]

Model		Unstandardized Coefficients		Standardized Coefficients	t	Sig.	95.0% Confidence Interval for B	
		B	Std. Error	Beta			Lower Bound	Upper Bound
1	(Constant)	-.618	.058		-10.728	.000	-.730	-.505
	Log New SLOC	.291	.014	.411	20.886	.000	.264	.319
	Log Modified SLOC	.289	.014	.392	19.908	.000	.260	.317

a. Dependent Variable: Log MB Effort

Table 4.3. Coefficients.

Figure 4.1 is a histogram of the standardized residuals. Residuals are the difference between each project actual and the regression model's estimate. The histogram is overlain with a normal curve, and since the values are standardized, the horizontal axis is the number of standard deviations the residual is from zero. We see on the graph that the model accounted for almost all of the systematic variation, because the residuals have a good fit to the normal curve.

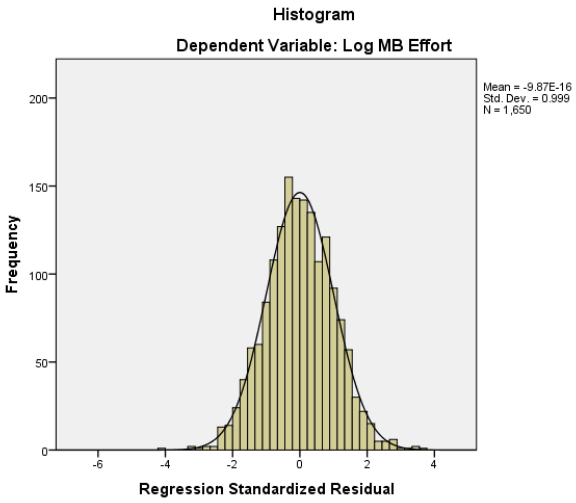


Figure 4.1. Histogram of the standardized residuals.

The relationship between size and duration has more scatter than that for size and effort. There is more variation in duration than effort at a given size. This can be seen in a lower adjusted R square (blue circle in Table 4.4, compared to the blue circle in Table 4.2).

Even so, the relationship is enough to allow us to draw a conclusion. This can be seen in the significance in the coefficients in Table 4.5 (all less than .05). The coefficients are slightly different (0.09 versus 0.1, in red circle) but there is a large overlap in the 95% confidence intervals (green oval in Table 4.5), preventing us from concluding that the impacts are actually different. Since the coefficients are essentially identical, then we should treat them as such by weighting the new and modified code equally.

Model Summary[b]

Model	R	R Square	Adjusted R Square	Std. Error of the Estimate
1	.408[a]	.167	.166	.27106

a. Predictors: (Constant), Log Modified SLOC, Log New SLOC

b. Dependent Variable: LogMBDur

Table 4.4. Model summary.

Coefficients[a]

Model		Unstandardized Coefficients		Standardized Coefficients	t	Sig.	95.0% Confidence Interval for B	
		B	Std. Error	Beta			Lower Bound	Upper Bound
1	(Constant)	.090	.037		2.417	.016	.017	.164
	Log New SLOC	.090	.009	.235	9.885	.000	.072	.107
	Log Modified SLOC	.106	.009	.266	11.207	.000	.087	.124

a. Dependent Variable: LogMBDur

Table 4.5. Coefficients.

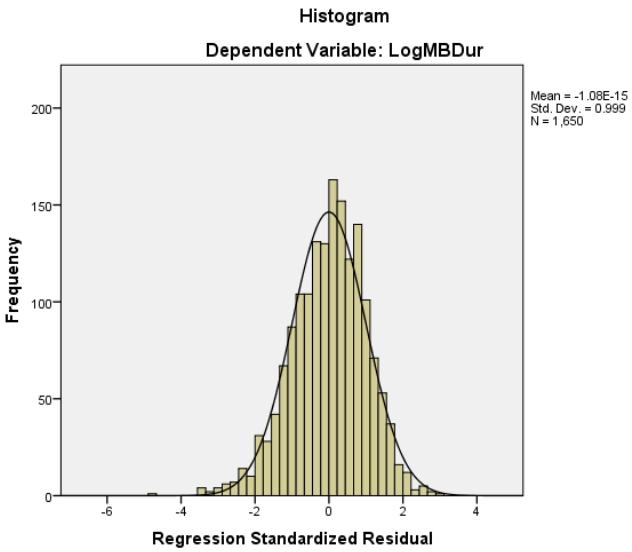


Figure 4.2. Histogram.

To summarize, there is nothing in the data to indicate there exists a systemic reason for (arbitrarily) using different multipliers for new and modified code. General purpose calculations of ESLOC should count them equally. However, what should estimators do about PI? We know that PI is higher for larger projects (productivity is higher for larger projects [3,4,5]). Should we adjust PI based on the percent of code that is being modified?

Productivity Index

Productivity Index (PI) is an exponential relationship of three factors: effort; duration; size. As mentioned previously, productivity increases with project size. Therefore, a regression of Productivity Index (PI) as a relationship of log SLOC would be expected to yield a positive coefficient. A regression using log modified SLOC and log new SLOC results in an adjusted R square of .347 (Table 4.6), and significant coefficients (Table 4.7).

The red circle on Table 4.7 highlights the fact that the coefficient for New SLOC is larger than that for modified code. There is no overlap of the 95% confidence intervals. This means that new code increases productivity faster than modified code. We will next examine quartile tables to make general recommendations for estimators.

Model Summary[b]

Model	R	R Square	Adjusted R Square	Std. Error of the Estimate
1	.589[a]	.347	.347	4.30363

a. Predictors: (Constant), Log Modified SLOC, Log New SLOC

b. Dependent Variable: PI

Table 4.6. Model summary.

Coefficients[a]

Model		Unstandardized Coefficients		Standardized Coefficients	t	Sig.	95.0% Confidence Interval for B	
		B	Std. Error	Beta			Lower Bound	Upper Bound
1	(Constant)	-4.885	.594		-8.226	.000	-6.049	-3.720
	Log New SLOC	3.266	.144	.478	22.697	.000	2.983	3.548
	Log Modified SLOC	1.591	.149	.224	10.644	.000	1.298	1.884

a. Dependent Variable: PI

Table 4.7. Coefficients.

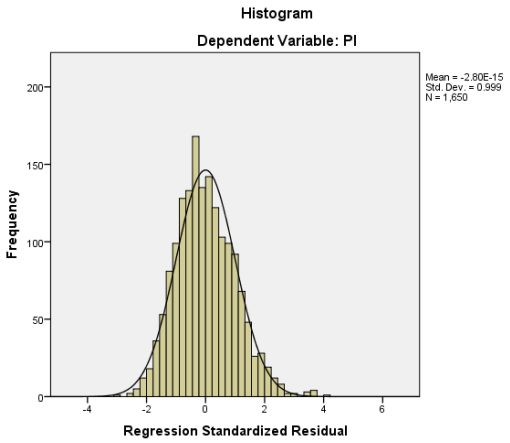


Figure 4.2. Histogram of residuals.

Table 4.8 shows median PI by deciles of percent of code that is modified:

Modified SLOC / (new + modified SLOC)

In general, PI (Productivity Index) decreases as the percent modified increases. This is in line with what we saw in Table 4.7.

% Modified	0	1 - 10	11 - 20	21 - 30	31 - 40	41 - 50	51 - 60	61 - 70	71 - 80	81 - 90	90 - 100
PI	14.3	14.0	12.9	12.5	11.6	13.5	12.1	10.4	10.2	11.0	9.9

Table 4.8. Modified code versus PI.

These observations can yield a useful rule of thumb:

- 10 percent or less modified has a median of around 14.
- 11 percent to 60% has a median in the range of 11.6 to 13.5.
- 61% and higher has a range of 9.9 to 11.

Therefore, when using historical data from new development projects to estimate projects that are modifying significant amount of code, a downward adjustment of approximately 1 to 2 PI is appropriate if the code is 11 to 60% modified, and a downward adjustment of 3 to 4 PI if the code is more than 60% modified. Productivity, measured by PI, decreases as the percent of modified code increases.

Summary

For most uses of the size metric in software development, especially in absence of any more detailed knowledge, new and modified code should be weighted equally in calculating software size.

Estimators should examine their historical data, and modify the PI if it is based off of new development projects and the project being estimated will have a significant (>10%) amount of modified code.

Works Cited

1. COCOMO II Model Definition Manual, version 1.4, USC, p 11.

2. Below, Paul. "Maximizing Value: Understanding Metrics Paradoxes through Use of Transformation." *The IFPUG Guide to IT and Software Measurement: A Comprehensive International Guide*. Boca Raton, FL: CRC Press (Auerbach Publications), 2012: 319-333. Print.

3. Beckett, Don. "An Analysis of Function Point Trends." *QSM Software Almanac 2014 Research Edition*. McLean, VA: QSM, Inc., 2014: 61. Print and Web.

4. "Performance Benchmark Tables." *QSM Software Almanac 2014 Research Edition*, McLean, VA: QSM, Inc., 2014: 191-193. Print and Web.

5. Below 319-333.

Sample Sizes and Trend Line Creation

Paul Below

*An article based on this research has been accepted for future publication in **Crosstalk, the Journal of Defense Software Engineering,** and is reprinted here with permission.*

What is a minimum sample size? The answer depends on the use of the sample. One of the most frequently asked questions of any statistical analyst is "how many do I need?" or "Is this enough?" The answer involves looking at the difference that would be considered important, as well as the variability in the data set.

To create metrics benchmarks or custom SLIM® trend lines for measures such as productivity, duration, or quality, effective grouping is important. We, therefore, need to determine the difference we consider to be important and examine the dataset to compute variation for candidate subgroups (strata).

Sampling

A sample is drawn from a "frame," which is part of a population [1]. The concept of a frame is helpful in creating a sampling plan or dealing with data that has already been collected. This is because we never sample directly from a population.

For example, let us say we want to sample from the people living in a particular country. Perhaps we would use telephone numbers to identify people. Our frame would then be phone numbers, not people. Many people have more than one phone number, while some people have none.

To identify a frame, follow a process. Let us say we are counting the number of people living in a household. Do we count Uncle Charley? He is going back to Cleveland soon (at least we hope he is!).

As W. E. Deming often said, there is no true number of anything. There is a number you will get if you follow a measurement and data collection process. Change the process and you will get a different number. It is important to define and understand the frame.

For a rule of thumb, when determining the mean and a confidence interval of a frame, strive for a sample size of at least 12 [2]. This is because, although confidence intervals shrink with an increase in sample size (at least to a point), the intervals shrink rapidly at first and then less

rapidly as the sample size increases. Somewhere in the range of 10 to 15 is where the curve flattens and additional sample items become less beneficial.

Stratified Sampling

In stratified sampling, the items in the frame are first divided into subgroups, or strata. This ensures proportionate representation of various groups in the sample. For example, a sample of counties in the U.S. would likely benefit from first grouping the counties into rural and urban, and sampling from each to make sure that each is accounted for since they are so very different. Likewise, sampling a large software application could group those items known to be especially large or especially difficult into a separate stratum.

This stratified sampling is done on factors that are known to be important and known to vary widely. The result can be a sampling plan that calls for a smaller sample than would be required if the frame was not first subdivided.

W Edwards Deming termed the major types of analysis as enumerative and analytical. In the enumerative problem, the question asked is of the nature "how many are there?" or "what kind are they?" Enumerative questions can be answered by a census of the population frame and by use of traditional statistical methods (mean, median, quartiles, standard deviation, bar graphs, etc.)

On the other hand, analytic problems answer questions like "why?" and "how many will there be?" These questions usually require a multistep process to answer.

Deming pointed out that, for stratification, these two types often require different approaches [3]. For the enumerative problem, stratification is not usually needed. In contrast, an analytic problem may benefit greatly from stratification. One example would be in investigating causes of software defects, in which case we may restrict the analysis to one type of application (such as business or engineering). For more on this important distinction, refer to "The Philosophy of Analysis" later in this Almanac.

Tests of Significance

The examples in this article make use of an independent sample T-Test to identify differences. This technique is described in any general statistics reference and is included in any statistics tool. The main reasons for using it here are that, unlike the Z-Test, it works with the smaller sample sizes we are frequently called upon to use, as well as in situations where the standard deviation of the population or frame is not known (which is essentially in every situation involving software metrics!). It does require data that is normally distributed, which means that for software metrics we will need to transform the data first [4].

A couple of tools are especially useful when reviewing the results of a significance test. One of these is the confidence interval. In the following formulas,

- X-bar is the estimate of the population mean,
- Sigma is the population standard deviation, s the sample standard deviation,

- n is the sample size, larger samples yield smaller confidence intervals
- Z is the standard normal distribution, t is the Student's t distribution

$$\overline{X} \pm Z^* (\sigma / \sqrt{n}) \qquad \overline{X} \pm t^* (s / \sqrt{n})$$

Another handy tool is standard error. It is the conditional standard deviation of a dependent variable given a value of an independent variable. At a glance, the size of the standard error should be small compared to the size of the associated variable. If the standard error is large, then the result is not very meaningful.

Although any statistics tool can compute it, the standard error of a small sample (15 observations or fewer) can be quickly estimated by dividing the range (max value minus min value) by the number of observations in the sample [5].

Here is an example of how the T-Test is used. We have two groups of software development projects, group A (6 projects) and group B (5 projects). These two groups differ in some way that we suspect may impact productivity. For simplicity, we will ignore project size, assuming that the projects are roughly equivalent in size.

For each project, we have computed the PI (Productivity Index). The PI is an exponential measure that incorporates size, duration, and effort. Since the PI exhibits a normal distribution, we will not have to worry about transformation [6].

Group A PI: 15; 14.5; 14.1; 13; 12.9; 12

Group B PI: 16; 14.5; 14; 13.9; 12.5

We want to know if these two strata are really different and, if so, whether the sample sizes are big enough to create useful benchmark trends.

Standard error:

- A: Highest PI is 15, lowest is 12. Standard error is approximately (15-12)/6 = 0.5
- B: Highest PI is 16, lowest is 12.5. Standard error is approximately (16-12.5)/5 = 0.7

In other words, a quick look shows the standard errors to be approximately the same.

Mean and Standard Deviation:

- A: 13.6 mean, sample standard deviation 1.1
- B: 14.2 mean, sample standard deviation 1.3

A T-Test reveals this difference of sample means (14.2 – 13.6 = 0.6) is not statistically significant (regardless of whether variances are assumed equal or not), either in P value (significance value is greater than .05 at .43) or confidence interval (-2.2 to 1.0). The confidence interval includes zero, so it is possible there is no difference in means at all) [7].

For use of the project metrics, we should assume that the two samples are from same frame

(since we have no evidence otherwise).

Combining the samples, what is uncertainty of mean?

- Highest PI is 16, lowest is 12. Standard error is approximately (16-12)/11 = 0.36. By the way, the standard error of the sample is 0.35 computed with the detailed formula.
- Mean is 13.85, and the 95% confidence interval on mean is 13.1 to 14.6

Returning to the two questions we posed, first, are these two strata (economically) different? They might be, but we have no evidence that they are, so we can treat them as one sample for now. Deming frequently made this point:

"The difference between two treatments or between two areas or two groups of people will show up as 'significantly different' if the experiment be conducted through a sufficient number of trials, even though the difference be so small that it is of no scientific or economic consequence." [8]

Second, are these sample sizes large enough to create a useful trend line ? This leads to the question of how accurate does an estimate of PI need to be? With the 11 projects we have in hand, we are 95% certain that the true mean is between 13.1 and 14.6. To put that in context, an increase of 1 PI can lead to a reduction in schedule of about 10%, or a decrease (e.g., in SLIM-Metrics®) in effort of about 30%.

The logical next question would be, "what would be the cost of obtaining more projects?" But let us now leave this contrived example and look at an example with real data.

Trend Line Creation

Here are the steps I have used to determine the strata to use for benchmarking an output metric:

1. Normalize or transform the variable, as required, to obtain a normal distribution, and account for variables that are known to be important (such as software size)
2. Create boxplots, one box per candidate stratum (e.g., project type), or create histograms or tables of measures of center and spread
3. Examine boxplots for apparent significant differences in either median (center) or variation (spread)
4. Conduct significance tests on those that appear to be different to verify that the center or spread is significantly different
5. Combine strata that are not demonstrably different
6. Create trend lines for each strata or combined strata that is significantly different from the others

The following is an example using actual data from completed software projects, from a sample of the QSM database [9]. The example follows the above steps and examines application type as a candidate stratum for benchmarking.

As in the previous example, PI will be the output metric used. The first step is to normalize the PI, because we already know that larger projects have higher productivity and that the relationship between size and effort is exponential [10]. To normalize the PI, a regression was created of PI versus log of size.

Standardized residuals were computed for each project, resulting in a number that represents the number of standard deviations an individual project is above or below the regression line. In other words, this is the number of standard deviations better or worse the project was than a typical project of that size.

Since the residuals are standardized, they fit a normal distribution and can be used to compare projects, regardless of the project size. This can be seen in Figure 5.1. The mean of the standardized residuals is extremely close to zero, and the standard deviation is close to one. We can now go to the next step.

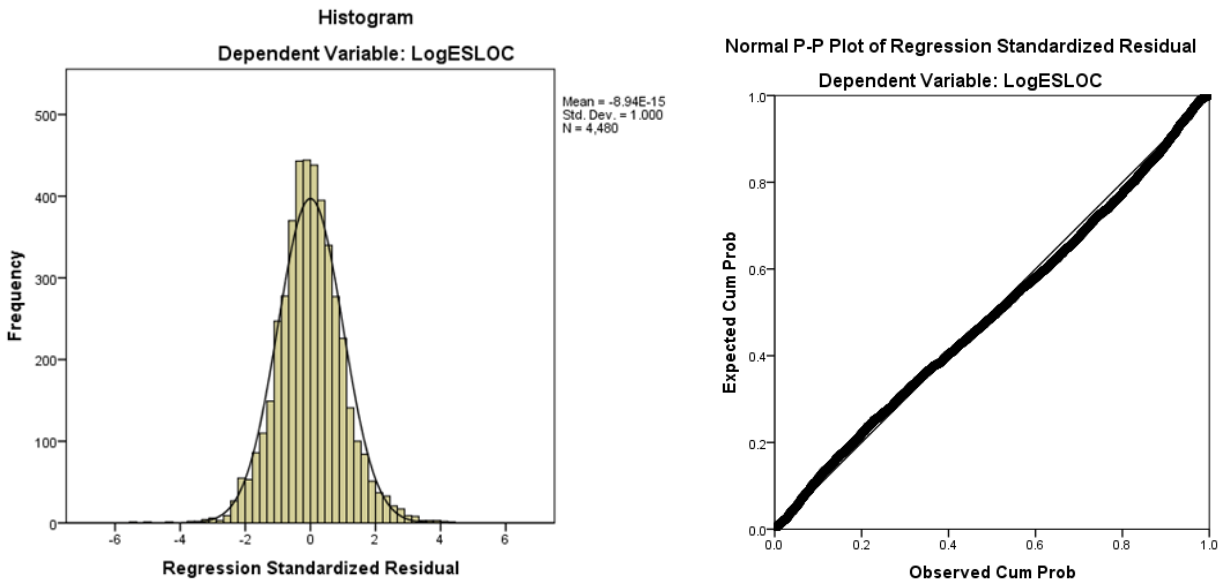


Figure 5.1. Histogram and Plot of standardized residuals.

The second step is to create an exploratory analysis graph, to look for any major differences. I find boxplots to be a very useful tool for this step. In the next graph (Figure 5.2), the vertical scale is the standardized residual of PI. Each box represents the projects from a single

application type. The accompanying table (Table 5.1) lists the sample sizes (number of projects in the frame for each application type).

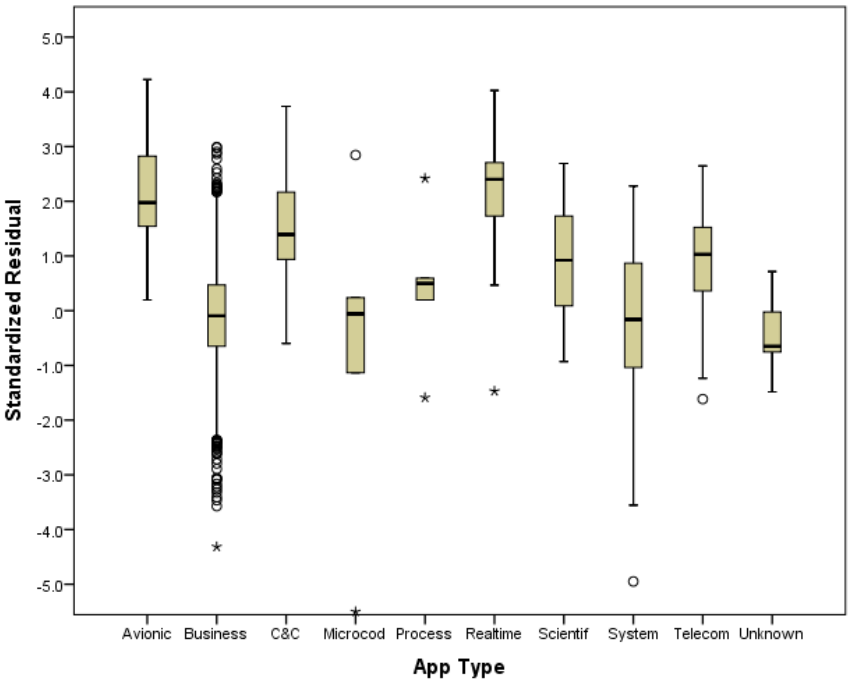


Figure 5.2. Exploratory analysis graph showing standardized residuals of PI.

		Cases	
		Valid	
	App Type		
Standardized Residual	Avionic	45	100.0%
	Business	4000	99.8%
	C&C	130	100.0%
	Microcod	5	100.0%
	Process	5	100.0%
	Realtime	46	100.0%
	Scientif	27	100.0%
	System	159	100.0%
	Telecom	50	100.0%
	Unknown	13	100.0%

Table 5.1. Case processing summary (number of projects in the frame for each application type).

At a glance, we can draw some conclusions (Figure 5.3). For example, the PI distribution for scientific and telecom systems seems very similar, so there is no reason to treat those as different. On the other hand, there may or may not be an important difference between the avionic and C&C (command and control) systems. Further analysis of these two strata will provide more information.

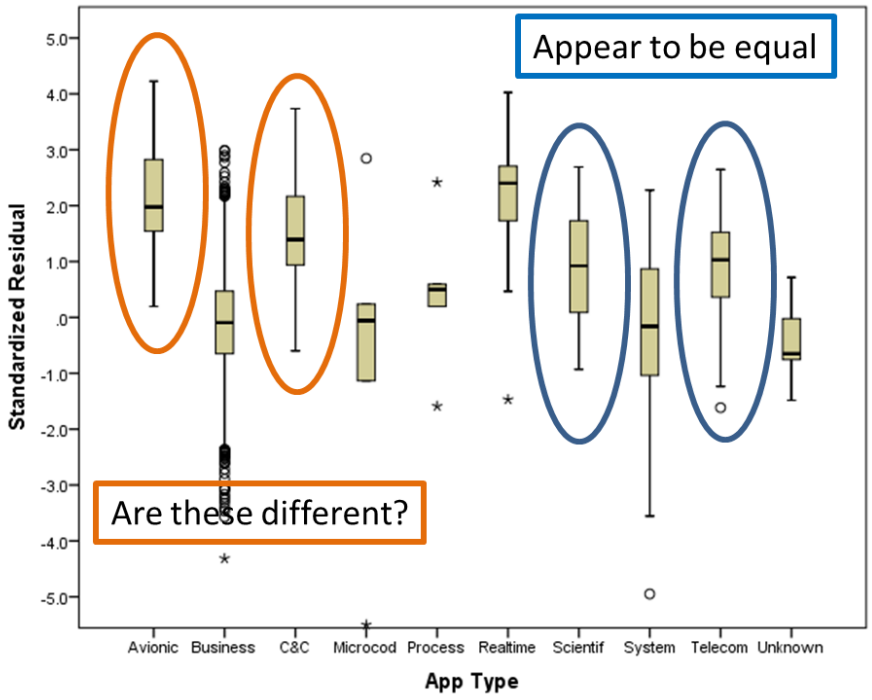


Figure 5.3. Initial conclusions.

Should C&C and avionic be treated separately? The difference in their mean residuals is about 0.6 standard deviations (2.1–1.5), and their sample sizes are 45 and 130 projects (Table 5.2).

Group Statistics

	App Type	N	Mean	Std. Deviation	Std. Error Mean
Standardized Residual	Avionic	45	2.1207876	1.05454789	.15720272
	C&C	130	1.5150850	.84727600	.07431102

Table 5.2. Summary of means and standard deviations.

The first test is of variances (Table 5.3). The null hypothesis is that the variances are equal. We fail to reject the null hypothesis (the significance > 0.05), therefore we have no evidence that the variances are different.

Independent Samples Test

		Levene's Test for Equality of Variances	
		F	Sig.
Standardized Residual	Equal variances assumed	2.410	.122

Table 5.3. Test of variances.

Next is the T-Test (Table 5.4). We are 95% certain that the true difference in their means is between 0.3 and 0.9 standard deviations. We get almost the same result, even if we assume the variances are not equal.

Independent Samples Test

		t-test for Equality of Means						
		t	df	Sig. (2-tailed)	Mean Difference	Std. Error Difference	95% Confidence Interval of the Difference	
							Lower	Upper
Standardized Residual	Equal variances assumed	3.872	173	.000	.60570257	.15644200	.29692183	.91448331
	Equal variances not assumed	3.483	64.758	.001	.60570257	.17388163	.25841236	.95299278

Table 5.4. T-Test example data.

In this example, I would recommend treating avionics and C&C projects as different strata for the purpose of trend line creation. Figure 5.3 is what the trend lines look like. The PI is about equal at the very small size, around 1,000 SLOC. The difference widens as projects grow bigger, eventually reaching a difference of around 5 PI, which is certainly economically important.

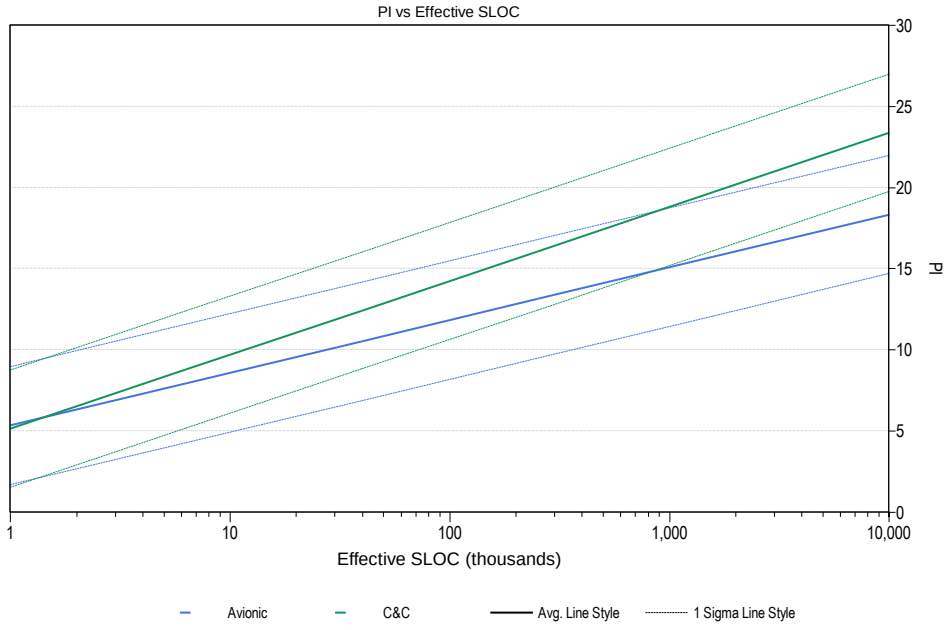


Figure 5.3. Scatter plot of avionics and C&C trend lines.

Summary

The decision of whether or not to group projects is an important one for creating effective benchmarks. Here are some tips for stratification to create metrics benchmarks:

- Know the economic importance of your output variable (what size difference is economically significant?)
- Stratification is very useful in an analytic problem, such as creating a benchmark for

future estimating

- Use the process steps to determine whether to create or combine strata.
  - Boxplots are very useful as a first step
  - Standardized residuals can normalize for known factors (such as software project size) that strongly influence the output variable
  - Tests for variance and t tests are good techniques for determining difference in means

---

Works Cited

1. Deming, W. Edwards. "On Probability as a Basis for Action." *The American Statistician* 29.4 (Nov 1975): 146-152. Print.

2. van Belle, Gerald. *Statistical Rules of Thumb*. Hoboken, NJ: John Wiley & Sons (Wiley-Interscience Publication), 2002: 18. Print.

3. Deming 146-152.

4. Below, Paul. "Maximizing Value: Understanding Metrics Paradoxes through use of Transformation." *The IFPUG Guide to IT and Software Measurement: A Comprehensive International Guide*. Boca Raton, FL: CRC Press (Auerbach Publications), 2012: 319-333.

5. van Belle 26.

6. Putnam, Lawrence H., and Ware Myers. *Measures for Excellence: Reliable Software On Time, Within Budget*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1992. Print.

7. Calculated with IBM SPSS™ version 22.

8. Deming 149.

9. http://www.qsm.com/resources/qsm-database

10. Below 328.

2. MANAGEMENT

"If you focus on results, you will never change. If you focus on change, you will get results."

–Jack Dixon,
American author

"The goal of management is to remove obstacles."

–Paul Orfalea,
American businessman and
Kinko's founder

"Management is, above all, a practice where art, science, and craft meet."

– Henry Mintzberg, Ph.D.,
McGill University

The Problem of Prolific Process: Balancing the Quantity and Quality of Documented Process

Phillip Glen Armour

*This article originally appeared in the May 2015 online edition of **CrossTalk, the Journal of Defense Software Engineering** and is reprinted here with permission.*

Should we have more process quantity, more process detail, more process options (and more rigorous enforcement of process)? Or should we just leave developers to figure out what they need to do as and when they do it? On one side we have the view that if process is good, then more process must be better—such philosophies can generate enormous volumes of paper-based process documents or their electronic equivalent. On the other hand, there are advocates of process so lightweight it hardly exists; with this approach, developers are pretty much left to their own devices to work out what to do.

"Big process" assumes that developers will (a) read the immense amount of process documents before or during development (b) understand what is written (c) figure out how to apply it to their situation and (d) make any necessary process adjustments while staying true to the original intent if not the letter of the documented process. This approach also assumes that all this adherence to pre-defined process will make for higher quality systems or make the process faster and less costly or provide a better basis for system compatibility, extension or maintenance. The advocates and authors of such process rarely seem to concern themselves with any negative effects on the morale, creativity, or sense of achievement the developers might experience when they work this way.

On the other hand, those who espouse very lightweight (if any) process assume that developers will (a) actually remember all the activities needed to build a system (b) consistently apply all these steps (c) apply their innate creativity (now liberated by freedom from oppressive process) to more than compensate for anything they miss. These advocates also assume that the developers will have the requisite experience and skill to do all this.

The Second Law

It is clear that the answer lies somewhere in the middle. Predefining everything we should do to build a system just is not possible. If it were, we could automate the process and we would not need people at all. However, allowing each project and each developer to make it up

how they work each and every time they build something is a recipe for anarchy. We did that 30 years ago and it did not work very well; in fact, the move to big process was fueled in part by the erratic results *laissez-faire* development gave us. And then the move to agile was driven by the reaction to the stifling overhead of big process.

It seems that developing process documentation at just the right level is hard. I described this difficulty in the Second Law of Software Process: We can only define software process at two levels: too vague and too confining [1].

The irony is intentional and it reflects the dilemma we have when writing process:

- Too Confining: if the written process attempts to define all activities under all conditions for all projects building any kind of system, or even a reasonable subset of the same, it becomes very large. Simply because it is very large people will be reluctant to read it. It also becomes difficult to dig through the mountain of documents to find the relevant bit of process just when it is needed. Even more problematic is the constraint that overly large process may enforce. While detailed process is helpful in defining what has occurred before, it cannot explicitly define how to build or test something that is new. In fact, defined process tends to force solutions similar to those that have been built before—specifically the solution scenarios that were used to build the process. It is this inhibiting of the creative process that most lightweight process advocates dislike.
- Too Vague: if the written process consists of high-level guidelines, a loose meta-process framework within which developers operate freely, ignoring it, modifying it and adjusting it as they wish, the process does not add much value. That is, working with the process and without the process is pretty much the same thing. In this case people complain that the process does not provide useful guidance and direction—the process has no "meat."

Balancing Act

Caught between the hard place of too much documented process and the rock of not enough, how can we find the sweet spot? It is a balancing act. But we also need to take a look at what process is, how we get it, what we expect it to do for us, and how we make sure it works. For an example of how balanced process might be built, let us go back to October of 1935.

Failing Fortress

On its second evaluation flight, Boeing's Model 299 (the prototype of what would become the B-17 Flying Fortress heavy bomber) crashed. It was flown by Major Ployer Peter Hill who, as one of the Army Air Corp's most experienced test pilots, had flown and evaluated nearly 60 of the Air Corp's newest aircraft. The crash was caused by the pilot's failure to disengage the B-17's gust locks (devices designed to lock control surfaces while the plane was parked). In dealing with the novel and complex demands of preparing and flying an experimental four-engine bomber, Hill forgot a very important step. He just forgot and it cost him his life.

The solution to this kind of problem was not more experience or more training; Major Hill and his co-pilot had plenty of both. The solution was simple process. It was from this beginning that the pilot checklist was born: a simple list of things to do to ensure the plane was set up safely.

Floating Flight 1549

At 3:27 p.m. on January 15, 2009, US Airways Flight 1549 struck a flock of Canada Geese at 2,800 feet on its climb out from La Guardia Airport in New York City. Immediately after impact, Captain Chesley Sullenberger took the controls while First Officer Jeffery Skiles began working the three-page emergency checklist on how to restart the engines. Four minutes later, Captain Sullenberger landed the unpowered 42-ton aircraft in the Hudson River to the west of 50th Street.

The incredible feat of safely landing a huge airplane on water at around 150 mph received widespread publicity, and the pilots and crew were accorded well-deserved accolades. The use of the emergency checklist was not so well known.

Essential Process

The story of Flight 1549 gives us clues to what constitutes good process and where process has its limits.

- Value Added: given the criticality of the situation, the pilots did not have the latitude to make a mistake in attempting the engine restart. Simply forgetting one step, or working steps in the incorrect order, might have had catastrophic consequences. When stress is high the human brain may not function flawlessly and a simple reminder can help avoid a lot of problems. With their passengers and their own lives at stake, the pilots would not have used any process that did not add immediate value.
- Routine, Well-defined: the restarting of a jet engine is mostly done the same way each time. There is no value to be added by experimenting with novel ways of powering up a jet turbine and, in this situation, there could have been a lot to lose by using an ineffective process. Process works best for things which are precise, repeatable, well-defined and for which there is no point in doing things differently.
- Not for "New:" Captain Sullenberger did not use a checklist to actually land the plane in the water; no such checklist exists. Even if a set of rules for landing a large commercial jetliner in a river next to a major metropolis did exist, the crew would not have had time to reference it and land the plane. When something is "new," there are intrinsic limits to what process can achieve.
- Not if Too Many Specific Conditions: the pilots had to deal with an enormous amount of information on the wind, the behavior of the plane, communicating with the cabin crew, the passengers, and the Air Traffic Control. The combination of these conditions was quite specific to this particular situation. Any "process" would necessarily have to abstract the situation to a set of generalized conditions and the pilots, with only four minutes available to them, would have had to decode these generalizations. Even when there is previous experience available and the situation is not entirely

"new," if there are specific conditions that apply to a particular situation, attempting to apply a pre-defined process will take more time and will be considerably less valuable.

- Succinct: there are many valuable books on flying airplanes in difficult situations. These pilots did not have time to reference and process them. The engine restart checklist contains only and exactly what is needed to restart an airplane engine under emergency situations.
- Process works best when it contains only what is essential.

Novel Projects

To some extent, software projects are always "new." We are always building something we have not built before—otherwise we should simply use what we built last time. That said, much of what we do in the business of software is repetitive. There are many aspects of our work that can and should be done the same way over and over. But there are also things for which previously defined process does not quite apply at the prescriptive level. Perhaps this is where we can define the boundary of process and extemporization.

What We Know, What We Do Not Know

Building systems consists of two kinds of work: the application of what we already know and the discovery of what we do not know (followed, of course, by its application). By "application," I mean the translation of that knowledge into the executable form we call "software." What we already know, we can call "Zero Order Ignorance" —provably correct knowledge (or its inverse, lack of ignorance).

What we do not (yet) know can be divided into several categories: those things we know we do not know, or "First Order Ignorance" (where we have a well-formed question, but do not have the answer) and what we do not know we do not know, or "Second Order Ignorance" (where we do not know enough to form even a good contextual question) [2].

Well-defined prescriptive process can work well for Zero Order Ignorance (0OI) and some of First Order Ignorance (1OI), but it cannot work well for the more complex 1OI and for Second Order Ignorance (2OI). Since software projects contain all of these, the process must flex.

Well-defined

Prescriptive process can be developed and should be used for those aspects of systems development which are boring and repetitive and for which there is no value in experimenting or learning a new way of working. A good example of this might be the check-out/check-in of code from a configuration management system. Once a good process has been defined, there is little point in doing it in any other way. Indeed, a lot of bad things might happen if people tried to circumvent the process. These processes always deal with 0OI or the simpler 1OI (for which the well-defined questions typically have a menu-driven answer selection). Here there is value in process.

Innovative

For those aspects of system development that are novel, the process must be intentionally sparse. Developers must be allowed to explore options free from restrictions that might constrain the solution. The developers are dealing with the remainder of their 1OI and also what they might be quite unaware of—their 2OI. Here there is value in explicit lack of process.

Process Transition

As systems development progresses, there can be a natural transition between processes. For example: when we start testing a system, we do not (and cannot) know exactly what to test since to some extent we are looking for things we do not know are not there [3]. Much of the time we are seeking to expose those things we do not know about the system (like what it does do that it should not do). To design tests and test processes, we cannot be highly prescriptive since we do not know what we are looking for. We might have general indications: that tests should focus on predicate boundaries or cover representatives of all (known) input classes, but we cannot say exactly where we will find defects. This process requires opening up the process to the innovative creativity of the testers.

However, once tests have been created, run, and proved, testing can be transitioned to the usually highly prescriptive process we call "regression testing." Setting up an automated regression process before the knowledge is obtained is ineffective and it might force early testing into a high restrictive process mold that constrains testing to the point where it does not find what it needs to find.

Write, Test, Measure, Reduce

Good process focuses on the value it delivers. This depends on what has to be done: old or new? Repetitive or innovative? Restart the engines or land in the Hudson? Good process does not over-prescribe where that is not valuable. But there are other aspects of process definition that are often missed:

- Test the Process: in many decades of working in software, I have rarely seen documented (i.e., on paper) process actually tested to see if it works. Paper documented process is often written by people who do not actually use the process they are defining. Even more often, these process writers themselves do not use a well-defined, tested, and measured process—which is a little ironic. Commercial pilot checklists are written by a team of pilots, aircraft primes, engine manufacturers, and the FAA. They are written by people who use the process. Once the checklists are created, they are tested in simulators and in the field to ensure they provide the value that is essential to keeping people safe.
- Measure the Process: software process is rarely measured to find out if it does, indeed, reduce defects, speed up the process, improve the lot of maintenance staff, or any of the other attributes used as rationale for writing, using, and enforcing the process.
- Reduce: a further step is necessary and that is to reduce the process. As pilot checklists are tested and the effectiveness measured, much effort goes into making them more concise, more pertinent, more valuable, and smaller.

Prolific Process

This intentional and careful reduction of process does not occur in software development—quite the opposite. Once documented process is created, it tends to grow and grow as it attempts to deal with more and more different conditions, to identify more and more different situations, and to cover wider ranges of application.

The documented process gets bigger and bigger, more and more complex, requiring more and more effort to read, to understand, and to apply. In doing so, it becomes more and more unwieldy and less and less valuable and, thus, less likely to be used at all.

Projects do not crash as spectacularly as the B-17 prototype. But they do crash. To bring them in to a safe landing, we need process that truly supports the business we are in: both the boring repetitive parts and the interesting innovative aspects of what we have to do. The process for each of these aspects should be designed for and support the true nature of the work. Such process needs to be more focused and more concise, and we should test it and measure it in operation to ensure it is really delivering value.

And we should make it smaller.

Works Cited

1. Armour, Phillip G. *The Laws of Software Process*. Boca Raton, FL: CRC Press (Auerbach Publications), 2004: 13. Print.

2. Armour 8.

3. Armour, Phillip G. "The Unconscious Art of Software Testing." *Communications of the ACM* 48.1 (Jan 2005). Print.

Philosophy of Analysis

Paul Below

Metrics analysis, using a variety of tools, is undertaken by software teams. The audiences for the analysis and reporting are as varied as the metrics themselves. The distinction of metrics types is key to organizing business discussions and metrics services so that they are aligned to stakeholders' needs. Appropriate and effective use of metrics analysis and reports will improve decision making.

The purpose of metrics (as well as statistics) is to take action. All types of analysis and reporting have the same high-level goal: to provide information to people who will act upon that information and receive a benefit in doing so.

Metrics offer a means to describe an activity in a quantitative form. There is as much truth as well as humor in this situation.

> "It ain't so much the things we don't know that get us in trouble. It's the things we know that ain't so." -- Artemus Ward, 19th Century American Humorist

> "The more you know about what is wrong with a number, the more useful it becomes." -- John Tukey, 20th Century statistician

Analysis Types

I have identified four high-level processes of deriving and communicating knowledge from raw metrics data. It is helpful to consider which of the four is pertinent to any particular request or need for information. This ensures that appropriate decisions are made and prevents, for example, attempts to do causal analysis solely from canned reports. An organization that is not capable of conducting causal and statistical analysis will be unable to pursue opportunities that arise from reporting or, worse yet, will take actions that turn out to be misguided due to drawing conclusions from non-significant variation.

> *You go to your tailor for a suit of clothes and the first thing that he does is make some measurements; you go to your physician because you are ill and the first thing he does is make some measurements. The objects of making measurements in these two cases are different. They typify the two general objects of making measurements. They are (a) to obtain quantitative information, and (b) to obtain a causal explanation of observed phenomena.* (Walter Shewhart, "Father" of statistical process control)

Enumerative versus Analytic

There is one further distinction that is important to understand when defining an analysis process based on types of analysis requests: Enumerative versus Analytic.

This distinction was first made by W.E. Deming, who distinguished between them as follows:

Enumerative. Action will be taken on the material in the frame studied. The action to be taken on the frame depends purely on estimates or complete counts of one or more specific populations of the frame. The aim of a statistical study in an enumerative problem are descriptive. The aim is not to find out why there are so many or so few units in this or that category, merely how many. Example questions to be answered are "how many are there?" or "what kind is most common?" or "how many defects have been fixed?" We can use our basic statistical techniques to answer this type of question, including sampling, measures of center (e.g., mean, median), and measures of spread (e.g., quartiles, standard deviations), and classification, such as Pareto charts.

Analytic. Action will be taken on the process or cause system that produced the frame studied, the aim being to improve practice in the future. Questions to be answered often start with "why?" or "how many will there be?" Interest centers in future product, not in the material studied. Actions taken might be (1) to determine whether to adopt a new process or continue using the old one; (2) to reduce defects in the future; or (3) to adjust schedules if the forecast shows too much risk.

Returning to Shewhart's example, the tailor was making enumerative measures in order to make you new clothes, while the doctor was doing analytic work in order to make a diagnosis and prescribe treatment.

The Four Types of Analysis

Here is a list of the four types of analysis that I have encountered:

1. **Ad hoc**: This type is used to answer specific questions, usually in a short time frame. By definition, the results are intended to be a one-time output. The questions are usually *enumerative* in nature (e.g., "How many are there?" or "What was the outcome? "). Often, these questions are in support of a sales engagement due to their tight deadlines.
2. **Reporting**: This type generates a predefined output (graphs, tables) and publishes or disseminates the information to a defined audience, either on demand or on regular schedule. The purpose is usually status-related (e.g., identify problems, trends or changes). Reporting is commonly used to answer enumerative questions, although the report can trigger analytical questions that cannot be answered by the report alone. By definition, requirements are gathered in advance, and the reports are designed so the output is repeatable. Therefore, reporting includes the definition, design, creation, testing, scheduling, distribution, and revision of "canned" reports, pivot tables, online reports with drill down capability, and graphs.
3. **Exploratory Analysis**: This type uses statistics and statistical thinking to investigate questions and reach conclusions. The questions are usually *analytical* in nature (e.g., "Why did this happen?" or "How many will there be?"). Analytical questions often require root cause analysis and estimating to answer. The analyst directs each step of the process, drawing upon the expert judgment of process owners and statistical tools to select factors to investigate. Statistical techniques are used to determine

which factors are important. Sampling and surveys are often used to gather additional information.

4. **Data Mining**: There are many definitions of data mining, with variation even among the experts. Most definitions agree that data mining involves discovering patterns of information within large data repositories, especially when there are many factors to consider. Data mining starts with data definition and cleansing, followed by automated knowledge extraction from historical data. Analysis and expert review of the results is required.

The decision tree below (Figure 6.1) shows the process of identification of the appropriate analysis type.

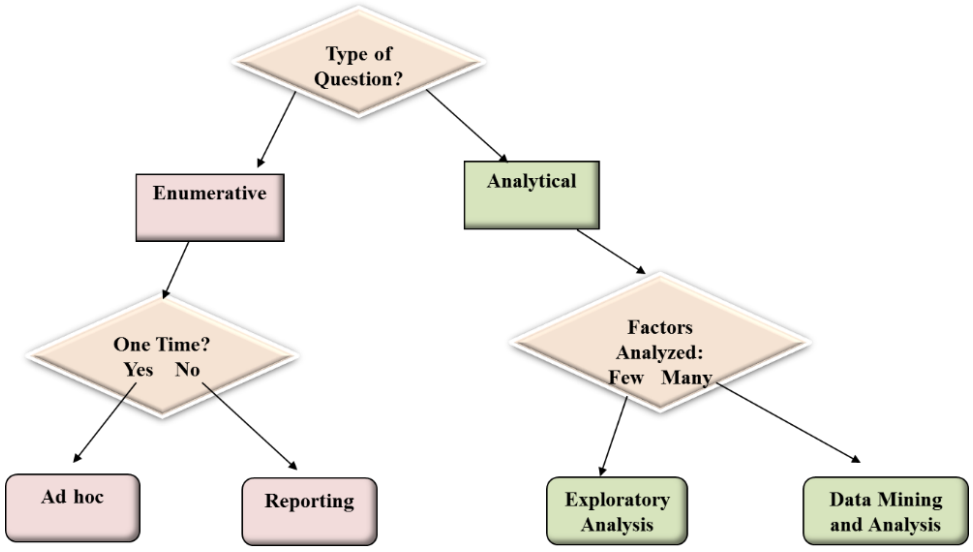


Figure 6.1. Decision tree for identifying analysis type.

Additional tips for producing metrics analysis:

1. Produce clean graphs and tables to display important information. The information may be used by various people for multiple purposes. Explanations should be clear, with good organization for users to find information of interest.
2. Recognize that it takes too long to analyze everything -- interpretations for every graph produced cannot be expected. Results are often superficial due to the limited time allotted for analysis. This is why working to answer specific questions is recommended.
3. "Special analysis" is preferred because it allows dedicated focus on one topic and the ability to study it in depth. This type of analysis can be completed in a reasonable time and results in something useful to the audience.
4. Ongoing feedback from the audience is crucial to obtaining useful results.

Summary

Metrics offer a means to describe an activity in a quantitative form that would allow a knowledgeable person to make rational decisions. However:

- Good statistical inference on bad data is no help.
- Bad statistical analysis, even on the right variable, is malpractice.

Therefore, effective metrics use requires implemented processes for metrics collection, requirements determination for reporting, metrics analysis, and metrics reporting. An organization needs to have capable analysts and organizational processes in place for this entire spectrum.

> "…the fact that the criterion which we happen to use has a fine ancestry of highbrow statistical theorems does not justify its use. Such justification must come from empirical evidence that it works. As the practical engineer might say, the proof of the pudding is in the eating. Let us therefore look for the proof." - Walter Shewhart

---

Works Cited

Below, Paul. "Data Mining for Process Improvement." *Crosstalk: The Journal of Defense Software Engineering*. Hill AFB, UT: Software Technology Support Center (Jan/Feb 2011): 10-15. Print and Web.

Box, George E.P., William G. Hunter, and J. Stuart Hunter. *Statistics for Experimenters: An Introduction to Design, Data Analysis, and Model Building*. Hoboken, NJ: John Wiley & Sons (Wiley-Interscience Publication), 1978. Print.

Deming, W. Edwards. "On Probability as a Basis for Action." *The American Statistician* 29.4 (Nov 1975): 146-152. Print.

Deming, W. Edwards. "On the Distinction Between Enumerative and Analytic Surveys." *Conference Proceedings, Institute of Statistics*. Chapel Hill, NC: University of North Carolina, 21 Jul 1952, 244-255. Print.

Marriott, Nigel. "The Future of Statistical Thinking." *Significance* (Dec 2014): 78-80. Print.

Shewhart, Walter A. *Economic Control of Quality of Manufactured Product*. Milwaukee, WI: American Society for Quality Control, 1980 (reprint). Print.

Tukey, John W. *Exploratory Data Analysis*. Boston: Addison-Wesley Publishing, 1977. Print.

The Most Common Reasons Software Projects Fail

Taylor Putnam-Majarian

*This article originally appeared in the July 13, 2015, edition of the **InfoQ** online journal,* and is reprinted here with permission.

When launching a new software project, best practices suggest enlisting the help of a subject matter expert, who is knowledgeable about software development and can assist in the early stages of project planning. This strategy has proven to greatly improve the project outcome, yet at the end of the project you are staring at a failure. How did this happen?

Project failure can be defined as one or a combination of cost overruns, late deliveries, poor quality, and/or developing a product that does not get used. Regardless of their involvement during the planning stages, more often than not, software developers bear the brunt of the responsibility for such situations; after all, they're the ones who built the application. However, closer examinations of the projects do not always show evidence of incompetence.

When assessing these failed projects, some of these perform "reasonably" when compared with industry trends, yet to the organization they are deemed failures. The reason is that overwhelming majority of the problems can be tied to flawed estimation or poor business decision making at the very outset of the project.

To avoid this, organizations first need to use a standardized set of estimation terms. We often find that individuals and organizations use a number of key terms interchangeably when they each have a unique meaning. Here is a basic list of necessary terms:

Target: A goal, what we would like to do or achieve.

Constraint: Some internal or external limitation on what we are able to do.

Estimate: A technical calculation of what we might be able to do at a defined level of scope, cost, schedule, staff, and probability.

Commitment: A business decision made by selecting one estimate scenario and assigning appropriate resources to meet a target within a set of constraints.

Plan: A set of project tasks and activities that will give us some probability of meeting a commitment at a defined level of scope, budget, schedule, and staff.

Being clear on these definitions ensures projects get off on the right foot with realistic targets and an understanding of the project's constraints. Not doing so can send a project on a death march from the start due to one of more of the following factors.

1. Accepting a forced schedule or mandated completion/milestone dates without substantial data and analysis.

Someone in your organization publicly speculates that the project will be done by a particular date, thus unintentionally commits the team to that deadline. Perhaps your budget cycle dictates that the money allocated to this project must be spent by the end of the year or the next release will not get funded. Maybe the stakeholder wants the project finished by Christmas so that he/she can enjoy the holiday in peace, knowing that the project is complete. Or maybe the stakeholder just really likes round numbers and wants the project to release the first of the month. There are many reasons why a development team would be given an arbitrary project completion deadline. The unfortunate reality is that an overzealous schedule often results in overstaffing the project, the next reason why software projects fail.

2. Adding excessive personnel to achieve unrealistic schedule compression.

How do project managers deal with an overly optimistic schedule? One common response is to staff up the project, often adding way more people than necessary to complete the project. Not only does this drastically increase the cost of a project, but it also decreases the quality. Having more people involved in the project increases opportunities for miscommunication and also makes it more challenging to integrate the different sections of code together. Additionally, Frederick Brooks (1975) would argue that "adding people to a late project only makes it later," which makes sense. The people have to come from somewhere, often from other projects. This puts the other projects further behind schedule and requires that the new staff get caught up by veteran staff, thus decreasing productivity across the board.

3. Failing to account and adjust for requirements growth or change and making necessary adjustments to the schedule and budget forecasts.

"Wouldn't it be great if…?" Those five words can be some of the most dreaded, especially when heard halfway through the construction portion of a project. While there is certainly a time and a place for brainstorming, those activities should take place in Phases 1 and 2. In fact, the purpose of these two phases is to determine whether or not a project is feasible and what features the application should have. If you think of it this way, Phase 2 helps you figure out what you are building, while in Phase 3, you build whatever you decided upon in Phase 2. While some overlap may exist between the two high-level phases, by the time you are in Phase 3, the scope of required functionality for a given product release should be clear.

Requirements growth is a problem if you are adding functionality without allotting more time and budget for development. In essence, you are asking for more work in a shorter amount of time, a technique that we have already found does not work.

Changes to existing requirements can also be a problem depending on the nature and timing of the change. Projects using agile methods can manage changes to requirements details, as long as they occur before construction of a given iteration. However, any changes to requirements of software architecture that cause rework of code already written will almost certainly have an impact to the schedule and budget.

4. Emotional or "intuition-based" stakeholder negotiation that ignores facts and statistics.

At some point or another we have all become attached to a particular project that we have worked on and became emotionally invested in its outcome. For many, this may be the project where your reputation is on the line — the one that is too-big-to-fail — and we often let our emotions get the best of us.

When the pending success or failure of a software project puts an individual's career on the line, it is likely that any related business decisions will be impacted. Stress can cloud one's thinking, especially when the stakes are high. A stakeholder may call for a 12-month schedule in order to impress a customer despite the fact that reports from previous projects of similar size all show a 15-month lifecycle. The stakeholder may dismiss advice from team members, stating that he has "a feeling" that the team will be able to pull through. In such situations, following your gut could be extremely detrimental and potentially lead to project failures.

5. False, but common belief that the proverbial IT silver bullet alone can solve project throughput or process issues.

When all else fails, a common next approach is a change of strategy. "We must be doing it all wrong" and "What are our competitors doing?" are common thoughts at this point. This is when musings of an "IT silver bullet" may start circulating around the office. For instance, someone may suggest that the organization needs to adopt the latest bandwagon development approach. While that might be a great direction in which to take the organization, a decision like this should not be taken lightly. Whichever development methodology your organization decides to use, it is only as good as your implementation. Merely switching development methodologies is not enough to transform your development operation. Regardless of what you decide, in order to make the implementation successful, there needs to be buy-in from all sides, individuals need to be trained, and everyone needs to be held accountable to the same standards. Otherwise, your development strategy will be roughly equivalent to waiting for the stars to align perfectly and hoping for a miracle each time you begin a project. If implemented without thought, this strategy is not only incredibly risky, but also reduces opportunities for team members to give feedback mid-project.

In short, software projects fail for a number of reasons. Take a moment to reflect on whether any of the above reasons may have been the cause of a project failure in your organization. Now what can you do about it? As an executive leader you can do a lot, but it will require

some courage to support your development operation. They need to function within a reasonable set of expectations. Obviously, they will still have to be held accountable, so you will also need historic performance data as evidence of their capability.

You will need to collect data on several different metrics, including the project's schedule duration, the effort expended, the scope of the work performed, the project's overall quality, and the productivity level achieved. In this case, productivity was measured in index points (Productivity Index, or PI), a calculated proprietary QSM unit which ranges from 0.1 to 40. It allows for meaningful comparisons to be made between projects, and accounts for software development factors, including variables such as management influence, development methods, tools, experience levels, and application type complexity.

Once a repository of an organization's completed projects has been established, custom trend lines can be calculated to use for creating the baseline. These trend lines serve as a reference point that can be used for comparing projects within your organization. Where your projects fall relative to the trend line will indicate better or worse performance than the average. This will give insight into your organization's current capabilities.

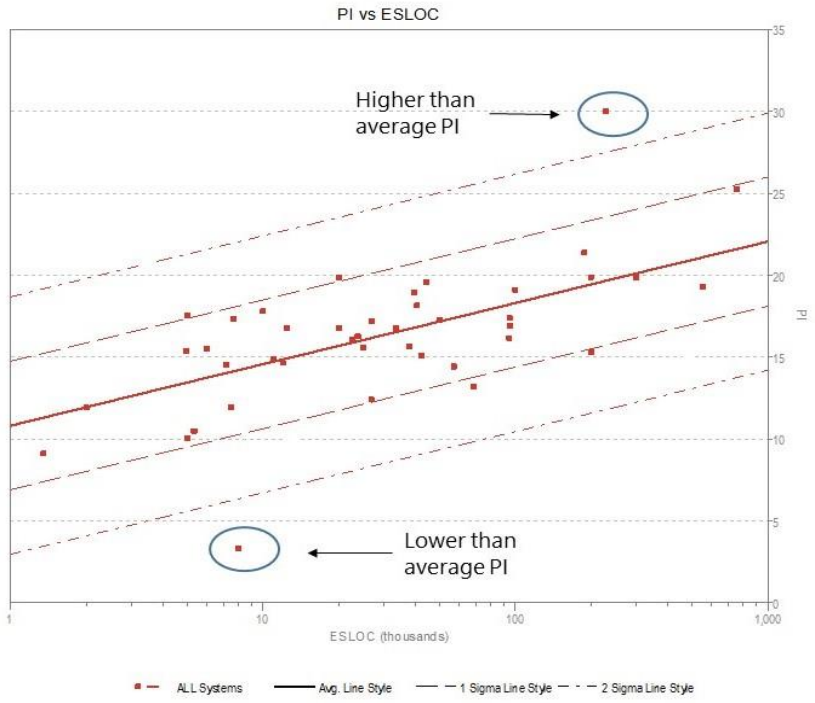


Figure 7.1. Company project portfolio plotted against its baseline productivity trend.

We can learn a lot about what we do well and what we can improve upon from looking at our projects relative to the baseline. Examining how far a project deviates from the various trends can help isolate best- or worst-in-class performances. Project outliers can also provide great insight into this. Figure 7.1 above displays the company project portfolio against its baseline average productivity. Two of the projects stand out since they fall outside two

standard deviations, one above the average and one below respectively. Further examining the factors that influenced these projects (i.e. new technology, tools and methods, personnel, or project complexity) will help shed some light on why these projects performed so well or so poorly. Mimicking what went well for best-in-class projects and avoiding what did not for the worst-in-class projects can help improve the performance of future projects.

Understanding the baseline for your current development operation can help set reasonable expectations for future projects by showing what has been accomplished in the past. If the desired project parameters push the estimate into uncharted territory, you can use the historical baseline to negotiate for something more reasonable. This baseline can also be used for contract negotiation, evaluating bids and vendor performance, and navigating customer constraints, thus allowing you to achieve your cost reduction and process improvement goals.

Doing this will put you in a much more powerful position to negotiate with customers and business stakeholders. The ideal outcome is one in which every entity can be successful. In all likelihood this will require some compromise, but through this process you will be sowing the seeds of success.

Works Cited

Brooks, Frederick. *The Mythical Man-Month: Essays on Software Engineering*. Boston, MA: Addison-Wesley Professional, 1975. Print.

Combining Soft Skills and Hard Tools for Better Software Estimates

Carol Dekkers

This article originally appeared in the 2014 ***Project Management Institute Virtual Library*** *and is reprinted here with permission.*

Why are we so woefully poor at project estimating when software is involved? We have some of the world's best minds at work, yet we cannot seem to get it right. Could it be that estimating goes beyond mere numbers and Gantt charts? This article looks at what might be behind our seeming inability to get good estimates and suggests steps to improve the process.

Anyone in the IT industry knows that software projects are the perfect combination (some might say the perfect storm) of art and science. There is the creative, innovative (right brain) aspect of designing a new system and the engineering (left brain) analysis and construction required to get the project off the conceptual drawing board and into production. The industry (and its stakeholders) houses a cultural melting pot of professionals, boasting artists (musicians, designers, writers) and technical gurus (computer scientists, engineers, mathematicians, accountants, etc.). IT may be a young industry, but it is a uniquely diverse one.

IT-centric jobs comprise four of the top ten featured in the 2013 edition of U.S. News and World Report's "The Top 25 Best Jobs"—#4 Computer Systems Analyst, #6 DBA, #7 Software Developer, #9 Web Developer—with percentage growth rates predicted to be in the double digits. Information technology remains for many an exciting career choice that promises innovative and creative opportunities for technology professionals.

It may not seem surprising that such a young and burgeoning industry is also plagued with burnout and unintentional dysfunctional behavior. After years of rework (over 40% on most software projects), 60+ hour workweeks, weekends, and midnight programming binges, beleaguered IT pros leave the industry. Those who make it to retirement can attest that there is never enough time to do it right the first time, but there is always enough time to go back and do it again.

When you couple talent drain with global studies citing a mere 30% project "success" rate (with success defined as on-time and on-budget delivery), you might wonder where the industry is headed. However, there is good news. If we combine lessons from psychology (specifically, human behavior and communication) with some new science-based estimation tools, we could multiply our present success and create a better future for IT professionals.

Dysfunctional Behaviors in IT

There is a joke circulating on the internet that goes like this:

> *The project manager walks into his boss's office and says, "Here is the bottom line budget needed for the success of the project."*
>
> *The boss says, "What can you do for half the money?" The project manager says, "Fail."*
>
> *The boss says, "When can you get started?" The project manager says, "I think I just did."*

Here are a few major problems that threaten our projects before they even start:

1. Overly optimistic and unrealistic estimates (often based on concepts rather than solid product scope);
2. Management distrust (and flawed assumptions about software development);
3. Lack of respect for expertise on both sides ("father knows best" syndrome); and
4. Negotiation rather than collaboration (a culture of blame and risk transfer/avoidance).

This article focuses on the first one (the latter three are subjects for future posts), and looks at applying aspects of psychology and human behavior for insight.

The Essence of Estimating

As professionals (defined as people with knowledge to do specialized work) throughout a corporation, we have somehow altered the definition of "estimate." Dictionary.com defines the word "estimate" as:

- An approximate judgment or calculation, as of the value, amount, time, size, or weight of something.
- A judgment or opinion, as of the qualities of a person or thing.
- A statement of the approximate charge for work to be done, submitted by a person or business firm ready to undertake the work.

Yet when it comes to IT, the word estimate is often used interchangeably with terms like budget, schedule, and guarantee. An estimate is never a guarantee, but rather a best guess of how big, how much, and how long something should take to build/develop/deliver given the data that we know and is available at the time.

It should be a no-brainer that preliminary estimates based on incomplete or missing data, or based purely on theory (rather than history), will end up being wrong. As such, the notion of project success defined as being on-time and on-budget when compared to preliminary estimates is sheer folly. However, since perception is reality, if we want to increase project success, it is really a matter of getting better estimates.

In other words, the industry definition of project success hinges on the ability of the estimator to predict the right schedule and budget, since projects that go over budget or fall behind are deemed failures. This is where the challenge lies.

How can one estimate a project while it is still as amorphous as a cloud in the sky? It is difficult, but not impossible, if you put limits on your estimate and base it on historical reality—and, I believe, we should always verify our estimates with a second opinion.

Estimating Psychology

Estimators often find themselves tasked to create estimates before the requirements are complete—sometimes budget cycles ahead of time—and must factor in priorities, constraints, and even future regulations. When this "best guess" (most often a single estimate) ends up being wrong in terms of the project being late and over-budget, estimators end up looking bad. Even with the best models (top-down, bottom-up, work breakdown structure (WBS)-based, spiral, agile, etc.), passion for the job, and the best intentions, accounting for all this uncertainty can seem like an uphill battle.

A speaker at the University of Washington commencement address shared ten aphorisms with the graduates. While citing from a graduation speech might seem completely unrelated, a couple of statements struck me as being relevant to this post:

> #7: Define yourself by what you love… Be demonstrative and generous in your praise of those you admire;
>
> #8: Respect those with less power than you. (Dekker's note: I would amend this point to add "or knowledge")

I believe that the majority of software developers, estimators, and project managers love what they do, and do the best job they know how to do, and I believe that there is a lack of respect for such expertise when things (such as estimates) go wrong. The typical reaction in the IT industry when an estimate is wrong is to blame, deflect, deny, denigrate, fire, or demand change from those involved—none of which helps the project. If we followed the two points above, a better response would be to examine what changed (since the time of the estimate) and find ways to reaffirm/adapt/better the estimating assumptions.

Often, consultants are hired to fix the estimating process and bring with them new tools/models to replace current methods. When this happens, it is tantamount to disrespecting the masses, when in fact, the new tools/methods should augment, supplement, and tweak the existing methods to provide a solid second opinion.

Are Bottom-up Estimates Flawed?

Bottom-up estimation occurs when a project is decomposed into a set of project tasks. Time or effort required to complete each task is estimated based on expert judgment as to how long it should take to do, then a project total is derived by summing the individual task estimates. There is nothing wrong with bottom-up estimating *per se*; however, one issue that arises is the assumption of optimistic conditions to complete tasks that preclude scope or historical realities. Estimated hours seldom consider the overall project size or take into account the realities of rework, contingency, or interdependencies. For example, when a task such as "code module x" is estimated, the hours typically represent how long it should take a dedicated programmer to sit down with well-defined requirements and code it perfectly the first time. This seldom happens—interruptions occur, rework happens (40–60% of project effort is rework), requirements are incomplete, upstream tasks delay downstream tasks, related requirements come into play, the specifications are more complex than anticipated, code is copied and pasted (without customization), and a myriad of other issues occur. When we estimate hundreds (or thousands) of these individual tasks in relative isolation, the estimate looks good at face value because it is so detailed. What is missing is a range for the estimates, incorporation of time for rework and contingency, and a way to account for the interdependencies of individual tasks. In addition, because the estimate is so detailed with precise sounding numbers (often including decimal point fragments of hours), anyone looking at the estimate assumes the estimate is almost a guarantee of what the budget and schedule will be.

Combining Soft Skills and Hard Tools to Move Forward

Bottom-up estimating does not work well as the only estimating method—even if we could increase the estimate to address the issues outlined above. I believe we need to find a more robust and empirical basis for project estimates. (Note, this does not mean discarding the bottom-up estimating, it simply means augmenting it using additional techniques.)

The following points can be used to advantage with estimating:

- Historical project totals (effort, duration, costs) are facts. It matters not what the original estimates may have been; the reality is that completed projects are reality.
- History repeats itself and is far more reliable than theoretical models.
- Human behavior is repeatable—we are overly optimistic that we can do better next time even without changing how we do projects. Even when the last 15 projects had a particular issue, we still think that whatever caused it is an anomaly. It is not.
- Projects are seldom smaller and easier than we anticipate. Again, we pride ourselves on being good communicators and project managers who can overcome problems

that arise. Even though we know that projects grow (the rule of thumb is 1.5% per month of the project), we tend to ignore this when performing estimates.

- Rework is not a legitimate task. Even though research on completed projects shows rework figures hovering between 40% and 60%, we cannot include rework as a lump sum task on projects. Because of this, our estimates end up being theoretical (based on idealized task effort or perfect work conditions), rather than practical (based on past performance).
- Project teams want and deserve respect from their peers and customers. When professionals are blamed or disparaged for work they diligently and knowledgeably perform (such as estimating or project work), they lose faith in the process and the project. Apathy, in terms of not contributing to planning or estimating, ends up creating worse outcomes. Respect for people and their knowledge (given their training) is an important part of project success.

Today, there are tools on the market that address and incorporate these factors. These tools allow estimators to use their own completed projects for trend line comparisons and calibrate the estimates to their corporate behaviors and proven ability to deliver software.

Education Equals Knowledge Equals Respect and Improvement

When we boost up our estimating knowledge using proven tools to augment our existing approaches, there is no doubt that our skills, confidence, and success rates improve.

On software projects, it is about time we started improving our project estimates, which in turn leads to higher levels of project success (on time and on budget), which in turn leads to higher respect and better overall corporate morale. Using these tools allows us to scientifically do this with the confidence of knowing that past projects are often better predictors of future performance than theoretical models.

It is time we use psychology (soft skills) and science (robust estimation software) to better our project success rates, and to increase the confidence of the business, industry, and our peers on software projects.

The Shape of the Work When Estimating Agile Releases

Dr. Andy Berner

In a recent webinar on using the SLIM® tools and methods in an agile environment, we showed a template for agile projects included with SLIM-Estimate®. I was asked, "Since agile teams are a fixed size group that stays together throughout the work on the release, why doesn't the agile template use the level load shape?" My answer was the typical short answer to a complex question, "It is complicated and it depends." In this article, I will examine some of those complications and dependencies.

The Team, the Whole Team, or Nothing but the Team?

When using SLIM-Estimate® to estimate the effort required for a release, agile or not, you have to decide, "Whose effort am I interested in?" When describing a team in a scrum project, we usually talk about three major roles: product owner, scrum master, and team member. These people are, in general, full time on the project. If you are only interested in estimating the effort of these full time team members, then you may want to use the "Level Load" shape in SLIM-Estimate®. The total effort of these people is based on the number of people multiplied by the duration of the work on the release.

But you may want to use SLIM-Estimate® to estimate the total labor on the release, and there will almost certainly be people beyond that core team that work on the project. For example, the heart and soul of scrum are the conversations about the user stories—the requirements for the release emerge through these conversations. Although the product owner may be ultimately responsible for the decisions on what stories are included, many agile gurus have pointed out that "it takes a village" to flesh out the details that should go into the release and to review the software as it is being built. Those details emerge through conversations with subject matter experts, process owners, customers or customer surrogates, and other stakeholders. Also, there are often specialists (yes, even in agile projects!) that share their time on multiple projects. For example, there are likely deployment and installation specialists, or training and documentation specialists. These people will not be full-time on the project, but the success of the release depends on their participation. You may want your effort estimate to account for their effort. Judiciously using the various Rayleigh shapes available in SLIM-Estimate® can predict the total effort needed.

Splitting Time between Two Types of Work

SLIM-Estimate® estimates effort based on the Phase Tuning settings, and one of the key settings is which phases are included. What, phases in an agile project? Sounds too much like waterfall! But "phase" within SLIM-Estimate® has a specific meaning that applies to agile projects just as well as other methodologies. Think of "phases" to mean "types of work" rather than "sequential periods of time." In particular, for agile development, we suggest you distinguish, estimate, and track (at least) two types of work: Story Writing and Story Development. Story Writing includes deciding what stories to include and creating the typical agile user story "cards" with brief descriptions. It also includes the work of refining the details, whether that's written down on paper, in electronic form, or, as most agile gurus suggest, perhaps not written down at all but brought out through conversations among all the participants. It's key to agile success that this work is a team effort. The conversations include the "village" of stakeholders and the product owner, of course, but also directly include the developers, testers, and other team members that will create the code. Story writing also includes the prioritization discussions that help choose the development order, and other activities to "groom the backlog."

Story Development translates the stories into working software. This includes coding, testing, documenting, and all other activities to prepare the software for deployment and use. This is done by the "team," together, as we noted, with people in specialist roles that may not be full time.

In the SLIM® tools and methods, you can estimate and track these two types of work separately. There are both theoretical and practical reasons for doing so. The theoretical reasons involve how the Putnam model and the software equation, together with statistical trend data, are used in the SLIM-Estimate® computation engine. The details are beyond the scope of this article. The practical reasons are that the people involved likely track their time differently using different tools. In particular, the stakeholders from the business side are not likely entering time against story records in an agile tracking tool, and even the developers involved may only be entering their coding and testing time in the agile tracking tool. The effort for story writing may be tracked elsewhere, or you may need to approximate it through observation of the work—that's a polite way of saying someone may make an informed guess at the actual effort of the people involved to track it. But such guesses are often made well and provide valuable data both while the project is on-going, and also become the basis for later use to compute trends to use for estimating new projects.

When you look at how the total effort is split among these two types of work or phases, three things stand out:

- Story writing has significant participation from people that are not full-time on the project;
- The people who are full-time split that full-time between the two types of work in an uneven way; and
- The amount of effort expended on each individual type of work is NOT constant throughout the project.

We described this in more detail in one of our QSM webinars, and a link to the recording of those webinars is included at the end of this article.

The upshot is that the shape of these two types of work or phases in SLIM-Estimate® can be best predicted by Rayleigh shapes, rather than level load. Which Rayleigh shapes depend a lot on the specifics of your methods (there's that answer "it depends" again!). We have some suggestions in our agile template based on some common patterns, but you may want to adjust those.

So how does this fit with the idea that "the team stays together full time?" If you combine the Rayleigh curves for the multiple phases, you ALMOST get the level shape. It includes the full-time product owner, scrum master, and team members for the duration of the project, but also includes the varying, part-time participation of specialists and the "village" that's grooming the backlog.

Maximum Utilization versus Maximum Throughput

One of the arguments against moving resources on and off a project in order to maximize resource utilization comes from the Lean Development movement. One of the lessons manufacturers have learned from the Toyota Production System is that keeping machines working at maximum capacity is counter-productive, because you do not have the capacity you need to handle unexpected bumps in demand. Maximizing throughput is more valuable than maximizing utilization. A great reference for this is Don Reinertson's book, "The...Flow." So the argument for keeping the team together is that when a particular sprint for any of a number of reasons requires some extra effort, the team members are there to provide it.

But there's a long jump between making sure there is extra capacity when you need it and keeping all resources active all the time. You can staff above the minimum estimate without necessarily keeping the team constant. Putnam-Norden-Rayleigh curves, used by SLIM-Estimate® to predict effort needed over time, can provide a guide for shaping that excess capacity. While no project will follow the Rayleigh curve exactly, it's still the best predictor of the natural flow and ebb of the project.

Relay Races, the Olympics, and September Baseball

Going a bit out on a limb, let's examine the assumption that agile teams should be a fixed size that stay together throughout the work on the release. Agile gurus propose this staffing pattern for a good reason. Agile is all about team communication, and having everyone together the whole time, often working in the same room, can promote maximum communication. The argument against it is an attempt to maximize resource utilization. If you roll people on an "off project" as they are needed, one person can accomplish work on multiple projects instead of "sitting around" waiting for work on a single project.

Agile methodologists love sports analogies (the term "scrum" itself comes from rugby). One analogy used against the idea of "maximize resource usage" is the relay race. When four runners form a team to run a relay race, they all line up together and they all participate fully from beginning to end of the race. You can see the intensity on their faces as they wait for

their individual leg. It's a team from beginning to end and no runners would ever consider doing something else while their teammates are running the other legs. This sense of intensity, readiness, and communication is the essence of teamwork, and agile teams apply these to their work of coding software.

So what is the analogy of the race in agile software development? Probably the closest is the "sprint," a short (maybe two-week) time-boxed effort where a specific set of stories will be communally developed by the team. It makes a lot of sense to me that keeping the team together during a sprint is a great idea.

But does it immediately follow that this means you should not add team members onto a project or move team members off as the amount of work changes across multiple sprints? Will optimizing the number of team members at any time make the work more efficient or will the disruption caused by changes to the team make it less efficient? We at QSM will not try to tell you what is right or wrong here, and there is research to be done as we often do at QSM by looking to customer data to see the trends. But let me offer a couple of other sports analogies that may be at least as applicable as the relay race.

When the entire Olympic team participates in the opening ceremonies each year, you can see the comradery and the team spirit. But as the games progress, athletes come and go based on when their events are held. While the team for an event stays together for that event like the relay race, the overall team is not constant for the duration of the Olympics. And the athlete that is completely focused on her team during the 4 x 100 relay may "roll onto another project" the next week to run in the 400-meter race.

I am personally not much of a sports guy, but I do like baseball. Through most of the season, the roster has exactly 25 people, but they are not the same people all year. A right-handed relief pitcher may be sent down to the minors when there is a stretch against teams where a left hander would be more useful. It is particularly exciting in September, when the team expands to 40 for the end of the pennant race, as the major and minor league teams trade off players based on the need in minors for playoff games and the need in the majors for extra help down the stretch.

Consistency on a software development team brings advantages with the close teamwork it can engender, but let's be careful about all or nothing solutions to any problem in software development. One size does not fit all.

So What Is an Estimator to Do?

We see that for a variety of reasons, a simple level load shape in SLIM-Estimate® may not always be the most appropriate for estimating agile projects. So what is the right shape to use? You can start with the shapes for the phases in the SLIM® agile story point template that is included when you install SLIM-Estimate®. But as you gain your own organizational history and you see what staffing patterns are most useful for your variations on agile methods, you will want to modify and customize that template. You may need multiple templates for different types of releases. Work on new products or major innovations to existing products

may need different shapes than more steady minor enhancement releases. Mission-critical projects may need more "front loaded" shapes for story writing so that people throughout the organization can give input early enough. There is no simple, single answer that will work for everyone. After all, it is complicated and it depends.

---

3. BEST PRACTICES

"Efficiency is doing things right; effectiveness is doing the right things."

– Peter F. Drucker, Austrian-born American management consultant, educator, and author

"An idea not coupled with action will never get any bigger than the brain cell it occupied."

– Arnold Glasow, American businessman and humorist

"Speed is useful only if you are running in the right direction."

– Joel Barker, American author and futurist

A Vendor Management Best Practice: The Challenges of Procuring Contracted Software Systems

Joseph A. Madden, Jeffrey "J.D." Ottenbreit, and Douglas T. Putnam

The movement toward IT outsourcing began back in the 1990s. It was going to revolutionize the way we procured and built systems, and promised to produce all kinds of cost efficiencies.

Fast forward to 2015.

Today, procurement of contracted software systems is one of the biggest challenges faced by both public- and private-sector organizations across the globe. All too often the procured system is delivered late, over budget, or with significantly less capability than was originally planned.

The most visible recent example of this phenomenon was the procurement and development of Healthcare.gov, the U.S. federal government's healthcare exchange required by the Patient Protection and Affordable Care Act. At the initial delivery, the system was so unstable it was essentially useless, frustrating millions of Americans who wanted to sign up, and it was months before it was stable enough for the system to be usable. In addition to poor performance of the initial software delivery, many of the capabilities required by law were deferred for a year so that they could be actually developed and adequately tested. In short, the initial project release of Healthcare.gov was a disaster!

So the question is why is the Healthcare.gov situation more the norm than the exception? We believe it is the result of a procurement process that often lacks realistic expectations and fails to address the differences in the motivations of purchasers and vendors. Let's take a look at that.

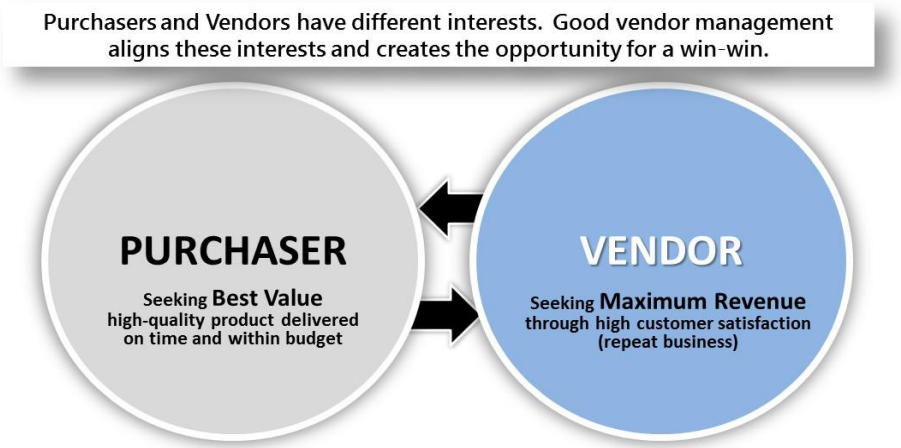


Figure 8.1. Purchasers and Vendors have different motivations. The aquisition process must reconcile this disconnect to achieve a positive outcome.

The purchaser wants to achieve a best value scenario. For them, that is defined by receiving the required software capability that works at an acceptable reliability for the best price and schedule that can be achieved. The supplier, on the other hand, needs to be competitive enough to secure the business, but then they are more concerned with maximizing revenue and profit and hopefully earning repeat business. How does one reconcile these fundamentally different motivations?

Any good vendor management solution needs to recognize and address these fundamental motivations. This inherent friction between purchasers and vendors needs to be taken into account in order to create a winning solution for both partners on any given procurement.

A Software Procurement Vendor Management Methodology

The QSM vendor management solution is broken down into four primary phases (Figure 8.2). The first phase, Pre-Acquisition, is all about doing a credible job of understanding the requirements of the stakeholders and then doing an independent "should cost" estimate that includes an assessment of expected effort, staffing, and schedule duration to meet a target reliability. The independent estimate should explore all of the viable options. If done right, this should lead to reasonable program parameters and expectations that will be specified in the RFP (request for proposal) when it is issued.

The second phase, Request for Proposal, is making sure that all the information that will be required to quantitatively assess each vendor is in the package.

The third phase, Award, is the analytical process of objectively assessing the bidders and scoring their solution.

The fourth phase, Post Award, assesses progress against the contract baseline to ensure the program is on track. If changes in direction are proposed, they need to be understood and quantified in order to adequately evaluate the impact to schedule and cost.

Let's take a look at each of these key areas in a bit more detail.

Figure 8.2. The QSM vendor management solution.

Pre-Acquisition - Phase 1

The preparation work done in the pre-acquisition phase is really important because it sets the stage for a successful program. At QSM, our research shows that the two most common reasons that projects fail are:

1. Unrealistic customer-specified short schedule; and
2. Inadequate requirements definition leading to requirements volatility and growth.

Given that information, the best place to start is with the customers' requirements. To do this correctly, we need to achieve a deep understanding of the business stakeholders' requirements. So why is this important? Fundamentally, the requirements determine the functionality. Once the functionality is understood, then it is possible to do a parametric analysis of cost and schedule. There may be several viable alternatives, and all of them should be assessed in order to determine how well they fit into the customer's desired budget and schedule. The typical ones include:

1. Build a custom solution
2. Enhance a legacy system
3. Purchase a package, configure it, and add customizations, as needed

It is also important that the "should cost" estimate is based on what the "average" vendor could accomplish. In setting the purchaser's expectation, we need to build the cost and schedule case around what has typically been achieved by others in industry for similar types of systems.

This is accomplished by using a good sized sample of historical data similar to the application that is being considered. With that data we are able to do some simple curve fits and determine the average amount of schedule, effort, and staffing for various levels of functional capability. Using this quantitative data, purchasers are then able to use objective criteria to defend their procurement decisions (see Figures 8.2 and 8.3).

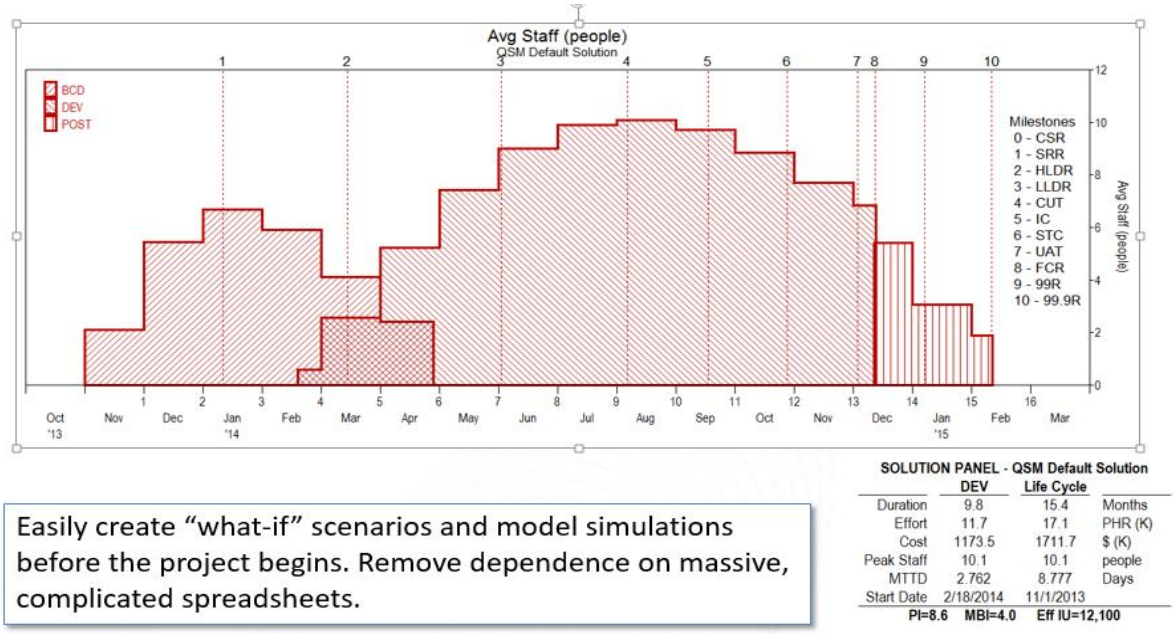


Figure 8.2. Independent parametric estimate (staffing, schedule, effort, cost and reliability).

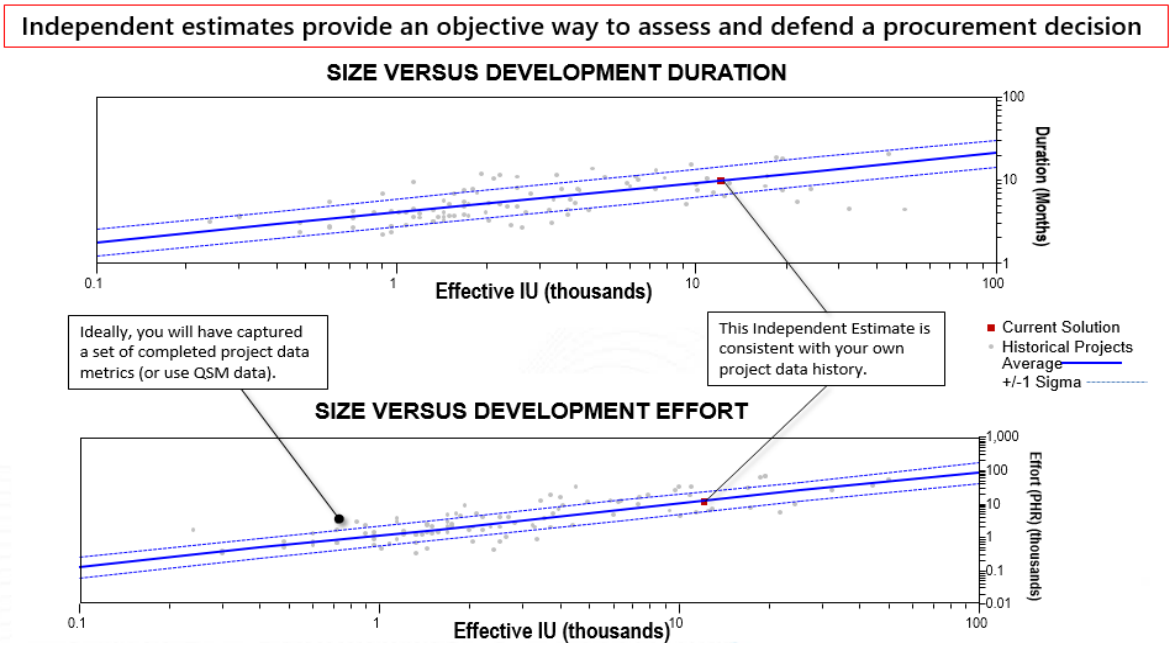


Figure 8.3. Estimate compared to historical data, assuming average capability.

In summary, the benefits of doing the pre-RFP assessment are:

1. You have a good understanding of the business stakeholders' requirements;
2. You are able to estimate all the viable alternatives that vendors may propose as their solution;
3. You are able to ensure that the business stakeholders have realistic expectations for schedule and cost;
4. You are in a position to put out an achievable RFP out for solicitation; and
5. You have an objective baseline from which to defend your procurement decisions.

Request for Proposal - Phase 2

During the preparation of the RFP, there are some key items that ensure one is in a good position to evaluate all the vendors on equal grounds.

First, the RFP must have a clear statement of the scope of required functionality, to include a context diagram that defines the application boundary (i.e., the border between the software being bid on and what is external to the application, to include end users and external interfaces with other systems). This is so important because it is the basis of the estimates that each vendor will be required to provide. Requirements details can be worked out later, but the statement of scope needs to include, at a minimum, a logical data model and the number of required elementary processes. The latter can be expressed in the form of use cases, user stories, or functional requirements.

Do this well and none of the vendors will be able to protest based on fuzzy or ill-defined requirements. Another added benefit to well-defined requirements is during the execution phase, as this will put the purchasing organization in a stronger position to negotiate a fair price for changes in requirements. The requirements change order game is often the way vendors who won the original contract with a "lowball" bid later attempt to recover an equal or greater amount of money than they would have made from a realistic bid.

The second important component of the RFP is to state any project constraints that were derived from the independent estimate. This could include guidance on a desired solution based on a commercial off-the-shelf product. It should also include any constraints on schedule and milestones, effort, staffing, cost, or quality at initial delivery.

Finally, the RFP should outline the data that each vendor will be required to provide as part of their bid. There are three parts of this data package. The first is the data that will quantify their bid. The second is relevant vendor historical project data, if available, in support of the bid. The third is the data that will be required for monthly tracking during contract execution.

Quantification of the Bid. Each bidder should be required to provide their bids for each of the following metrics. For each, it is important to know the total quantity for a given metric and the monthly rates and monthly cumulative total values. As a minimum, the bid metrics package should include the following metrics categories:

- **Size/Functionality.** This is the vendor's size estimate of the required software functionality and their plan for building that functionality over time. It tells you how

your product is being constructed. For example, if you are 80% of the way through the schedule and have consumed 85% of the budget and have only 25% of the product coded, you have a problem. The size metric(s) give you visibility into the construction of the product and enables one to ensure that it is in sync with how the money is being spent. Typical size metrics include source lines of code, function points, agile stories, use cases, and process configurations and RICEF objects for package implementations. It is not so important which size measure is used in the RFP, but there are some that make more sense for certain architectures and product domains. Each vendor should be required to identify how much code will be newly developed, how much will be 100% reused with no changes, and how much will require some form of modification. Clear definitions and counting rules are a must, so spend the time and get this right; you will be glad that you did. This metric provides real transparency in the contract execution phase. No more smoke and mirrors!

- **Schedule and Major Milestones.** This is the vendor's expression of the timeline and when major design and development maturation events will take place. Make sure they start with contract award and end with the conclusion of a software warranty phase. If Operations and Maintenance are part of the bid, then show that as a separate item. Make sure that major milestones that are important to the purchaser are included. This might be start and end of system level testing or start and end of user acceptance testing.
- **Effort and Cost.** This is the total hours and/or cost; the expected monthly invoicing over time. For time and materials contracts, require labor rate by each skill category.
- **Staffing.** Require the vendors to provide a staffing profile over time, even if it is a fixed price contract. This is very important as QSM research has repeatedly shown that overstaffing has an adverse impact on quality and productivity. There is a tendency for vendors to want to staff up quickly in order to maximize cash flow or in an attempt to meet an aggressive schedule. This is rarely the most optimum staffing profile to maximize productivity, cost and quality. By requiring the staffing profile by skill over time, the purchaser is in a position to identify non-optimum staffing profiles.
- **Defects.** These metrics are rarely thought about at bidding time, but become the most critical factor at delivery time (remember Healthcare.gov). It is important to know the predicted defect discovery rate during testing, at delivery, and throughout the warranty period. This is essential in order to get visibility into the rate at which new problems are likely to be encountered. There is a minimum acceptable reliability at the initial delivery. For business systems, it has historically been no more than 20 new defects discovered per month on IT applications, and is about five new defects per month for engineering and real time applications. This data allows us to see if the vendor's reliability growth plan is realistic and consistent with what is required for successfully fielding the application. If possible, ask that the defect data be broken out by severity category. Figure 8.4 shows an example data response expected from a bid package.

Estimate size ant Completion		96,679								
						Defect Arrival Rate				
Months from Start	Ending Month	Milestone	Milestone date	FTE Staffing	Cumulative Code	Priority 1	Priority 2	Priority 3	Priority 4	Priority 5
1	Jan			22						
2	Feb	SRR	2/15/2007	35						
3	Mar			44						
4	Apr	PDR	4/22/2007	45	0					
5	May			52	12,234					
6	Jun			61	24,678					
7	Jul	CDR	7/12/2007	65	37,456					
8	Aug			67						
9	Sep			67	73,456					
10	Oct			67						
11	Nov			69						
12	Dec	CUT	12/7/2007	72	85,436					
13	Jan			69						
14	Feb			75						
15	Mar			68	92,136	4	6	18	12	3
16	Apr			67		2	11	12	16	8
17	May	SIT		65		6	13	9	11	12
18	Jun			65	94,296	12	24	45	25	18
18	Jul			68		8	16	31	14	32
20	Aug			67		7	20	16	12	14
21	Sep			69	96,456	9	14	25	16	5
22	Oct			68		4	12	14	8	4
23	Nov			72	96,563	3	8	12	3	7
24	Dec	TRR		67		1	3	16	4	3
25	Jan			68	96,675	2	4	8	2	2
26	Feb			45	96,679	0	2	5	3	1

Historical Data. Evaluate each vendor on whether or not they are able to provide historical project data. If historical data is available, request that each vendor provide no fewer than three historical projects that have been successfully completed and deployed. They should provide the historical data in the same format as the project they are bidding on that is described in the section above. In addition to the data, require a client reference that can verify the information and provide other observations about their experience working with the vendor. Request that the history be reasonably current, out of the same business enterprise that is bidding, and of the same general application domain and technology. Having this data allows you to compare the historical performance with the bid to ensure that there are no "gross" inconsistencies with the vendor's own history.

Contract Monitoring Data. The RFP should also require that the vendor provide monthly or weekly actual performance data that is consistent with the information that is being required in their bid. If Figure 8.4 represents the bid, then this would simply add an actuals column next to each bid column. This will provide a baseline from which actual contract performance can be highly visible and transparent.

Summary. By providing a clear statement of the required functionality, bidders will have a sound knowledge base from which to create their bid. The bid can be quantified in terms of size, schedule, effort/cost, staffing, and predicted defects. The bids should provide totals, monthly rates, and monthly cumulative totals. If available, historical past performance data on no fewer than three vendor projects should be provided, which will be used to compare to the bid to ensure reasonable consistency.

Award Decision Activities - Phase 3

The RFP has been released and now the bids are in. The first step in the bidder evaluation process is evaluating the historical data of each vendor. There is a reason that there is such an emphasis on historical data: it is both objective and unbiased, and allows the evaluation team to improve the traditional technical evaluation process (Figure 8.5).

Technical Evaluation **without** Quantitative Criteria traditionally poses some issues:

The Myth	The Reality
Technical evaluation accurately assesses vendor's ability to perform ▪ "Excellent technical and management approach" ▪ "Clearly understands our environment"	Technical evaluation assesses the skill of the **proposal writer**, not the development team ▪ Proposal writer probably is not a key person on the delivery team ▪ May not even be an employee of the company
Technical Evaluation Panel (TEP) is always made up of technical individuals - who have the necessary tools and project metrics to perform a robust, objective analysis.	TEP often consists of whoever is available and they don't have the necessary tools and project metrics.
There is a positive correlation between high technical evaluation scores and actual vendor performance.	Very doubtful based on anecdotal data. Should be measured statistically to improve the procurement process.

Recommendation: fully leverage unbiased, quantitative criteria to address some of these issues - making the technical evaluation process much more objective.

Figure 8.5. The dangers of technical evaluations without quantitative data.

This is done by taking the historical past performance data of each vendor and positioning it against a set of benchmark trends. The benchmark trend can be created from the purchaser's database if they have been doing these types of assessments for a while or provided by QSM. The key point is that every vendor is compared to one common set of trends. This is done for each of the core metrics schedule, effort, staffing and predicted defects and a profile is established for each bidder.

Figure 8.6 shows that Vendor 1 consistently uses more effort to produce a product when its performance is compared to the base line trend, and this behavior is consistent across all size regimes. The next step would be to compare the vendor bids to the independent estimate and the base line trend. Here we are looking for when bids are grossly outside the bounds of what has been seen before. It could be too aggressive on schedule and cost or it might be too conservative. Neither is a good situation. This type of analysis is performed on each of the core metrics and the vendor profile is updated.

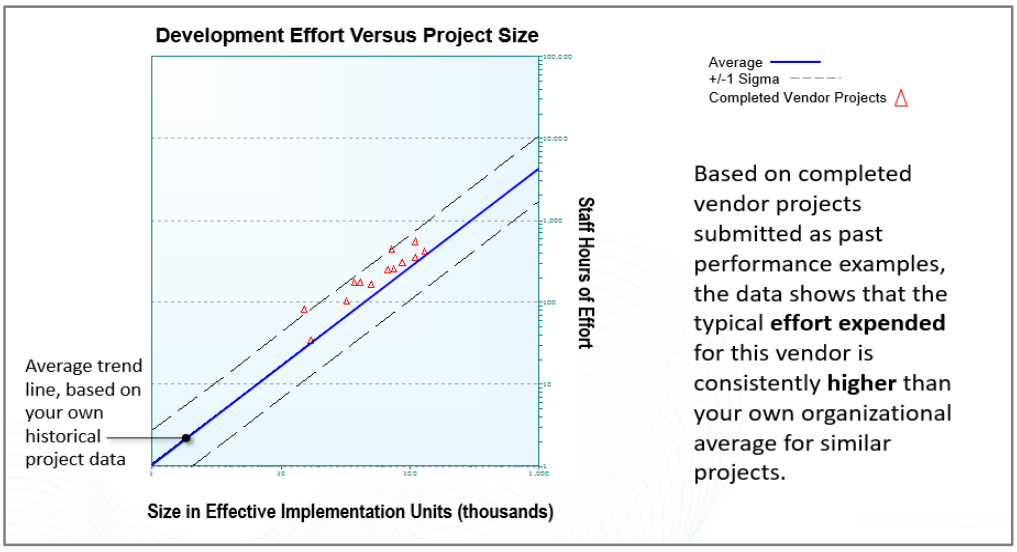


Figure 8.6. Historical position of Vendor 1 for effort versus size.

In Figure 8.7, we can see this is action when we assess the bids of two vendors and compared their estimated costs against the independent estimate and base lines. Vendor 1's cost (as manifested through the amount of effort) appears to be too high (expensive) and suggests that this vendor may be "padding" its bid, while Vendor 2's cost (amount of effort) looks to be much lower than what the independent estimate shows and suggests that this vendor might be underbidding in order to win the contract.

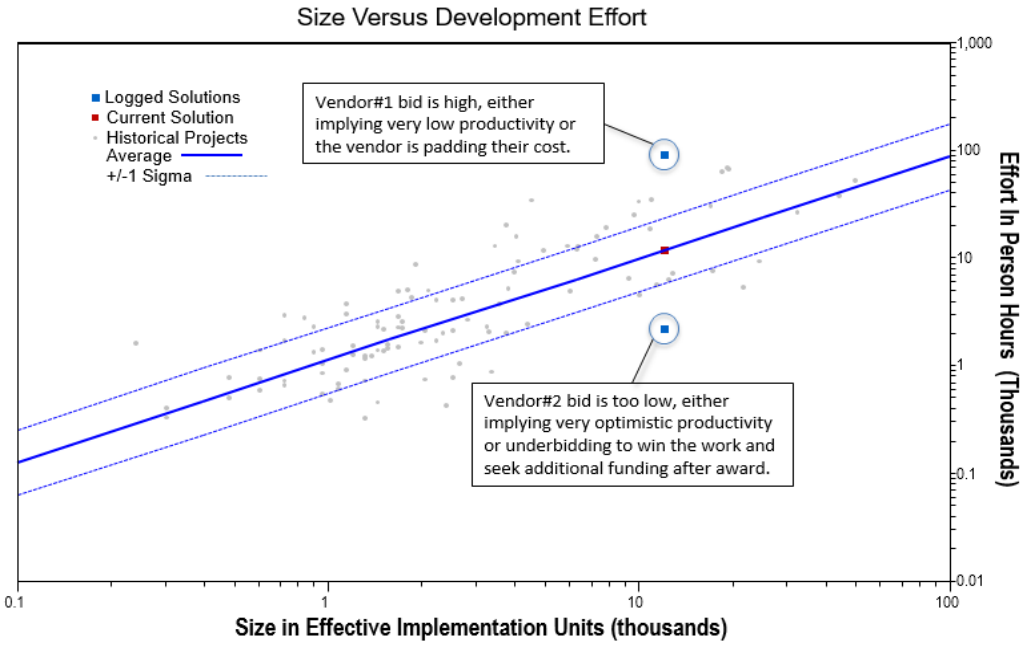


Figure 8.7. Vendor bids compared to the baseline trend and the independent estimate.

An analysis of the staffing plans is then performed (Figure 8.8). It is natural to have a gradual buildup and ramp down on a software development. We have found that a Rayleigh staffing curve is the most optimal way to staff a project, based on when work is available to be executed. We often find that vendors are more concerned about ramping up and getting a steady cash flow and thus typically propose level load staffing profiles. On large programs, this is a wasteful practice and generally results in cost overruns because money was wasted early in the project when people were applied but no work was available to be completed.

Finally, the results of the assessment are compiled into a scoring matrix (Figure 8.9).

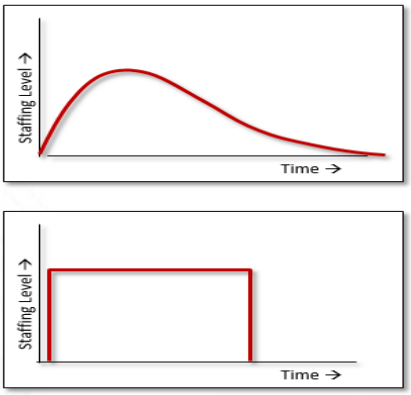


Figure 8.8. Optimum Rayleigh staffing profile versus inefficient level load staffing profile.

Depending on how each bidder performed they may achieve a rating anywhere from unsatisfactory to outstanding.

QUALITY OF METRICS PROVIDED	BENCHMARK COMPARISON W/ INDUSTRY, CLIENT	KEY PERSONNEL INCLUDED	ASSESSMENT	RATING
Incomplete/low quality/cannot be verified	N/A	N/A	High risk. Likely low process maturity (below CMMI Level 2)	**Unsatisfactory**
Satisfactory	Below Average	N/A	Medium-high risk. Productivity below average.	**Marginal**
Satisfactory	Average or Above Average	No	Medium risk. Favorable past performance, but proposed personnel did not work on those projects.	**Acceptable**
Satisfactory	Average	Yes	Low risk. Key personnel worked on past performance projects with average productivity.	**Above Acceptable**
Satisfactory	Above Average	Yes	Low risk. Key personnel worked on past performance projects with above average productivity.	**Outstanding**

Figure 8.9. Example scoring matrix showing criteria that would lead to ratings from unsatisfactory to outstanding.

The software vendor analysis is now complete, and this input is then combined with other acquisition decision criteria and relevant materials.

Summary. Quantitative assessment criteria have the advantage of being objective and defensible. If the vendors have incomplete or poor quality historical past performance data, that certainly tells us something about project risk. We want to make sure that their bids are credible when compared with historical projects and our own independent estimate. Finally, if the vendor has positive past performance, we want to note whether any of the key personnel being proposed actually worked on those projects.

Post Award - Phase 4

During the post award phase, the objective is to perform monitoring and control of contract execution by performing monthly health checks. Since the RFP required the winning vendor to report monthly progress using the same metrics they provided for their bid, we can use simple rate charts with control bounds to assess progress and risk (Figure 8.10).

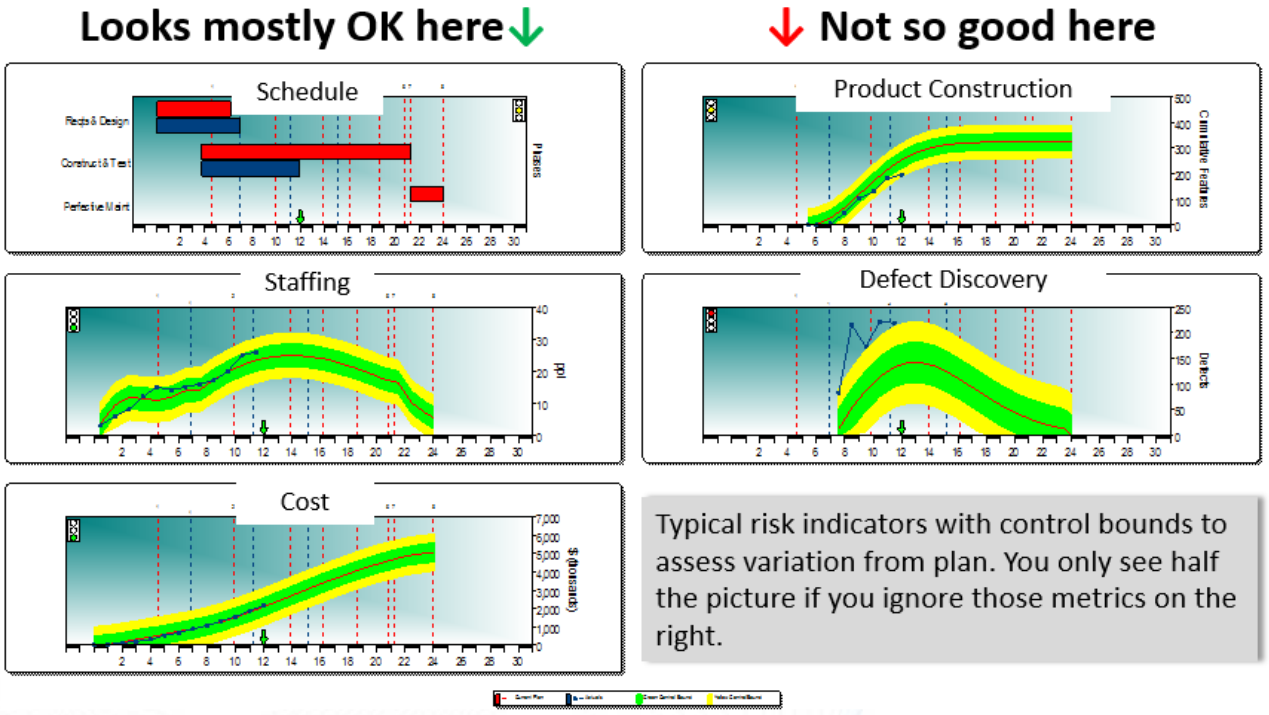


Figure 8.10. A typical project control dashboard showing the vendor bid with control bounds and actual performance data. An important point is that we need to link product construction and defect discovery to the consumption of resources. One without the other provides an incomplete and potentially misleading picture.

It is not unusual for changes in direction to occur, or sometimes things just do not go as planned. In this case, the control bounds identify the risk. The SLIM-Control® tool has the ability to forecast completion based on the actuals to date (Figure 8.11). It behaves very similar to the GPS in your car. If you miss a turn, it recalculates the best route from where you are right now. SLIM-Control® does the same thing (without the annoying voice). It provides a new

route and models the schedule, effort, staffing, and reliability consequences. Then, one can look at alternatives, such as adding people or reducing functionality, to simulate the possibilities. We call this adaptive forecasting. The advantage to this is that as management actions are taken, they either result in improved performance or they do not. Since SLIM-Control® reacts to the actual data, if a positive result is achieved, it will be reflected in any new forecast at completion. In effect, it is a learning system.

A nice point is that at the end of the contract, the purchaser has another new data point to add to their vendor performance database, which can be now be used to support the next

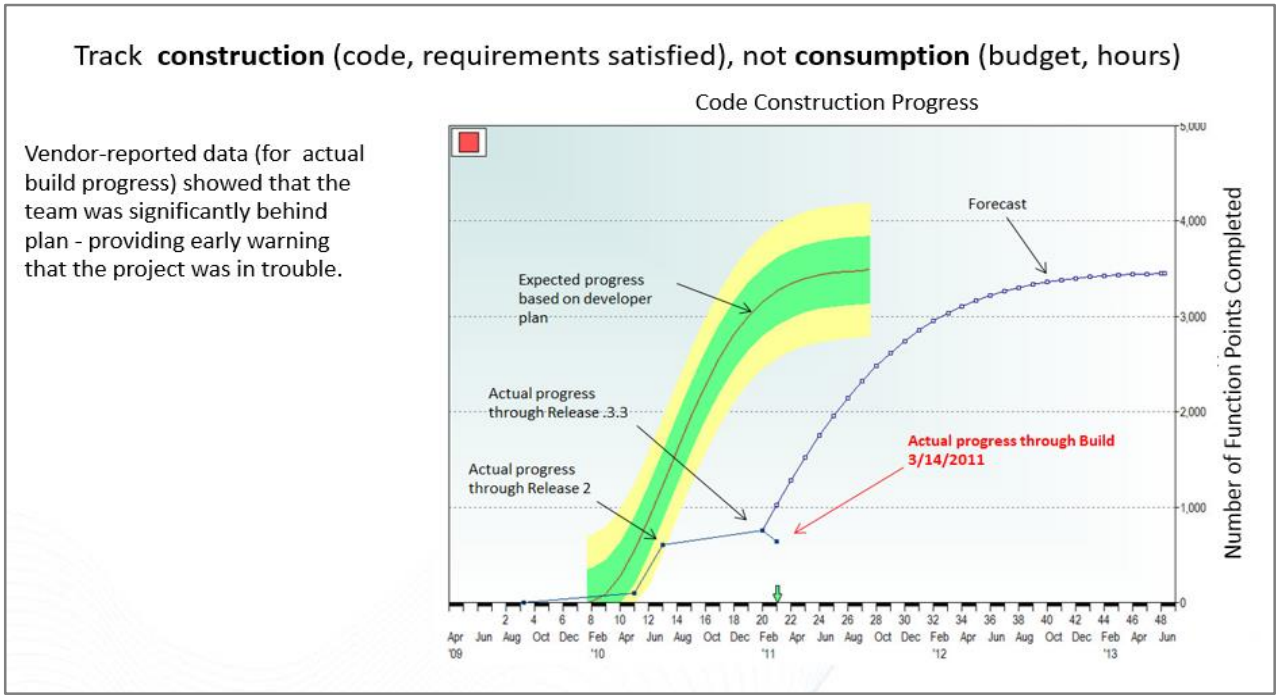


Figure 8.11. Example forecast to completion for code production. In this real example, the vendor has fallen well behind its bid production and a very significant schedule slippage is being forecast.

procurement.

Summary of Benefits

A quantitative vendor management process has many benefits and is worthy of serious consideration. Here are a few that can be realized:

1. The independent estimate builds realism into the purchaser's expectations.
2. Requiring vendors to quantify their bids and provide historical past performance data, when available, allows the purchaser to flag both "lowball" and overly conservative bids.
3. The assessment process identifies a vendor that provided "good value" and also one that has a high probability of successful execution.

4. The quantitative bid can be used for monitoring and controlling contract execution, providing stakeholders with transparency of progress and the opportunity to take corrective action when needed.
5. Adaptive forecasting allows one to embrace proposed changes and quickly model the potential impact to cost/effort, schedule duration and quality.
6. The process has the added benefit of helping to build a historical database of vendor past performance for future bid assessments.

How Much Software Is in Your Car? Is It Cyber Secure?

Joseph A. Madden

*An article based on this research appeared in the 2015 issue of the **Scarsdale Concours d'Elegance** auto show magazine and is reprinted here with permission.*

It is easy to imagine there is a lot of complex computer software code required to operate and control a fully autonomous self-driving car, such as the prototype recently unveiled by Google, and that advanced systems engineering and software life cycle management techniques are required to successfully manage its development. However, you may be surprised to find out that nearly every vehicle under 30 years old on the road today also depends on computer software - and lots of it.

According to an IEEE Spectrum article by Robert Charette entitled "This Car Runs on Code," the first production car to incorporate embedded software was the 1977 General Motors Oldsmobile Toronado which had an electronic control unit (ECU) that managed electronic spark timing (Figure 9.1). By 1981, GM had deployed about 50,000 lines of engine control software code across their entire domestic passenger car line. Other auto manufacturers soon followed the same trend.

Figure 9.1. 1977 General Motors Oldsmobile Toronado (Flickr).

Automotive Software Size

Around the same time software was being used for the first time in cars, QSM, Inc., founder, Lawrence Putnam, Sr., was discovering the "physics" of how engineers build software by successfully modeling the nonlinear relationship between the five core metrics of software product size, process productivity, schedule duration, effort, and reliability. One of the first presentations of his findings, entitled "A General Solution to the Software Sizing and Estimating Problem," was given at the Life Cycle Management Conference of the American Institute of Industrial Engineers in 1977. In 1978, Mr. Putnam invented the Software Lifecycle Management (SLIM®) tool based on these algorithms, and began collecting a benchmark database of historical software projects.

Fast forward to the present. The amount of software used in all industries, including the automotive industry, has increased exponentially in both size and complexity. Premium cars, such as the Mercedes-Benz S-Class, now depend on millions of lines of code running up to 100 networked ECUs throughout the body of the car, which control and monitor everything from the powertrain to the airbag system. Even lower-end cars have up to 50 ECUs. In a QSM productivity benchmark study for a major automobile manufacturer, we found that powertrain software can be just as sophisticated as real-time, embedded systems found in the military and aerospace industries.

Meg Selfe Divitto, former GM powertrain engineer and IBM Vice President of the Internet of Things, was quoted in an *Embedded* article as saying that the Volt had 10 million lines of code. While a significant portion of that was probably reused or auto-generated from model-driven development, it is nevertheless a massive amount of code greater in size than the avionics software in the U.S. Air Force F-22 Raptor and Boeing 787.

Few companies leverage software more than Tesla Motors, a U.S. company founded in 2003 and named after electrical engineer and physicist Nikola Tesla. Its CEO, Elon Musk, made his fortune in software startup companies, and Tesla Motors boasts cutting-edge software as a critical part of almost every aspect of its business. With an intense focus on agile innovation, the company began production of the Tesla Roadster in 2008 and the Model S sedan in 2012—both 100% electric cars that took the use of software to a whole new level. The Model S has a 17-inch touch screen with a Linux-based computer system that controls nearly every function of the car—from performance to the entertainment system. In fact, there are only two manual buttons: one for the hazard lights and one to open the glove box. Software updates, which include both bug fixes and new functionality, are pushed to the car remotely via a 3G cellular network.

Horsepower Driven by Software

Embedded software is the secret sauce in the electric powertrain of the Model S and one of the top areas of investment, according to CTO JB Straubel in an interview with *PCWorld*. The newest dual motor P85D version, released in late 2014, digitally adjusts torque at the millisecond level between the front and rear motors in order to achieve a staggering 864 foot-pounds of torque (more than a twin turbo V-12 engine), 691 horsepower, and 1G of

lateral acceleration going from 0-60 mph in 3.1 seconds. YouTube videos of the Tesla Model S P85D drag racing supercars such as the Ferrari 458 Italia have gone viral.

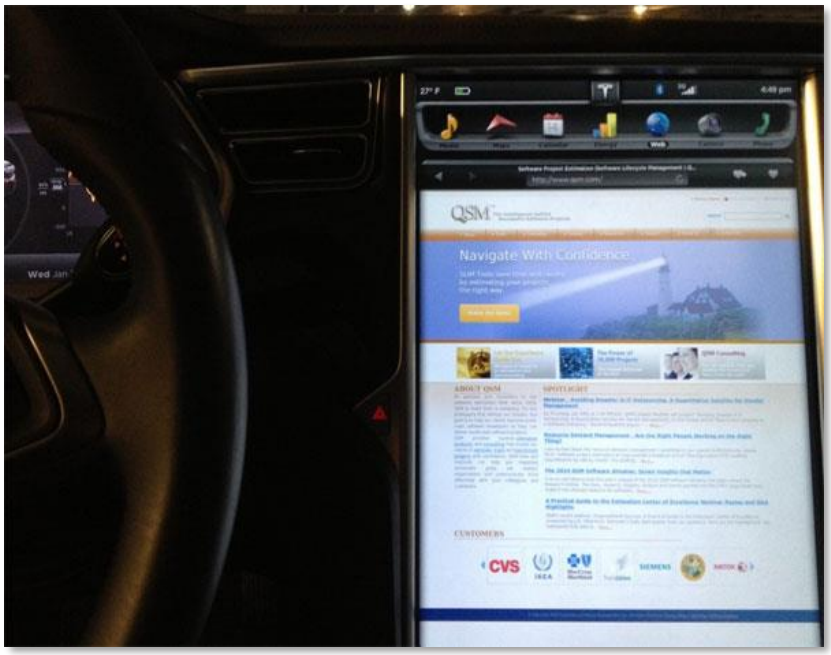


Figure 9.2. Tesla Model S 17" Touchscreen (author's photo).

Today's auto manufacturers have truly become software companies, presenting both opportunities and challenges. With all of this complex software comes the need for systems engineering and sophisticated project, program, and portfolio management carefully balanced against the triple constraints of schedule, effort/cost, and quality. Unlike some other industries, however, auto industry executives do not have the luxury of ignoring any one part of the triple constraint. Being late to market with innovative technology may mean that a competitor captures the majority of market share. If development effort and costs are too high, it puts more pressure on sales volume to reach an acceptable margin. Software reliability issues can result in expensive recalls and even lawsuits that tarnish a company's reputation and impact future sales.

So what does the future hold for the automotive industry? Some cars in production today, like the Tesla Model S, already offer advanced safety features and autopilot capability where software integrates data from cameras, radar, and 360-degree sonar sensors with real-time traffic updates. However, a ten-year look ahead gives us concept cars like the Mercedes-Benz F015 Luxury in Motion—so advanced that they appear as if they were straight out of a science fiction movie. Other visionary efforts, such as Project 100, are going beyond cars, and are rethinking the entire transportation system.

Will these concepts become reality ten plus years from now, when my young children are driving age? It is hard to say, but one thing is almost certain: there will be a lot of complex software involved and the companies that succeed will be the ones that combine

innovation with successful execution and a thorough understanding of the five core metrics of software development.

But Just How Cyber Secure Is Your Car?

Along with this explosion of complex software resident in our "wheels," we must now also contend with having to determine the cyber worthiness of the cars we plan to drive. This was abruptly brought to our attention this past July (2015) with the first cyber security recall in automotive history. Fiat Chrysler issued a formal voluntary recall of 1.4 million vehicles after security researchers Charlie Miller and Chris Valasek demonstrated to *WIRED* magazine how they could exploit a software vulnerability in Chrysler's Uconnect® dashboard computers. They remotely hacked into a 2014 Jeep Grand Cherokee over the Internet, taking over dashboard functions, transmission, steering, and brakes. Most notably, they did so from their basement while *WIRED* author Andy Greenberg was driving the vehicle on the highway!

Figure 9.3. 2014 Jeep Grand Cherokee (stock photo).

Though this was the first time an automotive manufacturer issued a recall for cyber security, it is not the first time security risks have been found in automotive software. As pointed out earlier in this article, nearly every vehicle less than 30 years old on the road today depends on lots of computer software, and, thus, is potentially vulnerable to hacking, especially newer models that are connected to the Internet.

One automotive manufacturer that has been very proactive with cyber security, however, is Tesla Motors. The Tesla Model S was built to be a connected car from the "ground up," and the protection of owners' security and privacy has always been a top priority, with the company collaborating strongly with security researchers throughout its development. After auditing the cyber security of the Tesla Model S, Kevin Mahaffey, co-founder and CTO of mobile security firm Lookout, concluded that "the Tesla Model S has a very well designed security architecture that we believe should serve as a template for others in the

industry…overall, I feel more secure driving in a Tesla Model S than any other connected car on the road."

Figure 9.4. 2014 Tesla Model S (stock photo)

Examples of good architecture decisions in the Tesla Model S observed by Mahaffey and his colleague, Marc Rogers, principal security researcher for CloudFlare, included an over-the-air update process to patch security vulnerabilities quickly, isolation between vehicle systems and infotainment systems, account password rotation, and the ability of the car to handle sudden power loss in a graceful way, enabling the driver to safely pull over to the side of the road in such an event.

However, even the Tesla Model S had a number of areas where cyber security could be improved. While the Model S has good perimeter security, Mahaffey and Rogers demonstrated how someone with physical access to the interior of the vehicle could plug into a diagnostic Ethernet port behind the instrument cluster and hack into the infotainment systems. From there they exploited a number of vulnerabilities in the infotainment systems, enabling them to remotely start and stop the car, unlock the doors, and open both the trunk and frunk. In response, Tesla immediately pushed out an over-the-air firmware security patch to all Model S owners, and Tesla CTO JB Straubel publicly thanked Mahaffey and Rogers for their findings at the Def Con Hacking Conference in Las Vegas.

So what can we do to improve cyber security across the entire automotive industry? Some best practice recommendations for consumers include the following:

- If you learn of a cybersecurity vulnerability applicable to your vehicle model year, promptly install the security patch from your vehicle manufacturer as soon as it is available.

- In the same way you would not allow someone you do not trust to have unsupervised physical access to your home computer, do not allow anyone you do not trust to have unsupervised physical access to your car.

- When considering the purchase of a new vehicle, do not be afraid to ask tough questions about cyber security and privacy in the same way that you might ask questions about the vehicle's safety rating, fuel efficiency, cargo capacity, etc.

- Encourage best practice collaboration between industry, government and security researchers. Consider supporting projects like the Five Star Automotive Cyber Safety Program. There is also an automotive anti-hacking bill that was recently introduced in the U.S. Senate.

Some best practice recommendations for automotive manufacturers:

- Incorporate cybersecurity across the entire software development life cycle (SDLC). Find and remove cybersecurity vulnerabilities early in the SDLC, when they are less expensive to fix. Include cybersecurity requirements in the criteria for peer review inspections and testing.

- Identify, quantify, and prioritize cybersecurity requirements as a formal part of the scope of a planned project release.

- Build cybersecurity into the software architecture. Mahaffey posted the following architecture best practice recommendations in his *Lookout* blog:
    - Establish an over-the-air update process to ensure security patches can be pushed out quickly.
    - Insolate vehicle and infotainment systems and ensure that any gateway between them undergoes an intense security review.
    - Harden the security of each individual component so that if one component is hacked, the others are still protected.
- Create a predictive model that forecasts software reliability. Researchers at Colorado State University found that between 1% and 5% of total software defects were cybersecurity vulnerabilities. Any software process improvement or management decision that reduces the total number of defects will likely reduce the number of potential security vulnerabilities.
- Ensure there is adequate budget to address cybersecurity requirements. For each planned software release, use a scope-based parametric software estimation tool to analyze various estimation scenarios that balance schedule, effort/cost, and quality. Do not overstaff the project to try to compress the schedule. QSM's quantitative research has repeatedly shown that large teams produce significantly more defects than small teams when building the same project scope. An increase in the total number of defects will almost certainly result in more cybersecurity vulnerabilities.

Works Cited

Alhazmi, O. H., Y. K. Malaiya, and I. Ray. "Security Vulnerabilities in Software Systems: A Quantitative Perspective." Conference Paper: Data and Applications Security XIX,

19th Annual IFIP WG 11.3 Working Conference on Data and Applications Security. Storrs, CT: Colorado State University, 7-10 Aug 2005. Print and Web.

Atiyeh, Clifford. "Senate Introduces Automotive Anti-Hacking Bill." *Car and Driver.* Web. 21 Jul 2015.

Charette, Robert N. "This Car Runs on Code." *IEEE Spectrum*. Web. 1 Feb 2009.

Flickr. "That Hartford Guy." Web. 4 July 2012.

Greenberg, Andy. "After Jeep Hack, Chrysler Recalls 1.4M Vehicles for Bug Fix." *Wired*. Web. 24 Jul 2015.

Greenberg, Andy. *Wired*. "Hackers Remotely Kill a Jeep on the Highway—With Me in It." Web. 21 Jul 2015.

I Am the Cavalry: Technology Worthy of Our Trust. "*Five Star Automotive Cyber Safety Framework*." Feb 2015. Web.

Mahaffey, Kevin. "The New Assembly Line: 3 Best Practices for Building (Secure) Connected Cars." *Lookout*. Web. 6 Aug 2015.

Mahaffey, Kevin. *Lookout.* "*Hacking a Tesla Model S: What we found and what we learned*." Web. 7 Aug 2015.

Putnam, Lawrence H. "A General Empirical Solution to the Macro Software Sizing and Estimating Problem." *IEEE Transactions on Software Engineering* SE-4 *(*Jul 1978): 345–361. Print.

Selfe, Meg, as quoted by Ron Wilson. "Ten Million Lines in 29 Months: Model-driven Development on the Chevy Volt." *Embedded: Cracking the Code to Systems Development*. Web. 12 Apr 2011.

Thomson, Iain. "Tesla Tech Top Dog Downs Slug, Hikes Bug Bounty to $10k: Classy Way to Thank Team Who Hacked His Car." *The Register: Biting the Hand that Feeds IT*. Web. 9 Aug 2015.

Williams, Martyn. *PCWorld*. "Tesla CTO Talks Model S, Batteries and In-car Linux." Web. 14 Aug 2012.

Zetter, Kim. "Researchers Hacked a Model S, But Tesla's Already Released a Patch." *Wired*. Web. 6 Aug 2015.

Successful Staffing for Successful Estimation

Victoriano Fuster III

When an organization wants to proactively manage their software activities from inception through development and sustainment, an enterprise software estimation or acquisition Center of Excellence (COE) is a great solution. A significant portion of our professional services business at QSM is helping companies design and stand up enterprise COE operations.

There are three main components to a successful COE implementation. They are:

- People: Finding people with the right characteristics and developing their skills;
- Business Processes: developing the right business processes to support decision making; and
- Tools: Acquiring and configuring analytical tools to support the business processes.

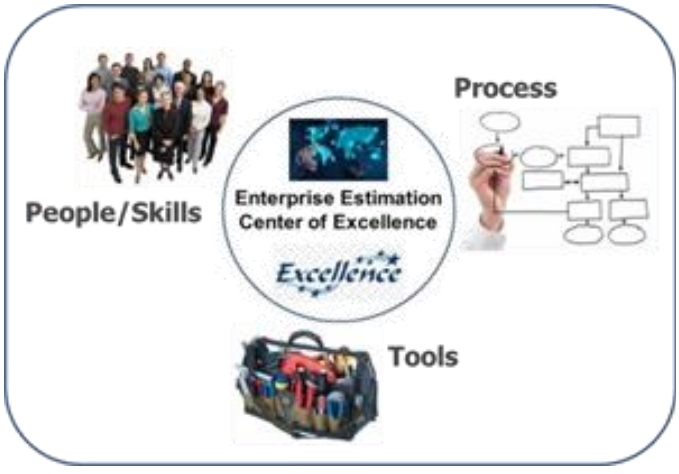


Figure 10.1. Components of a successful COE implementation.

Our clients often ask us to identify the best characteristics and skills for a person that they plan to staff into a COE. We went back and looked at our most successful implementations, and here is what we found.

Ideal Enterprise COE Skill Set

- **Basic Development Skills**. Parametric analysis tools (such as SLIM®) model system development projects. A basic understanding of the technical work being performed enables the estimator to configure the tool to different project and group development environments. Incorporating the appropriate terminology, development methods, and workflow aids both in the model calibration and the gathering of historical data.
- **Analytical Skills.** Parametric models are analytical decision support tools. As a user of such tools, one has to be comfortable working with data and numbers. A basic understanding of business statistics is good knowledge to have. One should have the ability to analyze data, identify trends, and determine how to effectively use those trends in future analysis.
- **Communication Skills.** The use of parametric models involves dealing with a diverse group of people within the organization. Software size and productivity inputs to the tool generally come from technical staff, while project constraints come from stakeholders or commercial managers. People interested in consuming the results are project managers, stakeholders, customers, financial managers, technical managers, and project staff. So it is quite important for the parametric analysts to have good communication skills. They need to be able to extract the information that they need in order to do the analysis in a non-threatening way, and to allow them to package the resulting assessment into a decision document or presentation appropriate to the audience.
- **Presentation Skills.** The outputs of parametric modeling tools are used to help make decisions, so the packaging of the relevant information into a digestible briefing is a large part of the analyst's task. Make sure to find someone who has good packaging and presentation skills.
- **Mediation and Conflict Resolution Skills.** Project estimation might better be called project negotiation. There are always trade-offs among the capability and the customer's desires, the time and money that is available, and the required reliability at deployment. Negotiations often result in heated debates among developers, customers, managers, and sales personnel. To make progress toward a solution, an analyst needs to be able to defuse the situation and then provide objective and practical alternatives that allow the interested parties to better understand the realities and move toward a compromised solution that is the best for all. Thus, conflict resolution and mediation skills are an important consideration.
- **Planning and Execution Skills.** If you are considering a large enterprise deployment, then planning and execution becomes an important skill set. Having a logical plan for data collection, template building, training, and deployment are all critical success factors.

- **Mentoring Skills.** If you are staffing up an estimation function, you want people who can share their knowledge and experience with the newer members of the team. This allows you to quickly jump start the group and leverage corporate experience, process, and intellectual property.
- **Functional Sizing Skills**. One of the biggest challenges in getting started is finding a functional sizing method to which engineers will be able to relate. It gets at the fundamental question of identifying what has to be built. There are various methods that can work successfully. Some common methods are function points, standard components, requirements, use cases, stories, and story points. If your candidates have experience in any of these areas, it can be a significant benefit.

The importance of selecting the right people for implementing an estimation COE cannot be overstated. Combined with the right tools and business processes, you will have a capability that will support a variety of levels within the enterprise.

- C-level: Performance benchmarks and dashboards, strategic program risk assessment, demand management, and portfolio performance optimization;
- Project managers: Realistic schedules and budgets, resource optimization, negotiation, and risk assessment;
- Business stakeholders: Transparency and a partner relationship with a development organization; and
- Acquisition stakeholders: Visibility into supplier capabilities and performance, best price/value.

Ultimately, an estimation COE solution provides the foundation and resources to achieve such organizational benefits.

Demand Management

Douglas T. Putnam and Taylor Putnam-Majarian

*This article originally appeared in the March 20, 2015, online edition of **Software Magazine** and is reprinted here with permission.*

If you think about it, enterprise application capacity planning can be a difficult juggling act. On one side of the equation you have business demand. The business exists in a fluid, competitive environment. With stakeholders hoping to gain an edge over their competitors, they are constantly looking to innovation and technology to help them improve business performance and increase their profitability. The IT organization stands on the other side of the equation, and is responsible for satisfying the demands of the stakeholders. Their capacity is limited by their facilities, the volume of developers and their specific skills, and the infrastructure that is in place to enhance their productivity. This leaves the business and technology executives in the unenviable position of trying to balance the demand for IT development with their current capacity levels (Figure 11.1).

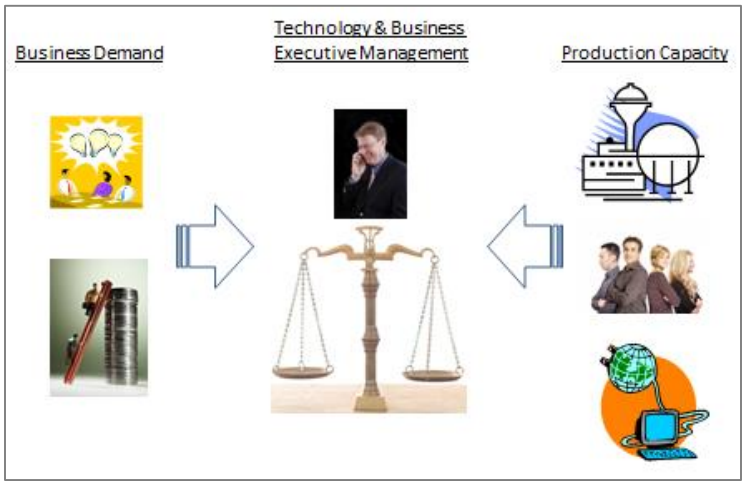


Figure 11.1. Conflicting demands on executive management.

Moreover, in a large enterprise, there are thousands of projects moving through the development pipeline, making the balancing process *that* much more difficult. In order to address these complexities many organizations have implemented enterprise Project and Portfolio Management (PPM) systems. These PPM systems are great for allocating known

resources against approved projects, and aggregating the data into a portfolio to give a current picture of how the IT capacity satisfies business demands.

In order for the capacity planning process to work, it is important to have a robust demand estimation capability. This is a discipline that has proven to be difficult for many organizations. Why is it so difficult? There are many reasons, but two stand out.

The first is that most organizations only estimate an aggregate number of hours by skill category and have no estimate of the schedule or how the skills are applied over time on the project. The most common practice today is based on using a detailed task-based estimation method. Imagine a spreadsheet with a generic list of hundreds of tasks that typically are performed on any given project. It is the estimator's job to review the requirements and estimate the number of labor hours required for each task. The task estimates are rolled up into a total labor estimate which is then allocated out to labor categories. The problem is that this method does not provide details for how the skilled labor categories are required to ramp on or taper off the project. This is an important component of capacity planning process. So using today's most common estimation practices, one ends up with a project estimate that looks something like Figure11.2:

Skill Category	Effort (PHR)	% Effort
Project Manager/Lead	2,076	12.79
Business Analyst	1,461	9.00
Data Architect	1,487	9.16
Developer	6,343	39.08
QA andTest	3,974	24.48
Database Administrator	280	1.72
Architect	611	3.76
Total	16,231	100.00

Figure 11.2. Typical bottom-up estimate without project schedule.

Unfortunately, this method also does not include a schedule. The schedule is usually decided arbitrarily by the stakeholders and often does not align with the estimated effort. QSM research shows that unrealistic schedule expectations are the number one reason why projects fail. This is clearly a big deficiency. The second issue is that task-based estimation is not well matched for the information that is available during the early stages of a project when the estimates and business decisions are typically made. The initial estimates are typically performed when there are only a few pages of business requirements. This clearly is not sufficient information to do a detailed task-level estimate. Moreover, doing a task-based estimate requires a lot of time and energy, so it would not be appropriate if numerous alternative scenarios are requested.

Top-down Estimation Is Very Effective in the Early Stages of the Project Lifecycle

Top-down parametric estimation is particularly effective in the early stages of the lifecycle. It utilizes gross sizing and productivity assumptions to produce schedule and effort estimates. If the estimates do not match expectations, top-down estimation tools can quickly evaluate

what can be accomplished with the available resources or determine how much additional time and effort will be required to achieve the business objectives (see Figure 11-3).

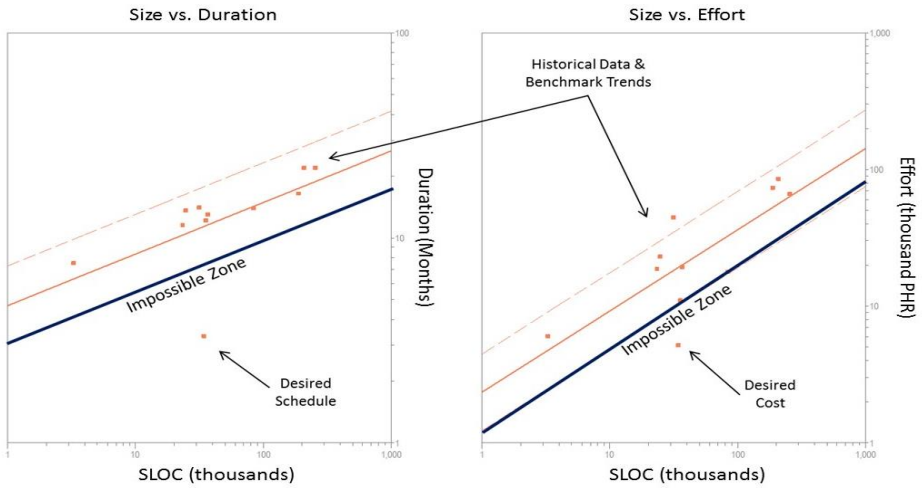


Figure 11.3. Desired schedule and cost compared against historical data and benchmark trends.

The beauty of the top-down method is that it uses crude size measures like business requirements, agile epics, use cases or function points as a direct input. Coincidentally, these are the design artifacts that are most likely to be available during the early stages of estimation.

With this basic information, top-down methods can quickly identify both high risk and conservative expectations and, if appropriate, suggest more reasonable alternatives. Let's take a look at a typical estimation situation early in the lifecycle. Suppose, for example, that a business has identified some new capabilities that they would like to have implemented. They have a five-page document that describes 19 business requirements or capabilities, which they would like to have available within three months with a budget of $250,000. Figure 11.4 shows their desired outcome.

When compared to historical data, the desired outcome is determined to be "Risky" and an alternative estimate of 5.7 months and $571,000 is recommended. Now a negotiation between the stakeholders and the IT department can take place to explore more viable options. For example, 11 business requirements could be completed in three months using the previously outlined resources, or all 19 requirements could be completed in four months using a team of 12 people and a budget of $1.8 million. The main purpose for estimating at this stage of the demand management process is to smoke out grossly unrealistic expectations and negotiate realistic options in order for all parties to be winners in this development.

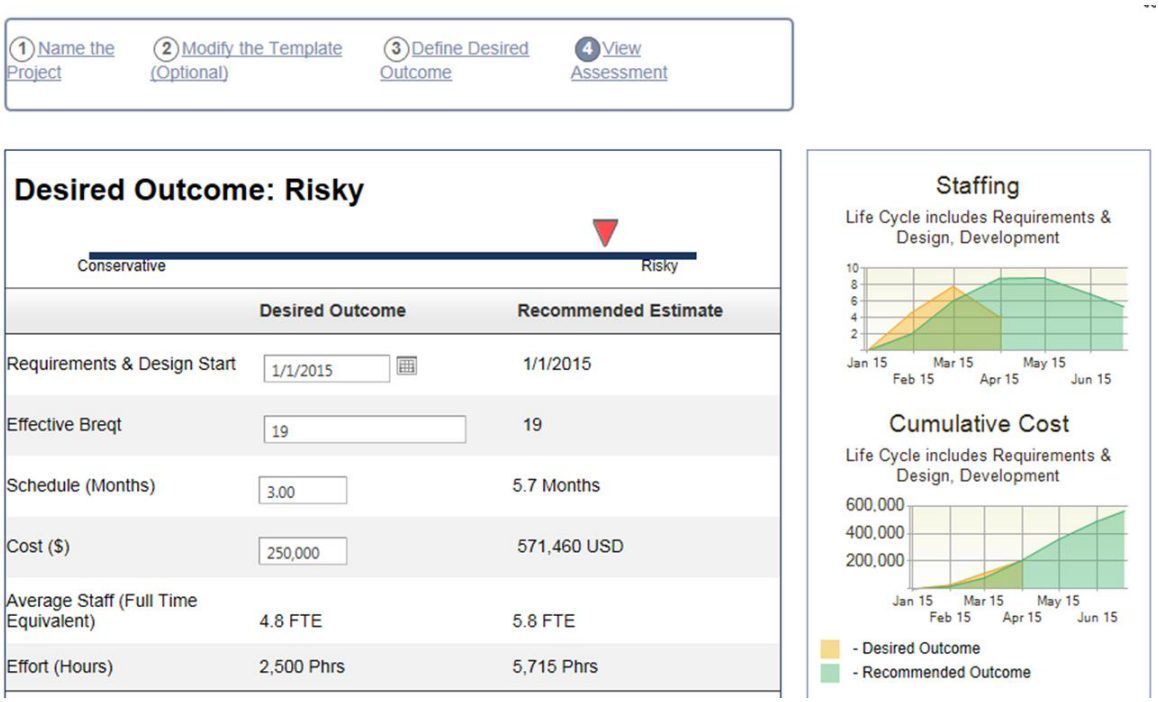
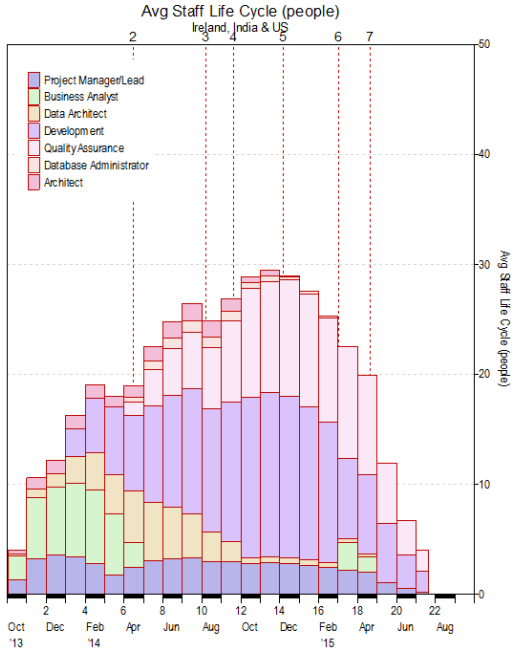


Figure 11.4. Desired, but risky, outcome, compared with the recommended, more realistic estimate.

Turning Hours into Skills and Staffing Requirements

The most "common practice" labor estimates just are not good enough to support capacity planning. What is really needed is an estimate of how the various skills will ramp up and taper off over the course of the project (see Figure 11.5).



Avg Staff Life Cycle (people) - Ireland Project 001
Ireland, India & US

Months'	Month	Project Manager/Lead	Business Analyst	Data Architect	Development	Quality Assurance	Database Administrator	Architect
1	10/13	1.39	2.31	0.46	0.00	0.00	0.00	0.46
2	11/13	3.34	5.57	1.11	0.00	0.00	0.00	1.11
3	12/13	3.67	6.21	1.34	0.24	0.00	0.00	1.24
4	01/14	3.42	6.79	2.49	2.75	0.00	0.00	1.36
5	02/14	2.85	6.76	3.47	5.14	0.00	0.00	1.35
6	03/14	1.85	5.56	3.71	6.30	0.00	0.00	1.11
7	04/14	2.51	2.37	4.82	6.93	1.35	0.47	1.18
8	05/14	3.22	0.00	5.35	8.82	3.30	0.92	1.38
9	06/14	3.37	0.00	4.76	10.25	4.36	1.01	1.52
10	07/14	3.44	0.00	4.13	11.38	5.25	1.08	1.61
11	08/14	3.08	0.00	2.77	11.28	5.67	1.01	1.52
12	09/14	3.09	0.00	1.94	12.84	7.41	0.90	1.24
13	10/14	2.93	0.00	0.59	14.67	9.98	0.59	0.59
14	11/14	3.00	0.00	0.60	15.00	10.20	0.60	0.60
15	12/14	2.95	0.00	0.59	14.73	10.70	0.36	0.13
16	01/15	2.80	0.00	0.56	13.98	10.35	0.28	0.00
17	02/15	2.57	0.00	0.51	12.85	9.51	0.26	0.00
18	03/15	2.30	2.58	0.46	7.40	10.21	0.01	0.00
19	04/15	2.03	1.56	0.26	7.31	9.12	0.00	0.00
20	05/15	1.23	0.00	0.00	5.54	5.54	0.00	0.00
21	06/15	0.70	0.00	0.00	3.16	3.16	0.00	0.00
22	07/15	0.44	0.00	0.00	1.99	1.99	0.00	0.00

Figure 11.5. Staffing skill categories by month.

With this information, PPM systems can match individuals with their respective skills to the project's needs at the time they are required and free them up when their work is done. It is important to recognize that one size does not fit all projects. For a large enterprise there might be several skill profiles or templates to support the different project types and methodologies. Typical methodology and project examples today include agile, traditional waterfall, package implementation, infrastructure, etc. The key point is to have just enough flexibility to provide adequate coverage but not so much that the organization becomes a slave to template creation and maintenance.

Configuration within the SLIM® tool is flexible enough to add, change, and delete labor categories and rates. It is also intuitive enough to allocate the buildup and ramp down of skill categories over time (see Figure 11.6).

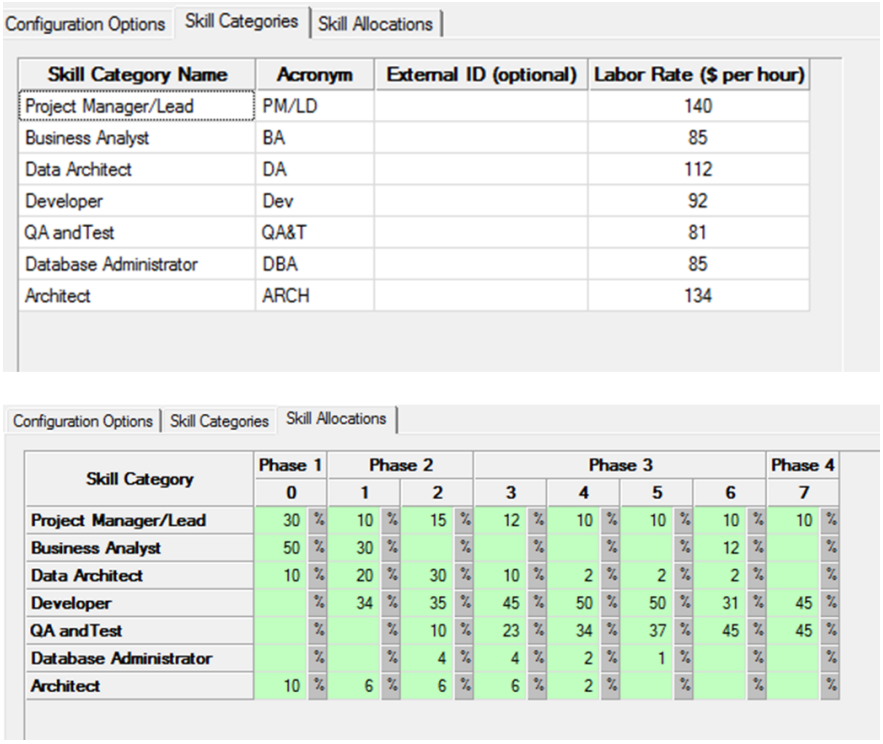


Figure 11.6. Configuring skill categories.

With a method to produce estimated skill allocation over time, it is possible to incorporate realistic demand into the planning process and match demand with the available capacity. The SLIM® estimation tool has a PPM integration framework, which allows SLIM® to integrate with any PPM program (see Figure 11.7).

The PPM integration allows SLIM® to assess and adjust the planned resources and/ or start dates to projects already present in the PPM system or to those prior to entry. This feature can be particularly useful for projects that require further evaluation and validation.

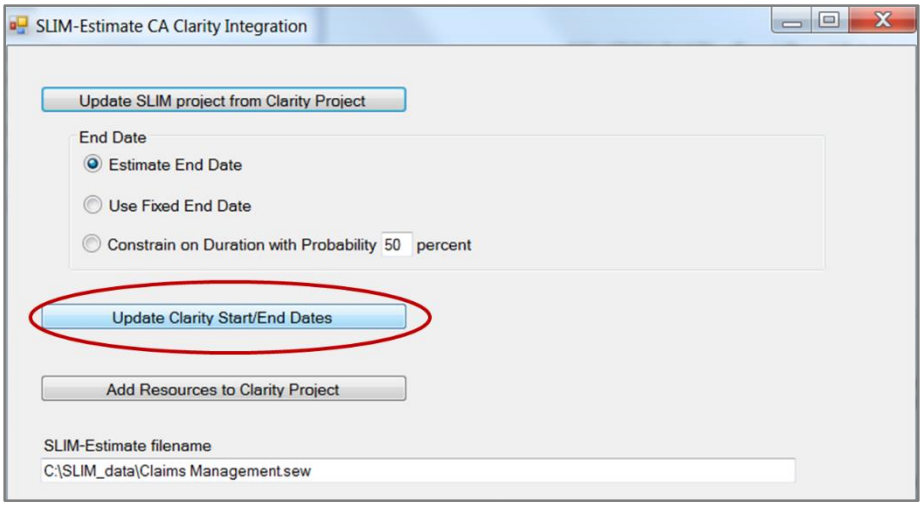


Figure 11.7. SLIM® integration with PPM tools.

Match Demand to Capacity

Now we have a way to realistically estimate schedule, effort, cost, and skill requirements over the duration of a project, and can feed that information into a PPM system. By implementing this discipline throughout the enterprise, it is possible to evaluate how demand matches capacity and determine whether they are out of sync. If we discover that they are out of alignment, we can evaluate practical options for matching demand and capacity.

Let's use an example to illustrate the process. Our example enterprise has three different data centers: one is located in the U.S., while the other two are in India and Ireland. The total capacity for these three centers is limited to 250 IT developers, and there are currently 35 projects that have been approved for production (Figure 8.8).

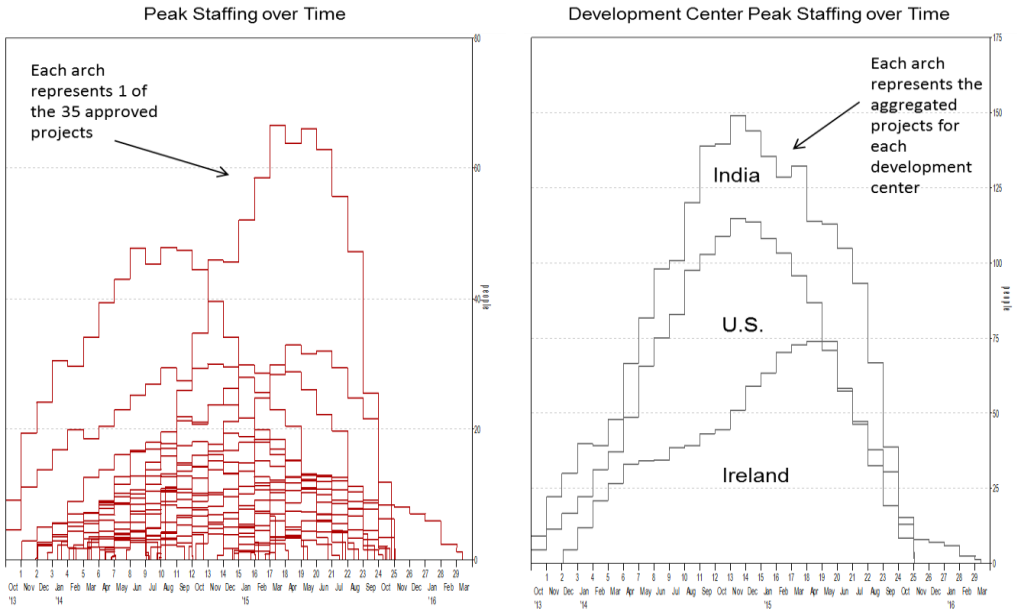


Figure 11.8. Peak staffing over time for individual projects and aggregated by development center.

Each of these projects is in a different stage of production, with some in the early planning stages, others beginning coding, and some that are approaching delivery. At any point in time, it is fairly typical to have a portfolio with projects at all of these production stages. Additionally, it is common for a portfolio to have a wide range of project scopes and team sizes. Each project can be estimated individually, and they can be rolled up to show the aggregate data at either the data center level or the enterprise level to help determine the capacity and demand levels. For the sake of simplicity, we will focus at the enterprise level in this example. When the data is aggregated at this level, we notice that for a nine-month period, between August 2014 and May 2015, the demand for developers exceeds capacity by 65 full time equivalents (FTEs) (see Figure 11.9).

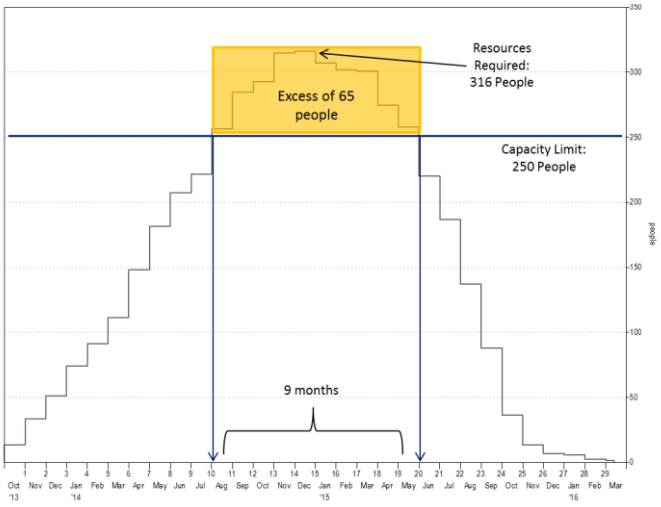


Figure 11.9. Demand exceeded capacity by 65 people for nine months.

When the demand for developers exceeds its capacity, there are usually three options:

1. Eliminate projects (usually not a practical option, since there is a definitive business needs;
2. Push back the start dates on projects that have not yet begun; or
3. Make a select staffing reduction on projects in-progress or those yet to begin, and relax their respective delivery dates.

In this example case, we chose option 2 and pushed back the start dates. In a relatively short period of time we were able to optimize the staffing to meet the capacity limit of 250 people. The resulting impact this had on the eight projects was a delay that lasted anywhere from two to eight months (see Figure 8.10).

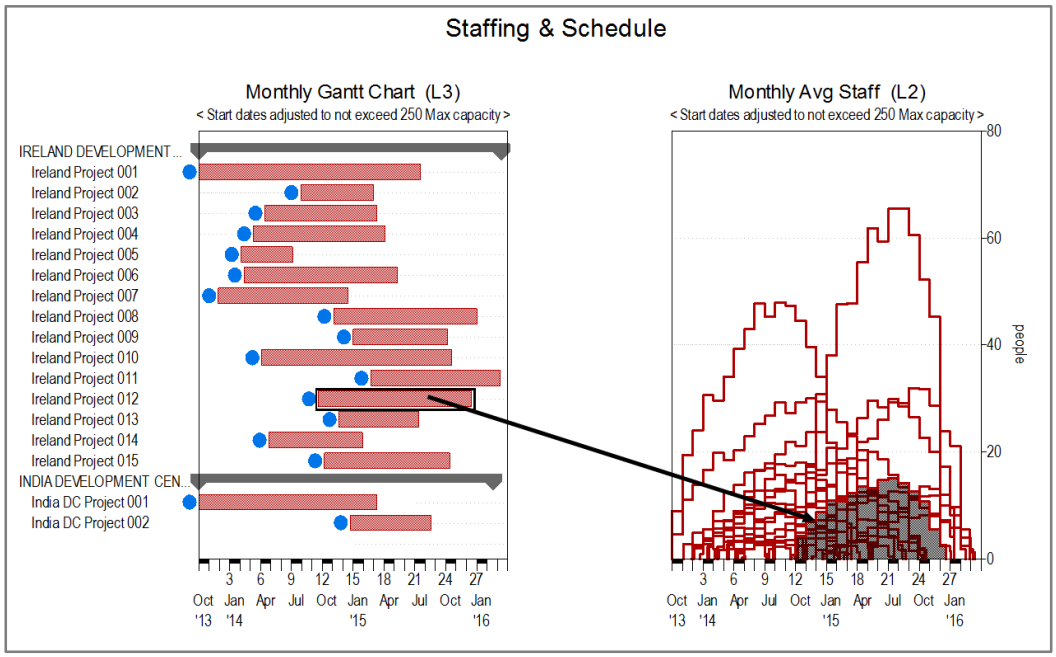


Figure 11.10. Project start date delayed.

If we want to assess the skill requirements and match those to our pool of available resources, we need to aggregate the skill categories to match that to our skill capacity.

Figure 11.11 gives a graphic depiction of the necessary resources at the enterprise level. It shows the aggregated demand for each of the skill sets at various times within a 28-month period. This information is useful to have because it lays out what you will need to successfully complete the approved projects and compare them with the resources currently available.

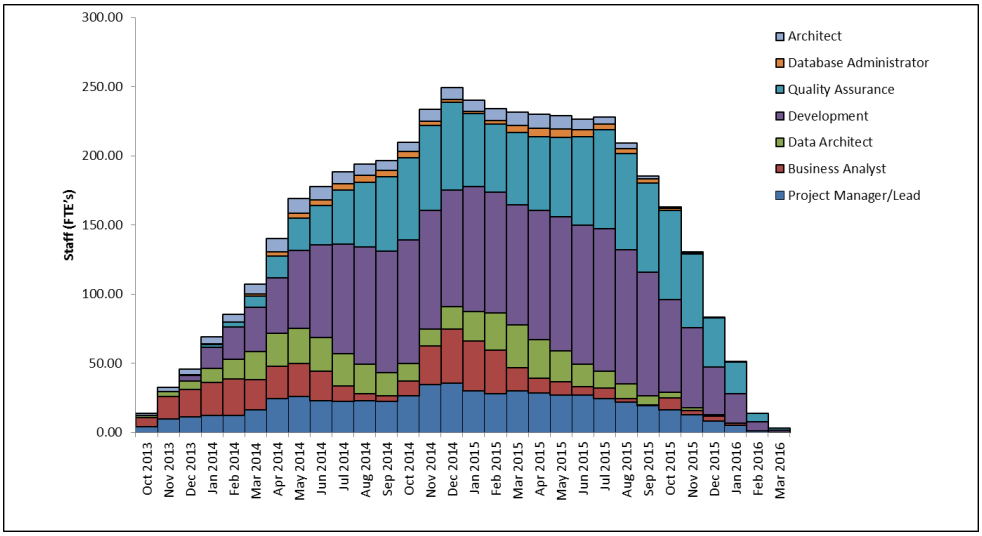


Figure 11.11. Staffing resource profile.

If there are imbalances between supply and demand, you now know what areas need to be supported and can take actions such as hiring or training additional employees to help

fill the void. By having this kind of information available and understanding its significance, great progress in capacity planning can be realized by the IT organization.

Summary

Capacity planning and demand management go hand in hand. To manage them effectively, one needs to be able to:

1. Realistically forecast demand early in the lifecycle;
2. Negotiate functionality, schedule and effort with business stakeholders when their expectations are unrealistic;
3. Have the ability to forecast effort by skill category, by month, or by week, and feed this information to PPM systems; and
4. Be in a position to perform quick what-if analyses on the portfolio when demand exceeds capacity.

If you take action to address these issues, then you can have a best-in-class IT capacity planning solution for the enterprise.

Glossary

Duration: Includes the time measured in months for all activities beginning with Requirements Determination (Phase 2) through Initial Release (end of Phase 3).

Effort: Includes the number of person hours for all activities completed in Phases 2 and 3.

Average Staff: Includes the number of staff, measured in FTE, for Phases 2 and 3.

MTTD: Refers to the Mean Time to Defect, determined in the first 30 days after delivery.

Hammer, Saw, or Chisel: Choosing the Right Tool

Donald Beckett

There is an old adage that if your only tool is a hammer, everything looks like a nail. We use the lessons learned and experience we have gained to address current issues. But if the problem (or software project) we face today is fundamentally different from those with which we have dealt previously, past experience is not the proper framework. In effect, we will be using a hammer when a saw or a chisel may, in fact, be more suitable.

The solution, of course, is to first gain an understanding of the problem at hand. What are its defining features? How does it behave? Only then can a proper solution be designed and the appropriate tools selected.

To a large degree, our understanding of how products are developed comes from knowledge gained from manufacturing since the beginning of the Industrial Revolution. Mentally, our first instinct is to try to apply those lessons learned to software development. But there is a huge problem with this approach. The creation of software is not a manufacturing process, but rather a knowledge acquisition and learning process that follows different rules. Here is a simple example. If I have an assembly line and want to double my output, I have several choices. I might add a second shift of workers, or I could install an additional assembly line. Because manufacturing is a repetitive process in which design problems are solved before product construction begins, the relationship between labor required and output remains fairly constant. In a nutshell, we already know exactly what we need to do (and how to do it).

Software is a different beast. Because design often "evolves" during product construction, there is always some uncertainty about what is required and how to do it. Software development is the process of resolving those uncertainties, the result of which is the delivered software product. If this were not so, the problems with software development would have been solved with Case Tools in the 1980s: analyze the problem, design the solution, feed it into the Case Tool, and, *Bingo,* out comes a functional software product! Case Tools had limited success in large part because it is impossible to completely eliminate uncertainty about what to do and how to do it in advance. Since software is a learning (rather than a manufacturing) process, it follows a different set of rules.

In the manufacturing example above, you could double the output or halve the development time (within the limits of machine capacity) by doubling the staff. The relationship between output (products manufactured) and the inputs to production (labor and machinery) is linear. Doubling the staff on a software projects, however, does not cut the schedule in half; it just costs more and ends up producing an inferior product.

Here are a few rules about how software development behaves. The business leader who adheres to these will improve his or her project success rate and may also experience the benefits of lower blood pressure:

- **Rule 1**. Your own project history is the best predictor of how you are going to perform at the present. Plan within your demonstrated capabilities. Projects planned outside that region lead to failure. Capabilities can and should be improved over time, but this is a long term endeavor; it is one that cannot simply be willed into being.
- **Rule 2**. While new tools and processes may provide long term benefits, the learning curves associated with them initially lowers schedule and cost productivity because the team has no experience working with them. They may be good long term investments, but you will not see the results immediately; they are not a short term fix!
- **Rule 3**. Small teams are more effective than large teams. They cost less, complete the software in about the same time, and produce fewer defects (and they are also easier to manage). There are two reasons why small teams work better. First, as staff is added to a software project, the number of communication channels increases in a decidedly non-linear fashion, making communication within the project more complex. It goes without saying that good communication is critical to software project success. Second, as more staff is added, the work becomes more and more optimized. While the individual developers may be doing their work well, they will lack the big picture of how their part contributes to the whole project. This can produce serious problems during integration testing.
- **Rule 4**. The relationship between software schedule and cost/effort is profoundly non-linear. One unit of schedule reduction is purchased at the cost of many additional units of labor (cost).
- **Rule 5**. Reducing a project's schedule below what is optimal is not only costly (see Rule 4); it is dangerous. Fredrick Brooks, the author of *The Mythical Man Month*, captured this succinctly when he said, "More software projects have gone awry for lack of calendar time than for all other causes combined. Why is this cause of disaster so common?"

RESOURCES

"Data is a precious thing and will last longer than the systems themselves."

–Professor Sir Timothy Berners-Lee, English computer scientist, inventor of the World Wide Web

"If you can't explain it simply, you don't understand it well enough."

– Albert Einstein, German-born American theoretical physicist, developed general theory of relativity

Sizing Infographic

Available for download at:
http://www.qsm.com/infographic/software-sizing-matters/

Software size, the amount of functionality in a given software release, is arguably the most important of the five core metrics of software estimation. There is little point in tracking effort, duration, productivity, and quality if you are unable to quantify what you are building.

Yet, despite its critical importance, software sizing is often a difficult concept for many to understand and use properly in the estimation process. Sometimes a picture is better than a thousand words. With that ideal of visual simplicity in mind, we developed a software sizing infographic that helps explain:

- Why we care about size
- Challenges in sizing
- When size should be measured during the software development life cycle (SDLC) to narrow the cone of uncertainty
- The difference between functional and technical size
- The most popular sizing methods and when to use them

The infographic begins by introducing the five core metrics of software estimation (size (scope), schedule (duration), effort (cost), quality (defects), and productivity) and the nonlinear relationship among them.

Next, it outlines the four generic phases in the software development life cycle and why estimators need to use different sizing methods, depending on where the project is in the life cycle and what information is available. At each stage of the software development life cycle, the cone of uncertainty narrows and the number of sizing techniques that can be used increases as more information is known about the project and the required functionality.

It then introduces the concepts of functional size and technical size, and how every sizing method can be normalized to a common unit. Technical size is the amount of software logic (source lines of code or configurations to a commercial off-the-shelf package (COTS)) that creates the functionality.

- All technical sizing methods can be converted to implementation units (IUs). An IU is equivalent to writing one source line of code or one technical step in configuring a commercial COTS package. Developers typically care more about technical size than functional size because it is a measure of how much technical work they need to do.
- Functional size is a technology-independent measure of the amount of software functionality delivered to the end user. All functional sizing methods can be converted to function points (the most widely used ISO standard for functional sizing

is the International Function Point Users Group (IFPUG) method). Function points can, in turn, be converted to IUs, based on the QSM Function Point Languages table. End users typically care more about functional size than technical size because it represents the software functionality that provides business value to them.

Building on the above concepts, the infographic next provides a table of the most common sizing methods with definitions.

- Examples of functional sizing methods include higher levels of abstraction (e.g., business requirements, epics, or use cases), medium levels of abstraction that can be prioritized for planning purposes (e.g., functional requirements, functional capabilities or user stories), and low level ISO standard function point techniques (IFPUG, COSMIC, MARK-II, FISMA, NESMA).
- Examples of technical sizing methods include business process configurations and RICEFW objects, technical components (screens, reports, forms, tables, etc.), source code files, and source lines of code. (Note: RICEFW is an acronym that stands for reports, interfaces, conversions, enhancements, forms and workflows which represent customizations to a COTS package.)

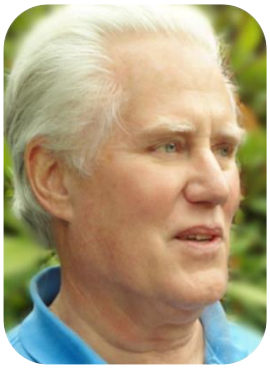
Finally, the infographic summarizes the whole sizing process by overlaying the cone of uncertainty atop the four software life cycle phases and the recommended sizing methods for each phase.

This infographic is a useful visual reference that puts you on the fast track to more successful estimation. A copy of this infographic can also be downloaded from the QSM website (http://www.qsm.com/infographic/software-sizing-matters).

INDEX

CONTRIBUTING AUTHORS

Phillip Glen Armour has been in the software development business for over 30 years, and has built industrial process control and management systems for steelworks, native language parsers for communications systems, data management systems for large retail operations, and support systems for airlines. Phil was a software process engineer in the mid '80s before there were such things, consulting with and implementing changes to the systems development processes and organizations of many companies, large and small, in the U.S., Europe, Asia, and the Middle East. He is a contributing editor and author of a regular column in the Association for Computing Machinery's (ACM) flagship magazine, *Communications of the ACM*, entitled "The Business of Software," and the author of *The Laws of Software Process: A New Model for the Production and Management of Software*. Phil earned a bachelor's degree in physics and pure mathematics from the University of Sheffield, United Kingdom.



Don Beckett has been active in software as a developer, manager, trainer, researcher, analyst, and consultant for 30 years. Since 1995, the focus of his work has been software measurement, analysis, and estimation; first with EDS (now HP) and, since 2004, with QSM. He has worked for many years with parametric models and tools to estimate and create forecasts to completion for software projects, and has created estimates for over 2,000 projects. In recent years, Don has worked extensively for the Department of Defense to evaluate requests for proposals and monitor the progress of large ERP implementations. More recently, he has studied the productivity and quality of software projects that follow the agile methodology.

Paul Below has over 30 years of experience in technology measurement, statistical analysis, estimating, Six Sigma, and data mining. As a Principal Consultant with QSM, he provides clients with statistical analysis of operational performance, process improvement, and predictability. He has written numerous articles for industry journals and is co-author of the *2012 IFPUG Guide to IT and Software Measurement*, and regularly presents his technical papers at industry conferences. He has developed courses and been an instructor for software estimation, Lean Six Sigma, metrics analysis, and function point analysis, and has taught metrics for two years in the Masters of Software Engineering Program at Seattle University. Paul is a Certified SLIM® Estimation Professional, and has been a Certified Software Quality Analyst and a Certified Function Point Analyst. He is a Six Sigma Black Belt and has one registered U.S. patent. He has a bachelor's degree in geological engineering from the South Dakota School of Mines and Technology and graduate studies in economics and systems engineering at Arizona University.

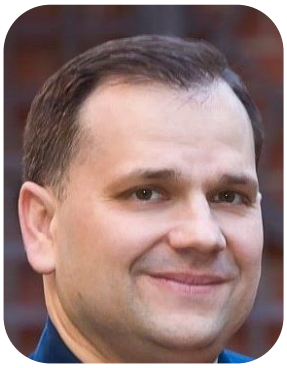
Dr. Andy Berner has helped organizations improve their software development processes for over 20 years. He has "hands-on" experience with almost every role in software development. He is on the QSM software development team and is leading the work at QSM to incorporate agile techniques into and enhance the resource demand management capabilities of the SLIM-Suite®. He has recently published several articles on agile methods and practices, focusing on planning projects to set realistic expectations. He has spoken at numerous conferences on software tools and methods, often with an emphasis on how to make sure that tools serve the team, rather than the other way around. He has an A.B. *cum laude* in mathematics from Harvard University and a Ph.D. in mathematics from the University of Wisconsin-Madison and has seven registered U.S. patents.

Carol Dekkers, PMP, CFPS, P.Eng (Canada), is a consultant, author, and speaker who has presented and taught in over 30 countries. She has been the primary U.S. expert for IFPUG and ISO software engineering standards for 20 years, and is a technical advisor to the International Software Benchmarking Standards Group (ISBSG.) She was recently re-elected to the IFPUG Board of Directors in Oct. 2015 (having served as its President in 1998-99). Carol is the co-author of two books: *The IT Measurement Compendium: Estimating and Benchmarking Success with Functional Size Measurement* and *Program Management Toolkit for Software and Systems Development*, and is a contributor to a dozen more, including both IFPUG textbooks on software metrics. Carol also holds designations as an Agile Expert Certified (AEC) and Certified Scrum Master (CSM). She earned her master's degree in mechanical engineering from the University of Calgary, Canada.

Victoriano Fuster III leverages dynamic leadership and technical abilities gained from a career across government, military, and commercial sectors. His 15+ years of professional experience includes specialty work in quantitative software analysis, multidisciplinary security analysis, and research and technology protection. He uses this unique background for QSM clients by providing custom solutions based on his diverse organizational experience, and has presented his focused DoD topics and courses reflecting these insights and experiences at national conferences and working groups. He earned his master's degree from Eastern Kentucky University and is a Certified SLIM® Estimation Professional. He is a proud U.S. Marine Corps veteran.

James Heires has over 25 years' experience in software development, project management, and management consulting. He is a member of the Project Management Institute and is a certified Project Management Professional. James has worked with SLIM® tools since the late 1980s. In addition to achieving QSM's Software Estimation Professional designation in 2013, James has used PRICE® and COCOMO® estimating models professionally. His responsibilities at QSM include consulting on major program acquisitions with the Department of Defense. James is a Certified SLIM® Estimation Professional, and earned his bachelor's degree in engineering from the University of Nebraska.

Joseph A. Madden leads the QSM consulting division, which offers a wide range of professional services to clients in both the public and private sectors. He has more than 23 years of experience in IT management and consulting. This includes more than six years as an officer in the U.S. Air Force and 10 years with consulting firms KPMG and BearingPoint. At those firms, he led many high-visibility projects and played a key role in successful SW-CMM and CMMI process improvement efforts. Joe is a graduate of the Yale School of Management. He earned his bachelor's degree in computer science from Marquette University and his master's degree in software systems engineering from George Mason University.



Jeff "JD" Ottenbreit has nearly 20 years of experience in the consulting industry, most recently learning to leverage QSM's well-known SLIM® tool suite to provide software modeling, best practice comparisons, and metric and benchmark studies for application development and IT decision makers. Also a former Division I Men's Ice Hockey coach, he has led consulting teams both domestically and internationally as part of a former Big Four corporation, a Fortune 50 company, and as an advisor to various Federal Government agencies and departments. He has a bachelor's degree from the University of Saskatchewan and a Masters of Business Administration from Sacred Heart University, Canada.

Douglas T. Putnam is Co-CEO for Quantitative Software Management (QSM) Inc. He has 35 years of experience in the software measurement field and is considered a pioneer in the development of this industry. Mr. Putnam has been instrumental in directing the development of the industry leading SLIM® Suite of software estimation and measurement tools, and is a sought-after international author, speaker, and consultant. His responsibilities include management and delivery of QSM software measurement services, defining requirements for the SLIM® product suite of tools, and overseeing the research activities derived from the QSM benchmark database.

Taylor Putnam-Majarian has over nine years of specialized data analysis, testing, and research experience. In addition to providing consulting support in software estimation and benchmarking engagements to clients from both the commercial and government sectors, Taylor has authored numerous publications on agile development, software estimation, and process improvement, and is a regular blog contributor for QSM. Most recently, Taylor presented research titled *Does Agile Scale? A Quantitative Look at Agile Projects* at the 2014 Agile in Government conference in Washington, DC. Taylor is a Certified SLIM® Estimation Professional, and holds a bachelor's degree from Dickinson College.