QSM Software Almanac

Application Development Series

2014 Research Edition

QSM Software Almanac

Application Development Series

2014 Research 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

First Edition

TABLE OF CONTENTS

(Hyperlinked Enabled – Click on Titles to Navigate)

EXECUTIVE SUMMARY

"The ability to simplify means to eliminate the unnecessary so that the necessary may speak."

– Hans Hofmann, German-born American abstract expressionist painter

"A person who is gifted sees the essential point and leaves the rest as surplus."

– Thomas Carlyle, Scottish Writer

"When you have mastered numbers, you will in fact no longer be reading numbers, any more than you read words when reading books. You will be reading meanings."

–W. E. B. Du Bois, American sociologist, historian, civil rights activist, Pan-Africanist, author and editor

Researching Success

Angela Maria Lungu, Editor

"If we knew what we were doing, it would not be called research."
-Albert Einstein

"Success is a lousy teacher. It seduces smart people into thinking they can't lose."
-Bill Gates

In this ever-changing world of software development, it is critical to maintain pace with current technologies, methodologies, and trends. Recently, Forrester released its list of top technology trends for the three-year time horizon. Mobile devices, cloud computing, virtualization, and cyber security are the top trends and concerns, and leaders in both the private and public sectors are affected by the daunting list of resultant challenges. Not only must the CIO stay ahead of the changes that are afoot, but he or she must also do so securely in an increasingly vulnerable environment.

Software is constantly evolving. The world of software development always moves at an incredible pace, and it can be difficult to keep up with the latest trends at the best of times. Yet, despite this constant change, some things have remained constant: software development projects are still difficult to estimate.

As Larry Putnam, Sr., observed more than 35 years ago and which is still relevant today (just read the latest Defense News headlines if you have any doubts), "200 to 300 percent cost overruns and up to 100 percent time slippages have been common, frequent, almost universal, as if there were no pattern, no process, no methodology, no characteristic behavior to the software development process" (Putnam 5). For decades, the goal for software developers has been to find predictable and repeatable processes that improve quality and productivity, and they must continue to do so with the added challenges of our modern world.

Influenced in part by the recent economic downturn, business and IT managers alike are looking for new methods of improving productivity, increasing employee efficiency, and optimizing their overall business and IT processes. Rather than chasing the latest technology, they must instead focus on applying proven practices for streamlining their development efforts and implementation processes. And this begins with looking inward, gaining basic insight into trends, behaviors, and technologies that affect organizational and enterprise software development.

The focus of this year's almanac is *Research*. As an industry, we do a surprisingly poor job of measuring the work that we do and how well we do it. We fail to measure enough, measure the right things, or know what to do with what we do measure; we cannot even agree on how to size things or determine how productive we are! How can we *get* better, or know that we *need to get* better, or know if we *are getting* better, if we can't or don't measure something? Without quantifying – offering proof – how can we determine if a new process is working, or make a business case for additional staff, or justify operating expenses?

We bring together in this almanac thought leadership and insights from client engagements around the world and research on our own database, in order to share the knowledge and allow managers and developers to better understand how their own processes and methodologies can be improved. We highlight how other organizations are effectively using core metrics to improve quality, reduce costs, shorten schedules, and, at the top of everyone's list, increase productivity. Just as important is to identify and examine those that are not successful and learn from those experiences, as well: "Deciding what not to do is as important as deciding what to do" (Steve Jobs). Because no two situations are ever the same, it is essential to look at these cases from all perspectives to better understand the full complement of factors impacting project success.

We begin by describing what we are seeing in terms of project demographics, examining the key characteristics of current software development, and establishing benchmarks across all size regimes, domain type, and industry, as well as other categories. We try to answer "What is a typical project?" to help you better assess your own relative position and start point.

From these basic observations of trending development, we delve deeper and examine the five core metrics (or five "levers") that have been used effectively for over 35 years to improve the management of software projects: time (duration), effort, size, productivity, and reliability (quality). The articles discuss the various interrelationships of each of these metrics, and present case studies and observations of their impact on project success from a variety of perspectives. Team size, defect forecasting, sizing, function point analysis, and data mining are only a few examples of the topics covered. Understanding this interrelationship among the core metrics allows a more sophisticated and nuanced approach to understanding the tradeoffs between time and effort, how to identify risk early on, and how to achieve effective project management.

We have chosen to include a section on Agile to examine such areas as how best performers have implemented this particular methodology, its potential for scalability across project sizes, and key planning factors and considerations when using Agile development. The results of our research may surprise you.

Next, we look at how to apply this shared knowledge for process improvement and enhanced planning for successful projects. Both best-in-class and worst-in-class performers are presented, in order to learn what works and, just as importantly, what does not.

Complementing this is the final section that presents the comparative project performance metrics over the last twenty years to identify emerging development trends in a range of interest areas, from methodologies, programming language, industry, and functional domain, to name only a few. We have also included a resources section with some useful references. The "Function Point Language Table" provides industry averages, organized by programming language, for number of source lines of code required to implement a function point, while the "Performance Benchmark Tables" provides a high-level benchmarking reference for industry average duration, effort, staff, and SLOC (or FP) per person month for size regimes of each trend group.

This research is provided in the hopes that this shared knowledge and current, focused insights will help you navigate today's challenges and help make your projects and organizations more successful. In this demanding environment, we all need the best tools available to make the most effective decisions possible. Given today's environment and all its challenges, known and unknown, can you really afford not to?

Work Cited

Putnam, Lawrence H. *Software Cost Estimating and Life-Cycle Control: Getting the Software Numbers*. New York: The Institute of Electrical and Electronics Engineers, Inc., 1980. Print.

ToC | Next Section

1. DEMOGRAPHICS

"Not everything that can be counted counts, and not everything that counts can be counted."

– Albert Einstein, German-born theoretical physicist and philosopher of science

"Data! Data! Data! I can't make bricks without clay!"

– Sir Arthur Conan Doyle, British physician and writer who is most noted for his fictional stories about the detective Sherlock Holmes

"There is divinity in odd numbers."

– William Shakespeare, The Merry Wives of Windsor, V.I.3

The QSM Project Database

Kate Armel

The QSM database is the cornerstone of our business. We use validated metrics collected from over 10,000 completed software projects to keep our products current with the latest tools and methods, to support our benchmarking business, to inform our customers as they move into new areas, and to develop better predictive algorithms.

Data Sources

Since 1978, QSM has collected completed project data from licensed SLIM-Suite® users and trained QSM consulting staff. Consulting data is also collected by permission during productivity assessment, benchmark, software estimation, project audit, and cost-to-complete engagements. Many projects in our database are subject to non-disclosure agreements; but regardless of whether formal agreements are in place, it is our policy to guard the confidentiality and identity of all data contributors. To preclude identification of individual projects/companies or disclosure of sensitive business information, we release industry data in summary form only.

In 1994, QSM began collecting project data continuously, updating the database every 2-3 years. Over the last 5 years, we have added an average of 200 validated projects each year.

Data Quality

Only projects rated Medium or High confidence are used in QSM's industry trend lines and research. Before being added to the database, incoming projects are carefully screened. On average, we reject about one third of the projects screened per update.

Data Metrics

Our basic metric set focuses on size, time, effort, and defects (SEI Core Metrics) for the Feasibility, Requirements/Design, Code/Test, and Maintenance phases. These core measurements are supplemented by nearly 300 other quantitative and qualitative metrics. Approximately 98% of our projects have time and effort data for the Code and Test phase and 70% have time/effort data for both the R&D and C&T phases.

Industry Data

QSM data is stratified into 9 major application domains (Avionics, IT, Command & Control, Microcode & Firmware, Process Control, Real-time, Scientific Systems, System Software, and Telecom) and 45 sub-domains. Software projects predominate, but we have a growing number of hardware and infrastructure (non-software call center) projects as well.

Data contributors include DoD; civilian commercial firms; and national, state, and local government entities. In addition to domain complexity bins, our data is also broken out by major industry and industry sector. Major industries include the financial sector, banking, insurance, manufacturing, telecommunications, systems integration, medical, aerospace, utilities, defense, and government.

Methodology Data

The QSM database includes a variety of lifecycle and development methodologies (Incremental, Agile, RUP, Spiral, Waterfall, Object Oriented) and standards (CMM/CMMI, DoD, ISO).

Language Data

Over 700 languages are represented with most projects recording multiple languages. Common primary languages are Java, COBOL, C, C++, C#, VISUAL BASIC, .NET, IEF / COOLGEN, PL/1, ABAP, SQL, ORACLE, POWERBUILDER, SABRETALK, Java SCRIPT, DATASTAGE, HTML. Frequently used secondary languages include JCL, SQL, Java, COBOL, ASSEMBLER, C++, HTML, VISUAL BASIC, XML, ASP.NET, and JSP.

Country Data

QSM has collected and analyzed software projects from North America, Europe, Asia, Australia, and Africa. About 50% of our data is from the U.S. Another 35-40% is from India, Japan, the Netherlands, the United Kingdom, Germany, France, and other major European countries.

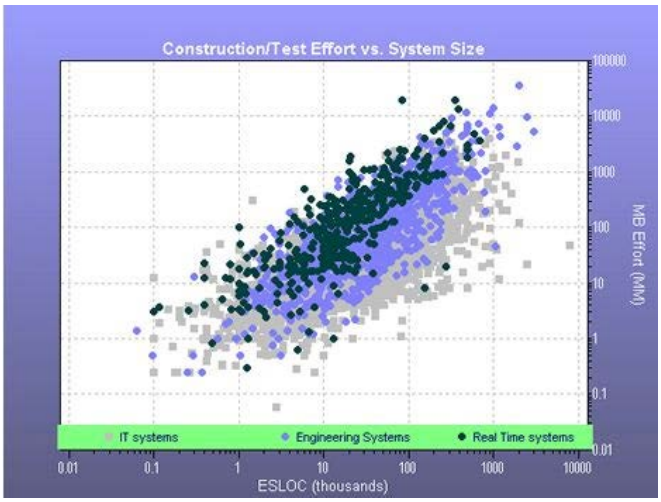


Figure 1. Construction & Test Effort for IT, Engineering, and Real-time Systems

The SLIM-Metrics® chart at Figure 1, above, shows Construction & Test effort for completed IT, Engineering, and Real-time systems as System Size increases. QSM stratifies project data into homogenous subsets to reduce variation and study the behavioral characteristics of different software application domains.

QSM Industry Trend lines

QSM industry trend lines are available for nine, high level application domains, five application subgroups, and three application supergroups. The nine application domains are:

Business	***Command & Control***	***Scientific***
System Software	***Telecommunications***	***Process Control***
Avionics	***Microcode/Firmware***	***Real-time Embedded***

Stratifying the data by application type reduces the variability at each size range and allows for more accurate curve fitting. One application domain, Business IT projects, has been further stratified into several sub-groupings:

Business Agile	***Package Implementation***
Government	***Web Systems***
Business Financial	

Three application supergroups are also available to benchmark projects of mixed or unknown application domains:

Real-time Group	***Engineering Group***
All Systems	

There are several ways QSM clients can access the QSM database:

- **Our SLIM® Tools Contain Current Industry Performance Trends:** QSM provides up to date trends for 17 application types with QSM software project estimation and benchmarking tools.
- **We Answer Your Basic Questions:** Our support team is happy to answer basic questions about the QSM database. Let us do the research to answer your estimation and benchmarking questions! We can provide graphs and summaries that allow you to compare your projects against both industry trend lines and actual projects that are similar in size, application type, and complexity. Basic questions usually involve research that we can accomplish in less than 4 hours. *Note: more extensive research is available through our consulting group.*
- **We Provide Benchmarking Services**: QSM's benchmarking service can help you assess your current levels of productivity and quality, identify exceptional or underperforming projects, analyze the root causes of poor performance and provide a roadmap for improvement, and build a business case for process improvement initiatives. Our benchmark service will position your projects against relevant industry data and measure your variation from the norm.
- **We Provide Client-Directed Research:** QSM can be engaged to perform to research and answer specific questions about performance, cause and effect relationships, or the impact of various tools or practices on software development projects. Current research from our database analysis is also available for download from our website and blog.

---

2. FIVE CORE METRICS

"The price of light is less than the cost of darkness."

– Arthur C. Nielsen, American market analyst who founded the ACNielsen Company

"You don't have to be a mathematician to have a feel for numbers."

– John Forbes Nash, Jr., American mathematician whose works in game theory, differential geometry, and partial differential equations have provided insight into the factors that govern chance and events inside complex systems in daily life

Predictable Change: Flexing the Five Core Levers of Software Development

Dr. Andy Berner

THE MEANING OF THE FIVE CORE METRICS

Software developers resist uniformity. "My project is different," they say. And they are right. As Ken Schwaber and Mike Beedle noted, software development is not very amenable to being managed like a repeatable manufacturing process. If every project is unique, how can we estimate unique projects in a consistent, repeatable way?

As different as they are, all software projects share certain characteristics. Larry Putnam, Sr. and Ware Myers describe "Five Core Metrics" that characterize the performance of software development projects: size, productivity, time (duration), effort, and reliability. These metrics represent five "levers" we can use to embrace and manage project change.

For over 30 years, QSM has studied the relationships among these five levers and has built a large and growing database of information from projects of differing sizes, application types, and development methodologies, to include, most recently, Agile projects.

Time (Duration) and Effort—The "Easy" Ones

Time refers to the calendar duration (in months, weeks, etc.) for the entire project. Effort is the number of person months (work hours, FTE years, etc.) of all the team members on the project. There is nothing "Agile specific" about time, effort, or the units in which they are measured. In practice there can be subtleties such as, "When does the project start and finish?" The notion of "potentially releasable software" and the changing decision of when to release introduces some ambiguity, but these issues are not new; continuous delivery occurred in mainframe development decades ago. In many Agile projects the team is kept constant, and since the major cost in software development is the cost of the people, we may switch between looking at cost, effort, and team size. For both time and effort, we find the basic meaning and measurements are very natural to project managers.

Size—The Measurement of Scope

While it is clear that some projects are "bigger" than others, specific measures of system size are not often used by project managers or development teams. Scope—the requirements, functions, and stories to be developed—is the qualitative version of size and is more natural. Of our five key levers, this is the one that most shows "All projects are different, yet alike." No two software projects will develop exactly the same set of stories but frequently, different sets of stories may be about the same size.

For size-based estimation the details of the project scope are not important. What matters is the size of the proposed project, relative to other projects completed by your organization or, if your own historic data is not available, to relevant industry data.

Over the years, sizing techniques such as function points (see IFPUG website for more information) have been developed, but until recently explicit size metrics were rarely used after the initial estimate. Agile methodologies are changing this. Agile teams routinely measure the size of their backlog in story points or counts of like-sized stories. These units are relative size measures, yet they are still very useful for planning iterations. But if we want to use past projects to predict future performance of entire projects, we must relate "relative size" in the context of one project to "normalized size" that can be used to compare projects.

Productivity

As much a relative measure as size might seem, productivity is even worse. We know some teams are more productive than others, but it's difficult to quantify how much more. We are challenged even to define the term and certainly hard pressed to give productivity a number.

Agile is changing this too. A key Agile metric is "velocity"—the number of story points completed per iteration. To be most useful, velocity depends on two constants that are common to Agile projects: the size of the team remains steady and all iterations in the project are the same length. When both these conditions are true, we can assert that the higher the velocity, the more productive the team.

However, this is true only within a team. If one team has 10 members and another has five, we would expect the larger team to accomplish more in the same time. How much more is an very interesting issue we'll explore later, but the primary use of "team velocity" is within a given project with a fixed team, to predict how much will be accomplished in future (same length) iterations of that project.

QSM uses a more sophisticated metric called "productivity index" (PI) which empirically measures how three of the core metrics (size, effort, schedule) interrelate on different projects. We have found it is essential to factor in this explicit and quantitative measure of productivity to provide useful estimation.

Reliability/Quality

Quality can also appear to be a vague and somewhat subjective metric. There are aspects of quality that can be measured in different ways for different purposes. At QSM we have found that the defect arrival rate (or its reciprocal, Mean Time to Defect during development) is most useful for measuring the effect quality has on the duration and effort needed to complete a project. When quality drops significantly, correcting it can wreak havoc on the best planned and resourced schedule.

Agile methods address the quality lever with some key techniques rather than direct measurements. A goal of Test Driven Development (TDD) is to catch defects on the developer's desk before they get into a build, thus increasing the mean time to defect in builds and reducing the overall project time.

EMBRACE CHANGE: FLEXING THE FIVE CORE LEVERS

No project has complete flexibility, but likewise no organization can truthfully assert, "We will deliver exactly this scope, in this amount of time, for this cost, with zero defects." All projects are constrained in some dimensions, but can be flexible in others. Often the delivery date is the primary constraint, and Agile organizations often plan with the duration lever fixed and meet the deadline by flexibly adjusting how much of the backlog is delivered in the release.

For some projects, the lever of size and scope is less flexible than the others. It wouldn't do much good to deliver an income tax program on time if it handled only deductions but not income. The notion of "Minimum Releasable Scope" is gaining favor among Agile teams who recognize that even if we have working software of "potentially releasable quality," the customer may require more features before we have a viable release.

For other projects, the quality lever is the most inflexible. On a commercial aircraft, failure of the on-board entertainment system is annoying; failure of the navigation system is something else.

Sometimes the lever that is most rigid is effort or team size. Since effort is the major cost driver, this lever may be totally inflexible due to budget constraints. In another situation we may only have a fixed number of developers available, limiting the flexibility of the staffing lever.

Since we embrace change, we don't expect all five levers to stay in a fixed place throughout a project. Some levers are more rigid than others. Which levers are more flexible varies from project to project, so we need tools and methods that let us take varying constraints into account and "solve" for optimal choices where we have the most flexibility.

USING HISTORY TO PREDICT FUTURE PERFORMANCE—UNDERSTANDING THE RELATIONSHIPS AMONG THE LEVERS

The Agile community is learning to predict what a team can do in a single iteration or sprint based on metrics from previous iterations. Constants that exist within a single project (team size, iteration length) help us do this. But how can we use the information from previous projects to decide how to position the five levers at the start of a new project? Can we predict what will be feasible if we are using a new team of a different size building a system of different scope over a different duration?

To move from using the five core levers on a single project to predict the feasibility and performance of a new project, we need two things:

a) Historical data from previous projects for the five core levers (size, effort, duration, defects, productivity) from our organization or from industry data which we can compare to our own performance.

b) The relationships among the levers – how will adjusting one lever affect the others?

The good news is that most Agile projects already capture raw data around these metrics. However, leveraging the relationships among software metrics to help predict future behavior is more challenging.

Let's look at why this isn't easy. We are starting a new project, one that is key to our competitive position. We need to deliver in six months. Our competition is already in the market, so we must at least match them on features and quality. The Minimum Releasable Scope is twice as large as other projects we've done recently, but the project is important enough to put our best people on it. We've collected metrics from previous projects we can use to estimate this one. Two teams stand out—their velocity is consistently high on the projects they've completed. If we put those teams together, their combined velocity should do the trick!

Here's why it's tempting to think this works:

- We compared the new project to previous projects and we flexed the size lever to twice the previous size, so we should expect to lengthen the project.
- If we assume the same velocity from a team, it would double the number of iterations needed for the previous project.
- Unfortunately, we can't afford to double the duration. Instead, we combine two teams that both achieved the same high velocity. Shouldn't this double the expected velocity and bring our schedule back to the six month window?

Unfortunately it is not that easy: the relationships among the levers are not that simple. Adjusting one lever affects the others: usually in the directions we expect, but not in the amount we expect. The relationship between schedule and effort is nonlinear—doubling the team size on a project does not halve the schedule.

The nonlinear tradeoffs that exist between effort and schedule apply to effort and defect creation. In general, increasing team size for a project of a given duration and size lowers the quality. Adding people increases the number of communication paths, which leads to

more defects and thus more time spent correcting them and re-testing the product. The result is we dramatically raise the total effort and cost. We see the same nonlinear tradeoffs when we adjust project size— the larger scope increases project duration, but this time the tradeoffs work in our favor: doubling the project size (keeping team size, productivity, and quality fixed), lengthens the schedule, but doesn't double it.

The simple reasoning that led us to believe that combining these teams will let us meet the schedule doesn't work. We need tools that model the complex relationships among the five core levers, let us specify which constraints will be most stiff on a particular project, and bend the more flexible levers to get feasible plans that fit our historical capabilities.

LEVELS OF CONFIDENCE AND RISK

The question we should ask during initial planning is not just "How long will it take?" Better questions would be: "How likely is it that we can make our schedule? How likely is it we will meet this cost constraint? How likely is it that our team size is sufficient?" The questions should include this qualifier: "How likely?"

Likewise, the answer should not be "12 months." It should include the risk. "It's likely we can do this in 12 months. Planning for 14 months would be very conservative. Ten months Is plausible, but quite risky. But there is no way it will get done in 6 months. Not only has our team never done that much that fast, but nobody in the industry has!" Quantifying the risk and expressing the level of confidence allows us to keep all levers as flexible as possible. We can plan for contingencies, allowing the most flexible levers to adjust as the project is carried out so we meet our constraints.

How can we predict what it takes to deliver a new project? We can use our historical data from the five core levers as a starting point. We can account for unique inputs and constraints from the project we are estimating. We can use tools that account for the nonlinear relationships among our core levers and adjust the more flexible ones in the right proportions. We make uncertainty an explicit part of the estimate. Our estimates will reflect the level of risk we can accept and allow us to plan for the changes we know are coming.

Works Cited

Schwaber, Ken, and Mike Beedle, *Agile Software Development with Scrum*, Upper Saddle River: Prentice-Hall, Inc., 2002. Print.

Putnam, Lawrence H., and Ware Myers, *Five Core Metrics—The Intelligence behind Successful Software Management*, New York: Dorset House Publishing Company, Inc., 2002. Print.

They Just Don't Make Software Like They Used to… Or Do They?

Taylor Putnam

In this article, I thought I'd take a moment to share the updated QSM Default Trend Lines and how they affect your estimates. I will focus on the differences in quality and reliability between 2010 and 2013 for the projects in our database. Since our last database update, we've included over 200 new projects in our trend groups.

Here are the breakouts of the percent increases in the number of projects by application type:

- Business Systems: 14%
- Engineering Systems: 63%
- Real-time Systems: 144%

Figure 1 (on the next page) is an infographic outlining some of the differences in quality between 2010 and 2013.

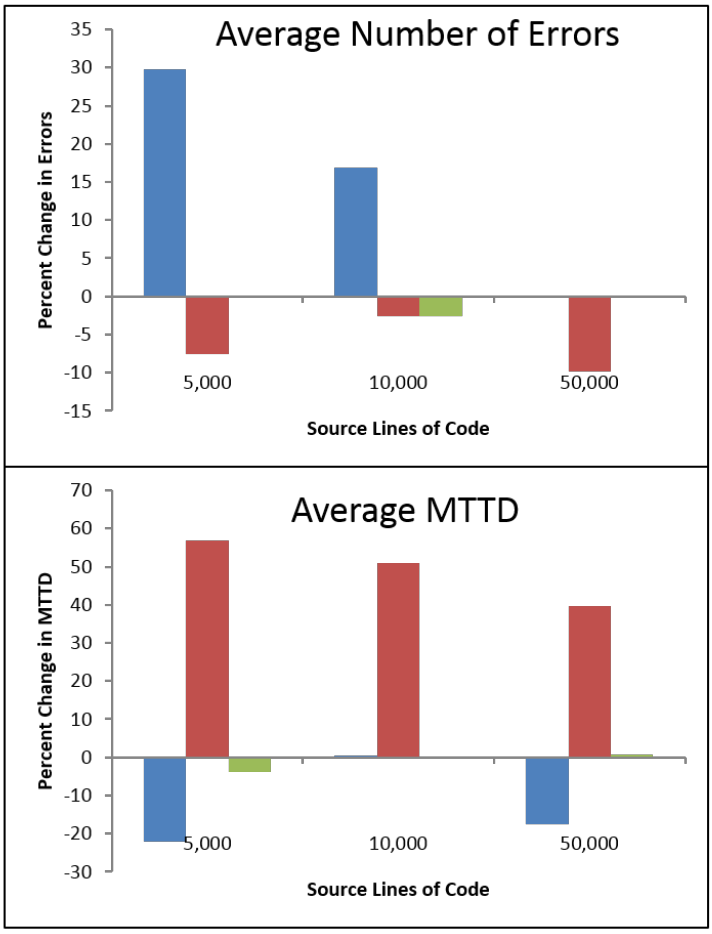
From the set of charts, we can see some trends emerging which could indicate the changes in quality between 2010 and 2013. By looking at the data, it's apparent that two distinct stories are being told:

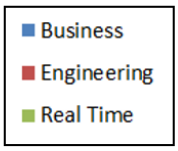
1. The Quality of Engineering Systems Has Increased

Overall, the Engineering Systems showed a decrease in reported errors and an overwhelming increase in Mean Time to Defect (MTTD). Although they had a slight increase in project size, with a greater number of projects that were larger than 5,000 SLOC, Engineering projects also decreased their team sizes and schedule durations. Using smaller teams can drastically decrease potential miscommunications among developers, thus reducing the overall number of errors generated. With fewer errors present in the system, the amount of rework is minimized which, in turn, reduces the overall schedule duration. Additionally, fewer errors present at the ship date also lead to a higher MTTD and quality rating.

They just don't make software like they used to…
Or do they?

Changes in Reported Project Quality between the 2010 and 2013 Trend Lines

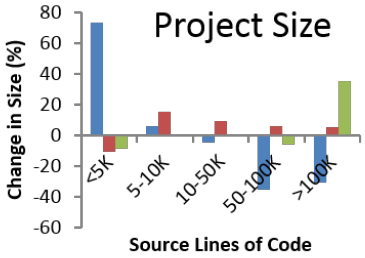




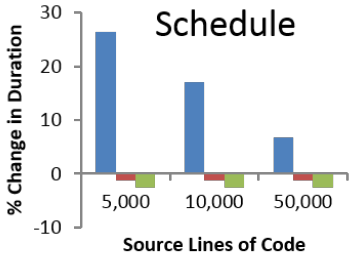
Quick Facts:

- Overall, Engineering systems are reporting fewer errors in the 2013 trends.
- Engineering projects had significant increases in Mean Time to Defect between 2010 and 2013.
- Real Time Systems remained relatively constant in reported errors and MTTD between 2010 and 2013.
- Business Systems reported more errors at 5K and 10K SLOC in 2013 than in 2010.
- Business Systems had a lower Mean Time to Defect at 5K and 50K SLOC in 2013 than in 2010.

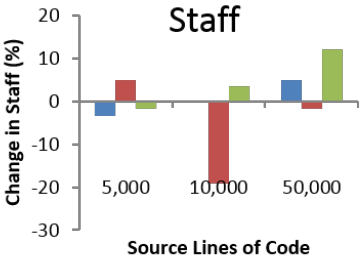
How do the other metrics support this finding?



Business systems appear to be building more projects at smaller sizes. Engineering sand Real Time systems are shifting to slightly larger project sizes.



Business projects are taking longer to deliver software, whereas Engineering and Real Time systems are taking less time.



At 10,000 SLOC Engineering projects are using fewer staff, thus decreasing the chance of rework created by communication issues.

Figure 1. Software Quality Differences: 2010-2013

2. The Quality of Business Systems Has Decreased

On the other hand, Business Systems showed an increase in errors generated and a decrease in MTTD. With more Business Systems shifting their project scopes to smaller sizes (<5,000 SLOC), likely due to the popularity of Agile methods, it's natural to see the decrease in staff size at 5,000 SLOC, because there is less work to be done. However, Business Systems also reported more errors and lower MTTDs than in 2010, yet they're taking longer to deliver. It's possible that one of the reasons we're seeing this is that projects are decreasing their functionality (perhaps to help meet a schedule deadline), and do not have enough time to properly test the software for defects. Consequently, their first ship date might therefore occur at the height of defect arrival rates, before developers have had time to correct the bugs. This ultimately results in extending the project schedule to allow for defect removal later on in the project lifecycle.

So what does this mean for your projects?

While it's good to examine the error counts or MTTD's individually, you also need to look at how that affects your project holistically. In this dataset it appeared that having an increase in errors was related to a decrease in quality. While that may be true, that is not necessarily the only outcome of a high number of errors present. If you found that you had a high number of reported errors but also a high MTTD, it could indicate that developers are finding more errors in testing and **fixing** them before shipment, thus indicating a good quality project. Therefore, it is important to examine how all these metrics could affect each other. With development methodologies shifting to smaller scopes with shorter schedules and smaller staff sizes, it's more important than ever to collect defect data on your projects. Meeting the goal of a shorter schedule deadline may be good, but if it it's at the expense of the projects quality, is that really a desired outcome? In short, collecting defect data and analyzing its effects can help indicate whether or not your organization is moving in its desired direction.

Data-driven Estimation, Management Lead to High Quality

Kate Armel

This article originally appeared in the American Society for Quality's ***Software Quality Professional****, volume 15 (March 2013), pp 25-47, and is reprinted here with permission.*

Quality assurance comprises a growing share of software development costs. To improve reliability, projects should focus as much effort on upfront planning and estimation as they do on remedial testing and defect removal. Industry data show that simple changes like using smaller teams, capturing average deviations between estimates and actuals and using this information as an explicit input to future estimates, and tuning estimates to an organization's historical performance result in lower defect creation rates. Access to accurate historical data helps projects counter unrealistic expectations and negotiate plans that support quality instead of undermining it.

INTRODUCTION

Software projects devote enormous amounts of time and money to quality assurance. A recent study found that roughly 30 percent of software developers believe they release too many defects and lack adequate quality assurance (QA) programs. A stunning one-quarter of these firms do not conduct formal quality reviews at all (Seapine). Despite these holdouts, the National Institute of Standards and Technology (NIST) estimates that about half of all development costs can be traced back to defect identification and removal (NIST 36).

Unfortunately, most QA work is remedial in nature. It can correct problems that arise long before the requirements are complete or the first line of code has been written, but has little chance of preventing defects from being created in the first place. By the time the first bugs are discovered, too many projects have already committed to fixed scope, staffing, and

schedule targets that fail to account for the complex and nonlinear relationships between size, effort, schedule, and defects.

Despite the best efforts of some very hardworking and committed professionals, these projects have set themselves up for failure. But it doesn't have to be that way.

Armed with the right information, managers can graphically demonstrate the tradeoffs between time to market, cost, and quality, and negotiate achievable deadlines and budgets that reflect their management goals. Over the last three decades, QSM has collected performance data from more than 10,000 completed software projects. The company uses this information to study the interactions between core software measures like size, schedule, staffing, and reliability. These nonlinear relationships have remained remarkably stable as technologies and development methods come and go. What's more, these fundamental behaviors unite projects developed and measured across a wide range of environments, programming languages, application domains, and industries. The beauty of completed project data is that they establish a solid, empirical baseline for informed and achievable commitments and plans. Proven insights gained from industry or internal performance benchmarks can help all organizations achieve their management goals. Over the last three decades, QSM has collected performance data from more than 10,000 completed software projects. The company uses this information to study the interactions between core software measures like size, schedule, staffing, and reliability. These nonlinear relationships have remained remarkably stable as technologies and development methods come and go. What's more, these fundamental behaviors unite projects developed and measured across a wide range of environments, programming languages, application domains, and industries. The beauty of completed project data is that they establish a solid, empirical baseline for informed and achievable commitments and plans. Proven insights gained from industry or internal performance benchmarks can help even the most troubled firms reduce cost and time to market and improve quality. The best performers in the industry already know and do these things:

- Capture and use "failure" metrics to improve future estimates rather than punishing estimators and teams
- Keep team sizes small
- Study their best performers to identify best practices
- Choose practical defect metrics and models
- Match reliability targets to the mission profile

If these practices were as simple to implement as they sound, every project would be using them. But powerful incentives and competing interests that plague projects can present formidable barriers to effective project and quality management.

UNCERTAINTY IS A FEATURE, NOT A BUG

How good is the average software development firm at meeting goals and commitments? One frequently cited study—the Standish Group's Chaos Report—found that only one-third

of software projects deliver the promised functionality on time and within budget (The Standish Group). A more current study of recently completed software projects (Beckett) points to one problem: While nearly all of the projects in the QSM database report actual schedule, effort, and size data, only one-third take the next step and formally assess their actual performance against their estimated budget, schedule, or scope targets.

Of the projects that reported over- or under-run information, a significant number overran their planned budget or schedule by 20 percent or more (see Figure 1). Projects were significantly more willing to overrun their schedules or budgets than they were to deliver less functionality.

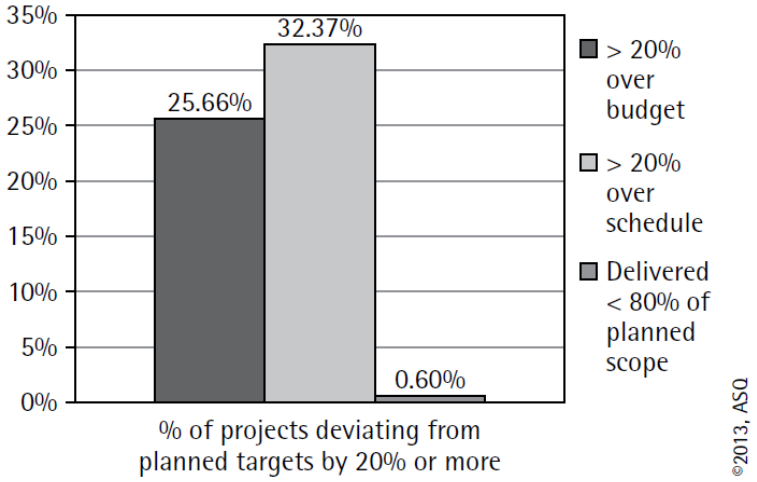


Figure 1. Project Failure

It's surprising—and a bit disconcerting—to see so many projects failing to meet their goals. These are companies that have formal metrics programs in place. They have invested in state-of-the-art tools, processes, and training. So why aren't they using project actuals to improve the accuracy of future estimates? Why do so many projects continue to overrun their planned schedules and budgets? One possible answer may lie in the way companies typically use—and misuse—estimates.

Initial estimates are often required to support feasibility assessments performed very early in the project lifecycle. Thus, they are needed long before detailed information about the system's features or architecture is available. The exact technology mix, schedule, team size, required skill set, and project plan have rarely been determined at the time the estimate is requested. This timing problem creates a fundamental mismatch between the kind of detailed estimate needed to win business and price bids competitively and the information that exists at that point in time.

When not much is known about the project, risk and uncertainty are high. Later on as design and coding are under way and detailed information becomes available, reduced uncertainty about estimation inputs (size, staffing, productivity) translates to less risk

surrounding the final cost, schedule, and scope. Changing uncertainty and risk over the project lifecycle are demonstrated in Figure 2 (Armour 13-16).

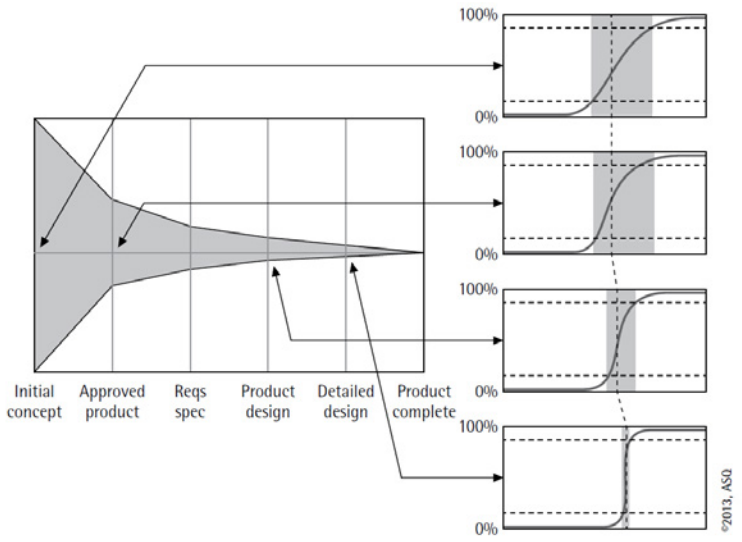


Figure 2. Commitments Early in the Lifecycle Must Account for Greater Uncertainty Surrounding Estimation Inputs

Unfortunately, most organizations don't have the luxury of waiting for all the details to be nailed down before they submit competitive bids. To keep up with the competition, they must make binding commitments long before accurate and detailed information exists.

This dilemma illustrates the folly of misusing estimation accuracy statistics. Measurement standards that treat an estimate as "wrong" or a project as "failed" whenever the final scope, schedule, or cost differ from their estimated values effectively punish estimators for something outside their control: the uncertainty that comes from multiple unknowns. Estimates—particularly early estimates—are inherently risky. In the context of early estimation, uncertainty is a *feature,* not a bug. Uncertainty and consequent risk can be minimized or managed, but not eliminated entirely.

Does that mean one shouldn't track overruns and slippages at all? Absolutely not. In fact, it's vital that projects capture deviations between estimated and actual project outcomes *because this information allows them to quantify a crucial estimation input (risk) and account for it explicitly in future estimates and bids. If measurement becomes a stick used to punish estimators for not having information that is rarely available to them, they will have little incentive to collect and use metrics to improve future estimates.*

USE UNCERTAINTY TO IMPROVE (NOT DISCOURAGE) ESTIMATION

Looking only at project "failures," however defined, can easily lead to the conclusion that efforts to improve the quality of estimates are a waste of time and resources. But successful estimation that actually promotes better quality, lower cost, and improved time to market generally requires only a small shift in focus and a little empirical support.

This is where access to a large historical database can provide valuable perspective and help firms manage the competing interests of various project stakeholders. Sales and marketing departments exist to win business. They are rewarded for bringing in revenue and thus have a vested interest in promising more software in less time than their competitors. Developers long for schedules that give them a fighting chance to succeed and access to the right skill sets at the right time. And, of course, clients want it all: lots of features and a reliable product, delivered as quickly and cheaply as possible. But if development firms are to stay in business and clients and developers are to get what they want, the balance between these competing interests *must* be grounded in proven performance data. Winning a fixed price contract to build a 500,000 line of code system in 10 months isn't a good idea if the organization has never delivered that much software in less than 18 months.

Without historical data, estimators must rely on experience or expert judgment when assessing the impact of inevitable changes to staffing, schedule, or scope. While the wisdom of experts can be invaluable, it is difficult to replicate across an enterprise. Not everyone can be an expert, and concentrating knowledge in the hands of a few highly experienced personnel is not a practice that lends itself to establishing standardized and repeatable processes. Without historical data, experts can *guess* what effect various options might have, but they cannot empirically demonstrate why adding 10 percent more staff is effective for projects below a certain threshold but usually disastrous on larger projects. They may suspect that adding people will be more effective early in the lifecycle than toward the end, but they can't show this empirically to impatient senior managers or frustrated clients who want instant gratification. Solid historical data allow managers and estimators to demonstrate cause and effect. They remove much of the uncertainty and subjectivity from the evaluation of management metrics, allowing estimators and analysts to leverage tradeoffs and negotiate more achievable project plans.

A CASE STUDY

The preceding studies show what happens when firms get estimation wrong. What happens when software development firms get estimation right—when they capture and use uncertainty data as an explicit input to new estimates? The experience of one organization, a global telecommunications giant, should serve as a powerful antidote to depressing industry statistics about failing projects (Putnam and Myers):

> "In the year before using SLIM®, 10 of 12 projects (83 percent) exceeded budget and schedule. The cost of this excess was more than $15 million. QSM was hired to implement a formal estimation process.
>
> "Within the first year of using a SLIM®-based approach, the percentage of projects over schedule/budget decreased from 83 percent to 50 percent—with cost overrun reduced from $15 million to $9 million.

"After full implementation of SLIM® in the second year, the percentage of projects over schedule/budget dropped to 10 percent and the cost overruns were less than $2 million."

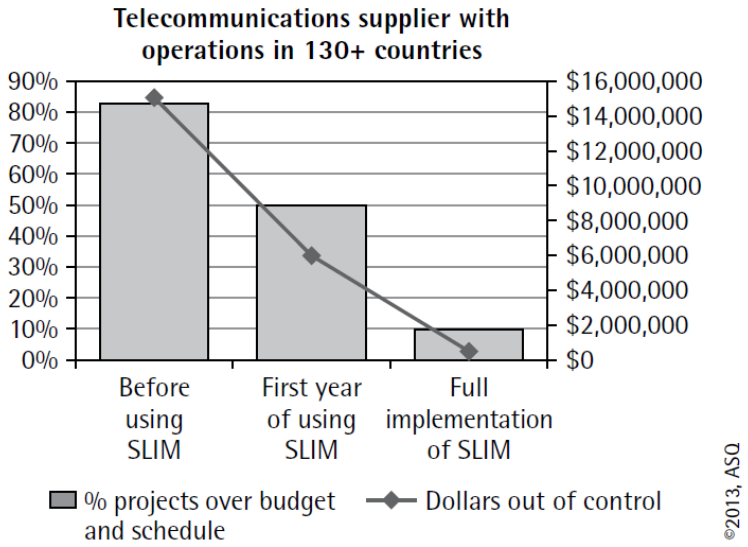


Figure 3. Case Study Source: *Five Core Metrics: The Intelligence behind Successful Software Management*

Like many developers, the organization shown in Figure 3 wasn't unaware of recommended industry best practices. But the key to overcoming objections to effective project management proved to be historical and industry data. Armed with the *right* information, they were able to counter unrealistic expectations and deliver better outcomes.

TO IMPROVE QUALITY, TRY SMALLER TEAMS

When projects do sign up to aggressive or unrealistic deadlines, they often add staff in the hope of bringing the schedule back into alignment with the plan. But because software development is full of nonlinear tradeoffs, the results of adding staff can be hard to predict. More than 30 years of research show that staffing buildup has a particularly powerful effect on project performance and reliability.

To demonstrate this effect, the author recently looked at 1,060 IT projects completed between 2005 and 2011 to see how small changes to a project's team size or schedule affect the final cost and quality (Armel 16-22). Projects were divided into two staffing bins:

- Small teams (four or fewer FTE staff)
- Large teams (five or more FTE staff)

The bins span the median team size of 4.6, producing roughly equal samples covering the same range of project sizes. For large team projects, the median team size was 8.5. For small team projects, the median team size was 2.1 FTE staff. The ratio of large to small team size along the entire size spectrum is striking: approximately 4 to 1.

The wide range of staffing strategies for projects of the same size is a vivid reminder that team size is highly variable and only loosely related to the actual work to be performed. Because the relationship between project size and staff is exponential rather than linear, managers who add or remove staff from a project should understand how the project's position along the size spectrum will affect the resulting cost, quality, and schedule.

The author ran regression trends through the large and small team samples to determine average construct and test effort, schedule, and quality at various project sizes (see Figure 4). For very small projects, using larger teams was somewhat effective in reducing schedule. The average reduction was 24 percent (slightly over a month), but this improved schedule performance carried a hefty price tag: project effort/cost tripled and defect density more than doubled.

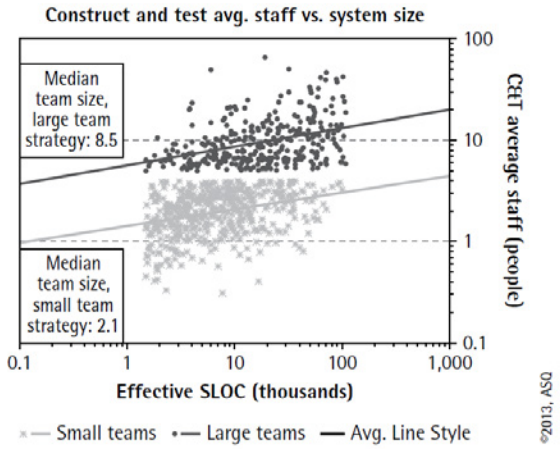


Figure 4. Regression Fits for Average Staff vs. System Size (Large and Small Team Samples)

For larger projects (defined as 50,000 new and modified source lines of code), the large team strategy shaved only 6 percent (about 12 days) off the schedule, but effort/cost quadrupled and defect density tripled. The results are summarized in Table 1 (Armel 16-22).

5K ESLOC	Schedule (months)	Effort (person hours)	Defect density (defects per K ESLOC)
Small teams	4.6	1260	3.7
Large teams	3.5	4210	9.2
Difference (large team strategy)	*-24%*	*334%*	*249%*
50K ESLOC			
Small teams	7	3130	1.2
Large teams	6.6	13810	3.9
Difference (large team strategy)	*-6%*	*441%*	*325%*

Table 1. Large Team Strategy Increased Effort Expenditure (334-441%) and Defect Creation (249-325%) while Yielding Only Marginal Reductions to Schedule

The relative magnitude of the tradeoffs between team size and schedule, effort, and quality is easily visible by inspection of Figure 5.

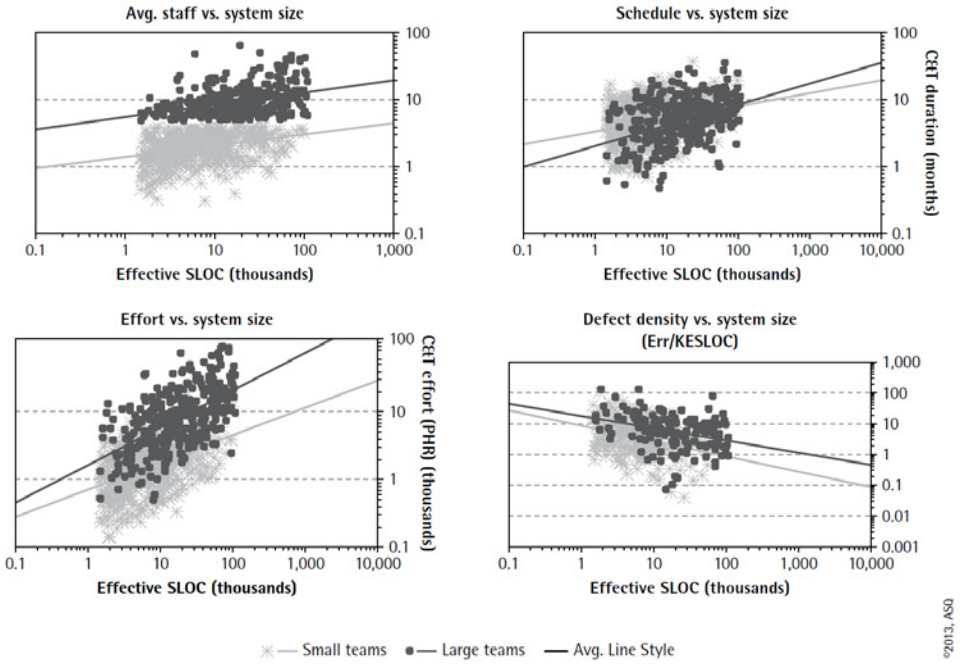


Figure 5. Larger Teams: Little Impact on Schedule; Large Impact on Effort Expenditure and Defect Density

Large teams achieve only modest schedule compression (note the large overlap between large and small team projects in the top right-hand "Schedule" chart) while causing dramatic increases in effort and defect density that increase with project size. This shows up in the relatively wider gap between the effort vs. size and defect density at the large end of the size (horizontal) axis.

For firms that understand the importance of staffing as a performance driver and are willing to use that knowledge, the benefits can be impressive. Recently the company worked with an organization with nearly 50,000 employees and a budget of more than $25 billion. The director of enterprise systems development was tasked with overseeing a daunting number of programs and contractors: more than 1,000 application and system vendors fell within his purview. He asked QSM to review the project portfolio and determine the staffing levels required to deliver the agreed-upon functionality within the required timeframe. The research team demonstrated that a 50 percent reduction in staff/cost would result in minimal schedule extension. The agency acted swiftly, reducing average head counts from 100 to 52. The results were dramatic: Cost fell from $27 million to $15 million and schedule increased by only two months.

BEST-IN-CLASS PERFORMERS USE SMALLER TEAMS

Studying large samples or individual case studies is one way to assess the influence of staff on project performance. Another way involves identifying the best and worst performers in a group of related software projects, then comparing their characteristics. The *2006 QSM IT Software Almanac* (QSM 2006) performed this analysis using more than 500 completed IT projects. The study defined best-in-class projects as those that were 1σ (standard deviation) better than average for both effort and time to market. Conversely, worst-in-class projects were 1σ worse than average for the same two variables. Another way to visualize this is that best-in-class projects were in the top 16 percent of all projects for effort and schedule, while worst-in-class projects fell in the bottom 16 percent for both measures (QSM, Inc.):

> "Staffing was one area where best and worst projects differed significantly. Conventional wisdom has long maintained that smaller teams deliver more productive projects with fewer defects. The data confirmed this. Figure 6 shows average staff for best- and worst-in-class projects. The median team size was 17 for the worst and four for the best-in-class projects. Looking at the average team size, the trends for the two datasets run parallel with the worst projects using 4.25 times as many staff.
>
> "Strikingly, only 8.8 percent of the best-in-class projects had a peak staff of 10 or more (the maximum was 15), while 79 percent of the worst projects did. This underscores an interesting finding: Visual inspection of [Figure 6] appears to suggest that, at any given project size, larger team size is not a characteristic of more productive projects. That they don't do well on cost efficiency is not surprising; after all, they use more people, and that costs money.
>
> "The more interesting finding is that they don't appear to do any better on speed of delivery either!"

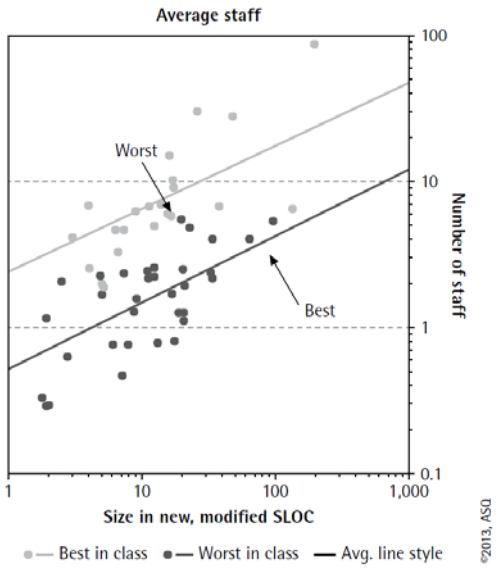


Figure 6. Worst-in-Class Projects: Average of 4.25 as Many Staff at Same Project Size as Best-in-Class

How much better did best-in-class projects perform against both average and the worst performers? The differences, shown in Figure 7, are striking.

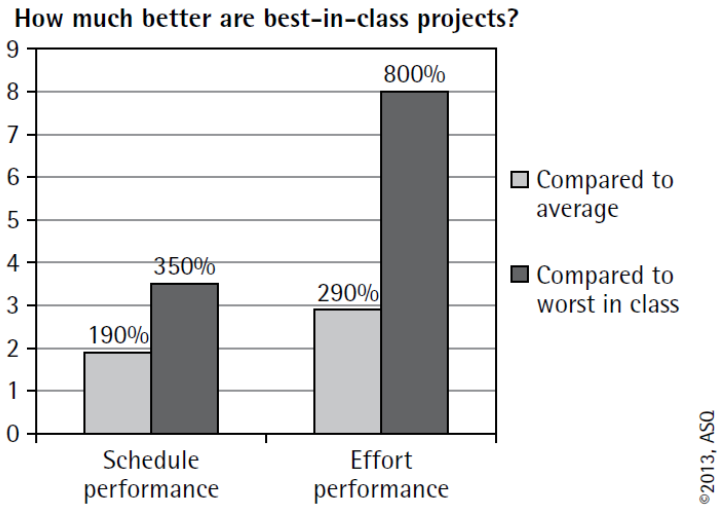


Figure 7. Best-in-Class Project Performance vs. Average and Worst-in-Class

This analysis was repeated in 2013 using a sample of 300 engineering class projects, and the results were consistent with the IT best-in-class study. On average, worst-in-class projects used four times as much staff and took three times longer to complete than the best-in-class projects. These results underscore the author's belief that using the smallest practical team can result in significant improvements to cost and quality with only minimal schedule impact.

BEST-IN-CLASS QUALITY

QSM focuses on two primary quality measures: prerelease defects and mean time to defect (MTTD). MTTD represents the average time between discovered errors in the post-implementation product stabilization period. Interpreting defect metrics in isolation can be anything but a straightforward task. Without the right contextual data, it can be difficult to know whether high pre-delivery defect counts indicate poor quality or unusually effective defect identification and removal processes. For this reason, the company combines several reliability metrics to predict and assess quality at delivery. The 2006 Almanac found that IT best-in-class projects were far more likely to report defects (53 percent) than the worst-in-class projects (21 percent) (QSM, Inc.). Quality comparisons were hampered by the fact that so few worst performers provided defect counts. Engineering projects display the same characteristics, but the quality reporting disparity between best and worst performers is even more dramatic: (Beckett)

> "Eighty-five percent of the best-in-class projects reported pre-implementation defects while only 10 percent (one project) of the worst in class did so. Of the remaining engineering projects (ones that are neither best nor worst in class) 62 percent reported prerelease defects. Overall, the best-in-class projects reported

fewer prerelease defects than their engineering peers, but the small number of projects makes accurate quality comparisons impossible.

"Defect tracking isn't just a 'nice to have' addition to software development. It provides critical information about the quality of the product being developed and insight into which management practices work well and which require improvement. That so few worst-in-class projects formally reported defects suggests organizational process problems."

CHOOSING A PRACTICAL AND REPEATABLE DEFECT PREDICTION MODEL

Once a project is under way and the first defect counts begin to roll in, what is the most effective way to use that information? The best method will involve metrics that are easy to capture and interpret and allow the project to produce reliable and repeatable reliability forecasts. The methods outlined in this article have been in use for more than three decades and have worked well for organizations at all levels of process maturity and development capability.

Defect prediction models can be broadly classified as either static or dynamic. Both have advantages and may be useful at various points in the lifecycle. *Static models* use final defect counts from completed projects to estimate the number of defects in future projects. *Dynamic models* use actual defect discovery rates over time (defects per week or month) from an ongoing project to forecast.

Research performed by Lawrence H. Putnam, Sr. (Putnam and Myers) shows that defect rates follow a predictable pattern over the project lifecycle. Initially, staffing is relatively low and few project tasks have been completed. Defect creation and discovery increase or decrease as a function of effort and work completion. As people are added to the project and the volume of completed code grows, the defect discovery rate rises to a peak and then declines as work tails off and the project approaches the desired reliability goals. This characteristic pattern is well described by the Weibull family of curves (which includes the Rayleigh model used in SLIM®) (Figure 8).

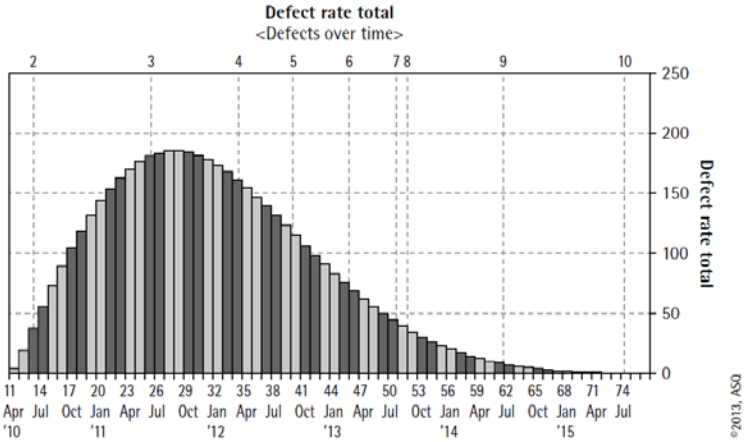


Figure 8. Defect Rates over Time Follow a Rayleigh Distribution

Time-based models offer several advantages over static defect prediction methods. Both static and dynamic models predict total defects over the lifecycle, but time-based models provide a more in-depth view of the defect creation and discovery process. The incorporation of actual defect discovery rates allows managers to estimate latent defects at any given point in time. By comparing reliability at various stages of the lifecycle to the required reliability and mission profile, they can tell whether testers are finding enough defects to avoid delivering a bug-ridden product.

Defect rates have another useful aspect; they can be used to calculate the MTTD. MTTD is analogous to Mean Time to Failure. It measures reliability from the user's perspective at a given point in time, typically when the system is put into production for the first time. Though more complicated methods exist, it can be calculated quickly simply using the following formula (QSM, Inc.):

> "To calculate MTTD, take the reciprocal of the number of defects during this month and multiply by 4.333 (weeks per month) and the days per week for the operational environment. For example, if there were five errors during the first month of operation and the system runs seven days per week, the average MTTD value would be (1/5) * 4.333 * 7 = 6.07 days between defects. If there are no defects in the first month, the MTTD in the first month cannot be calculated."

MTTD makes it possible to compare the average time between defect discoveries to the software's required mission profile and predict when the software will be reliable enough to be put into production. Mission-critical or high-reliability software should have a higher Mean Time to Defect than a typical IT application. Software that must run 24 hours a day and seven days a week requires a higher Mean Time to Defect than software that is only used for eight hours a day from Monday to Friday. MTTD considers all of these factors explicitly.

UNDERSTANDING SIZE, STAFFING, PRODUCTIVITY, AND DEFECTS

If software were more like traditional manufacturing, estimation and reliability prediction would be far more straightforward. A manufacturer of widgets focuses on the repeatable mass production of identical products. Since the same tasks will be performed over and over, linear production rates for each type of widget can be used to estimate and allocate resources. Software development is different in that each new project presents a new type of "widget." The design from a completed project cannot be applied to a new project that solves a completely different set of problems. Instead, project teams must devise new technical solutions via iterative cycles that involve considerable trial and error, evaluation, and rework (feedback cycles).

Though software development does not produce the identical widgets described in the manufacturing example, even software projects share certain characteristics and behaviors. Over the past three decades these similarities have proven useful in managing and improving the software development process. Larry Putnam's (Putnam and Myers) research identified three primary factors that drive defect creation:

- The size (new and modified code volume) of the delivered software product
- Process productivity (PI)
- Team communication complexity (staffing levels)

As the size and staff increase, it makes sense that total defects would increase as well. But these relationships are nonlinear: a 10 percent increase in code or people does not translate to 10 percent more defects. As Figure 9 shows, defects increase with staffing and code volume. Moreover, the slope of the defects vs. staff curve is steeper than that of the defects vs. size curve. This supports the earlier observation that staffing is one of the most powerful levers in software development. When staff buildup is rationally related to the work to be performed, productivity and quality are maximized. Adding more staff than needed, or adding staff too early or too late in the process, complicates team communication and leads to additional defects and increased rework.

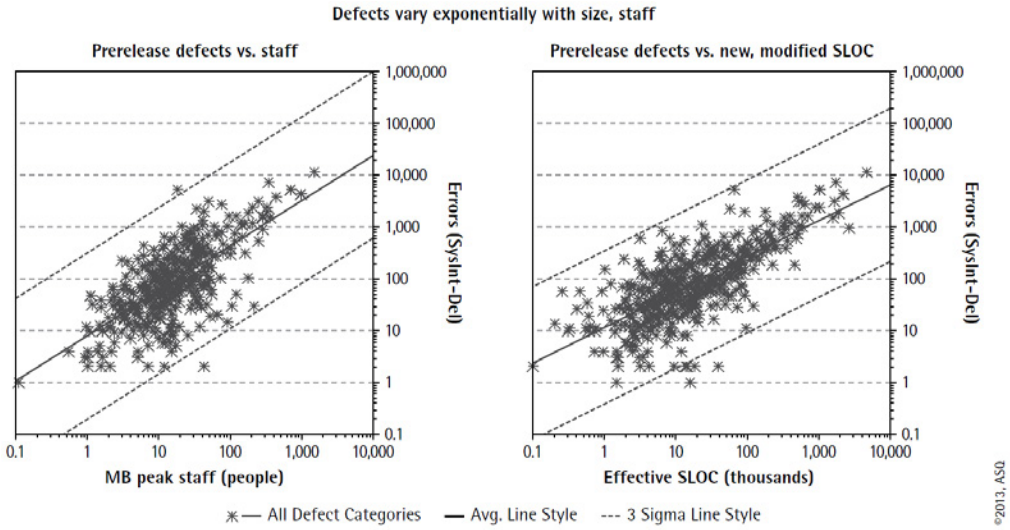


Figure 9. Defects Increase Exponentially with FTE Staff and System Size

The relationship between developed size and defects is complex and nonlinear because other factors also affect defect creation. One of these factors is team or process productivity. QSM uses the productivity index (or PI) as a macro measure of the total development environment. The PI reflects management effectiveness, development methods, tools, techniques, the skill and experience of the development team, and application complexity. Low PIs are associated with poor tools and environments, immature processes, and complex algorithms or architectures. High PI values are associated with mature processes and environments, good tools, effective management, well-understood problems, and simple architectures.

Figure 10 shows the relationship between productivity (PI) and defect density. As the PI increases, defect density for a given project declines exponentially.

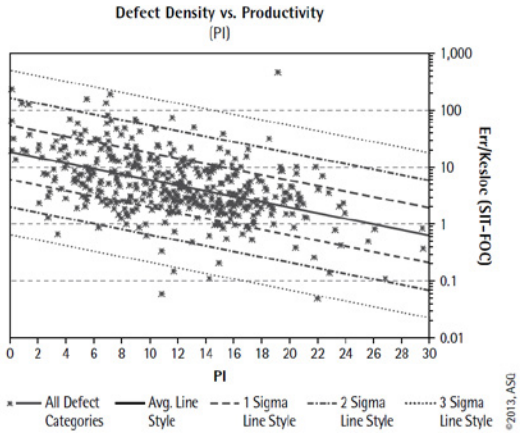


Figure 10. Defect Density Declines as Project Productivity Increases

This relationship is important, but productivity cannot be directly manipulated as easily as other inputs to the development process. It tends to improve slowly over time. The most important (and easiest to control) defect driver is people. It is no accident that the Rayleigh defect curve follows the same general pattern as the project staffing curve. As more people are introduced, the number of communication paths multiplies. More communication paths between people cause misunderstandings and miscommunication, which eventually show up in the software as defects.

CHOOSING EFFECTIVE DEFECT METRICS

What are the best defect metrics for organizations that want to capture reliability from completed or in-progress projects and use that information to improve future defect estimates? A good defect metric should be flexible and easy to use. It should model the defect creation process accurately. It must predict total defects and allow managers to forecast reliability throughout the project lifecycle. Finally, it should allow comparisons between observed and required reliability. Defect creation does not increase linearly with project size. This is true because size is not the only factor driving defect creation (nor is it even the most important driver). Productivity and staffing have a greater effect on the final number of defects in a system than size.

Used in isolation or as the sole basis for defect estimates, ratio-based metrics like defect density (defects/KLOC) do not adequately reflect the nonlinear impact of size, productivity, or staffing on defect creation. Because both the numerator (defects) and denominator (KLOC) change at different rates over time, the resulting numbers can be tricky to interpret. Ratio-based metrics imply a constant increase in defects as the volume of completed code increases, but real defect discovery data shows a nonlinear, Rayleigh defect curve with characteristic find and fix cycles as effort is alternately focused on discovering, fixing, and retesting defects.

The jagged "find and fix cycles" shown in Figure 11 typically smooth out once the project reaches system integration and test and product reliability begins to stabilize. For all of these

reasons, a straight-line model is poorly suited to measuring highly variable defect data that are exponentially related to core metrics like size, staff, and productivity.

Unlike ratio-based metrics, defect rates are relatively simple to interpret. They can be estimated and monitored over time, they accurately reflect the status of the project as it moves through various development stages, and they make it possible to calculate more sophisticated reliability metrics like MTTD that explicitly account for the required reliability at delivery.

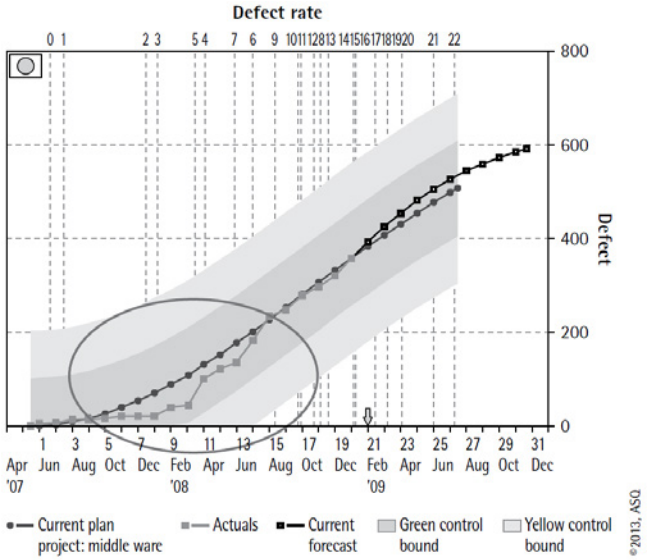


Figure 11. Prior to System Integration and Test, Actual Defect Data often Erratic with Jagged "Find and Fix" Cycles; Near Delivery, Defect Discovery Typically Settles Down and Closely Tracks Smooth Rayleigh Curve

MATCHING RELIABILITY STANDARDS TO THE MISSION PROFILE

How should organizations determine the right reliability standard for each project? A good place to start is by asking, "What does the software do?" and "How reliable do we need it to be?" Defect rates, taken in isolation, aren't terribly helpful in this regard. Software developers need to know how long the software should run in production before users encounter defects of various severities.

MTTD can be customized to reflect the project's unique mission profile. The mission profile, in turn, can depend on a variety of factors. Required reliability for onboard navigation software aboard a fighter jet may depend on how long it can stay in the air before refueling. For other types of software, human considerations like time on shift or time until a unit is relieved determine the required reliability. Different types of software will have different mission profiles. A flight management system may be operational 24 hours per day, seven days per week, 60 minutes per hour, and 60 seconds per minute. A billing application for a doctor's office, on the other hand, may only be required to operate eight hours a day, five days a week. Because the flight system operates continuously for a longer period of time, it requires a higher reliability (or MTTD).

Finally, MTTD can be calculated for total defects or customized to reflect only high-priority defects. End users may not care how long the system runs between cosmetic errors but for mission-critical applications, increasing the average time between serious and critical errors can literally mean the difference between life and death.

SAMPLE MISSION PROFILES

MTTD can be calculated in seconds, minutes, hours, days, or weeks. It can also be customized to reflect all defect categories (cosmetic, tolerable, moderate, serious, and critical) or a subset of defect categories (serious and critical, for example). In the examples that follow, only serious and critical defects were used to calculate the MTTD.

In Table 2, sample mission profiles have been identified for various engineering class applications. Engineering class systems include process control, command and control, scientific, system software, and telecommunications products. As the figure makes clear, acceptable defect discovery rates will vary depending on the application's mission profile. Safe operation of a flight navigation system requires it to run for six days between discoveries of a serious or critical defect, while a cruise missile requires an average reliability of only 45 minutes. For each mission profile, the monthly defect discovery rate that matches the desired reliability target has been calculated. The cruise missile system will be "reliable enough" when the defect discovery rate reaches 67 defects per month, but the flight management system must meet a far stricter threshold of four defects per month.

Application	Mission bin	Desired mission profile	Corresponding defect rate
Smart munitions	Seconds	30 seconds	67
Cruise missile	Minutes	45 minutes	67
X system	Hours	20 hours	31
Flight navigation system	Days	6 days	4
Y system	Weeks	1.2 weeks	3

Table 2. Appropriate Defect Rate Depends on Desired Mission Profile

Once again, factoring in the mission profile demonstrates the importance of context to metrics analysis. It allows one to see that a defect rate of eight defects a month has little meaning in isolation. Once the mission profile and required reliability have been taken into account, managers are in a position to make more informed decisions. Data (raw defect counts) have become information (MTTD and an appropriate target defect rate).

CONCLUSION

Regardless of which estimation and quality assurance practices are used, recognizing and accounting for the uncertainties inherent in early software estimates is essential to ensure sound commitments and achievable project plans.

The competing interests of various project stakeholders can create powerful disincentives to effective project management. Measures of estimation accuracy that punish estimators for being "wrong" when dealing with normal uncertainty cloud this fundamental truth and discourage honest measurement. For all of these reasons, deltas between planned and actual outcomes are better suited to quantifying normal estimation *uncertainty* than they are to misguided attempts to ensure perfect estimation *accuracy*.

How can development organizations deliver estimates that are consistent with past performance and promote high quality? Collecting and analyzing completed project data is one way to demonstrate both present capability and the complex relationships between management metrics like size, staffing, schedule, and quality. Access to historical data lends empirical support to expert judgments and allows projects to manage the tradeoffs between staffing and cost, quality, schedule, and productivity instead of being managed by them.

The best historical database will contain the organization's own completed projects and use the organization's data definitions, standards, and methods. If collecting internal benchmark data is impossible or impractical, external or industry data offers another way to leverage the experiences of thousands of software professionals. Industry databases typically exhibit more variability than projects collected within a single organization, but decades of research have repeatedly demonstrated that fundamental relationships between the core metrics apply regardless of application complexity, technology, or methodology.

History suggests that best-in-class performers counteract perverse incentives and market pressure by employing a small but powerful set of best practices:

- Capture and use "failure" metrics to improve future estimates rather than punishing estimators and teams
- Keep team sizes small
- Study the best performers to identify best practices
- Choose practical defect metrics and models
- Match quality goals to the mission profile

Software developers will never eliminate uncertainty and risk, but they can leverage past experience and performance data to challenge unrealistic expectations, negotiate more effectively, and avoid costly surprises. Effective measurement puts projects in the driver's seat. It provides the timely and targeted information they need to negotiate achievable schedules, identify cost-effective staffing strategies, optimize quality, and make timely midcourse corrections.

Works Cited

Armel, K. 2012. "History Is the Key to Estimation Success." *Journal of Software Technology* 15,1 (2012):16-22. Print.

Armour, Phillip G. 2008. "The Inaccurate Conception." *Communications of the ACM* 51.3 (2008): 13-16. Print.

Beckett, Donald. "Engineering best and worst in class systems." *QSM Blog*. 2013. Web. <http://www.qsm.com/resources/research/research-articles-papers/>.

National Institute of Standards & Technology. 2002. *Final Planning Report 02-3: The Economic Impacts of Inadequate Infrastructure for Software Testing*. May 2002. Research Triangle Park: RTI Heath, Social, and Economics Research, 36. Print.

Putnam, Lawrence H., and W. Myers. *Five Core Metrics: The Intelligence behind Successful Software Management*. New York: Dorset House Publishing, 2002. Print.

Putnam, Lawrence H., and W. Myers. *Measures for Excellence: Reliable Software on Time, within Budget*. Upper Saddle River: Prentice-Hall, Inc., 1992. Print.

Quantitative Software Management, Inc. *QSM Software Almanac: IT Metrics Edition*. McLean: QSM, Inc., 2006. Print.

Seapine Software. 2009. *Identifying the Cost of Poor Quality*. 2009. Web.

The Standish Group. "New Standish Group report shows more project failing and less successful projects." *The Standish Group*. 2009. Web.

Improving Forecasts using Defect Signals

Paul Below

An article based on this research is scheduled to appear in an upcoming edition of the Department of Defense journal ***CrossTalk****.*

Abstract

On large software development and acquisition Programs, testing phases typically extend over many months. It is important to forecast the quality of the software at that future time when the schedule calls for testing to be complete. Shewhart's Control Charts can be applied to this purpose, in order to detect a signal that indicates a significant change in the state of the software.

Introduction: Shewhart's Control Charts

Every process displays variation. Some processes display controlled variation and others display uncontrolled variation.

Control Charts were invented by Walter A. Shewhart in the 1920s, and for most of the rest of the 20th century, these concepts were popularized by people such as W. Edwards Deming. The driver for this development, as explained by Shewhart:

> "The engineer desires to reduce the variability in quality to an economic minimum. In other words, he wants (a) a rational method of prediction that is subject to minimum error, and (b) a means of minimizing variability in the quality of a given produce at a given cost of production." (Shewhart 9)

Over several decades, QSM has found that software metrics commonly follow a Rayleigh curve (Putnam Chapter 13). This results in a very different situation from the typical use of control charts, where the process being measured is expected or desired to have a consistent output each time, every time.

In this paper, I describe the use of control charts during testing phases of software development projects. This use is not to determine if the testing is in control, nor is it in order to improve product quality (although that has also been done) (Comps 15-18; Hale and

Rowe 4-8), but rather to determine when there has been a shift in quality. This is in order to improve mapping of project progress to forecast curves and thereby improve estimates of project schedule.

Therefore, let us look at a typical example.

Using Control Charts to Detect Signals

In order to improve defect forecasts, I use Individuals and Moving Range charts (*XmR)*. This is a type of control chart that is suitable for most real time situations, including the collection of periodic data such as defects detected in a given time period (such as week or month). The Individuals chart has each value plotted in time order. The Moving Range chart, on the other hand, plots the short term variation from one period to the next.

While most signals denoting a significant change in the underlying situation, such as stabilization of the product reliability, appear on the individuals chart, it is good practice to look at the moving range chart as well, as some signals will only show up on it.

Control charts are based on the long term average value as well as the average moving range value of one point to the next. It is important to calculate control limits correctly in order to not miss valid signals. The appropriate formulas can be found in select books (Wheeler; Breyfogle) and are also built into statistical tools such as *SPSS*®, *Minitab*® and *SAS*®.

Control limits provide a signal of sporadic or chronic problems. For tracking defects, however, the signal we are looking for is a change in the underlying quality of the software product. Hopefully, this will be a signal of an improvement and not a signal of a problem!

Rules

There are a number of rules that are used to detect signals. The number of rules used and the definitions of the rules vary slightly from one source to another. However, the traditional use of control charts is best met by keeping the number of rules to a minimum, thereby reducing the chance of obtaining a false signal.

All uses of control charts walk this decision line. Shewhart originally used three sigma limits because he wanted to minimize false signals, which would incur the unnecessary cost of researching a problem that didn't exist. In other words, when he saw a signal he wanted to be almost completely certain it was real.

In IBM *SPSS* 22, for example, there are 11 possible rules that can be turned on or off:

- One point above +3 Sigma, or one point below -3 Sigma
- 2 out of last 3 above +2 Sigma, or 2 of 3 below -2 Sigma
- 4 out of last 5 above +1 Sigma, or 4 out of 5 below -1 Sigma
- 8 points above center line, or 8 below center line

- 6 in a row trending up, or 6 trending down
- 14 in a row alternating up and down

Example Control Charts

In Figure 1, weekly defects detected are plotted. All the SPSS® rules are turned on. If the defect detection rate has changed significantly, that would show up as a special cause signal in the control chart. In this example, the balance between testing and fixing has not remained constant. Five of the points show up as red, meaning they violated one of the rules (see Table 1).

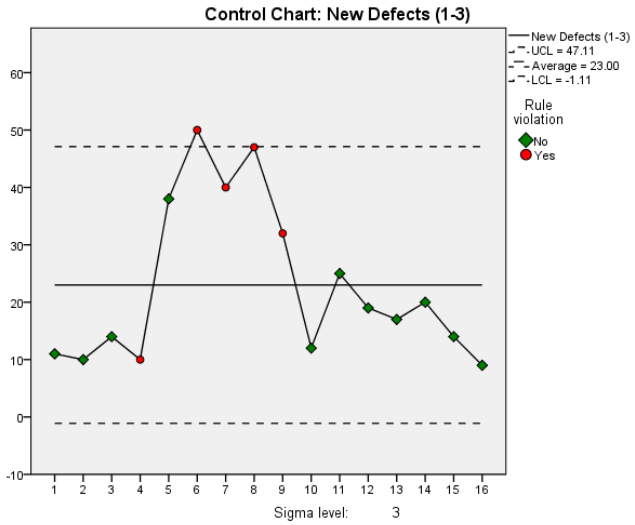


Figure 1. Control Chart Example

Point #	Violations for Points
4	4 points out of the last 5 below -1 sigma
6	Greater than +3 sigma
6	2 points out of the last 3 above +2 sigma
7	2 points out of the last 3 above +2 sigma
8	4 points out of the last 5 above +1 sigma
9	4 points out of the last 5 above +1 sigma

Table 1. Rule Violations

What can we surmise from this? These violations are not unusual. As mentioned previously, defect metrics commonly follow a Rayleigh distribution. In Figure 2, actual defects detected monthly are overlain on a defect forecast based on the current project plan (a parametric SLIM-Control® forecast based on historical defect rates and project type, size, staff, and duration). We can see the peak detected as a set of rule violations falls in line with the peak of the Rayleigh curve.

Defect estimation is outside the scope of this article, although an example is described in the next section.

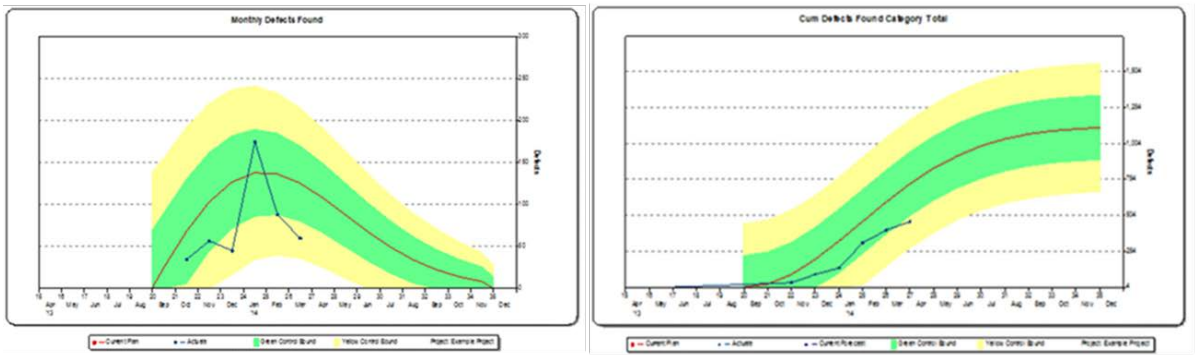


Figure 2. Rayleigh Example: Defects Detected and Cumulative Defects Detected

Figure 3 shows Individual and Moving Range charts for the ratio of defects discovered to defects resolved. This ratio measures the balance between defect detection in testing and defect repair. Values in the individuals chart (Figure 3, left chart) greater than 1 indicate more defects were resolved than were detected during that week. Both charts show rule violations. Point 15 on the individuals chart (Figure 3, left chart) provides evidence that the balance has shifted, supporting the conclusion that the project is truly on the downslope of the Rayleigh Curve.

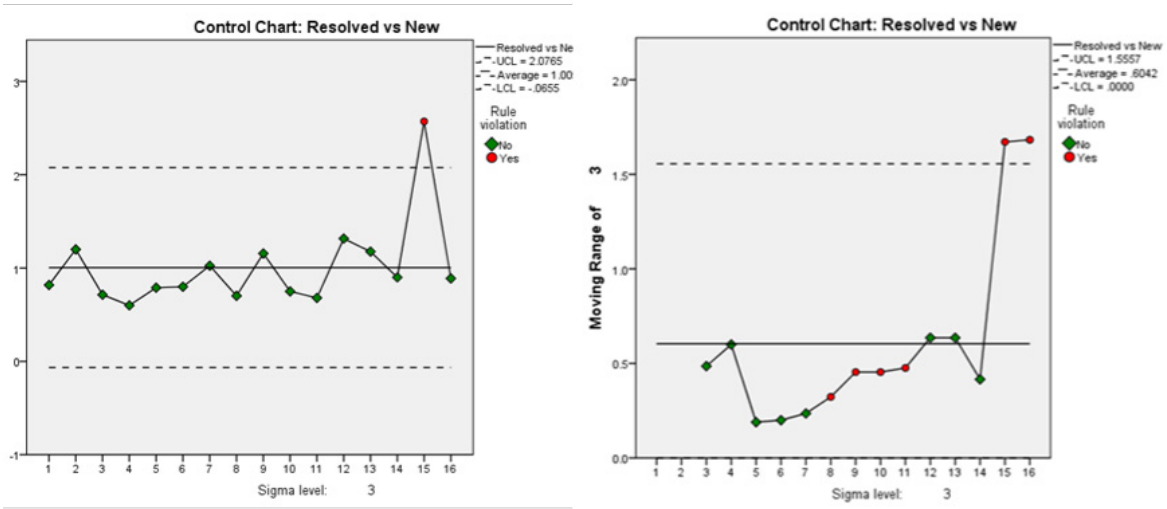


Figure 3. Individual and Moving Range Charts

Defect Prediction

To make good use of a defect signal, a defect estimate is required.

The plan in Figure 3 was based on parametric estimating. It is possible to create very useful estimates of defects based on only a few key metrics. For example, I created a regression analysis to predict defects based on over 2000 recently completed software projects from the QSM database. This resulted in an adjusted R square of .537 using only the input variables the log of peak staff, the log of ESLOC, and the log of production rate (ESLOC per calendar month). The output variable is log of defects (Why logs? For the explanation, see Below, 319-

333). The standardized residuals are plotted on a histogram in Figure 4. As can be seen, the residuals have a normal distribution with mean close to zero. The model is not skewed.

Large projects have multiple testing phases. Such models, with multiple control charts, can be used throughout. For example, with one Fortune 500 client, I found that merely using the number of prerelease defects was an excellent predictor of their go live release defects (R-square of over 0.7).

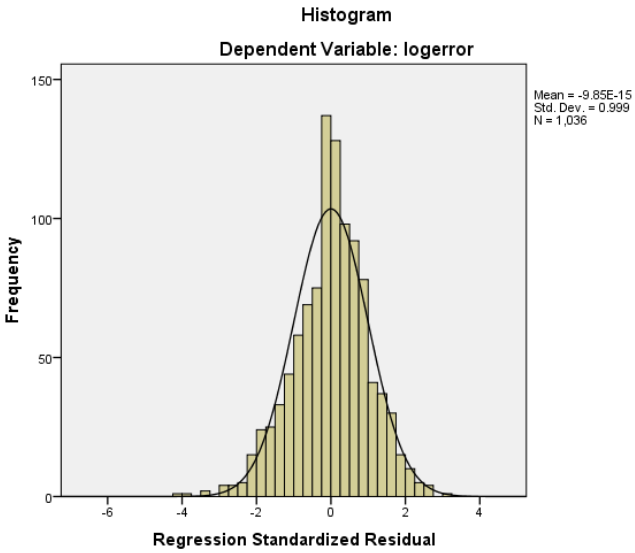


Figure 4. Standardized Residuals

Summary

Control charts can be used to determine whether apparent changes in defect rates are significant and especially, whether the peak of the defect detection Rayleigh curve been reached. One use for this knowledge is to create and improve forecasts of Program completion, or software quality at key Program milestones. Once the peak of the Rayleigh curve has been reached, the curve can be forecast into the future to predict the software quality at any given point.

Shewhart gave us this thought regarding updating forecasts:

> "…since we can make operationally verifiable predictions only in terms of future observations, it follows that with the acquisition of new data, not only may the magnitudes involved in any prediction change, but also our grounds for belief in it" (104)

Works Cited

Below, Paul. "Maximizing Value: Understanding Metrics Paradoxes through Use of Transformation." *The IFPUG Guide to IT and Software Measurement: A*

Comprehensive International Guide. Ed. IFPUG. New York: CRC Press, 2012. 319-333. Print.

Breyfogle, Forrest W., III. *Implementing Six Sigma: Smarter Solutions Using Statistical Methods*. Hoboken: John Wiley & Sons, 2003. Print.

Comps, Michael. "Why CMMI Maturity Level 5?" *Crosstalk* Jan-Feb 2012: 15-18. Print.

Hale, Craig, and Mike Rowe. "Do Not Get Out of Control: Achieving Real-time Quality and Performance," *Crosstalk* (Jan-Feb 2012): 4-8. Print.

Putnam, Lawrence H., and Ware Myers. *Five Core Metrics: The Intelligence behind Successful Software Management*. New York: Dorset House, 2003. Print.

Shewhart, Walter A. *Statistical Method from the Viewpoint of Quality Control*. New York: Dover Publications, 1986. Print.

Wheeler, Donald J. *Understanding Statistical Process Control*. Culver City: SPC Press, 2010. Print.

Counting Function Points for Agile: Iterative Software Development

Carol Dekkers

Abstract

Function points (FPs) are proven to be effective and efficient units of measure for *both* Agile/iterative and waterfall software deliveries. However, inconsistencies come to light when comparing FPs counted in Agile/iterative development with those counted in waterfall or combination development – and those inconsistencies can create confusion for cost, productivity, and schedule evaluations that span multiple software delivery methods.

IFPUG's definitions for such terms as "project," "elementary process," "consistent state," and "enhancement" do not directly translate to Agile/iterative delivery methods in which the term "project" is often interchanged with such terms as "sprint," "iteration," "release," or "story map." This paper seeks to marry IFPUG definitions with equivalent concepts in Agile/iterative processes so as to create a basis for consistent comparison.

Introduction

Functional size is a pre-requisite for estimating the cost, effort and duration of software development "projects" (labeled by IFPUG as *development projects* for the first "release" of a software product and *enhancement projects* for its subsequent adaptive maintenance and enhancement). Function points are a convenient and reliable common denominator of size and are often used as the basis for comparing project productivity (FP/effort), duration delivery (FP/elapsed time), maintainability (hours/FP), product quality (defect density) and other important aspects of software delivery.

With today's IT landscape dotted with Agile, iterative, spiral, waterfall and combination approaches to development, businesses are searching for the ultimate approach to delivering the right software (functionality- and quality-wise) at the lowest unit cost for the least amount of effort. But for estimation to succeed in this climate, we'll need consistent FP definitions across all methods of software delivery.

Agile Is Here to Stay

Gone are the days when Agile/iterative development methods were considered "rogue" and without structure; today, Agile methods are held in high esteem, even in conservative software development shops where waterfall still prevails. Indeed, the penetration of Agile in the IT marketplace has had numerous positive impacts, including:

- Change is no longer seen as the enemy;
- Users are more receptive to participating on projects and better understand the impact that non-involvement can cause;
- Business stakeholders are more engaged; and
- Developers can better respond to changing business requirements.

Now, with both Agile/iterative and waterfall methods at their disposal, businesses are searching for the optimal combination of talent, tools, techniques, cost, and schedule that will deliver good-enough- quality software for a reasonable investment of time and money. But finding that "sweet spot" relies on measuring the same elements in the same ways across various delivery methods. And consistency in measurement depends on consistency of definitions and the application of measurement techniques, such as IFPUG Function Points.

We need to start by aligning the IFPUG definitions and then looking at how to apply them consistently to count FP on Agile and waterfall deliveries alike.

IFPUG Functional Size Measurement Definitions

IFPUG FP methodology (IFPUG 4.3.1) gives us guidance on how to count FP based on projects and the delivery of unique (and complete) elementary processes that leave the business in a consistent state. Agile/iterative techniques deliver software incrementally. Defining what constitutes a "project" and the delivery of an "elementary process" in Agile/iterative is the KEY element for consistent function point counting across development approaches.

Since FP counting of software *development* is meant to measure the size of the functional user requirements delivered via a *project (development or enhancement)*, the methodology used to implement that functionality (be it Agile/iterative, spiral waterfall, or any other development method) should have no effect on the size of the delivered software product. The application FP (also called the baseline or installed application FP size) is the same regardless of the delivery method used and can be measured consistently at the completion of any type of software delivery. Application FP counts are of secondary concern for this paper; the primary concern is defining what constitutes a "project" (either development or enhancement) in Agile/iterative development.

Terminology Presents Challenges

Before we get into the issues and challenges of counting FP in an Agile environment, let's add a bit of strictness and consistency to a few of our terms:

- **Release:** A release is the distribution of the final version of an application. A software release may be either public or private and generally constitutes the initial generation of a new or upgraded application. A release is preceded by the distribution of alpha and then beta versions of the software. In Agile software development, *a release is a deployable software package that is the culmination of several iterations* (Rouse "Release").

- **Project:** *A collection of work tasks with a time frame and a work product to be delivered* (IFPUG). According to the Project Management Institute, a project is a *temporary endeavor undertaken to create a unique product or service* (PMI).

- **Iteration:** In Agile software development, an iteration is a *single development cycle, usually measured as one week or two weeks* (Rouse "Iteration").

  (Side note: Some proponents of Agile insist that all iterations be the same length, and that the particular length of iterations (anywhere from 2 to 6 weeks) is of less importance. For our purposes in this paper, the key element is that an iteration represents a single development cycle.)

- **Sprint (software development)**: In product development, a sprint is a *set period of time during which specific work has to be completed and made ready* (Rouse "Sprint").

For **waterfall development**, it is fairly easy to identify and count FP for discrete development and enhancement projects. "Release" and "project" are often used synonymously to refer to the scope of a self-contained software delivery.

For **Agile development**, a "project" is not so easily identifiable. "Sprint" and "iteration" are used more often than "release," and those terms are based on *elapsed calendar time or work effort* rather than functionality. "User stories" (or "use cases") are used to describe functionality and are useful for identifying functional user requirements, but there is no requirement that they constitute an elementary process or that they leaves the business in a consistent state – both of which are required for FP. The notion of using "story points" (a sizing approach intended to quantify the relative size of a user story) as equivalent to FP (as suggested by a few Agile advocates) is not feasible for the following reasons:

- Story points are not convertible to FP (there is no conversion factor);
- Story points are not standardized (FP are standardized through IFPUG and ISO); and
- Story points capture user story size differently (and based on different concepts) than FP.

When functionality is delivered in discrete and well-defined construction projects, as is intended with **waterfall-style deliveries**, counting FP is easy and based on a single set of functional user requirements. Even when a subset of the overall features is delivered in one release and then enhanced in a later release, the discrete "chunks" of new/enhanced functionality make counting FP a straight-forward process using IFPUG methodology.

With traditional waterfall delivery, the terms "developed" and "delivered" are almost always used interchangeably. But in **Agile/iterative development**, there is a difference

between "developed software" (which is not yet ready for mass deployment) and "delivered software" (ready for full deployment).

Therefore, with Agile/iterative forms of software delivery, counting the "delivered" functionality is not so easy. IFPUG (and other ISO-conformant) functional size measurement techniques are counted based on complete elementary processes or functions that leave the business in a consistent state, not on parts thereof. A "function point" count consists of delivered (or anticipated to be delivered) functions that are whole business processes. Partially delivered functions that are incomplete and cannot support the business without further work would not typically be counted as "function points delivered." For example, a business process such as create hotel reservation would not be an elementary process until a reservation is made and stored. If the first part of the reservation process was delivered in one sprint (such as checking the availability of a hotel for given dates) and the latter part was delivered subsequently (enter customer information and book reservation), FPs would be counted on the elementary process (both parts.)

Why Does All This Matter?

At this point, you may be asking why we don't just use the word "release" in place of the word "project" and be done with it.

Or, couldn't we simply count up the delivered function points at the end of a "release" no matter how the software is developed, and then compare the productivity across releases to perform our estimates?

Actually, yes. This is absolutely the right way to go, but only if our definition of "release" can remain consistent across our various forms of delivery. For starters, we'll have to determine the number of Agile iterations or sprints that constitute a "release." Let's consider the following situations:

- When a single software "product" (i.e., the result is a working piece of software) is delivered via two or more distinct releases, each of which is completed and implemented into production (i.e., fully-functioning software), the software application in place at the end of the two releases is exactly the same size as it would have been if delivered all at once. (Consider the analogy of a floor plan that is built in stages versus all at once – the resultant square foot size is the same.) Each release is discrete and self-contained, and the sum of the FP of the two releases likely will exceed the installed application base (because some of the functionality completed in release 1 may be enhanced through adaptive maintenance FP in release 2, yet may not increase the application size (also called "installed base" size or baseline). Think of how a house can be delivered in a first construction and then renovated in a second – the square foot size of the two constructions added together may exceed the overall size of the house.
- However, when the software is built iteratively over the course of a year in two week sprints, there is a lack of discrete delivery. (Think of building a house bits at a time and

slowly developing the underlying floor plan.) Where is the "elementary process" for FP counting? Likely, the "complete" functions were delivered through multiple user stories or use cases spanning a number of sprints or iterations. Therefore, the challenge to counting function points in Agile/iterative lies in the question of when and where a business process or function leaves the business in a consistent state.

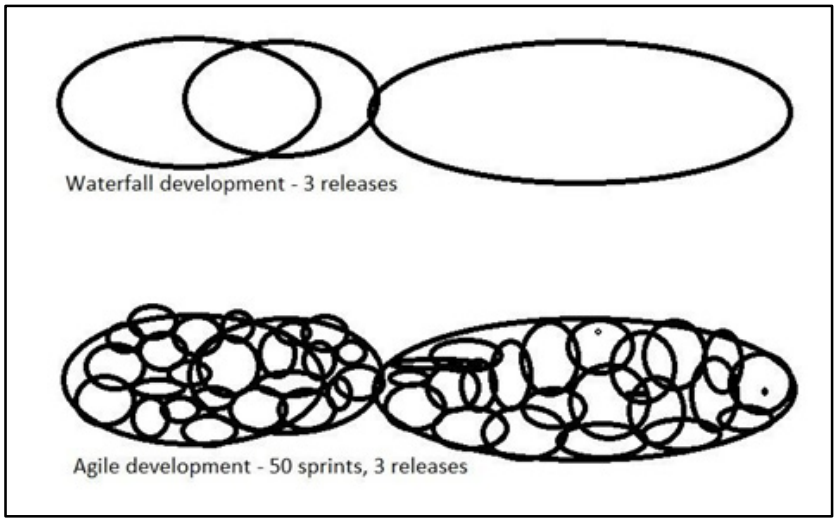


Figure 1. Waterfall Releases vs. Agile Releases

Some would suggest that each sprint or iteration should count as a discretely countable functionality in FP; however, this approach would contradict FP definitions and concepts including:

- Elementary process
- Self-contained
- Enhancement (defined as the adaptive maintenance of a delivered software product)
- Functional user requirements (which relies on elementary processes)
- Development *project*
- Enhancement *project*
- Counting scope (a set of Functional User Requirements)
- Base Functional Requirement (BFC) – elementary unit of Functional User Requirements
- Consistent state

Why would counting "sprints" violate these definitions? The answer is that a sprint, by definition, is based on elapsed time or work effort – not on functionality. Certainly user stories (and use cases) describe user functionality, but there is no requirement that they describe complete or self-contained functionality. Consider that two user stories for an airline reservation system might be written as:

a. Choose the flights on which you want a seat
b. Pay for the reservation

Clearly these are two different steps in the sequence of steps needed to complete the airline reservation. But each one is NOT its own separate elementary processes; they're part of a single elementary process where both steps are needed to leave the business in a consistent state.

Further examples are shown at Table 1:

Release #	Release (project) FPs	Application FPs
1	300	300
2	250 (200 new + 50 chg)	500

Table 1. Comparison of Release and Application Function Point Counts

FPs in Agile/Iterative Development

When we talk about implementing user stories or use cases, the assumption may be that each user story or use case equals at least one standalone and self-contained function. This is seldom the case. While the scope grows and morphs with each iteration or sprint, the requirements are often progressively elaborated. This means that one "functional user requirement" may span multiple user stories or use cases – especially if it is a complex one. Thus, the application may not satisfy the full user requirement for a process or transaction until after a set of sprints is implemented.

A full Agile software development implemented in a series of two- to six-week sprints will deliver functionality in a piece by piece fashion. It may not be easy to determine/predict when the functionality will actually be delivered, especially once the IFPUG definitions for "elementary process" and "leaving the business in a consistent state" are taken into account.

Using the home building analogy, Agile development is similar to pouring the foundation and building rooms a bit at a time, as the overall floor plan eventually comes into being. When a home is constructed in this manner, it is not ready to be occupied until the rooms are finished and a roof covers the structure. In Agile development, functionality is typically delivered partially – in sprints – and it isn't until several sprints are delivered that the business can begin to actually *use* the software. Yet there is a tendency to assume that sprints and iterations are akin to a new development project for the first sprint and enhancement projects for each sprint thereafter.

The challenge to counting FP on Agile projects lies in determining *which* functional user requirements have been satisfied (and *when* they are satisfied) by the software delivery.

For instance, if a function is "delivered" in a sprint (i.e., we count FP for the initial sprint, assuming that the user story completely describes an elementary, self-contained, and complete business function), and subsequently enhanced in a second sprint, was the original function:

a. Incomplete (i.e., the elementary process was NOT fully delivered in the first sprint) –and we shouldn't have counted/taken credit for FP in the first sprint?
b. Complete at the time of the first sprint but now enhanced due to changing requirement – and we should count delivered FP for sprint #1 and count the entire transaction's FP a second time for sprint #2?
c. Flawed in the first sprint – therefore the FP counted in sprint #1 should not be recounted in sprint #2 (because sprint #2 was only corrective maintenance)
d. Some other variation?

The value and beauty of FP is that they provide a methodology- and technology-independent assessment of software size based on the functional user requirements, which (at the end of both Agile AND waterfall development) are the same. The actual size of the installed application baseline (FP installed) is – *or should be* – the same, regardless of HOW the software is developed.

When does the delineation of FP across sprints become an issue? (Or, why is it important to count delivered FP in a consistent manner regardless of development methodology between Agile and waterfall projects?)

There are two significant situations where the FP "delivery" is critical to businesses:

1. **Productivity assessment.** Businesses want to compare the cost per FP or effort per FP between Agile and waterfall projects, but doing so requires a consistent baseline. If we count FP for each sprint the same way as we do for an entire project, the total project FP delivered in Agile (the sum of FP across all sprints) may be 10 or more times the total project FP delivered using waterfall (the sum of FP across several releases) thereby invalidating productivity comparisons;
2. **Outsourcing.** When businesses commit to paying for software as it is "delivered," it makes no sense that the business should pay over and over for partial delivery of functionality just because it is delivered using an Agile approach. From the client perspective, the overall delivered software (base) is the same size.

Therein lies the dilemma and the challenge in using function points on Agile projects – it is problematic to credit Agile projects that deliver same completed functionality (i.e., complete elementary processes leaving the business in a consistent state) with having produced MORE functionality than waterfall projects!

To recap, let's look at one example using the two different methods:

Waterfall software delivery: The business needs a new customer service application where the final installed software will equal 1000 FP. Through negotiation and agreement, three phases/projects are outlined, each of which delivers working software in production.

1. Phase 1: New development project = 300 FP (installed baseline at the end of the project = 300 FP).

2. Phase 2: New functionality of 200 FP and enhancement of 50 that were already delivered in phase 1. Project count = 250 FP (new installed baseline at the end of the project is now 500 FP).
3. Phase 3: New functionality of 500 FP and enhancement of 50 that were already in place. Project count = 550 FP (installed baseline at the end of the project is now 1000 FP).

Agile development: The business needs a new customer service application delivered using an Agile approach. User stories are iteratively documented and a series of 2-week sprints is agreed upon to allow developers and the business to discover the requirements and define them as they go. Twenty-five different sprints are worked on over a year period, and at the end of the "project(s)" the installed baseline software is 1000 FP.

1. Sprint #1 outlines the need for users to sign in and validate their password. (It is not yet certain where the data will reside. We cannot yet count a datastore definitively but it is envisaged that it will be maintained in either an ILF or EIF in a future user story/user case.) Sprint 1 also delivers the first of several screens needed to set up a new customer.

   – Estimated FP count = 1 Low Complexity Query (for user validation) + 1 Average complexity datastore (for customer) + 1 Average Input process (create customer) = 16 FPs (baseline = 16 FPs).

2. Sprint #2 adds a second screen of data for customer creation and identifies the need to allow changes to and deletion of customer records. Customer details (all of the data added across both screens) can be displayed. What should be counted in Sprint #2?
   a. The new functionality introduced: Change customer = 1 Average complexity Input; Display customer = Average complexity Query; Delete customer = 1 Low complexity Input EI;
   b. The datastore called Customer - it was already counted in the first sprint (and it is the same complexity). Should it be counted again in sprint #2? It seems nonsensical to do so;
   c. The add customer function – it was delivered partially in the first sprint because there wasn't enough time to deliver it fully. It was incomplete (i.e., did not leave the business in a consistent state) in the first iteration – the question is whether the FPs should be counted in sprint #1, in sprint #2, divided between sprints (e.g., ½ in each sprint), or as the entire number of FPs in both sprints (i.e., appearing as double the FPs).

Guidance on FPs Counting for Agile

The following list of recommendations is provided to increase consistency across the various forms of software delivery:

1. Identify the user stories and use cases that contribute to a single elementary process, group them together, and count the FPs for the elementary process (and document what contributed to the function);

2. Count an ILF only when its maintenance is introduced and consider future DETs and RETs it will include (i.e., count a Customer ILF only when the first transactional function to maintain it is delivered, and count its complexity based on all DET and RET envisaged in that release);
3. Count functionality at a release level according to #1;
4. Count as development project FP all functionality for the first release as long as working software is implemented (i.e., users can input data) and elementary processes are complete;
5. Count as enhancement project FP all functionality for subsequent releases as long as working software is implemented and adaptive maintenance is performed on each release;
6. If data or transactions describe code data, do not count (not as ILF, EIF, or any associated maintenance or query/drop down functions for such data). This needs to be spelled out because this is easy to overlook when counting from use cases or user stories;
7. Document your assumptions used in the FP count(s).

Comparative and consistent FP counts across various development "projects" can be done through consistent terminology, and the application of FP rules. Being careful not to size "bits" of functionality partially delivered, and instead grouping use cases and user stories into elementary processes according to IFPUG 4.3.1 will go a long way to creating consistent FP counts.

Works Cited

International Function Point Users Group (IFPUG). "Glossary." *Function Point Counting Practices Manual (Release 4.3.1)*. Princeton Junction: IFPUG, 2010. Print.

Project Management Institute (PMI). *A Guide to the Project Management Body of Knowledge (PMBOK Guide)*. Newtown Square, Pa: PMI, 2004. Print.

Rouse, Margaret. "Iteration." *SearchSoftwareQuality*. TechTopic Network. 2014. Web.

Rouse, Margaret. "Release." *SearchSoftwareQuality*. TechTopic Network. 2014. Web.

An Analysis of Function Point Trends

Donald Beckett

Introduction

Function point analysis has played an important role in software measurement and analysis for 30 years. This study looks at the QSM software project database and examines a set of validated projects counted in function points that have completed since the year 2000 to see what they tell about productivity, schedule, and staffing. We are fortunate to have several thousand projects in this sample to work with as this allows us parse to the data many different ways and still have enough projects to be statistically significant. For this study only unadjusted function points were used.

Demographics

Our sample contains 2,231 projects completed since the year 2000.

The "Average" Function Point Project

What does a representative project sized in function points look like?

- **Domain:** 98% business IT projects
- **Size (median):** 160 function points
- **Schedule (median):** 7.03 months from the start of analysis through implementation
- **Effort (median):** 21.85 person months
- **Average Staff (median):** 2.3 full time equivalents
- **Labor Cost (median):** $262,200 At $75/hour based on a 160 hour work month
- **Requirements Effort (median):**13% of project effort spent in analysis and high level design
- **Development Type:** 75% are enhancements to existing systems

Figure 1 shows the size distribution of the function point projects. Only 7% are larger than 1000 FP.

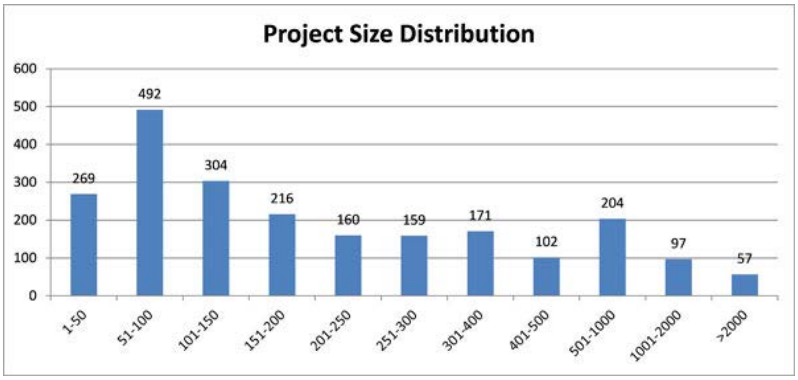


Figure 1. Project Size Distribution

New Development, Enhancements, and Maintenance

QSM classifies software development projects by the ratio of new code to modified, deleted, or reused code:

- New development (> 75% new functionality)
- Major enhancement (25% - 75% new functionality)
- Minor enhancement (5% - 25% new functionality)
- Conversion (< 5% new functionality)
- Maintenance

The majority (75%) of function point projects in the QSM database (shown at Table 1) are enhancements to existing systems. Conversion projects (those with less than 5% new functionality) are less productive than other project types.

	Median Productivity				
	New Development	Major Enhancement	Minor Enhancement	Conversion	Maintenance
% of Projects	16%	61%	14%	7%	2%
Median PI	14.00	10.10	11.20	9.90	10.40
Median Size (FP)	291	119	153	109	68
Median Effort Months	29.7	19.3	28.1	23.4	18.6
Median % Funct Effort	12%	11%	12%	10%	19%
Median FP/PM	9.16	5.79	5.19	5.06	2.70
Median Duration	7.57	7.23	6.42	6.43	4.73
Median Defects	37.0	16.0	38.5	35.0	16.0

Table 1. QSM Database Function Point Projects Breakdown by Type

New development projects are the largest and most productive, but they account for only 16% of the sample. While extensive code reuse in enhancement projects provides additional functionality that the project does not have to develop, it also creates a more complex development environment for the new and modified code which will require additional

analysis to ensure that it is compatible with the existing code and extensive regression testing to verify that there are not unforeseen impacts.

Productivity

Few topics in software measurement generate as much discussion and debate as productivity. While linear or ratio-based measures like hours per function point or function points per person month are widely used, QSM's productivity metric – the PI or productivity index, differs in two important ways from ratio-based productivity measures:

- It accounts for size, effort, **and** schedule performance.
- It accounts for the distinctly nonlinear relationship between these three metrics.

The scatter plot shown in Figure 2 allows the user to see how much a project or estimate is above or below average while providing a visual comparison to the organization's other completed projects. While past performance is no guarantee of the future (in the case of an estimate) it can serve to define the boundaries of what is possible.

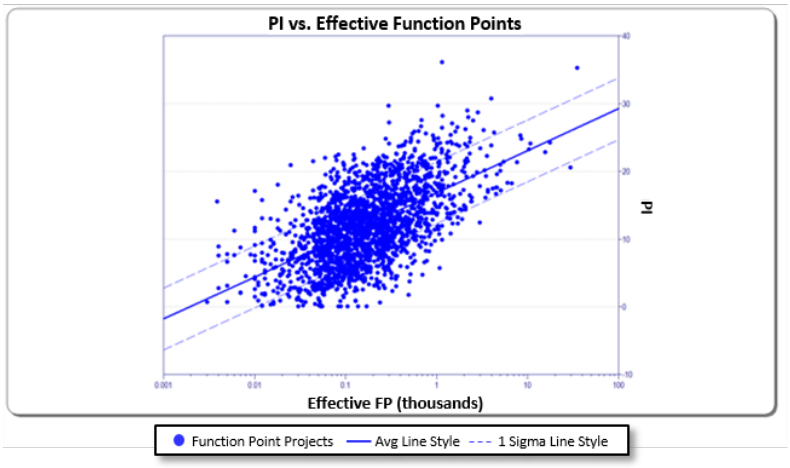


Figure 2. PI vs. Effective Function Points

Our research shows that project size has an important impact on productivity, whether measured by our PI or in function points per person month. Simply put, larger projects are more productive. While we can only speculate, here are several possible causes.

- Larger projects are more important to organizations. They cost more and have higher visibility. As a result, they benefit from more experienced developers and better project management and tools.
- Another reason is less positive. Large projects, as Capers Jones tells us, are more likely to be cancelled (Jones). Since we calculate project productivity from completed projects, the impact of cancelled or failed projects is not reflected in our productivity measures. This is not just a preference of ours. Failed and cancelled projects frequently have not had effective metrics processes in place, in which case the data do not exist. If they do exist, they are by definition

incomplete. And in any case, projects that fail are often more interested in covering their tracks than in publicizing what occurred. At the same time, small projects that are experiencing problems may be allowed to limp along to completion. They aren't going to bankrupt the company! Regardless of our preference, interest remains in how many function points per person month (or hours per function point) constitutes normal productivity.

Table 2 looks at the productivity of different size ranges by function points. Since projects differ in scope, we have included only effort from the beginning of analysis up to implementation into production to normalize the comparison. (These activities correspond to the Requirements & Design and Code & Test phases in the SLIM® model).

Productivity by Size Category		
Size (FP)	Count	FP/PM (Median)
≤ 50	269	3.49
51-100	492	5.13
101-150	304	6.54
151-200	216	6.67
201-250	160	7.65
251-300	159	8.49
301-400	171	9.55
401-500	102	9.72
501-1000	204	13.43
1001-2000	97	16.29
> 2000	57	23.10

Table 2. Productivity by Size Category

Project Effort

The majority of projects are not multi-million dollar endeavors with dozens of programmers. 61% of the projects in our study expended 30 or fewer person months of effort. From a cost perspective, at a labor rate of $10,000/person month, the labor cost of these projects was $300,000 or less. Over 75% of the projects had 50 or fewer person months of effort. While Table 2 above showed that productivity measured in function points per person month increases with project size, Table 3 below shows a different trend: as effort grows, productivity decreases. Evidently, larger projects that complete are more efficient in their use of labor. They are more likely to have full time dedicated resources that do not have to divide their attention between multiple projects.

Effort	Project Count	% of Projects	FP/PM (Median)	Median Size (FP)
≤ 10 PM	487	21.83%	11.60	72
>10 ≤ 20	551	24.70%	7.86	113
>20 ≤ 30	323	14.48%	5.87	139
>30 ≤ 40	194	8.70%	5.65	194
>40 ≤ 50	127	5.69%	5.44	247
>50 ≤ 60	109	4.89%	5.32	301
>60 ≤ 70	67	3.00%	4.45	292
>70 ≤ 80	60	2.69%	4.73	348
>80 ≤ 90	38	1.70%	3.45	292
>90 ≤ 100	30	1.34%	3.32	312
>100 ≤ 150	102	4.57%	3.12	359
>150 ≤ 200	52	2.33%	3.44	606
>200 ≤ 300	45	2.02%	2.66	597
>300 ≤ 400	12	0.54%	3.13	1,041
>400 ≤ 500	13	0.58%	3.33	1,477
>500 ≤ 1000	14	0.63%	3.11	1,989
>1000	7	0.31%	2.55	3,500

Table 3. Project Effort vs. Function Point Sizing

As Figure 3, below, demonstrates, project types differ in the average amount of effort they require. New development projects expend the most effort, followed closely by Minor Enhancements. As the median size of New Development projects is nearly twice that of Minor Enhancements, this is strong evidence that what is often considered "free" functionality from reuse actually has an associated productivity cost.

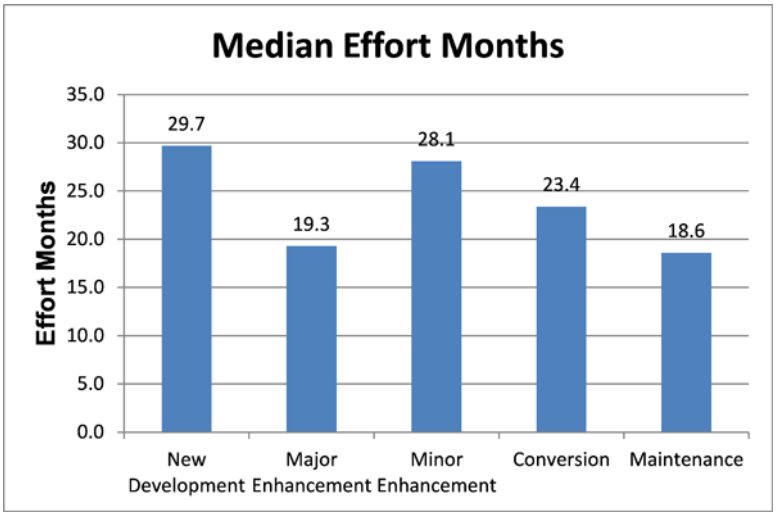


Figure 3. Median Effort Months

Schedule

In this section we look at schedule from two perspectives. First, we examine the schedule distribution of projects. Then, we look at how schedule compression and extension affect productivity.

Schedule Distribution:

As we have seen, the majority of the sample projects are enhancements to existing systems. The basic framework of the software they modify already exists. Their purpose is to add new features, modify how existing functionality works – which can include correcting defects, improving performance, or any combination of these. In a word, their objectives are normally better defined than new development which has more aspects of a learning process for both developers and end users. As Figure 3 illustrated, their median schedules are slightly shorter. Figure 4 illustrates project schedule distribution. Here are some takeaways:

- 50% of FP projects last 7 months or less
- 70% last 9 months or less
- 85% last a year or less

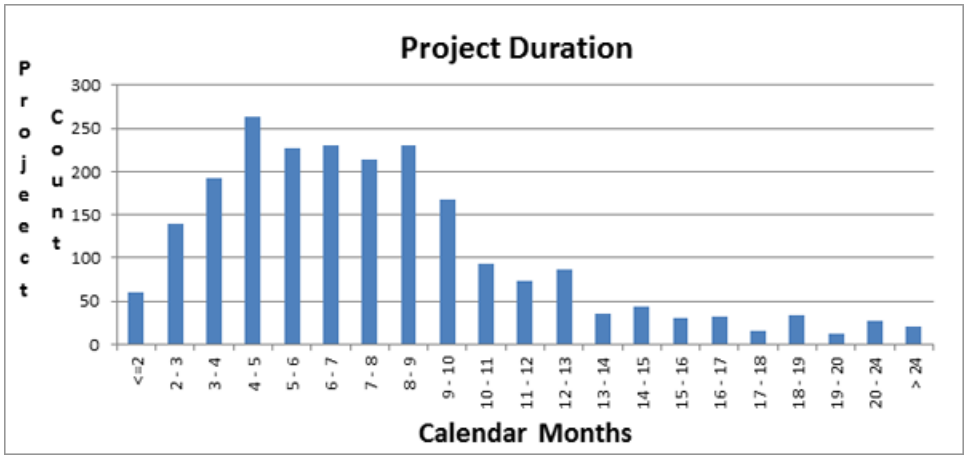


Figure 4. Project Duration

Perhaps an important reason why Agile has become so popular is that it works well with projects that last 7 months or less and produce about 160 function points using 30 person months of effort. Many lifecycle methodologies and process sets are too large and cumbersome to work effectively with projects where flexibility is a key requirement.

Schedule Compression and Extension

Figure 5 illustrates the relationship between schedule and size (in function points). The solid line in the center represents average duration at various project sizes. The dashed lines above and below the average line represent plus and minus 1 standard deviation from the regression line, with roughly 2/3 of the projects inside the dashed lines. As projects increase in size, their schedules generally grow longer, but there is considerable variability; not all projects of the same size complete in the same time frame and project size is not the only factor that influences duration. Look more closely at the scale on the two axes: they are logarithmic, not linear. Simply stated, doubling the schedule does not double the output in function points. Similarly, reducing schedules by 50% does not cut output in half.

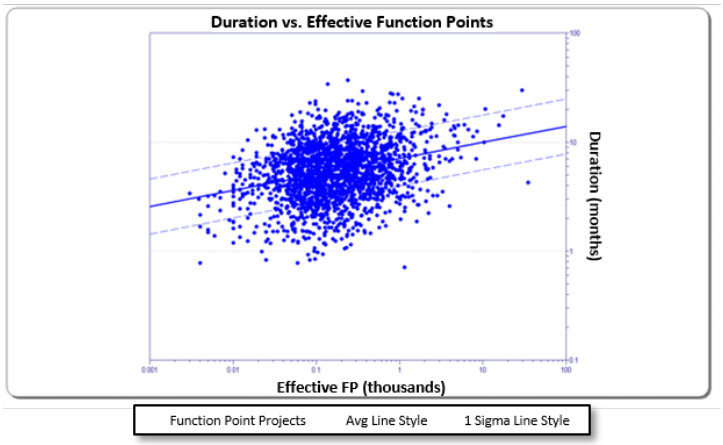


Figure 5. Duration vs. Effective Function Points

Since the relationship between size in function points and schedule is nonlinear, what happens to productivity when projects are planned with a schedule buffer? What happens when they must complete in less than average time? Figure 6 answers these questions conclusively: the more time projects are allowed, the better their productivity. While allowing a project more time than average to complete is not always an option, compressing a schedule should only be done in the full knowledge that it will lower productivity and quality while increasing cost.

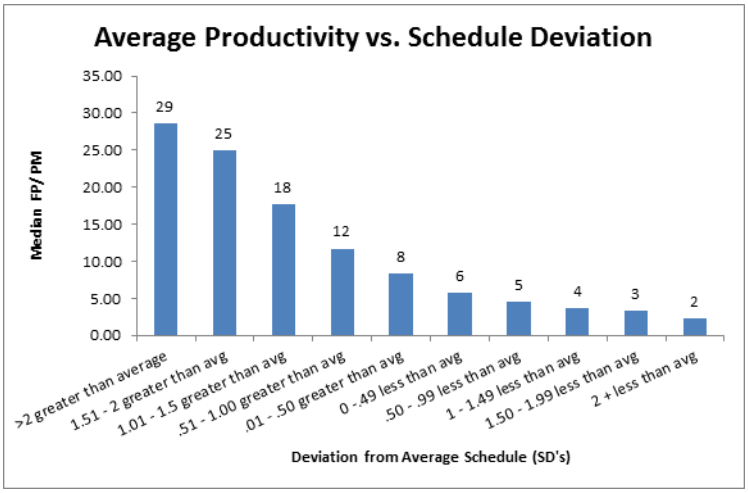


Figure 6. Average Productivity vs. Schedule Deviation

Impact of Analysis and Design

Several years ago QSM performed a study on how the amount of effort expended in analysis and design affects final productivity, quality, and time to market (Beckett). In that study, the median effort expended in analysis and design was 20% of total project effort. The projects were divided into two groups:

- Projects using more than 20% of total effort in analysis and design
- Projects using less than 20% of total effort in analysis and design

The results were striking. Projects that spent more effort on analysis and design completed sooner, had fewer defects, and were more productive. Here we repeat that analysis using our function point sample. The results are summarized in Table 4.

Impact of Effort Spent in Analysis & Design			
Medians	**>20%**	**<20%**	**Difference**
PI	14.9	11.04	29%
FP/Person-Month	7.93	6.20	28%
Duration	6.20	7.23	-14%
Total Effort	20.29	22.59	-10%
Average Staff	2.50	2.34	7%
Size in FP	171.00	157.00	9%
Defects	19.50	20.00	-3%

Table 4. Impact of Effort Spent in Analysis & Design

Once again, projects that allocated more than 20% of their effort to Analysis and Design completed sooner, expended less effort, and achieved higher productivity. They had fewer defects (and that is based on projects that averaged 9% larger). The data show conclusively that time and effort spent up front **defining** "**what** to produce" and **determining** "**how** to produce it" result in better productivity, reduced cost, higher quality, and shorter time to market. Projects that invest up front are better defined and run more smoothly: something both developers and management can appreciate.

Trends over Time

While the focus of this paper has been the analysis of function point projects completed since the year 2000, we also looked at how FP projects have changed – and remained the same - over a longer time frame. The following tables and charts include function point projects put into production since 1990.

Software Languages

In the year 2013, very few students planning a career in information technology would pick COBOL as their first choice for a software language to learn; but the old war horse still has a significant presence (see Table 5). While few new development projects are coded in COBOL, enhancements to existing systems (many of them written in COBOL or PL/1) ensure that the market for COBOL programmers is still robust.

The programming language rankings are based on the primary language of function point projects in the QSM database. COBOL, PL/1, and C++ have demonstrated staying power over time. Since 2000, Java has grown to be the leading software language. Powerbuilder was popular for a time; but is no more. What is not apparent from Table 5 is the increasing number of projects that use multiple languages. Combinations such as Java for the primary and COBOL as the secondary language are found as enterprises put Web front ends on their legacy systems.

Top 10 Software Languages			
1990-1994	1995-1999	2000-2004	2005+
COBOL	COBOL	COBOL	JAVA
PL/1	POWERBUILDER	JAVA	COBOL
NATURAL	C	PL/1	IEF/COOL:GEN
TELON	C++	C++	PL/1
SQL FORMS	VISUAL BASIC	VISUAL BASIC	Cognos Impromptu Scripts
C++	SQL FORMS	IEF/COOL:GEN	PACBASE
C	SQL	POWERBUILDER	.net
ASSEMBLER	PL/1	Oracle SQL Forms	LOTUS NOTE
CLIPPER	IEF/COOL:GEN	SQL	C++
IDEAL	ORACLE	Datastage Basic	J2EE

Table 5. Top 10 Software Languages

Productivity

During the 1990s, productivity, whether measured in function points per person month or by the QSM productivity index (PI), increased steadily. The upward productivity trend of the 1990s was followed by a precipitous decrease from 2000 on. Viewed in isolation, this reversal may seem surprising. But, this trend must be viewed in the context of what changes were going on in industry trends for project size and schedule. Figure 12 summarizes the productivity trend over time in the QSM function point database. To minimize the impact of outliers, median rather than average productivity has been used.

One factor contributing to the post-2000 decline in productivity has been the decrease in average project size. The median size of projects completed since 2005 is less than half of average project sizes from 1990 – 1994. As shown in Table 6, productivity increases with project size, so a significant decrease in size is accompanied by a decrease in productivity. Although it is outside of the scope of this analysis, investigating the reasons for this size decrease would make for an interesting study.

	Median Productivity			
	1990-1994	1995-1999	2000-2004	2005+
FP/PM	11.10	17.00	9.21	5.84
FP/Month	17.10	63.90	29.74	22.10
PI	15.30	16.40	13.90	10.95
Size (FP)	394.00	167.00	205.00	144.00

Table 6. Median Productivity

Schedule and Effort

Table 7 charts project duration and effort over time. No overall trend is apparent for duration: projects from 1990 – 1994 had a significantly longer median duration than has been

seen in later time periods. Median duration has varied up and down in a fairly tight range since then. The trend in effort has been continuously downward. Projects completed in the most recent time period expended 1/3 less effort than those completed from 1990 – 1994.

	Median Schedule and Effort			
	1990-1994	1995-1999	2000-2004	2005+
Duration (months)	10.06	6.67	6.57	7.13
Effort (person months)	32.00	26.45	23.00	21.60

Table 7. Median Schedule and Effort

Summary

Here are the most important observations we have drawn from the project data:

- **Function points have staying power.** While they are not the only sizing metric used in software, function points are widely used, especially in IT. They are used much less in telecommunications software and hardly at all in real time.
- **Function Point projects have gotten smaller.** Their median size is less than half of what it was 20 years ago.
- **Most function point projects modify existing systems.** 75% of them are enhancements.
- **Projects deliver faster and expend less effort than they did 20 years ago.** The average project from the beginning of analysis until implementation into production now lasts a little over 7 months.
- **Productivity has worsened.** Although the tools and methods available to developers are superior to those available 20 years ago, they have not improved average project productivity. The implication is that to improve productivity, the focus needs to be moved from the developers to other aspects of the software development process. One recommendation that stands out from our analysis would be to lessen schedule pressure.
- **Time spent in Analysis and Design is a sound investment.** Projects that spend over 20% of their total effort in Analysis and Design complete sooner, cost less (use less effort), and have higher quality (fewer defects). Note that this item and the previous one are areas in which enlightened program and project management can directly affect productivity, cost, and quality.

Works Cited

Beckett, Donald. "Using Metrics to Develop a Software Project Strategy." 7th Annual CMMI Technology Conference & User Group presentation. Denver, Colorado. 12-15 November 2007. Conference Presentation.

Jones, Capers. "Project Management Tools and Software Failures and Successes." *Crosstalk* (July 1998): 13-17. Print.

Prev Section | *Prev Article* | *ToC* | *Next Article* | *Next Section*

Why Are Conversion Projects Less Productive than Development?

Donald Beckett

While doing research on projects counted in function points, the sample size was large enough (over 2000 projects) to allow me to compare the productivity of different project types. The QSM database uses these project categories:

- New Development (> 75% new functionality)
- Major Enhancement (25% - 75% new functionality)
- Minor Enhancement (5% - 25% new functionality)
- Conversion (< 5% new functionality)
- Maintenance

I calculated the normalized PIs for projects in each development classification compared to the QSM Business trend lines (Table 1). The advantage of this is that it takes into consideration the impact of size and shows how the productivity of each project "application type" differs from the QSM Business IT average. The datasets included medium and high confidence IT projects completed since 2000. When I obtained the results, I went back over my selection process and calculations to make sure I hadn't made a mistake. The numbers were that surprising. But, no, I hadn't "fat fingered" anything (neither physically nor mentally). Average productivity for conversion projects was more than a standard deviation below the QSM Business IT average.

Normalized Productivity (PI)					
	Median	**Average**	**1st Quartile**	**3rd Quartile**	**% Below QSM Business Average**
Conversion	-1.20	-1.06	-1.86	-0.36	81%
New Development	-0.20	-0.25	-1.03	0.52	57%

Table 1. Normalized Productivity (PI)

Conversions are often undertaken with the idea of reusing existing processes and functionality rather than re-inventing (and debugging) them. The objective is to save time and money while taking advantage of existing processes. While the intention is admirable,

conversions are not always time and money savers. Here are a few of the confounding factors that should be considered before embarking on a conversion project.

- The staff that developed the system you want to convert may no longer be available. In fact, the actual work you are planning may be done in another country by a team that has no previous exposure to the system. When application knowledge is minimal, the conversion team will spend a lot of time understanding how the current system works. Being human, they'll make a few mistakes, too.
- Application documentation, if it exists at all, may not be current. New teams with minimal application knowledge may have few resources on hand to help them gain that knowledge.
- While the business processes may be the same, their technical implementation on another platform may differ significantly. You will know *what* you want to accomplish; but *how* to do it still has to be fleshed out.
- Applications are developed around the available technology. The system you are converting does things in ways that may not make sense (or even be possible) on the target system. Some processes will have to be rewritten. At this point you are developing, testing, and integrating new code with an existing system.
- Changes will creep in. One of the best reasons for converting to a newer technology is to take advantage of the features it offers. You probably don't want your new web-based system to mimic the touch and feel of the IMS-COBOL one it is replacing.

All of these factors can reduce productivity and should be addressed honestly before beginning a conversion.

To illustrate this, I modeled both the development and the conversion of a 500 function point project in SLIM-Estimate®. I used an average productivity factor (PI) for the development project and a PI that is standard deviation below average for the conversion project. I assumed 25% re-use for the conversion project which lowered the actual function points being developed to 375. A labor rate of $10,000/person month was used for both projects. The results are captured in Table 2.

Development/ Conversion Comparison		
	Develop	**Convert**
Size in FP	500	375
Schedule (Months)	10.2	15.5
Cost (in thousands)	336.6	385.6
Effort (Staff Months)	34	39
productivity index (PI)	15.2	11.7

Table 2. Development/Conversion Comparison

While the assumption of 25% re-use is arbitrary, it illustrates that a significant part of the functionality will not have to be redeveloped in the conversion project. However, in this example the lower productivity of the conversion project offsets this and cost, schedule, and effort are all greater than on the 500 function point development project.

What does this all mean to a software project estimator? While development projects are demonstrably more productive than conversions, he or she needs to keep in mind how that productivity is determined. Parametric estimation tools like SLIM-Estimate® use the amount of software that is developed or modified as the size basis for determining productivity. If function points are used, this would consist of added, changed, and deleted function points.

Ideally, in a conversion project a great deal of the functionality is not modified and would, therefore, not be counted as part of the project size. As a result, a conversion project may deliver significantly more functionality than is accounted for in traditional productivity measures.

The decision to convert or redevelop a software system needs to account for the total amount of functionality that will be delivered. A good way to compare the alternatives in SLIM-Estimate® is to create a scenario for the development project based on developing all of the functionality that will be converted using an average productivity index (or one that is appropriate for your organization). Next, create an estimate based on the size of the conversion using a PI a little lower than 1 standard deviation below average. Be careful not to underestimate all of the functions that will require tweaking. Then, compare the two scenarios keeping in mind the degree to which the confounding factors mentioned above will come into play on the conversion.

All of this is not to say that conversions cannot succeed; they can. The completed conversion projects in our database testify to this. But, before you embark on one, keep in mind that converting a legacy system can be far more complex than it initially seems. In some cases redevelopment may be a viable alternative. Comparing what it takes to convert a legacy system to the resources required to build it from scratch can help you decide on the best course.

Small Teams Deliver Lower Cost, Higher Quality

Kate Armel

For this study, Best in Class projects were those that **delivered more than one standard deviation *faster***, but **used more than one standard deviation *less* effort** than the industry average for projects of the same size. A key characteristic of these top performing projects was the use of small teams: median team size for best in class projects was **4 FTEs** (full time equivalent) people versus **17 FTEs** for the worst performers.

What is the relationship between team size and management metrics like cost and defects? To find out, I recently looked at 1060 medium and high confidence IT projects completed between 2005 and 2011. These projects were drawn from the QSM database of over 10,000 completed software projects. The projects were divided into two staffing bins:

- **Small team projects (4 or fewer FTE staff)**
- **Large team projects (5 or more FTE staff)**

These size bins bracket the median team size of 4.6 for the overall sample, producing roughly equal groups of projects that cover the same size range. Our best/worst in class study found a **4 to 1 team size ratio** between the best and worst performers (see Figure 1).

Interestingly, using team size as a selection criterion and including average projects as well as high and low performers produced a very similar team size ratio:

- Large team projects had a median team size of **8.5 FTE staff**
- On average, small team projects used only **2.1 FTEs**

Note the variability in team size even for projects of the same size. If team size is as much a function of management style and resource availability as it is of project scope and required technical skill sets, it stands to reason that managers who add or remove staff from a project need to understand how using smaller vs. larger teams will impact cost and quality.

Regression trends were run through each sample to pinpoint the average Construct & Test effort, schedule, and quality at various points along the size axis. On small projects (5000 new and modified source lines of code), large teams achieved an **average schedule reduction**

of 24% (slightly over a month). But this improved schedule performance was costly: **project effort/cost tripled** and **defect density more than doubled**.

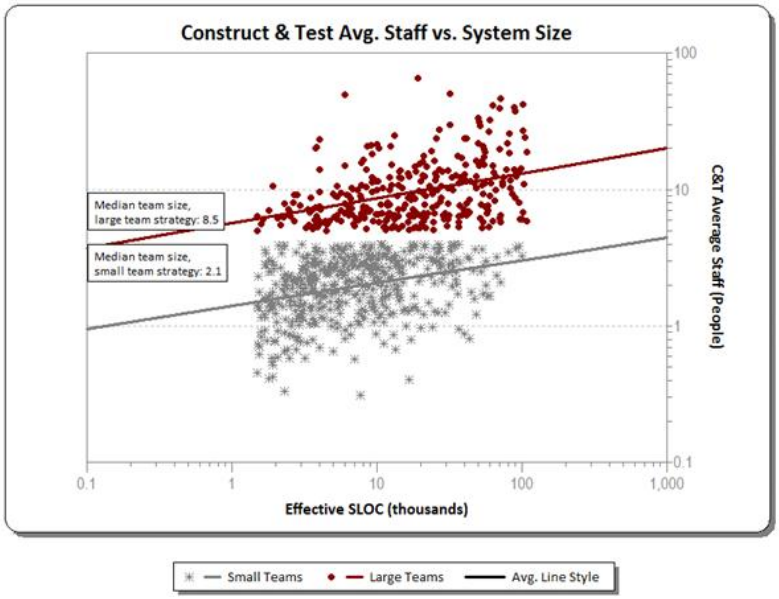


Figure 1. Construct & Code Average Staff vs. System Size

Larger projects (50,000 new and modified source lines of code) using large teams managed delivered **only 6% faster (about 12 days)** but **effort/cost quadrupled** and **defect density tripled**. The tradeoffs between team size and schedule, effort, and quality are easily visible by inspection of Figure 2 below. The schedule chart shows little difference between small and large teams (note the high degree of overlap between the two samples). But the effort and defect density charts show distinct differences (less overlap) between small and large team projects:

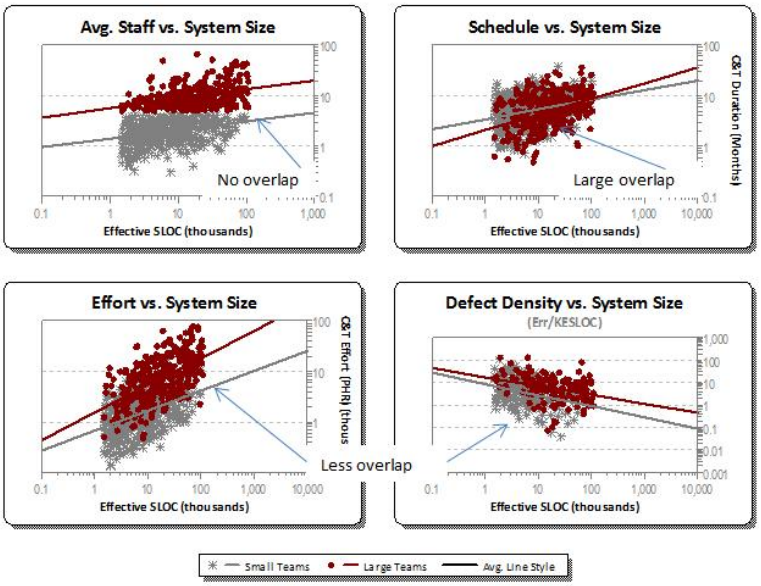


Figure 2. Effort/Staff Size vs. Defect Density for Small and Large Teams

So what might account for these results? Over three decades of data from the QSM database suggest that defect creation and density are highly correlated with the number of people (and hence communication paths) present on a given project. Larger teams create more defects, which in turn beget additional rework. Fixed code must be retested and the team may find new defects injected during rework. These unplanned find/fix/retest cycles take additional time, drive up cost, and cancel out any schedule compression achieved by larger teams earlier in the lifecycle.

In an earlier article on this topic, we saw that best in class software projects typically use smaller teams (four times smaller, on average) than poor performers. This post looked at the differences in cost and quality between small and large team projects. Larger teams realized modest schedule compression on small projects, but saw little or no improvement in time to market for larger projects. Regardless of project size, the large team strategy drastically increased cost and reduced quality.

Optimal Schedule Performance: Project/Environmental Factors with Most Impact on Schedule Performance

Paul Below

This article originally appeared in the Project Management Institute's conference proceedings, "PMI Global Congress 2013 – North America (2012), and is reprinted here with permission.

Speedy delivery is almost always one project goal, and often it is the primary goal or even a project constraint. Organizations, too, usually have shorter time to market as one of their primary goals.

In order to work on improving duration, we need to answer the question "What factors are most closely related to project duration?" After all, if we want to improve, we need to know where to focus our improvement efforts.

To answer this question we mined the QSM database, specifically projects that completed in the 21st century and had metrics that passed data quality checks. The techniques used are described in the Methodology section, including the computation of a standardized residual of duration. This standardized residual is used in this article to represent project duration.

Correlation factors

Examination of a large number of candidate factors (40 quantitative variables) revealed two general interesting areas that correlated with the standardized residual of duration versus size.

- Overlap in months shows a correlation with duration, positive correlation means more overlap yields longer durations
- The various error metrics and reliability metrics show correlation with duration
    - Error counts are positively correlated, more errors mean longer duration
    - Reliability (MTTD) is negatively correlated, higher reliability means shorter durations
    - Error per unit of size is positively correlated, higher defect density yields longer duration

Phase overlaps occur, generally, because project and program managers are attempting to shorten the overall project duration by having concurrent phases. How interesting, then, that overlaps are associated with longer durations.

The following two graphs at Figure 1 have the months of phase 2 overlap (Functional Design) on the horizontal axis. This is the number of months that Functional Design overlapped with Main Build. The vertical scale is the standardized residual of duration where higher values represent projects that had longer duration. The first graph (Figure 1, left graph) is for Business applications, the second (Figure 1, right graph) is Engineering. The sloping lines represent a linear regression and a 95% confidence interval on the mean.

In general, longer overlaps (in calendar months) result in higher duration that would be predicted.

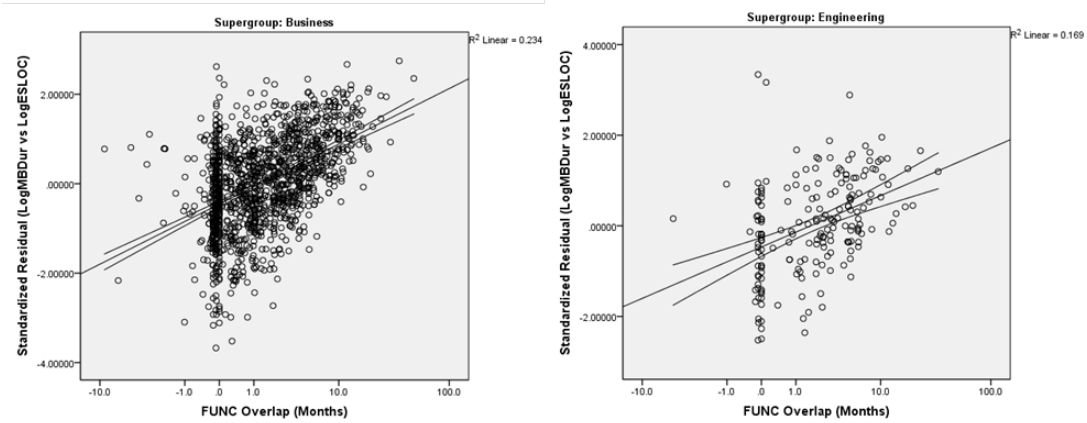


Figure 1. Functional Design Overlap Comparison for Business and Engineering Supergroups

What does a typical project with high overlap look like? And, how does it compare to a typical project with low overlap? Table 1 provides means and medians for the key metrics of effort, staff, duration and size in addition to functional design overlap months. The projects have been divided into two groups, based on whether the functional design overlap months was higher or lower than the median (Table 1).

Supergroup	Functional Design Overlap		FUNC Overlap (Months)	MB Effort (MM)	MB Duration (Months)	MB Peak Staff (People)	Effective SLOC	Standardized Residual (LogMBDur vs LogESLOC)
Business	Low	Median	.0	11.6	3.8	8.0	7714	-.35
		Mean	.2	37.8	4.6	14.0	20183	-.37
	High	Median	3.8	26.9	7.4	10.0	10809	.48
		Mean	5.2	68.4	8.4	18.2	48331	.47
Engineering	Low	Median	.0	27.5	5.2	7.8	12319	-.35
		Mean	.1	71.6	7.6	13.7	48087	-.38
	High	Median	4.5	178.8	16.0	15.8	79677	.29
		Mean	5.4	536.3	17.6	45.6	169833	.30

Table 1. Projects Grouped by Functional Design Overlap (Low or High)

To compare typical projects, the following graphs from SLIM-Estimate® use the median values for Business projects, and for simplicity include only phases 2 and 3.

Compare the amount of overlap and compare the PI in these two typical projects at Figures 2 and 3. The first graphs are for a low overlap median project, the second are for a high overlap median project. The PI for the low overlap is 14.5 and 11.1 for the high.

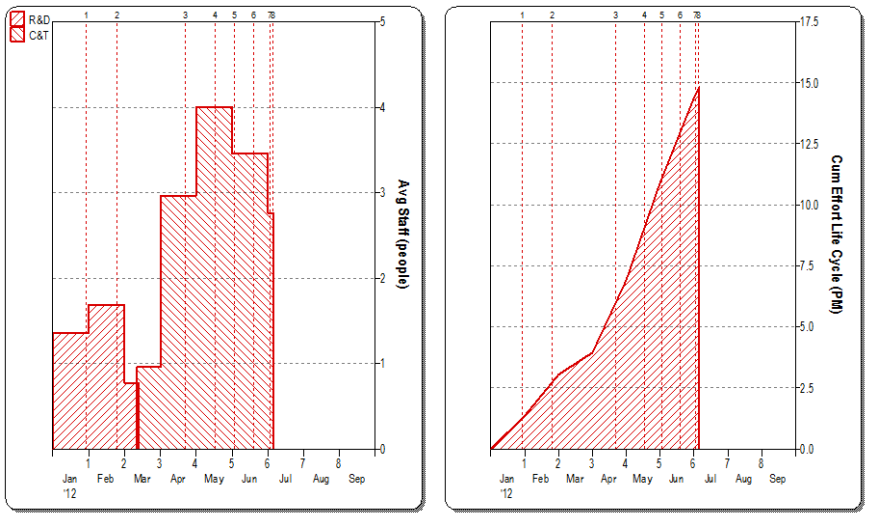


Figure 2. Overlap and PI Comparison

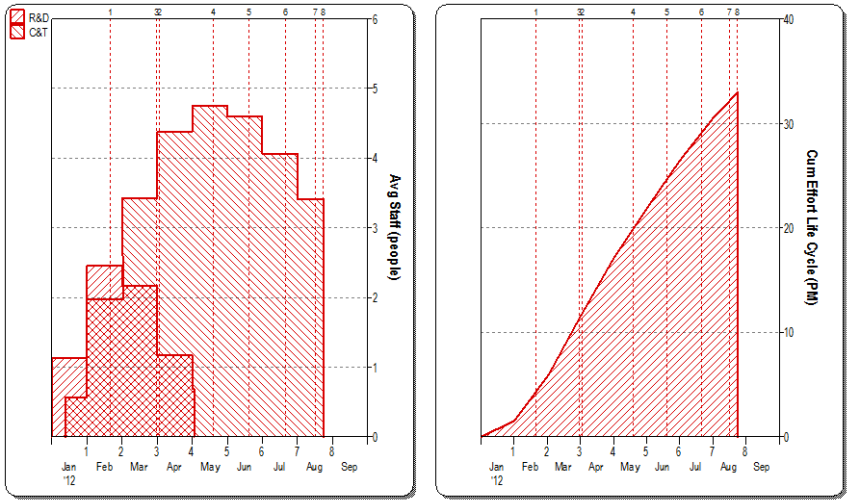


Figure 3. Overlap and PI Comparison

The second interesting correlation was quality factors. More errors results in longer durations, and correspondingly, higher reliability results in shorter durations. The following graph (Figure 4) is reliability (Mean Time to Defect) for business applications. Again, there is a trend line with 95% confidence interval on the mean placed on the projects.

One interesting item to point out is that the trend line is under zero across the entire range of reliability. This is because the standardized residual was determined for all business applications, but those projects that reported MTTD had, on average, shorter durations than those that did not report MTTD.

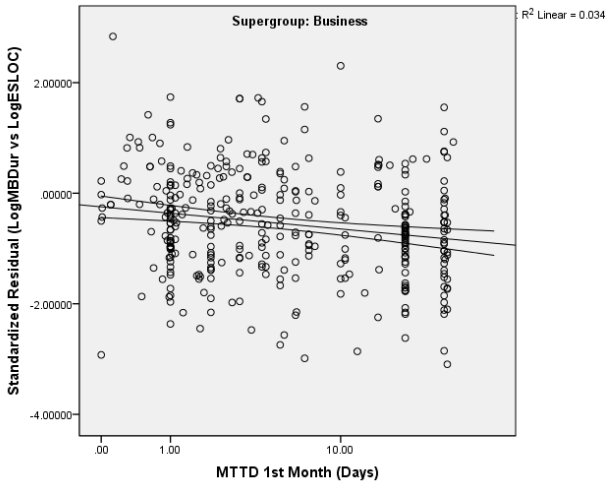


Figure 4. Standardized Residual vs. MTTD for Business Supergroup

"As the Japanese learned in 1950, productivity moves upward as the quality of process improves." W. E. Deming

So, for Business applications, we see that the initial quality of the product is the key. In other words, higher quality results in shorter durations. If two products are of similar initial quality, we would expect one with higher quality to have a longer duration because it would have undergone more thorough testing and debugging.

This is not exactly identical for Engineering applications. The correlation of MTTD and standardized residual is not significant for Engineering. In the following two box plots at Figure 5, each box represents a quartile of MTTD, so that the box on the left is the 25% of the projects with the worst reliability, the box on the right is the 25% of the projects with the best reliability.

Engineering is interesting, in a non-intuitive sort of way. Quality in engineering appears to be created by extending the duration (i.e., testing quality in), whereas in business the duration is more a direct result of the initial quality. In other words, in Engineering systems, the quality requirement drives the duration by affecting the testing and debugging time.

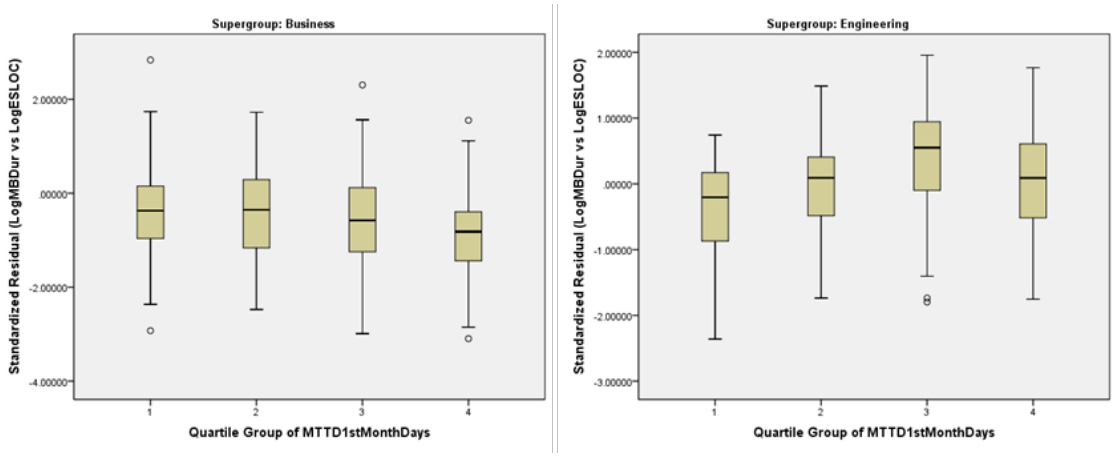


Figure 5. Comparison of MTTD vs. Standardized Residual for Business and Engineering Supergroups

Qualitative Assessment Factors

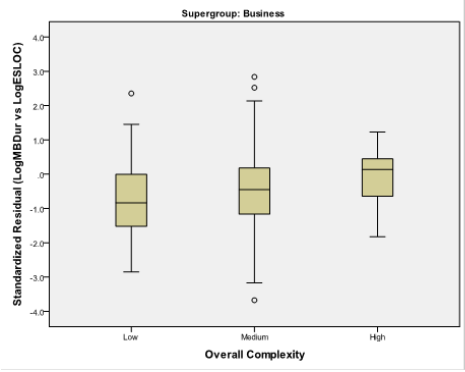
The assessment factors are on a scale of zero to ten, where zero means none, and ten means a high amount. The assessment factors that exhibit the strongest correlations or are the most important regression factors for duration prediction are described in this section.

Business Applications

Technical and communication complexity is important to the duration of business application development projects.

- **Overall complexity** is the overall technical complexity, higher numbers represent higher complexity. The coefficient is positive, meaning that higher complexity is longer duration.
- **Team Communication Complexity** is the level of team communication complexity. Higher numbers mean more complexity. The coefficient is positive, meaning that higher complexity is longer duration.

In the following boxplot (Figure 6), overall complexity of 1 to 4 is "Low," 5 to 8 is "Medium," and 9 to 10 is "High." The lowest complexity projects tend to have the shortest durations. The median business project with high complexity is 0.14 standard deviations above the duration trend line, whereas the median low complexity project is 0.83 standard deviations below the duration trend line. That is a difference of almost a full standard deviation.

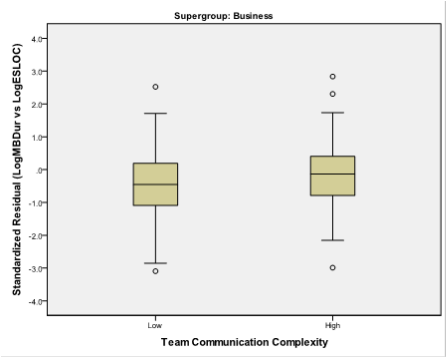


Standardized Residual (LogMBDur vs LogESLOC)

Overall Complexity	Median	Mean
High	.14	-.10
Medium	-.45	-.49
Low	-.83	-.74
Total	.02	.00

Figure 6. Overall Complexity Breakout for Business Supergroup

Projects with low team communication complexity tend to have the shortest durations. In the following box plot (Figure 7), Team Communication Complexity of 1 to 4 is "Low" and 5 to 10 is "High." Team communication complexity is a significant factor, although it does not have as strong an influence as overall complexity.



Standardized Residual (LogMBDur vs LogESLOC)

Team Communication Complexity	Median	Mean
High	-.13	-.15
Low	-.46	-.46
Total	.02	.00

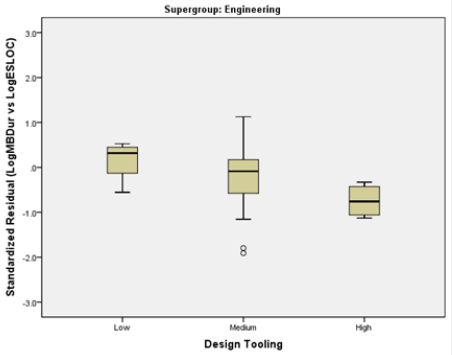
Figure 7. Team Communication Complexity Breakout for Business Supergroup

Engineering Applications

For engineering applications, three factors that had the highest significance to duration are:

- **Design tooling** is the capability of the design tool, 10 is high capability. The correlation is negative, so that higher capability results in shorter duration.
- **Closeness arch limit** is how close to the architectural limits of the development environment (memory, storage, etc.) The correlation is negative so that a higher closeness results in a longer duration.
- **Construction tooling** is the capability of the construction tool, 10 is high capability. The correlation is positive so a higher capability results in shorter duration.

Engineering projects with the best design tools tend to have shorter durations. In the following box plot (Figure 8), Design Tooling of 1 to 3 is "Low," 4 to 6 is "Medium," and 7 to 10 is "High."



Standardized Residual (LogMBDur vs LogESLOC)

Design Tooling	Median	Mean
High	-.76	-.74
Medium	-.09	-.20
Low	.32	.15
Total	-.00	.00

Figure 8. Design Tooling Complexity Breakout for Engineering Supergroup

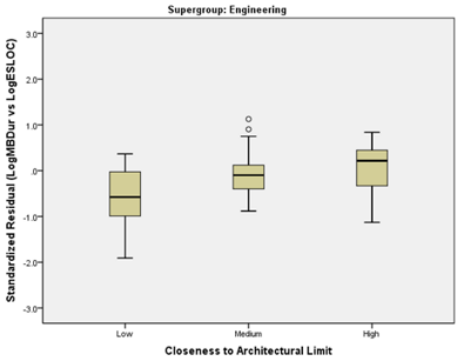
Durations became gradually shorter as Construction Tooling ratings increase from five to ten (Figure 9). Improving the tools from 1 to 6 makes little difference. In the following box plot, Construction Tooling of 1 to 6 is "Low," 7 to 9 is "High" and 10 is "Very High." A typical engineering project with construction tools rated as 10 has a duration that is a full standard deviation shorter than the typical engineering project.

Standardized Residual (LogMBDur vs LogESLOC)

Construction Tooling	Median	Mean
Very High	-.99	-.80
High	-.09	-.16
Low	.18	.20
Total	-.00	.00

Figure 9. Construction Complexity Breakout for Engineering Supergroup

For engineering projects, as the system approaches the architectural limits the duration increases. In the following box plot (Figure 10), Closeness to Architectural Limits of 1 to 2 is "Low," 3 to 5 is "Medium," and 6 to 10 is "High."



Standardized Residual (LogMBDur vs LogESLOC)

Closeness to Architectural Limit	Median	Mean
High	.22	.05
Medium	-.10	-.00
Low	-.58	-.57
Total	-.00	.00

Figure 10. Closeness to Architectural Limit Complexity Breakout for Engineering Supergroup

Myths

A number of factors are frequently considered to be important in determining duration of software projects. Among them are team skill levels and team size.

While there is a significant relationship between average team size and duration, the log of the average team size explains less than 3 percent of the variation in the duration residual. Other articles in this abstract look at the impact of staff size on various output measures.

Having a skilled and experienced team is certainly important for a number of reasons. However, team skill alone does not significantly impact project duration. Table 2 lists correlation coefficients (using a technique applicable for ordinal variables such as the qualitative assessment factors). The significance for Overall Personnel and Staff capability are high enough to cast doubt into whether a relationship actually exists (general, a significance of less than .05 is considered to be evidence for the existence of a relationship).

Correlation with Standardized Residual (LogMBDur vs LogESLOC)		
Overall Personnel	Correlation Coefficient	-.036
	Sig.	.252
Staff Capability	Correlation Coefficient	-.066
	Sig.	.138
Team Motivation	Correlation Coefficient	-.095
	Sig.	.035
Mgmt Effectiveness	Correlation Coefficient	-.106
	Sig.	.020

Table 2. Correlation Coefficients for Personnel, Staff, Team Motivation, and Management

- Overall personnel is the overall capability of the personnel involved in the project, where 1 is low and 10 is high.
- Staff capability is the capability and experience of the development team, where 1 is low and 10 is high.

However, there do appear to be some staff factors that impact duration. For example, team motivation and management effectiveness have a relationship with duration. Although the relationships are weak, the significance factor is sufficient to provide evidence that the relationships are real.

- Team motivation is the level of motivation of the development team, where 1 is low and 10 is high.
- Management effectiveness is the effectiveness of management and leadership, where 1 is low and 10 is high.

Both correlation coefficients are negative, which means that, in general, as team motivation or management effectiveness increase, duration decreases.

In the following boxplots (Figure 11), team motivation of 1 to 3 is low, 4 to 8 is medium, and 9 to 10 is high. Management effectiveness of 1 to 4 is low, 5 to 8 is medium, and 9 to 10 is high.

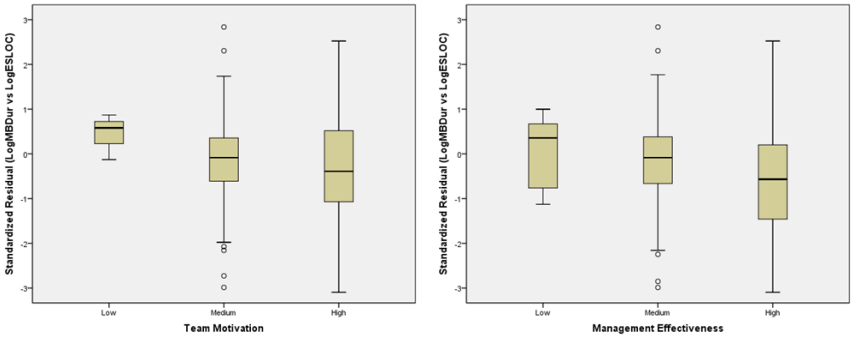


Figure 11. Team Motivation and Management Effectiveness Boxplots

Summary

Project duration (aka time to market) is often an important constraint. Organizations that want to shorten their project durations should improve their processes. This article has highlighted some of the factors that have a major impact on project durations. Organizations should identify and focus on their processes that control these factors.

In order to shorten project durations, it is important to:

- Improve the upstream quality of the product (inject fewer defects into the constructed product),
- Improve testing efficiency (especially in engineering applications) ,
- Track and use measures of product quality,
- Minimize the overlap between major phases (the four SLIM® phases),
- Reduce technical complexity and communication complexity where possible (especially for business application projects),
- Improve tools for design and construction (especially for engineering application projects),
- Either improve the architecture or modify designs so that the engineering projects are not close to the limits of the architecture (memory, storage, speed, etc.), and
- Keep the development team motivated, and retain effective managers and leaders.

Glossary

Box plot: a graph with boxes that represent interquartile range, the box represents the second and third quartiles, the dark line inside the box represents the median. A box plot is used as a visual aid in comparing multiple distributions.

Correlation Coefficient: the square root of r square, it can range from -1 to +1. The sign indicates the direction of the relationship, the absolute value represents the extent of the relationship.

Logarithm: frequently used to normalize data that has a skewed distribution. The logarithm of a number to a given base is the exponent to which the base must be raised in order to produce that number.

Mean: arithmetic average, sum of the values in the data set divided by the number of values. The mean is sensitive to outliers and skewed data sets.

Median: the middle value when all the values in a data set are arranged in ascending or descending order. It is the 50th percentile. The median is not skewed by outliers.

Ordinal: a metric that has order, but no ratio scale. The numbers represent ranks, indicate relative magnitude, but the difference between the ranks are not assumed to be equal.

R square: indicates the proportion of the variance in the dependent variable that is statistically explained by the regression equation.

Regression: the purpose of regression is to estimate an output variable (dependent) given the value of one or more independent variables.

Residual: The difference between the predicted value (e.g., using a regression) and the actual value.

Significance: the conditional probability that the observed statistic could occur by chance.

Standard Deviation: the square root of the variance. Standard deviation is frequently used as a measure of dispersion in a set of data that has a normal distribution.

Data Mining for Process Improvement

Paul Below

*This article originally appeared in the Department of Defense journal **CrossTalk** Jan/Feb 2011 (pp 10-14) and is reprinted here with permission.*

Introduction

What do you do if you want to create an estimate and you have 100 candidate variables to use in your estimating model?

This is also a common question for CMMI® high maturity organizations that need to create process performance models. According to SEI, process performance models are:

"A description of relationships among attributes of a process and its work products that is developed from historical process-performance data and calibrated using collected process and product or service measures from the project and that are used to predict results by following a process."

High maturity organizations typically use process performance models for operational purposes such as project monitoring, project planning, and to identify and evaluate improvement opportunities. They typically are used to predict many output variables including defects, test effectiveness, cost, schedule, and duration, requirements volatility, customer satisfaction, and work product size.

Data mining techniques can be used to filter many variables to a vital few to build or improve predictive models. Specific examples are provided in four categories: classification, regression, clustering, and association.

When creating an estimating model or a process performance model, the primary challenge is how to start. Regardless of the variable being estimated (e.g., effort, cost, duration, quality, staff, productivity, risk, size), there are many factors that influence the actual value and many more that could be influential.

The existence of one or more large datasets of historical data could be viewed as both a blessing and a curse. The existence and accessibility of the data is necessary for prediction, but traditional analysis techniques do not provide us with optimum methods for identifying key independent (predictor) variables

Data mining techniques can be used to help thin out the forest so that we can examine the important trees. Hopefully, this article will encourage you to learn more about data mining, try some of the techniques on your own data, and see if you can identify some key factors that you can control or use to build a predictive model.

What Is Data Mining?

There are many books on data mining, and each one has a slightly different definition. The definitions commonly refer to the exploration of very large databases through the use of specialized tools and a process. The purpose of the data mining is to extract useful knowledge from the data, and to put that knowledge to beneficial use.

Data mining can be viewed as an extension of statistical analysis techniques used for exploratory analysis, incorporating new techniques and increased computer power. A few sources with details on data mining are listed at the end of this article.

There are a number of myths that have grown up regarding the use of data mining techniques. Data mining is useful but not a magic box that spits out solutions to problems no one knew existed. Still required for success:

- business domain knowledge
- the collection and preparation of good data
- data analysis skills
- the right questions to ask

Techniques for cleansing data, measuring the quality of data, and dealing with missing data are topics that are outside the scope of this article.

Researchers have created a number of new data mining algorithms and tools in recent years, and each has theoretical advantages and avid proponents. However, for the purpose of getting started with estimate model creation, tool selection is not critical. The comparative theoretical advantages and disadvantages of the techniques and tools are not important to our purpose of identification of key factors. The practical advice is to try as many different techniques as possible, as the difficult time-consuming task is data preparation. Refer to a list of tools in the References section.

Model Creation Challenges

People love to interpret noise. Regardless of what the data shows, the audience will offer theories to explain the causes for what is observed. If a graph shows that performance has improved, someone will offer an explanation for why that happened. If you tell the audience that the graph was upside down, and performance has actually decreased, just as quickly someone will propose a reason for why that happened.

Figure 1 is an image of random noise. If you stare at it long enough, you will start to see some patterns. People are pretty good at pattern recognition, even if no pattern actually exists. That is one reason why statistical quality control, data mining, and hypothesis testing are useful – to help determine whether the patterns we think we see are real or whether they could be explained by randomness alone. Another reason is to help us find patterns that are real but are difficult to see.

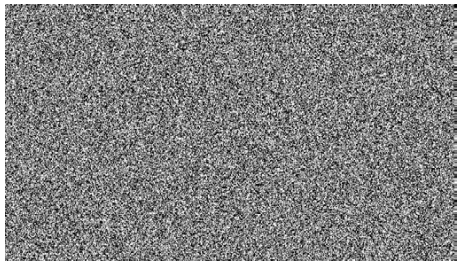


Figure 1. Random noise

Exploratory analysis, including data mining, utilizes existing data that has already been collected. There are challenges with using such data, including:

• The databases already exist and almost always were created without considering analytical needs.

• Databases generally are built by committees, or have evolved from older systems through multiple stages. The variables stored include items that were used long ago as well as fields that someone thought might be useful someday, mixed in with data that are currently necessary. Many of the fields have values that are hard to decipher, or were used inconsistently by different populations of users.

• The structure of the data is often bad or the keys are not appropriate, making data extraction difficult.

Regardless of the data mining tools used, data extraction and validation is a major undertaking.

Once the data is extracted and placed in a readable format, the analyst is faced with dozens of input variables. Which of those variables should be used in the model?

It is common for our variables to exhibit colinearity. Colinearity is when the variables are highly correlated with each other. In practical terms this means that those variables are measuring the same or similar things. Dumping all of these variables into a regression equation is not a way to receive a useful output.

Data mining can help us thin out the forest so that we can see the most important trees. Many of the data mining techniques can be used to identify independent variables that are influential in predicting the desired result variable. Success will depend more on the mining process than on the specific tools used.

Data Mining Models

> *"Statisticians, like artists, have the bad habit of falling in love with their models."*
> *- George Box*

Data mining can aid in hypothesis testing as well as exploratory analysis.

There are many pure data mining products on the market, but they are typically very expensive. Some of the common techniques, however, are supported by basic statistical analysis tools which are much less costly. These techniques include all of the examples provided in this document. Examples of statistical analysis tools that support data mining models can be placed into four categories as described in Table 1:

Category	Description	Purpose	Primary Data Type
Classification	Split the data to form	Predict response variable	Discrete is best
Regression	Best fit to estimating model	Predict response variable	Continuous (ratio or interval)
Clustering	Group cases that are similar based	Identify homogeneous groups of cases	Any
Association	Group variables that are similar	Determine co-linearity, identify factors that explain correlations	Ratio or interval (not categorical)

Table 1. Data Mining Models

We will now look at an example from each of the four categories.

Classification Example

One classification technique is a tree. In a tree, the data mining tool begins with a pool of all cases and then gradually divides and subdivides them based on selected variables.

The tool can continue branching and branching until each subgroup contains very few (maybe as few as one) cases. This is called overfitting, and the solution to this problem is to stop the tool before it goes that far.

For our purposes, the tree is used to identify the key variables. In other words, which variables does the algorithm select first? Which does it pick second or third? These are good candidate variables to be used in an estimating model, since the tree selected them as the major factors.

In Figure 2, we see an example that started with a data set of 841 cases, taken from a database of client information. Prior to running the tree, each of the 841 clients was assigned to one of four groups. The assignments were made based on information about customer satisfaction. The goal of the analysis was to see if there were key factors that could be used to predict which group a client would fall in. This prediction would then be used to identify clients that were likely to become less satisfied in the future, and determine actions that could be taken to improve client satisfaction.

In the top box of the tree, each group is listed with the fraction of the cases. So, for example, Group I contains 6.8% of the 841 cases. The total for the four groups will be approximately equal to 1 (100%) allowing for round off.

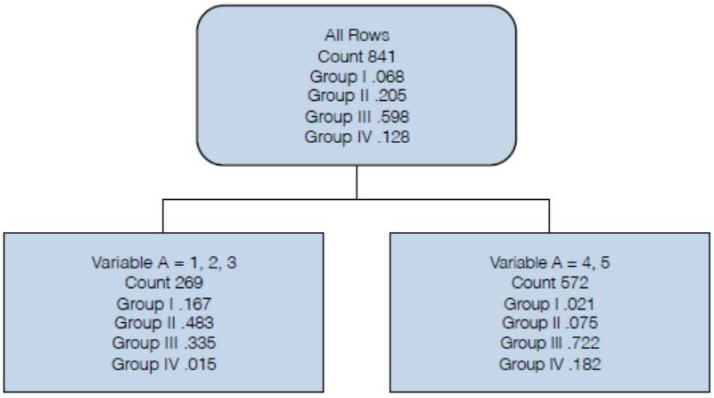


Figure 2. Tree Example

The tree algorithm examined all of the variables and selected Variable A to be the first branch. Variable A has possible integer values from 1 to 5. As we can see, the algorithm put the cases where Variable A is equal to 1, 2, or 3 in the left branch and those with Variable A equal to 4 or 5 in the right branch.

The left branch has 269 cases, including most of the cases in Groups I and II (the 269 cases are composed of 16.7% Group I and 48.3% Group II, compared to the right branch which is composed of only 2.1% Group I and 7.5% Group II). The right branch ended up with 572 cases, including most of the cases in Groups III and IV.

Variable A by itself is not a sufficient predictor to use as a predictive model. However, the tree is telling us that Variable A is one important factor. The tree would have additional branches, but Figure 2 is sufficient to aid in explaining how the tree is used.

Regression and Correlation Examples

The data used in the remaining examples came from industry data. It is based on a sample of 193 projects extracted from a corporate database.

The output in the examples is for illustrative purposes and should not be used to reach conclusions about performance of specific software projects.

Stepwise regression is a type of multivariate regression in which variables are entered into the model one by one, and meanwhile variables are tested for removal. It can be a good model to use when supposedly independent variables are correlated. Stepwise regression is one of the techniques that can help thin out the forest and find important predictive factors.

Table 2 is a summary output of a stepwise regression that went through nine steps to build the best model. It was created in SPSS, although other statistical packages produce similar results. The dependent variable being predicted was errors detected prior to deployment. The stepwise regression selected nine variables that fit the threshold for inclusion, while excluding 20 other variables (not listed).

Model	R	R Square	Adjusted R Square	Std. Error of the Estimate
9	.840	.706	.691	330.332

Model		Sum of Squares	df	Mean Square	F	Sig.
9	Regression	47883321.563	9	5320369.063	48.757	.000
	Residual	19968796.365	183	109119.106		
	Total	67852117.927	192			

Predictors: (Constant), Effective SLOC, Life Duration (Months), MB Time Overrun %, MB Effort (MM), Life Peak Staff (People), Data Complexity, MBI, MB Effort %, Mgmt Eff.
Dependent Variable: Errors (SysInt-Del)

Table 2. Regression Summary

The nine variables selected by the stepwise regression were, in the order the tool selected them: effective source lines of code; project life cycle duration in months; percent of duration overrun of Main Build (design through deploy); Main Build man months of effort; peak staff; data complexity; Putnam's Manpower Buildup Index; percent of effort expended in Main Build; and management effectiveness. Note that two of these nine variables (data complexity and management effectiveness) are qualitative, scored on a scale of one to 10 where five is average and 10 is high.

The first number to look at in Table 2 is the Sig (significance) in the rightmost column. The most commonly used significance threshold is .05, which means that the variable or model would be significant at the 95% level. In the example, the value .000 means that we have less than

a one in a thousand chance of being fooled by random variation into thinking this model is significant.

Although all nine variables selected are clearly significant, the overall model created has an adjusted R square of .691, which means that these nine variables taken together are explaining about 69% of the variation in errors found. This may not be the best model to use for estimating, but it is important to look at each of the nine variables if the intent is to create an estimating model or if we need to reduce the number of errors found in the future.

The coefficients of the stepwise regression formula are displayed in Table 3. Each variable is listed next to the coefficient B, which is the multiplier in the linear equation.

Coeffecients: Dependent Variable: Errors (Sysint-Del)

	Unstandardized Coefficients		Sig.	95% Confidence Interval for B	
Variable	**B**	**Std. Error**		**Lower Bound**	**Upper Bound**
(Constant)	-580.411	239.656	.016	-1053.255	-107.568
Effective SLOC	.001	.000	.000	.001	.001
Life Duration (Months)	27.633	5.832	.000	16.126	39.139
MB Time Overrun %	.026	.006	.000	.015	.037
MB Effort (MM)	1.535	.326	.000	.892	2.177
Life Peak Staff (People)	-7.438	1.905	.000	-11.197	-3.679
Data Complexity	66.840	18.269	.000	30.795	102.886
MBI	33.683	14.609	.022	4.859	62.507
MB Effort %	3.924	1.552	.012	.862	6.987
Mgmt Eff.	-50.012	22.775	.029	-94.948	-5.076

Table 3. Regression Coefficients

The equation that yielded the adjusted R square of .691 is:

Errors = -580 +(.001*ESLOC)+(27.6*Duration)+(.026*overrun%)+(1.5*MB Effort)-(7.4*peak staff)+(66+data complexity)+(33.68*MBI)+(3.9*MB effort %)-(50*Mgmt Eff)

The factors in the equation can be determined from reading the numbers in the B column.

A negative number means a negative correlation. One counterintuitive result of this example is the coefficient for peak staff. The negative coefficient means, in this model, that the larger the peak staff, the smaller the number of errors detected. This type of result is why it is necessary to evaluate the data in more depth and conduct additional analysis before using the model. Sometimes, negative correlations are expected. For example, management effectiveness has a negative coefficient, meaning that a higher effectiveness results in a lower number of errors.

The two rightmost columns of Table 3, the 95% confidence intervals, are useful as an indication of the uncertainty in the coefficients. The lower and upper bound for any variable

should not straddle zero. If it did, that would be an indication that we lack confidence in the factor B. Another method is to compare the value of the standard error to the value of the coefficient; ideally the standard error should be much smaller than the coefficient B. Also, the Sig should be small, ideally less than .05.

In addition to regression, correlation can be used to identify candidate important variables. This can be done by selecting the dependent variable first for the correlation and then the list of independent variables. There are different types of correlation that can be used. For ratio data, Pearson correlation can be used. For ordinal data, Kendall's Tau-B will work. For nominal (categorical) data, a chi square test can be used on a crosstab (two-way table) to determine significance.

It is important to note that these tests will determine linear correlations. Sometimes correlations exist but are nonlinear. One technique for exploring those relationships is transformation, which is not discussed in this paper.

Clustering Example

Cluster techniques detect groupings in the data. We can use this technique as a start on summarization and segmentation of the data for further analysis.

Two common methods for clustering are K-Means and hierarchical. K-Means iteratively moves from an initial set of cluster centers to a final set of centers. Each observation is assigned to the cluster with the nearest mean. Hierarchical clustering finds the pair of objects that most resemble each other, and then iteratively adds objects until they are all in one cluster. The results from each stage are typically saved and displayed numerically or graphically as a hierarchy of clusters with subclusters.

Table 4 is the output of a K-Means example run from the sample with the output constrained to create exactly three clusters. The tool placed the largest projects in the first two clusters. These projects had more errors, more staff, and higher productivity than the third cluster. One difference between the first two clusters is that the projects in the second cluster tended to have poor estimates of effort.

Final Cluster Centers

	Cluster		
	1	2	3
Project Count	5	22	166
Life Effort (MM)	750.7	617.8	89.1
Errors (SysInt-Del)	1898	1030	186
Errors First Month	138	117	8
Total FP	37167	26533	2648
Effective SLOC	1272194	298791	26444
Life Duration (Months)	21.3	18.4	9.3
Life Peak Staff (People)	56.5	61.1	15.4
Life Avg Staff (People)	23.8	26.5	7.1
MB Eff Overrun %	.0	62.0	45.8
SLOC/MB MM	2384.5	1606.4	910.9
Putnam's PI	24.4	21.5	14.1

Table 4. Cluster Example

We may want to stratify the projects into groups based on the above distinctions prior to conducting additional analysis. This may result in the need for more than one estimating model, or more than one process improvement project.

Association Example

Association examines correlations between large numbers of quantitative variables by grouping the variables into factors. Each of the resulting factors can be interpreted by reviewing the meaning of the variables that were assigned to each factor. One benefit of association is that many variables can be summarized by just a few factors.

In the following example using actual data, Principal Components analysis was used to extract four components. The scree plot in Figure 3 was used to determine the number of components to use. The higher the Eigenvalue, the more important the component is in explaining the associations. Selection of the number of components to use is somewhat arbitrary, but should be a point at which the Eigenvalues decline steeply (such as between components 2 and 3, or between 4 and 5). It turned out in this example that the first four components account for roughly half of the variation in the data set (included in other output from the principal components tool, not shown here), making four a reasonable choice.

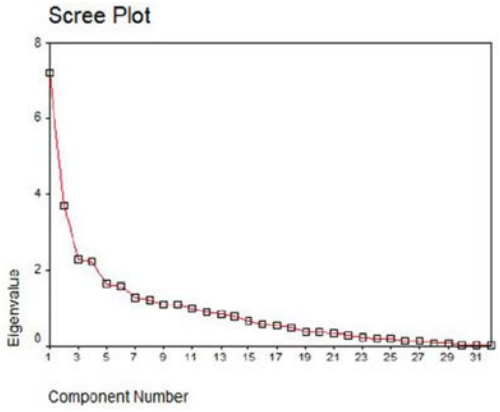


Figure 3. Scree Plot for Association Example

Table 5 displays variables with the most significant output for each component. The important numbers in the figure are those with relatively large absolute values and have been shaded for easy reference.

- Component 1 is composed of a market basket of variables related to effort and size (the variables aligned with the shaded numbers in component 1).
- Component 2 grouped variables related to the development team: knowledge, turnover, and skill.
- Component 3 isolated the Manpower Buildup Index, which is the speed at which staff is added to a project.

- Component 4 linked the percent of effort expended in functional requirements to the percent expended in the Main Build (design through deploy).

Variables that are seen to be related should be combined (or one should be chosen as the representative) as an input variable when creating prediction models or identifying root causes.

	Component			
	1	2	3	4
Life Effort (MM)	.920	-.152	.196	-.006
Effective SLOC	.652	.111	-.475	.106
Life Duration (Months)	.658	-.198	-.429	.066
Life Peak Staff (People)	.865	-.115	.338	-.137
Life Avg Staff (People)	.823	-.157	.381	-.156
FUNC Effort (MM)	.880	-.169	.151	.098
MB Effort (MM)	.925	-.160	.065	-.122
Func Effort %	-.241	.088	.236	.719
MB Effort %	-.059	-.072	-.247	-.765
Knowledge	.186	.770	.161	-.076
Staff Turnover	.083	-.717	.049	.110
Dev Team Skill	.133	.746	.029	-.225
MBI	-.006	-.011	.640	-.200

Table 5. Association Example Output

Summary

Once data has been collected and validated, the hardest work is behind you. Any data mining tools that are available to the researcher can be used relatively quickly on clean data. These data mining techniques should be used to filter an overwhelming set of many variables down to a vital few predictors of a key output (for example, quality).

Determination of the vital few is a key component of process improvement (such as Six Sigma projects) activities as well as prediction. With those key drivers or influencers of quality in hand, improvements can be designed and implemented with fewer iterations, effort, or time.

In addition to process improvement activities, we use the "vital few" to build error prediction models, and then use the models to tune parametric project estimates for specific clients. The project estimate and plan is thereby not only an estimate of duration and cost to complete construction, but also includes the prediction of when the system will be ready for prime time.

History is the Key to Estimation Success

Kate Armel

This article originally appeared in the Data & Analysis Center for Software (DACS) ***Journal of Software Technology*** *15-1 February 2012 (pp 16-22) and is reprinted here with permission.*

It was late afternoon in April of 1999 when the phone in my office rang. The conversation went something like this:

"This software estimate just landed on my desk and I need to finish it by close of business today to support a fixed price bid."

"What can you tell me about this project?"

"We're rewriting an existing mainframe billing system developed in COBOL. The new system will be written in C++, so it should be much smaller than the old system."

"Great –perhaps we can use the existing system as a rough baseline. How big is it?"

"I don't have that information."

"Will this be a straight rewrite, or will you add new features?"

"Not sure – the requirements are still being fleshed out."

"What about resources? How many people do you have on hand?"

"Not sure – the team size will depend on how much work must be done... which we don't know yet."

"Can we use some completed projects to assess your development capability?"

"Sorry, we don't have any history."

"That's OK –even without detailed information on scope, resources, or productivity we should still be able to produce a rough order of magnitude estimate based on relevant industry data."

"Rough order of magnitude??? My boss will never accept that much risk on a fixed price bid! Isn't there some general rule of thumb we can apply?"

Welcome to the world of software cost estimation where the things we know – the known knowns - are often outweighed by the things we don't know. Numerous estimation methods exist. Scope is described using effort, delivered code volume, features, or function points. Expert judgment, Wideband Delphi, top down, bottom up, parametric and algorithmic models each have their determined champions. But regardless of method, all estimates are vulnerable to risk arising from uncertain inputs, requirements changes, and scope creep. Skilled estimators and better methods can reduce this risk, but they can't eliminate it. Thus, the ability to identify and account for uncertainty remains a vital component of successful risk management.

Estimation Accuracy vs. Estimation Usefulness

How accurate is the average software cost estimate? Industry statistics vary as widely as the estimates they seek to measure. One oft-cited study – the Standish Group's Chaos Report – concludes that only one third of software projects deliver the promised functionality on time and within budget (The Standish Group, 2009). A later IEEE study (Eveleens and Verhoef, 2010) noted several gaps in the Standish Group's criteria for estimation accuracy:

> …the [Standish] definitions don't cover all possibilities. For instance, a project that's within budget and time but that has less functionality doesn't fit any category. … The Standish Group's measures … neglect under runs for cost and time and over runs for the amount of functionality.

When studies rely on different definitions of estimation success or failure, we should expect their assessments of estimation accuracy to exhibit considerable variability. The existence of different standards raises an intriguing question: what makes an estimate "accurate"?

Most quality control measures for estimates compare estimated cost/effort, schedule, or scope to their actual (final) values. The problem with this formulation is that "accurate" estimates are an integral part of feasibility decisions made very early in the project lifecycle; long before anything but the most generic information about the system's intended features or use can be known with reasonable certainty. The technologies used to implement the requirements may be unknown and the schedule, team size, required skill mix, and project plan have yet to be determined. As design and coding progress, the list of unknowns grows shorter and decreasing uncertainty about the estimation inputs lowers the risk surrounding the estimated cost, schedule, and scope. Unfortunately, most organizations must make binding commitments before detailed and reliable information about the project is available.

Given the degree of uncertainty surrounding early estimates – and the correspondingly broad range of possible time/effort/scope combinations - estimation accuracy may be less important than estimation usefulness. In an article for the ACM, Philip Armour explores the difference between these two concepts (Armour):

> The commitment is the point along the estimate probability distribution curve where we promise the customer and assign resources. This is what we need to hit, at least most of the time. It is not a technical estimation activity at all but is a risk/return based business

activity. It is founded on the information obtained from the estimate, but is not the estimate. Using Figure 1 as an example, if we needed an accurate commitment in the earliest (Initial Concept) phase based on how the diagram shows the project actually worked out, we would have had to commit at around a 75% probability. From the figure, committing to the "expected" result at Initial Concept would have led to a significant overrun beyond that commitment, and the project would have "failed." We can consider the 50% (expected) result to represent the cost of the project and the 25% increment to the higher commitment level to represent the cost of the risk of the project.

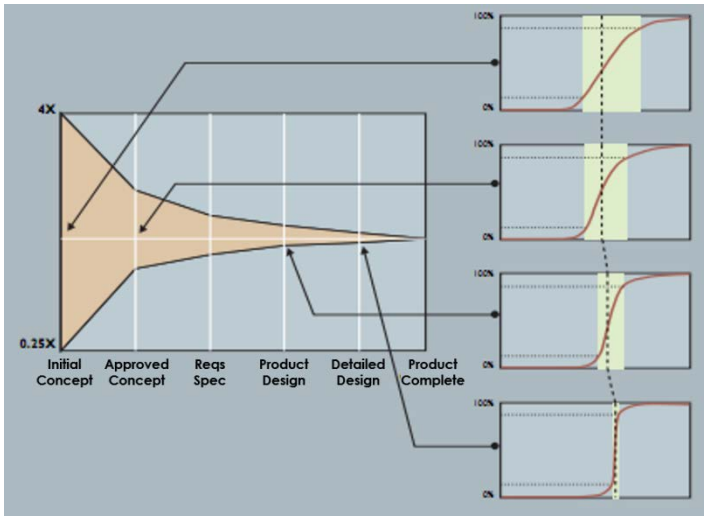


Figure 1. Commitments Made Early in Project Lifecycle Must Account for Greater Uncertainty Surrounding Estimation Inputs

Measures of estimation accuracy that treat an estimate as "wrong" or a project as "failed" whenever the final scope, schedule, or cost differ from their estimated values penalize estimators for something outside their control: the uncertainty that comes from incomplete information. We should measure deviations between estimated and actual project outcomes because this information helps us quantify estimation uncertainty and account for it in future estimates. But if measurement becomes a stick used to punish estimators, they will have little incentive to collect and use metrics to improve future estimates.

Understanding and Assessing Tradeoffs

Figure 2. Management Trade-offs

An old project management maxim succinctly summarizes the choices facing software development organizations: "*You can have it fast, cheap, or good. Pick two.*" Given that estimates (and therefore, commitments) are made early in the project lifecycle when uncertainty is high and the range of possible solutions is still wide, how do we select plans with a high probability of success? A thorough understanding of management tradeoff s can help. The idea behind the infamous Project Management Triangle (Figure 2) is simple but powerful: the tradeoffs between software schedule, effort or cost, and quality are both real and unforgiving. Thanks to the work of pioneers like Fred Brooks, most software professionals now accept the existence and validity of these tradeoffs but as Brooks

himself once ruefully observed, quoting famous maxims is no substitute for managing by them.

With so many unknowns out there, why don't we make better use of what we do know? Most software "failures" are attributable to the human penchant for unfounded optimism. Under pressure to win business, organizations blithely set aside carefully constructed estimates and ignore sober risk assessments in favor of plans that just happen to match what the company needs to bid to secure new business. Lured by the siren song of the latest tools and methods, it becomes all too easy to elevate future hopes over past experience.

This behavior is hardly unique to software development. Recently two economists (Carmen Reinhart and Kenneth Rogoff) cited this tendency to unfounded optimism as one of the primary causes of the 2008 global financial crisis. Their exhaustive study of events leading up to the crash provides powerful evidence that optimism caused both banks and regulators to dismiss centuries-old banking practices. They dubbed this phenomenon the "This Time Is Different" mentality. Citing an extensive database of information gleaned from eight centuries of sovereign financial crises, bank panics, and government defaults, Reinhart and Rogoff illustrate a pattern that should be depressingly familiar to software professionals: without constant reminders of past experiences, our natural optimism bias makes us prone to underestimate risk and overestimate the likelihood of positive outcomes.

The best counter to unfounded optimism is the sobering voice of history, preferably supported by ample empirical evidence. This is where a large historical database can provide valuable perspective on current events. Software development is full of complex, nonlinear tradeoffs between time, effort, and quality. Because these relationships are nonlinear, a 20% reduction in schedule or effort can have vastly different effects at different points along the size spectrum. We know this, but the human mind is poorly equipped to account for non-intuitive exponential relationships on the fly.

Without historical data, estimators must rely on experience or expert judgment when assessing the potential effects of small changes to effort, schedule, or scope on an estimate. They can guess what effect such changes might have, but they cannot empirically prove that a change of the same magnitude may be beneficial in one case but disastrous in another. The presence of an empirical baseline removes much of the uncertainty and subjectivity from the evaluation of management metrics, allowing the estimator to leverage tradeoffs and negotiate more achievable (hence, less risky) project outcomes. One of the most powerful of these project levers is staffing. A recent study of projects from the QSM database (Armel) used 1060 IT projects completed between 2005 and 2011 to show that small changes to a project's team size or schedule dramatically affect the final cost and quality.

To demonstrate the power of the time/effort tradeoff, projects were divided into two "staffing bins":

- Projects that used small teams of 4 or fewer FTE staff
- Projects that used large teams of 5 or more FTE staff

The size bins span the median team size of 4.6, producing roughly equal samples covering the same size range with no overlap in team size (Figure 3). Median team size was **8.5 for the large team projects** and **2.1 for the small team projects**, making the ratio of large median to small median staff approximately 4 to 1. The wide range of staffing strategies for projects of the same size is a vivid reminder that team size is highly variable, even for projects of the same size. It stands to reason that managers who add or remove staff from a project need to understand the implications of such decisions.

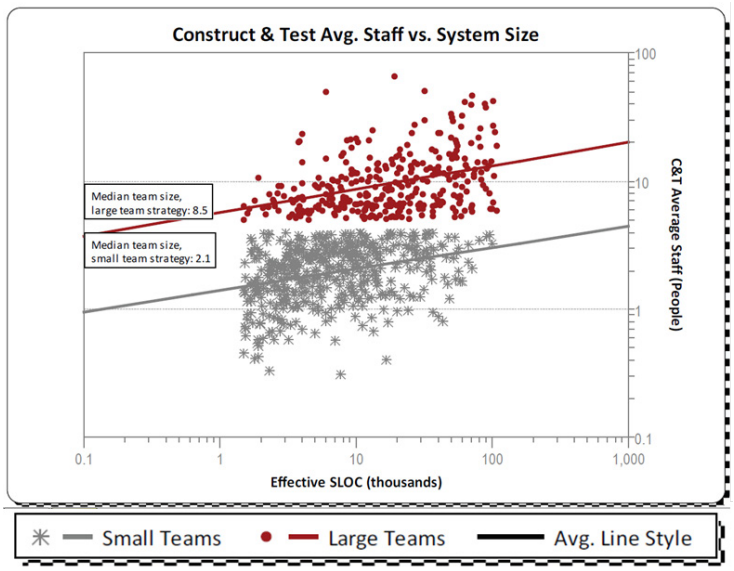


Figure 3. Construct & Test Average Staff vs. System Size

Regression trends were run through each sample to determine the average Construct & Test effort, schedule, and quality at various points along the size axis. Table 1 below shows the results.

At 5K ESLOC	Schedule (Months)	Effort (Person Hours)	Defect Density (Defects per K ESLOC)
Small teams	4.6	1260	3.7
Large teams	3.5	4210	9.2
Avg. Difference (Large team strategy)	-24%	334%	249%
At 50K ESLOC			
Small teams	7	3130	1.2
Large teams	6.6	13810	3.9
Avg. Difference (Large team strategy)	-6%	441%	325%

Quantitative Software Management, Inc.

Table 1. Small Team vs. Large Team Performance across Size Regimes

For very small projects (defined as 5000 new and modified source lines of code), using large teams was somewhat effective in reducing schedule. The **average reduction was 24%** (slightly over a month), but this improved schedule performance carried a hefty price tag: **project effort/cost tripled** and **defect density more than doubled**.

For larger projects (defined as 50,000 new and modified source lines of code), the **large team strategy shaved only 6% (about 12 days) off the schedule** but **effort/cost quadrupled** and **defect density tripled**.

The relative magnitude of tradeoffs between team size and schedule, effort, and quality is easily visible: large teams achieve only modest schedule compression while causing dramatic increases in effort and defect density (Figure 4).

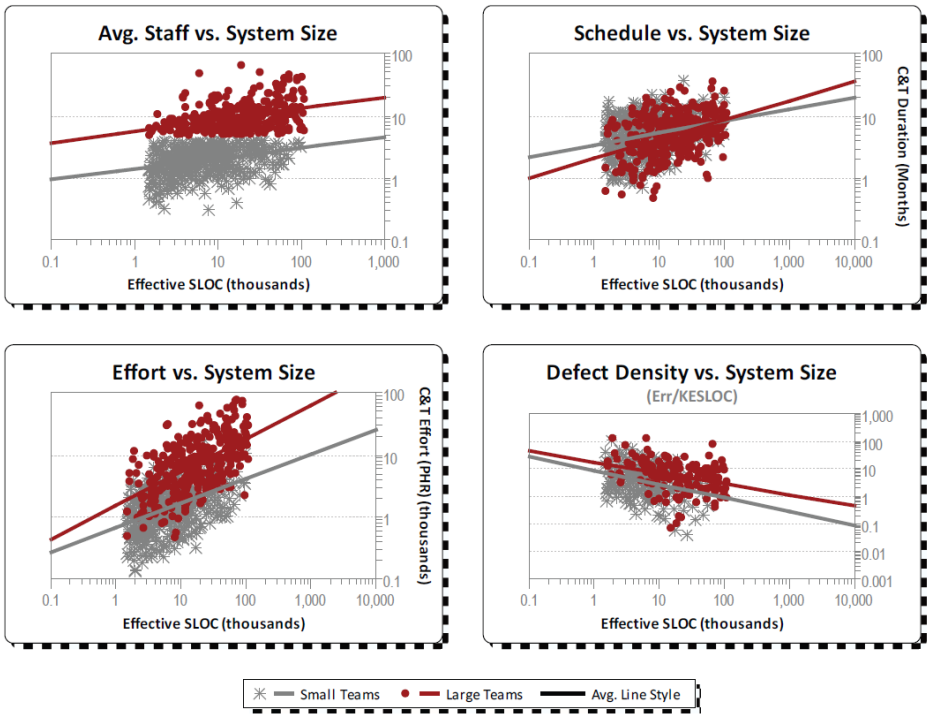


Figure 4. Small Team vs. Large Team Performance Metric Comparison across Size Regimes

What else can the data tell us about the relationship between team size and other software management metrics? A 2010 study by QSM consultant and metrics analyst Paul Below found an interesting relationship between team size and conventional productivity (defined as effective SLOC per unit of construct and test effort) (Below). To make this relationship easier to visualize, Paul stratified a large sample of recently completed IT projects into 4 size quartiles or bins, then broke each size bin into sub- quartiles based on team size. The resulting observations held true across the entire size spectrum:

- In general, productivity **increased with project size.**
- *With any given size bin* **productivity decreased as team size went up.**

To see the relationship between average productivity and project size, compare any four staffing quartiles of the same color in the graph below (Figure 5) from left to right as size (bottom or horizontal axis) increases:

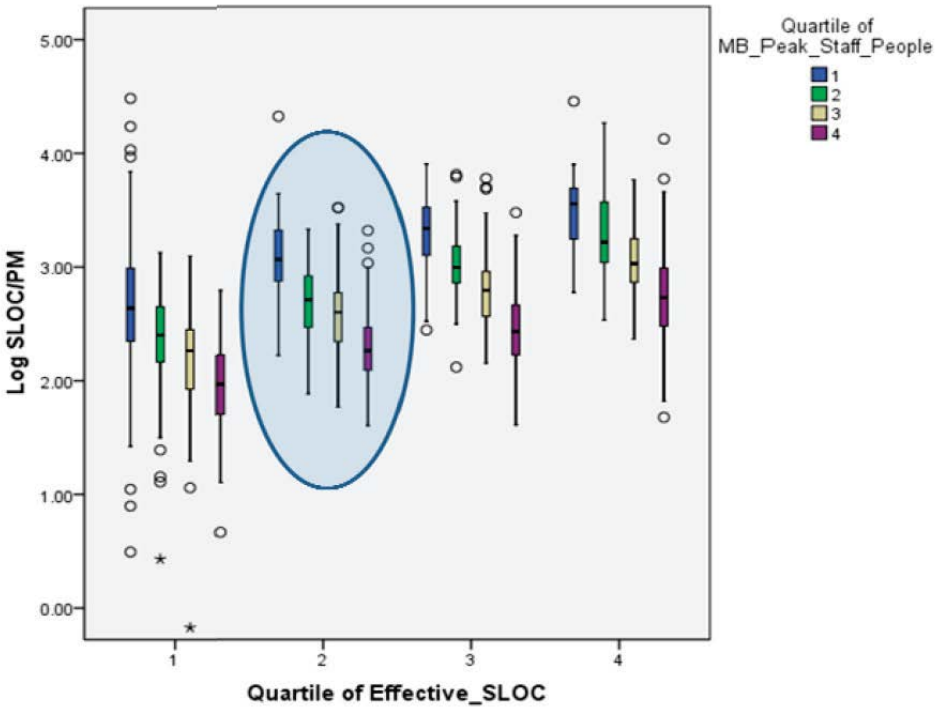


Figure 5. Staffing Quartile Comparison

As the quartiles increase in size (bottom axis), average productivity (expressed as SLOC per person month of effort on the left-hand axis) rises. The slope is reversed for projects of the same size (i.e., within a given size quartile). To see this, compare the four differently colored box plots in the second size quartile highlighted in blue. The size and staffing vs. productivity relationships hold true regardless of which Productivity measure is used: SLOC per person month, Function Points per person month, and QSM's PI (or productivity index) all increase as project size goes up, but decrease as team size relative to project size increases. The implication that the optimal team size is not independent of project scope should not surprise anyone who has ever worked on a project that was over or under staffed but the ability to demonstrate these intuitively sensible relationships between scope and team size with real data is a valuable negotiation tool.

Determining the Optimal Team Size for your Project

If the data suggest that optimal team size is related to project scope, it should be able to help us find the right staffing strategy for projects of various sizes. In a study conducted in the spring of 2011, QSM Consultant Don Beckett decided to explore the best team size for different project sizes and management goals. He divided 1920 IT projects completed since 2000 from the QSM database into four size bins: less than 4000, 4001 – 9400, 9401-25000, and over 25000 SLOC. For each of these size bins, he determined median effort (SLOC/PM) and median schedule (SLOC/Month) productivity values. Based on the results, he assigned projects to one of four categories:

Better than average for effort & schedule	Worse than average for effort & schedule
Better for effort/worse for schedule	Worse for effort/better for schedule

As Figure 6 shows, projects in the smallest size quartile (under 4,000 SLOC) using teams of **3 or fewer people** (blue bars) were the most likely to achieve balanced schedule and cost/ effort performance. Teams of **2 or fewer** (purple) achieved the best cost/effort performance and teams of **2-4** (yellow) delivered the best schedule performance. Teams that used **more than 4 people** achieved dramatically worse cost/effort and schedule performance (green bar). This process was repeated for projects in the next 3 size quartiles and the results were entered into a team size matrix (Table 2):

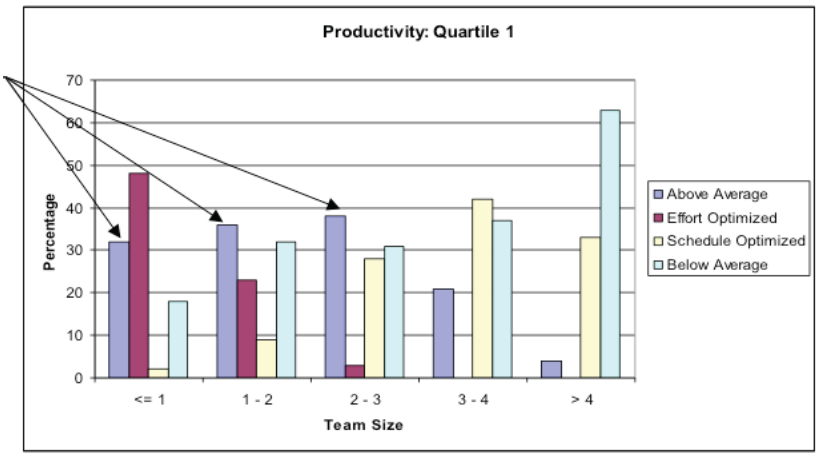


Figure 6. Productivity Quartile Comparison

Size Bin	Schedule	Cost/Effort	Balanced Performance
1 - 4000 ESLOC	2 - 4	2 or fewer	3 or fewer
4000 - 9400 ESLOC	2 - 6	3 or fewer	3 or fewer
9401 - 25000 ESLOC	2 - 4	4 or fewer	2 - 4
Over 25000 ESLOC	4 - 6	5 or fewer	2 - 6
Large Projects > 70000 ESLOC	10 - 20	10 - 20	10 - 20

Table 2. Team Size Matrix

Don's results confirm the findings from our previous two studies: the maximum optimal team size for cost/effort performance increases steadily with project size. The relationship between schedule performance and team size is less clear, with the optimal team size for balanced schedule and performance falling somewhere in the middle.

Expert Judgment vs. Empiricism

Regardless of which estimation methods are used in your organization, uncertainty and risk cannot be eliminated and should never be ignored. Recognizing and explicitly accounting for the uncertainties inherent in early software estimates is critical to ensure sound commitments and achievable project plans.

Measures of estimation accuracy that penalize estimators for being "wrong" when dealing with uncertain inputs cloud this fundamental truth and create powerful disincentives to honest measurement. Recording the difference between planned and actual outcomes is better suited to quantifying estimation *uncertainty* and feeding that information back into future estimates than it is to measuring estimation accuracy.

So how can development organizations counter optimism bias and deliver estimates that are consistent with their proven ability to deliver software? Collecting and analyzing completed project data is one way to demonstrate both an organization's present capability and the complex relationships between various management metrics. Access to historical data provides empirical support for expert judgments and allows managers to leverage tradeoffs between staffing and cost, quality, schedule and productivity instead of being sandbagged by them.

The ideal historical database will contain your own projects, collected using your organization's data definitions, standards, and methods but if you haven't started collecting your own data, industry data offers another way to leverage the experiences of other software professionals. Industry databases typically exhibit more variability than projects collected within a single organization with uniform standards and data definitions, but QSM's three-plus decades of collecting and analyzing software project metrics have shown that the fundamental relationships between software schedule, effort, size, productivity and reliability unite projects developed and measured over an astonishingly diverse set of methodologies, programming languages, complexity domains and industries.

Software estimators will always have uncertainty to contend with, but having solid data at your fingertips can help you challenge unrealistic expectations, negotiate more effectively, and avoid costly surprises. Effective measurement puts managers in the drivers' seat. It provides the information they need to negotiate achievable schedules based on their proven ability to deliver software, find the optimal team size for new projects, plan for requirements growth, track progress, and make timely mid-course corrections. The best way to avoid a repeat of history is to harness it.

Works Cited

Armel, Kate. "An In-Depth Look at the QSM Database." *QSM* Blog. September 2011. Web. <http://www.qsm.com/blog/2011/depth-look-qsm-database>.

Armour, Phillip G. 2008. "The Inaccurate Conception." *Communications of the ACM* 51.3 (2008): 13-16. Print.

Below, Paul. "Part II: Team Size and Productivity." *QSM Blog*. April 2010. Web. <http://www.qsm.com/blog/2010/part-ii-team-size-and-productivity>

Eveleens, J. Laurenz, and Chris Verhoef. "The Rise and Fall of the Chaos Report Figures." *IEEE Software* 27.1 (January-February 2010): 30-36. Print.

Reinhart, Carmen, and Kenneth S. Rogoff. *This Time Is Different: Eight Centuries of Financial Folly*. New Jersey: Princeton University Press, 2009. Print.

The Standish Group. "New Standish Group Report Shows More Projects Failing and Less Successful Projects." *The Standish Group*. 23 April 2009. Web.

3. AGILE

"If you are having everything under control, you're not moving fast enough."

– Mario Andretti, retired Italian American world champion racing driver, one of the most successful Americans in the history of the sport

"The way to get started is to quit talking and begin doing."

– Walt Disney, American business magnate, cartoonist, filmmaker, philanthropist, and voice actor

The Typical Agile Project

Taylor Putnam

While spending days at a time examining Agile projects within our database, I'm left with numerous data-driven questions. Therefore, I thought I would take this opportunity to examine what a typical Agile project looks like.

QSM's database contains over 100 Agile projects from the U.S. and abroad. The projects include a variety of application types and their top three programming languages were Java, C++, and VB.NET. Seeing this, I thought it might be interesting to examine the "typical" Agile project according to our data.

So what does the "typical" Agile project look like? For consistency purposes, I limited the sample to IT systems projects completed in the last six years. I measured the Duration, Effort, Average Staff, and MTTD at various project sizes to see how they compare.

Table 1 and Figure 1 below give demographic information about our "typical" Agile projects. Table 1 shows the values of previously mentioned metrics at various sizing units, and the scatter plot at Figure 1 shows the individual Agile projects compared against QSM's Business Agile trends.

The purpose of these figures was to give a benchmark of how Agile projects in our database typically perform. It is a good way to measure your organization against the industry. This can also be done using QSM's Business Agile benchmark trends. However, the most accurate way of estimating your projects should be from your own historical data. Since the data in this figure are merely descriptive, I'm going to leave the interpretation of the results up to the readers.

Size (SLOC)	Duration (Months)	Effort (PHR)	Average Staff	MTTD (Days)
10,000	3.9	3,420.0	7.3	2.3
25,000	5.2	6,200.0	10.4	1.8
50,000	6.3	9,740.0	13.0	1.4
100,000	7.8	15,320.0	16.4	1.2

Table 1. Performance Metrics across Size Domains

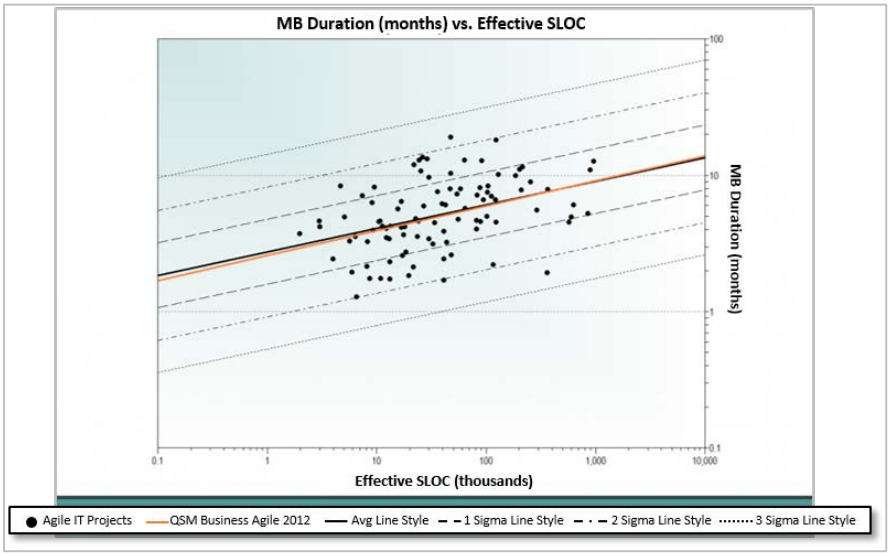


Figure 1. Duration vs. Size Scatter Plot

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.

Does Agile Scale?

Larry Putnam, Jr.

*This article originally appeared in the May-June 2013 edition (pp. 11-13) of **Modern Government** and is reprinted here with permission.*

Like Romeo and Juliet, government's flirtations with Agile software development practices have been the talk of the town. But there's one aspect of the story we tend to forget: government is big—really big. So what happens to Agile projects when they're forced to scale to the size of major government enterprise initiatives? We could speculate, of course. But instead, let's take a look at the data.

For this exercise, we analyzed 93 Agile projects, occurring between 2002 and 2012, mainly in the Government, Financial, and Health sectors. We divided the data into two datasets called Early Adopters (2002-2008) and Later Adopters (2009–2012), which gave us similar sample sizes in both groups. At the same time, we examined a sample of 93 non-Agile projects with the same sample demographics. Then, we charted how the projects fared as their sizes (measured in lines of source code) increased.

> "Large staff" projects cost significantly more, achieved negligible time savings, and produced many more defects than "small staff" projects.

Here's what we found:

Early Adopters vs. Later Adopters

- Early adopters had a significant proportion of smaller boutique software firms
- Later adopters were mostly large enterprises, with many from the financial services and banking sectors
- Early adopters exhibited shorter project durations (faster to market) than later adopters

- Later adopters tended to use more staff than early adopters and tended to look more like the non-Agile group

Agile vs. Non-Agile

- Duration of Agile projects was mostly lower on projects larger than 14,000 lines of code
- Number of defects created were lower on Agile projects than non-Agile projects

MB Average Staff vs. Effective SLOC

So, if our question is whether Agile projects can scale, the answer would seem to be yes. As the projects we analyzed grew in size, Agile methods produced: shorter project durations, fewer errors, and higher productivity. However, we also learned that as Agile projects grow, they tend to expend more effort and use more staff. And that's a problem, because we know from separate analyses (e.g., the QSM Almanac study in 2005) that high staffing levels correlate strongly with waste, regardless of development methodology.

We also know that government IT departments aren't exactly flush with cash these days, and that hiring an army of Agile developers is probably not in the budget.

"Large staff" projects keep adding more people as they add functionality, but the "small staff" projects do not.

So just for fun (yes, this is what we do for fun), we decided to take a closer look at Agile projects in relation to staff size. Projects with 7 or fewer staff members we classified as "small staff," and projects with more than 9 staff members we classified as "large staff." When we compared the two groups, here's what we found:

"Large staff" projects cost significantly more, achieved negligible time savings, and produced many more defects than "small staff" projects.

That's fairly definitive. But does it hold true even at maximum scale? 80,000 lines of code? 200,000? 5 million? The answer seems to be yes. And here's why:

As Figure 1 shows, the "large staff" projects keep adding more people as they add functionality, but the "small staff" projects do not. (As functionality increases, "small staff" projects add fewer than one full-time employee, whereas "large staff" projects grow fifty times in staff size.)

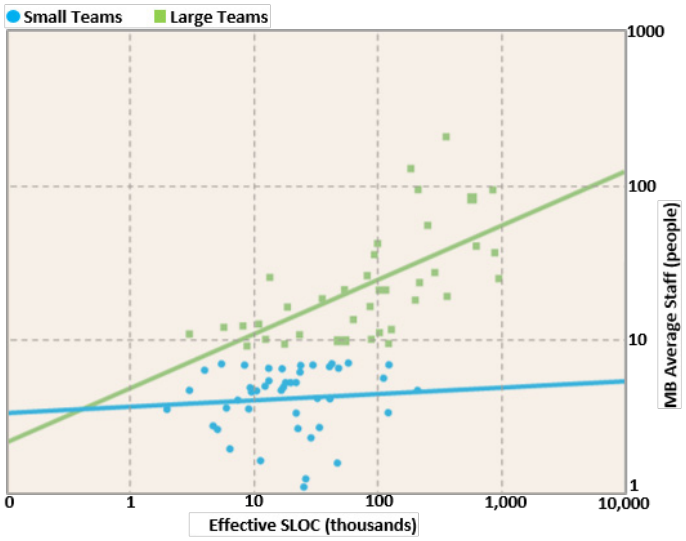


Figure 1. Number of Lines of Source Code (SLOC) vs. Average Staff Size of "Small Staff" and "Large Staff" Agile Projects

What this suggests is that large-scale Agile projects may not require big staffs after all. (Staffing up is just how most IT managers reflexively respond.) If, instead, we kept our Agile staff small, the data indicates we could complete large-scale enterprise initiatives much more efficiently, in nearly the same time frame.

This is phenomenal news for government, particularly during the dog days of sequestration— when keeping IT teams small and lean is worth its weight in U.S. bonds. Using this method, government organizations can keep costs down by influencing the way their contractors staff development projects, while at the same time producing more reliable products.

We must be careful, however, not to view Agile as government's panacea. If our data shows anything, it's that Agile projects are highly variable. We've seen how adopting Agile methods can potentially lead to modest schedule compression and fewer errors, but also a tendency toward bigger staff size and higher costs at scale.

So the question becomes: With all of these variables in play, how will government IT managers know when it's the right time—or the wrong time—to implement Agile methodologies?

The answer, we believe, is to do exactly what we've just done—run the numbers.

Agile practitioners might try to minimize upfront planning, but in large-scale government initiatives, there will always be a need for some semblance of pre-project assessment (even if it's done in an iterative, Agile fashion). In fact, one reason government IT managers have been reluctant to adopt Agile methods is for fear of losing that planning and predictability.

However, this is where software estimation and forecasting tools can bridge the gap. In a recent Harvard Business Review article ("Why Your IT Project May Be Riskier than You Think"), Bent Flyvbjerg and Alexander Budzier write:

> "[Smart managers] break big projects down into ones of limited size, complexity, and duration; recognize and make contingency plans to deal with unavoidable risks; and avail themselves of the best possible forecasting techniques..."

But to forecast effectively, you need data—and lots of it—on past and present Agile development projects. Because the truth is, Agile development in government is here for the long haul.

A Case Study in Implementing Agile

Taylor Putnam

This article originally appeared in the August 6, 2014 online edition of ***agileconnection*** *and is reprinted here with permission.*

During these times of economic austerity everyone is looking for a competitive edge. It's not surprising then, that solutions which promote decreased time to market and increased productivity would be appealing. As more organizations begin to implement Agile into their software development practices, it is logical to wonder whether Agile development methodologies truly differ from traditional Waterfall methods, and what quantifiable advantages may perhaps be realized by adopting Agile.

To tackle these questions with some objective numbers and data, QSM conducted a case study for a large technical business organization (that wishes to remain anonymous). Initially a Waterfall shop, this company attempted to adopt Agile on a small scale in 2010. Their results (at Figure 1, to the right) were less than optimal, primarily because they lacked the necessary infrastructure and organizational mind shift necessary to truly embrace the principles of Agile in their environment.

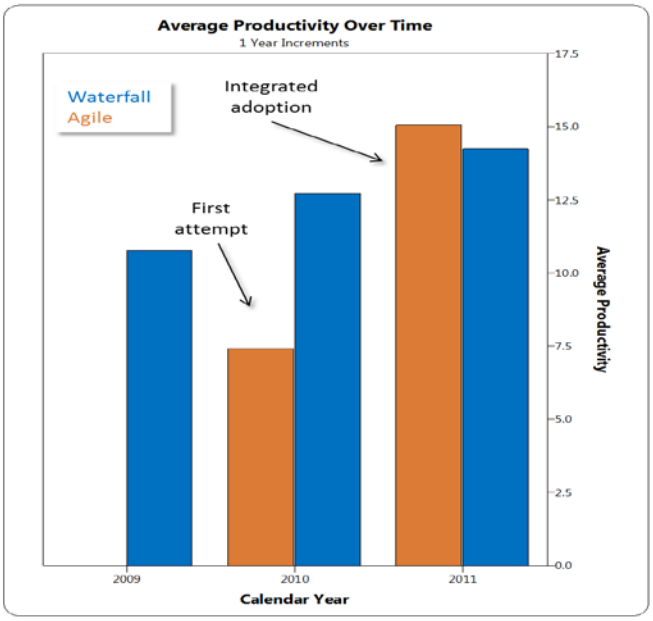


Figure 1. Average Software Development Productivity for Agile and Waterfall Methods over Time

In 2011 they made a second attempt, this time using a more integrated approach. To start, they had organizational support and "buy-in" from senior management and key stakeholders. Their process consisted of conducting an initial baseline assessment of their development systems, using an integrated set of tools and methods that would support Agile and help manage the backlog, as well as a training component.

What QSM found from this instance (and others) was that successfully adopting Agile may take upfront investment of both time and resources in order to realize optimal results. Figure 1 shows a comparison over time between the average productivity of Agile and Waterfall projects. Productivity was measured in index points, 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. The projects developed using Waterfall methods increased their average productivity ratings between 1.5 – 2 index points per year, which is fairly typical of this organization's industry. As organizations improve their software development techniques and become more efficient, they also tend to improve their productivity over time.

You can see that the projects developed using Agile methods did not have the highest productivity ratings when first adopted in 2010. However, by 2011 not only did their productivity increase dramatically, by 7.5 index points, but that it also surpassed the average productivity of the projects using Waterfall methods.

To put some additional numbers around that finding, we modeled the software development lifecycles of typical Agile and Waterfall projects of the same functional size and staffing (Figure 2) to better understand the differences between the two methodologies over time.

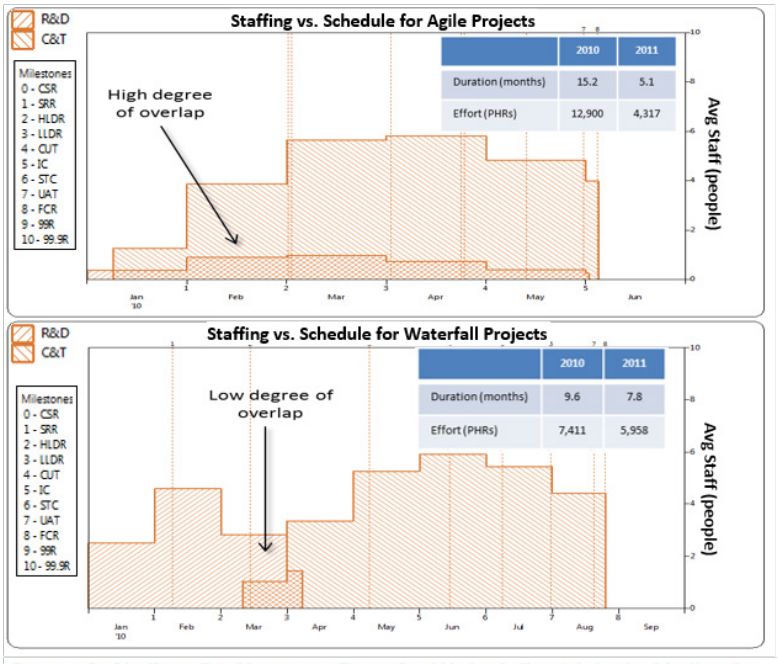


Figure 2. Staffing Profiles over Time for Waterfall and Agile Methods

One of the first and most glaring observations was that Agile methods utilize a much higher degree of overlap between High Level Design and Construction Phases than Waterfall methods, 97% versus 30% respectively (see Figure 2 above). This makes sense, as Agile methods leverage an iterative approach to release planning and delivery.

The second observation was the slope of the learning curve. When adopting Agile, development teams often have to learn a whole new methodology of defining requirements, writing code, and concurrent testing. The effects of this can be seen in the schedule duration and effort expended.

In 2010, when Agile methods were initially adopted, the projects using Waterfall methods delivered **58% faster** and **used 74% less effort**. This equates to about $550,000 in upfront costs for adopting Agile when using a normalized labor rate of $100/ person hour.

However, 2011 saw a shift after the integrated Agile adoption. This time, Agile methods achieved **34% faster deliveries** and **utilized 27% less effort** than Waterfall methods, resulting in a cost savings of $160,000 per project.

From this we can see that using an integrated management approach to adopting Agile yielded far more beneficial results than simply changing the technical development process by itself. However, realization of these benefits required an initial time investment in order to obtain organizational buy-in and allow adequate time for the necessary learning curve.

Knowing that adopting Agile can be a lengthy process, it may be valuable to examine whether adopting Agile is a worthwhile endeavor for an organization. We've all heard before that Agile methods thrive when developing smaller, less complex systems, but now it seems that people want to push the limits of what Agile can do. Can Agile scale to larger organizations and/ or more complex systems? Is there a "sweet-spot" in which Agile can realize the greatest benefits?

To answer these questions empirically, we compared the average trend lines of the Agile and Waterfall projects in a variety of areas, including duration, effort expended, staffing, and productivity (Figure 3).

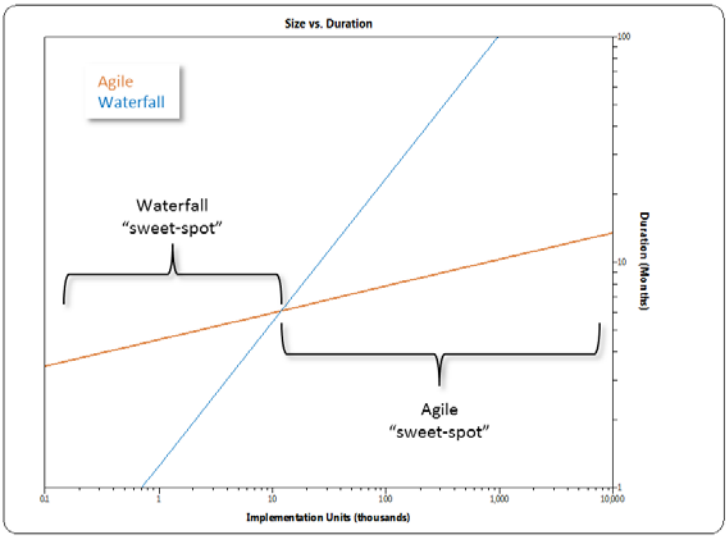


Figure 3. Average Durations for Agile and Waterfall Projects

Since the projects used different sizing methods, we normalized the project sizes into a common sizing unit called Implementation Units (IU) to allow us to make an apples-to-apples comparison.

What we found was that there was a scaling "sweet-spot" for this organization, whereby projects larger than 12,000 (IU) realized greater benefits from using Agile methods in terms of time to market, cost, and productivity. Projects that were smaller than 12,000 IU achieved better results using Waterfall.

This finding, though entirely based on one organization's environment, demonstrates that Agile has the potential to scale to larger project sizes and enterprises. While this finding may behave counter to commonly held beliefs that Agile methods should only be used for developing small software systems, we've seen similar findings with other large enterprise organizations as well, suggesting that this case study is not merely an outlier. That said, this finding is not generalizable to the software industry at large. We have seen other organizations achieve greater benefits from using Agile methods for smaller sized projects, indicating that the "sweet-spot" varies by organization.

The point we're trying to drive home is that Agile is not a silver bullet, and the decision to use Agile methods should be entirely situational. Within this organization (see Figure 3), there were situations in which using Agile resulted in the greatest benefits, and others in which using Waterfall was still the best option, suggesting that Waterfall was still a relevant development

method and should not be abandoned. When deciding between adopting Agile or sticking with more traditional methods, our suggestion would be to use the development method best suited for the project's intended environment.

This case study serves as an example for how adopting Agile can be extremely beneficial to an organization, as long as these factors are considered. If adopted impulsively without the appropriate cultural modifications and organizational support, Agile, or any new development methodology for that matter, has the potential to negatively impact organizational productivity.

Adopting a new development method is a strategic, long-term investment rather than a quick fix. Before making drastic changes such as those needed to properly implement a similar change in development practices, invest time at all levels of the organization to fully understand the task at hand and assess how long it will take to reap the benefits and achieve the expected return on investment. As in this case study, making deliberate, fully-formed decisions will ultimately lead to better outcomes.

Is It Bigger than a Breadbox? Getting Started with Release Estimation

Dr. Andy Berner

It's becoming clear to organizations adopting Agile methods that one still needs to estimate how long a project or a release of a product will take. It won't suffice for businesses to simply take guesses or accept unreasonable constraints. We must be able to derive credible estimates, based on a history of similar projects. But how can we estimate a project in advance, while still maintaining the ability to manage the backlog in an Agile manner?

In this article, we'll answer that question, compare release-level estimation to the techniques used for iteration estimation, and give some pointers on getting started with release estimation in an Agile environment.

If the Release Backlog Will Change, How Can I Estimate in Advance?

Arguably the biggest benefit of Agile methods is right in the name: Agile teams can respond quickly to changing priorities and conditions. But if our backlog—an organized list of project requirements to achieve a successful final product—changes throughout the course of our release, does it still help to estimate based on what we know in advance? Absolutely.

We can estimate in advance because it isn't necessary to know all the details of our backlog in order to develop an accurate prediction. The major driver for a release estimate is the overall size of the release, not the individual items on the backlog. Let's look at the ways backlogs change during the course of a project to see why our release estimates will remain viable.

1. The priority order of the backlog will change. (This is perhaps the most important aspect of staying Agile.)
2. The level of detail on the backlog will change. (We'll break down some of the high-level features or epics into user stories and other "developer-sized bites" that can be produced in an iteration.)

3. The specific items will change. (Some individual stories will be added and some removed.)
4. There may be a major change in the business, and some major features will be added or removed.

For change variables 1-3 above, we can expect a backlog to change frequently over the course of a project. However, these variables will not drastically change the overall size of the release, so an estimate based on overall size remains viable.

If a major change to the business occurs, our estimate may have to change, as will likely the entire justification for the project). Fortunately, while changes of this magnitude can occur quickly, without warning, they don't occur often and aren't major contributors to the changes that Agile teams typically respond to.

So barring major disruptions to the business, we can estimate our overall release while still maintaining the level of flexibility that Agile methods make possible.

Estimating Size: Differences between Release and Iteration Planning

To measure the size of our release backlog, we can borrow techniques from Agile iteration planning (Cohn). However, there are several differences between release and iteration planning, such as:

- Release backlog items are less uniform in size than stories refined for an iteration.
- Detailed comparison of size among the backlog items is not as important for release estimation.
- Consistency among multiple teams and multiple projects is more important for release estimation.

Let's look at each of these differences.

Non-Uniform Sizing

Many Agile experts suggest counting stories as the simplest way to estimate size during iteration planning. At this point, a team has refined the part of the backlog it needs to estimate into stories that are roughly comparable in size, so the size differences average out. For release planning, we'll need to group these items into "size buckets." For example, we might count totally new epics, new stories that enhance existing features, and modifications to previous stories, or—even more simply—count big, medium and small backlog items. Using just a few buckets makes it simple to count. To compare projects, we'll need to combine the bucket counts into a weighted count. We may decide that an epic is about the same size as five new stories or 15 modifications to previous stories, and then "weigh" projects according to this formula.

Detailed Comparison of Size

With techniques such as "planning poker," Agile teams compare backlog items that are targeted for a particular iteration in detail. But for release estimation, fine differences between items are not important; enough will change by the time these items are developed that the current details are immaterial. (If you're starting to ask questions like, "Is this item worth six story points instead of five, since it's slightly bigger than the last item we gave five points to?" you're getting too detailed and spending too much time.)

Consistency among Multiple Teams

When measuring iterations using story points, it's important that a team is consistent from iteration to iteration. When measuring for release planning, our goal is to compare the size of one project to another. Therefore, multiple teams must measure consistently across projects. This is sometimes called "normalizing story points." This is an example of what experimental researchers call "inter-rater reliability." Simply put, we want to ensure that different team members rate most backlog items the same way.

Getting Started

"The relationship among size, duration, and effort is complex. Larry Putnam, Sr., the founder of QSM, codified this in "The Software Equation" (Putnam). What's needed is a history of comparable projects to use for determining the effort/duration tradeoffs for a project of a particular size. You can collect this history for your own organization, or you can use industry-wide data."

While collecting data for your projects, you'll need to work on establishing consistent measurement methods (inter-rater reliability), so you can compare the size of projects. Here are some possible steps:

Build Consensus

Build consensus among your teams on the technique and scale you will use to measure size. The specifics of the technique are much less important than applying it uniformly, so keep the debate about "favorite technique" to a minimum. You may decide on a standard set of "buckets" to count ("feature, new story, enhancement, throw-in"; "huge, big, medium, small"; or some other set that fits your style). You may decide to measure rather than count, comparing items to each other. If so, be sure to set a consistent scale (e.g., story points using Fibonacci numbers, powers of two, or another unit or scale).

Practice Estimating

Ask multiple teams to estimate each other's backlogs using the chosen technique, and see if the ratings of the items are comparable. Discuss major differences to gain consensus on the meaning of the buckets or scale. Certain differences may arise because of a different

understanding of the backlog items (for example, one rater might be more familiar with the project than another); that's a consequence of the practice exercise, so you can ignore these differences. But through the discussion, you can build consensus on the meaning of the buckets or measurement scale, and improve comparisons for different projects.

Once you can get consistent size estimates, you're on your way to use these to estimate project durations and costs. Then, you can compare projects directly and make informed decisions for the best chance for success.

Works Cited

Cohn, Mike. *Agile Estimation and Planning*. Upper Saddle River: Prentice-Hall, Inc., 2005. Print.

Putnam, Lawrence H.., and W. Myers. *Measures for Excellence: Reliable Software on Time, within Budget*. Upper Saddle River: Prentice-Hall, Inc., 1992. Print.

Ready, Set, Go…and Ready Again: Planning to Groom the Backlog

Dr. Andy Berner

In an Agile project, the backlog--the prioritized set of requirements--is the main input to iteration planning. Many Agile teams are as careful in specifying the "definition of 'ready'" as they are in specifying the "definition of 'done'." The product owner must ensure that priorities are thought through, stories are at the Goldilocks level of granularity ("not too big, not too small") and stakeholders are prepared to discuss details.

Getting the backlog ready and the related concept of "grooming the backlog" doesn't come for free. You need to plan and budget for this work. Here are five aspects to consider.

Keep Two Views of the Backlog

1. ***User Story Mapping***--The Business View: This view lets the business stakeholders keep their eyes on the prize. In this view, they see user stories of all different sizes--some very large that are broken down into multiple levels of detail, some small and specific. Also, collections of stories are grouped into business scenarios. These dependencies among stories clarify how individual capabilities provide value to the users and stakeholders.
2. ***Prioritized List of Stories--The Classic View:*** The top items on the backlog get developed in the next sprint. The heart of iteration planning involves determining how far down the backlog each sprint will reach. To be ready for the upcoming iteration, the top items must be in "development-sized chunks." If an item is too large to be developed in a single, fixed-length iteration, it must be broken down into smaller partial stories.

These two views work together: While the prioritized list gives developers the roadmap for a particular sprint, the user-story mapping puts those development-sized chunks in context.

When Is the Work Done?

Getting the backlog ready overlaps almost all the coding and testing. It begins shortly before coding starts to get "just enough" of the backlog ready. In many projects, those initial

highest-priority items can be easy to find, even though prioritization down the line will be more contentious. In other projects, even those initial priorities require significant discussion; adjust the lead time before the first coding iteration accordingly.

Expect grooming the backlog to continue almost to the end of the release. Details about stories emerge over time. As stakeholders review already developed stories, priorities change and the backlog needs more grooming.

There are two milestones to aim for:

1. ***Minimally Marketable Features Defined***: As you break down top-level features in the user story map, you eventually identify what portions of stories are needed for the release to be useful. These may still be too large to develop in a single iteration, so they may need to be broken down further. Other portions of the original features may also still be included, but these would have lower priority. You need to reach this milestone quickly, since it guides the prioritization of early iterations.
2. ***Release Backlog Finalized:*** Toward the end of the release, you've made the decisions of what stories will stay in this release and what can be deferred. The user story will remain stable after this point.

The Work Contour

There are four tasks to groom the backlog:

1. Adding and removing stories from the backlog
2. Refining the user story map: breaking down high-level features into more concrete stories and grouping them into business scenarios
3. Prioritizing "development-sized chunks" for each iteration
4. Specifying the details of the stories

These don't all occur at a steady rate over the course of the project, and the people involved and their degree of involvement change. Early in the project, much of the work will be geared toward refining the user story map. This requires a high level of involvement from business stakeholders; getting consensus among the stakeholders is both difficult and critical. Their involvement will stay high at least until the minimally marketable features are defined. Stakeholders will still be needed to provide details, but once the main parts of the story map are stable this can be delegated to a working group of subject matter experts.

The development team will be involved at a fairly steady rate, primarily in discussions of story details. Overall, the work is front-loaded and intense until the minimally marketable features are defined, less intense as the User Story Map evolves, with only minor work remaining after the release backlog is finalized.

"Ready Again": Plan for revisions to previous work

Stories developed in earlier iterations may need to be revised based on new requirements. For example, in an online commerce site, "checkout" may have been defined and

implemented based on each registered user having a single address. Later in the project--when the story was added to allow a user to store multiple addresses--the checkout scenario needs to be revised. The business stakeholders will need to provide additional detail regarding business rules around billing addresses and shipping addresses.

Although many of us have been conditioned to avoid rework at all costs, Kent Beck years ago described some different economics in *eXtreme Programming eXplained*. By keeping things simple at first and adding complexity only later when you need it, you cover the cost of rework with two sources of savings:

1. You benefit from the simpler design making it faster to write code that will still work with the later, more complex design
2. You don't spend the time designing for complexity you may never need (The "YAGNI" Principle: "You Ain't Gonna Need It")

Beck was primarily describing coding techniques such as refactoring and automated unit testing, but the same principles apply to getting the backlog ready for development: don't spend time debating details abstractly until you know you need them. The time saved exploring in advance what *might* be needed makes up for the time spent later reworking only what *is* needed.

Planning and accounting for the work

When you are estimating the cost and resource requirements for a project or product release, include the work to groom the backlog:

1. Effort can be estimated as a proportion of development effort. Use history of past projects (either from your organization or industry-wide) as a guide. The proportion may be different for different types of projects, so be sure to compare projects that are similar in nature, with similar development methods.
2. Schedule time with business stakeholders who will develop the user story map and participate in discussions of story detail. They will be needed throughout the project, but not at a uniform rate. More work will be needed upfront, with more detail work at the later stages--likely from people with more specialized knowledge. Plan for rework as stories are revisited throughout the project.
3. Account for work that coders will do getting the backlog items ready. Expect discussions between the business stakeholders (perhaps represented by the product owner) and the coders.

Planning for this in advance will keep your backlog groomed and your Agile project humming along smoothly.

Works Cited

Agile Alliance. "Backlog." *Agile Alliance Guide*. 2014. Web.

Agile Alliance. "Definition of Ready." *Agile Alliance Guide*. 2014. Web.

Beck, Kent. *eXtreme Programming eXplained*. Boston: Addison-Wesley Publishing, 2000. Print.

Patton, Jeff. *User Story Mapping*. 2014. Web.

Rubin, Ken. *The Importance of the Product Backlog on a Scrum Development Project*. 2014. Web.

Shalloway, Alan. *Minimal Marketable Features: The Why of Enterprise Agility*. 2014. Web.

Constant Velocity Is a Myth

Dr. Andy Berner

Is your Agile team's velocity constant from sprint to sprint? No? That's not a surprise. Many teams assume that their velocity will be constant. In this article, we'll see why that's not the right expectation and how that affects how you use this metric.

What Is Velocity?

Velocity measures the amount of work accomplished in your project over time. In Agile terms, this is how much of your release's backlog is completed in each iteration. Velocity is measured in whatever unit you use to estimate stories; for example, story points per iteration or the count of stories per iteration.

Teams often assume that their velocity will be nearly constant, although most teams know that the velocity in early iterations may be lower than later ones. Since all the sprints are the same duration, this amounts to assuming that the team will complete the same amount of the backlog in each iteration. However, that's not the way real projects work! To see the significance of this, we need to look at how velocity is used.

Uses of Velocity

Teams use velocity in three ways:

- Iteration planning: How many stories should we plan for the next iteration? The assumption of constant velocity says we could plan the next iteration to match the previous one.
- Release planning: How many iterations should we plan for a new product release? Divide the total planned backlog by the velocity to find the number of iterations.
- Project forecasting: How many more iterations until we release? Divide the remaining backlog by the velocity and adjust the plan.

These simple techniques to accomplish difficult planning tasks make it really tempting to assume velocity will be constant. But just because that assumption is tempting doesn't make it correct.

Velocity Is Not Constant

If velocity were constant throughout a project, the graph of the cumulative work completed over time would be a straight line (Figure 1):

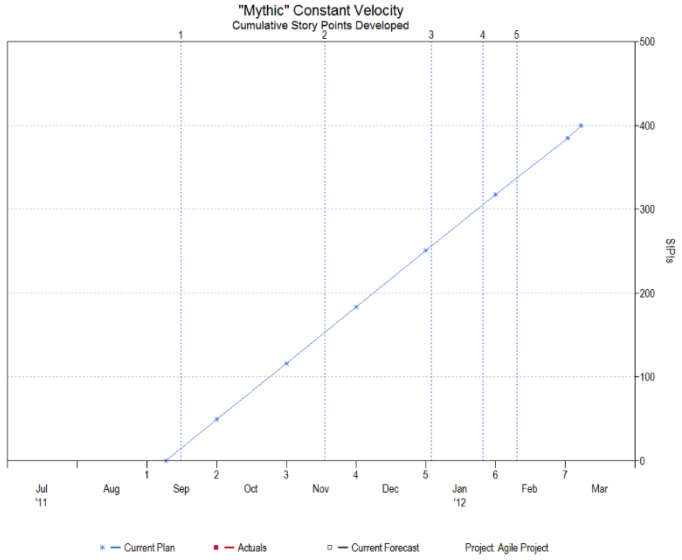


Figure 1. Mythic Constant Velocity

Years of research have shown, though, that key metrics--including the cumulative amount of work accomplished over time--follow an S-shaped curve, known as the cumulative Putnam-Norden-Rayleigh curve ("Putnam-Norden-Rayleigh Curve") (Putnam; Cheslon; Cohn). The reasons why projects take this shape vary. Different methodologies, including Agile, have different characteristics that cause this production curve to change in detail. But while the reasons vary, the basic shape remains the same (Figure 2).

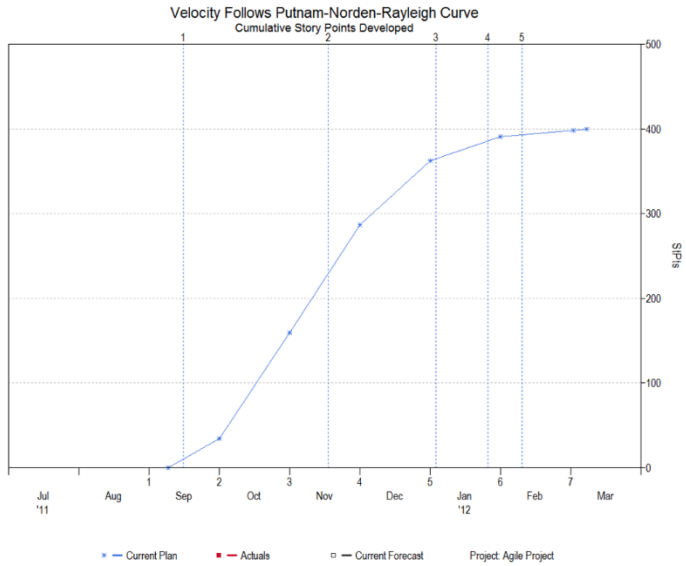


Figure 2. Velocity and Putnam-Norden-Raleigh Curve

You can see this curve starts slowly, but then rises more steeply and flattens out at the end. Notice that in the middle of the release, the curve is fairly straight. This means that for a period

of time, velocity will be close to constant but it will not stay constant. It will change later in the project when the curve flattens out again.

How can you use velocity even though it's not constant? Let's look at how the variability of velocity affects the three uses we pointed out earlier.

Iteration planning: Velocity doesn't change abruptly. If you choose to estimate how many story points you plan to develop in an iteration based on what you developed in the last few iterations, you can still do that with some confidence. If you can determine where you are on the S-curve, you can polish that prediction. Are you near the bottom of the S, so the velocity will increase? Are you in the middle, so you expect the velocity to be about the same as the previous iteration? Or are you nearing the top of the S, so you should plan for a lower velocity?

Note that many Agile coaches are wary of using velocity as a strong predictor of what you should estimate for the next iteration. Instead, you should look at the tasks required to develop the chosen set of stories to decide what you can commit to for the next iteration.[4] This is the case whether you assume constant velocity or not.

Release planning: Whether you assume velocity is constant or not, using velocity to plan a new release is difficult. Velocity is a measure of what one particular team is doing on one particular release. There are several reasons velocity is often team and release dependent:

- Team sizes vary among projects, but velocity is not proportional. It's not a surprise that a larger team accomplishes more than a smaller team in the same period of time, but it may surprise you that that how much more is quite difficult to compute. Doubling the team size does not double the velocity. So you cannot easily compare velocities of teams of different sizes.
- The team composition (both the experience of the team members as well as the team dynamics) affects the velocity.
- The nature of the product affects the velocity. For example, you would not expect the same velocity for a team building an informational website and a team building a life-sustaining medical device.
- Velocity depends on how the team measures the stories in Story Points, and also the chosen sprint length. This may differ from team to team or even from project to project.

To use velocity for release planning, you need to take a number of steps:

- Gather historical information from as many previous projects as you can. Instead of using the velocity from a single project, use trends computed from multiple projects.
- Work with multiple teams to normalize the way teams measure the backlog in Story Points ("Normalizing Story Point Estimation;" Berner).

- Use estimation tools or collect historical data to compute the "time/effort" tradeoff, and adjust expected velocity based on team size. Remember, it is definitely not linear--doubling the team size only increases velocity by a moderate amount.
- Use an estimation tool or otherwise adjust the release plan to account for the S-shape of the project.

Project Forecasting: Consider multiple metrics, not just velocity. For example, also consider the rate at which stories are defined using your "definition of ready." In addition, consider the burndown rate, which measures how the backlog is changing. If you swap out equal size stories, the change may not have much effect on the delivery date. But if you consistently add in more or larger stories than you take out, or if you decide not to deliver stories already developed, your original estimate may need to be adjusted.

As with iteration planning, when you're reforecasting and you use the average velocity you've achieved so far in the project, consider where you are on the S-curve.

More Research Is Needed

Projects follow the Putnam-Norden-Rayleigh cumulative S-curve whether they use Agile methods or not; but the methods you use do affect the shape. At QSM, we have been collecting data from thousands of projects for many years. We are starting to get enough data from projects using Agile methods to start drawing some conclusions, but more research is still needed in several areas. Here are some of the questions that require more research to answer:

1. Does the "middle of the S" cover more of the project, so velocity is closer to constant than with other methods?
2. How does the Agile principle of "embrace change" affect velocity? Is there more or less variance in the middle of the S because of scope changes?
3. How do refactoring and emergent design affect the shape? As project size increases and thus the eventual design must accommodate many diverse stories, does the amount of refactoring needed show up in the S-curve as a longer "tail"? Does size affect the overall productivity gains from Agile methods because of increased refactoring?
4. How does test-driven development affect the shape? Does continuous testing throughout the project shorten the tail, either because testing is spread, or because defects are removed early?

In Summary

It's not realistic to expect velocity on an Agile project to be constant. Depending on how you intend to use velocity, you must adjust your estimating methods by keeping in mind the natural cumulative S-curve of development metrics. Agile methods almost certainly have an effect on the detailed shape of that curve, but more research is needed to know precisely

how. In the meantime, use historical data plus your knowledge of your individual teams and projects to get the best estimates you can.

Works Cited

Berner, Andy. "Is it Bigger than a Breadbox? Getting Started with Release Estimation." *Project Management.com*. 18 July 2013. Web.

Chelson, Heather F., Richard L. Coleman, Jessica R. Summerville, and Steven L. Van Drew. "Rayleigh Curves—A Tutorial." SCEA Conference. Manhattan Beach, CA. June 2004. Conference Paper.

Cohn, Mike. "Why I Don't Use Story Points for Sprint Planning." *Succeeding with Agile: Mike Cohn's Blog*. 7 November 2007. Web.

"Normalizing Story Point Estimation." *Scaled Agile Framework*. 2014. Web.

Putnam, Lawrence H., and Myers, Ware. *Controlling Software Development*. New York: IEEE Computer Society Press, 1996. Print.

"Putnam-Norden-Rayleigh Curve." *Wikipedia, The Free Encyclopedia*. Wikimedia Foundation, Inc. 2014. Web.

Big Agile: Enterprise Savior or Oxymoron?

Larry Putnam, Jr.

Something strange is happening in enterprise software development—eager CIOs are launching "Agile" projects with teams of thirty people or more devoted to a single product release. And why not? We know Agile works well for small teams and small projects, and monster enterprise projects (like rolling out a new SAP financial solution to replace all your legacy systems worldwide) often require greater capabilities than a small team can provide. So why not scale up Agile teams to maintain the cost and efficiency benefits of the Agile process while accessing the necessary manpower to pursue complex global projects?

If it works, we'll be enterprise heroes—ready to have our portraits enshrined in the corporate IT hall of fame. But what if Agile only works when teams and projects stay relatively small? That's the question most CIOs want answered before investing scarce time, energy, or resources chasing the "big Agile" paradigm.

To get that answer, we turned to the only source we truly trust—cold, hard data from the QSM software projects database.

The Ground Rules

To find out whether Agile delivers the same benefits when applied to larger endeavors, we analyzed roughly three hundred recently completed IT projects, half of which reported using Agile methods and half of which did not.

Agile projects in the QSM database are those that were reported as such by the teams that developed and delivered them. The results of the study may be influenced by variability in how Agile methods were applied, but that seems only fitting for a methodology that espouses the freedom to "adapt as you adopt." We are actively collecting more Agile projects. However, at this point, the sample size is still relatively small—approximately one hundred fifty Agile projects. We'll be interested to see how our initial observations hold up over time.

To measure the relative size of software projects, we looked at the number of source lines of code delivered when the system was put into production. Though Agile projects frequently

estimate using story points, ideal hours, or counts of stories instead of code, we can empirically determine the average code volume per story point from completed project data by dividing the delivered code by the number of story points. This works well for our purposes because we need a "ruler" for measuring the volume of work to be performed that's independent of how the project is staffed.

In addition, we looked at time, effort, activity overlap, and productivity data for two high-level phases of each software project:

- The story writing, or requirements and design phase, encompasses requirements setting and high-level design and architecture. This includes the work in Agile projects that is sometimes referred to as "getting to ready" and grooming the backlog.
- The code, test, and deliver phase includes low-level design, coding, unit testing, integration, and system testing that leads to deployment.

In the traditional waterfall method of development, a large proportion of the requirements and design work precedes coding, testing, and release packaging. Overlap between the two phases is minimal. Conversely, Agile's iterative design cycles and just-in-time story detailing typically result in a great deal more overlap.

Hence, we were curious to study the proportion of time spent on requirements and design relative to the time spent on coding, testing, and packaging for delivery. In other words, how does time and effort spent on design work and story writing affect productivity?

Is Big Agile Effective or Not?

Enough process talk. Do Agile methods translate well to large-scale software projects? You didn't really expect a simple yes or no answer, did you?

Any measure of effectiveness must first define what "success" looks like. Software effectiveness measures typically include one or more of these high-level management goals:

- Cost efficiency
- Schedule efficiency
- High productivity

An organization's definition of project success should align with its top priorities. Hence, organizations with significant cost and resource constraints should focus on completing projects inexpensively. Others may prioritize time to market or optimal productivity (finding the right balance between resources, schedule, and quality).

Let's examine each of these goals individually.

Success Case #1: Cost Efficiency

To compare the cost efficiencies of numerous projects, we need to minimize the effects of varying labor rates. To that end, we viewed cost through the lens of effort hours expended (i.e., cost = effort * labor rate) (Figure 1). Using this method, we can see that Agile projects with fewer than thirty thousand source lines of code, or SLOC, are less costly than similarly sized non-Agile projects. The trend seems to reverse itself in projects above the thirty thousand SLOC threshold, but in a subtle way.

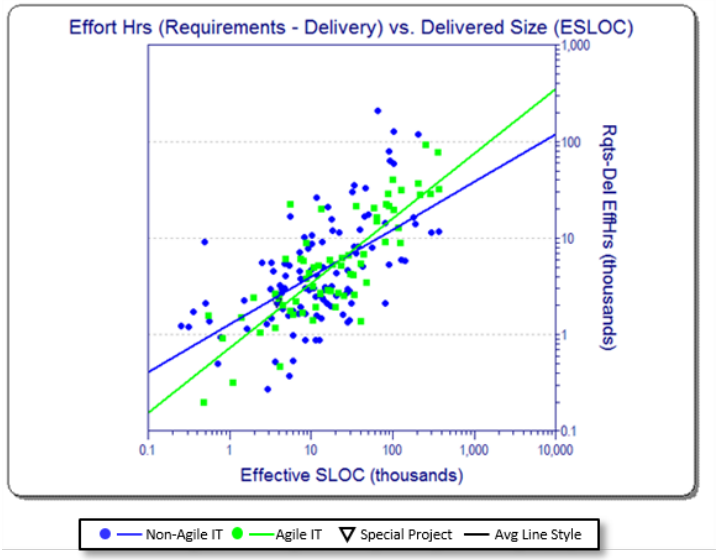


Figure 1. Cost Efficiency

The larger Agile projects are fairly closely clustered; that is, for a particular size, there is not much variation in effort and cost. The larger non-Agile projects, however, are either much less costly than their Agile counterparts or quite a bit more costly. On average, large Agile projects are a bit more costly, and this disparity seems to increase with size.

Bottom line: The Agile cost advantage phases out at around thirty thousand SLOC. Above that threshold, Agile projects may actually be more costly than traditional waterfall projects. If cost efficiency is your primary concern, both Agile and non-Agile projects should keep release sizes small.

Success Case #2: Schedule Efficiency

Our schedule analysis shows that time to market is consistently shorter for Agile IT projects, but that Agile schedule edge diminishes as the volume of delivered features increases (Figure 2).

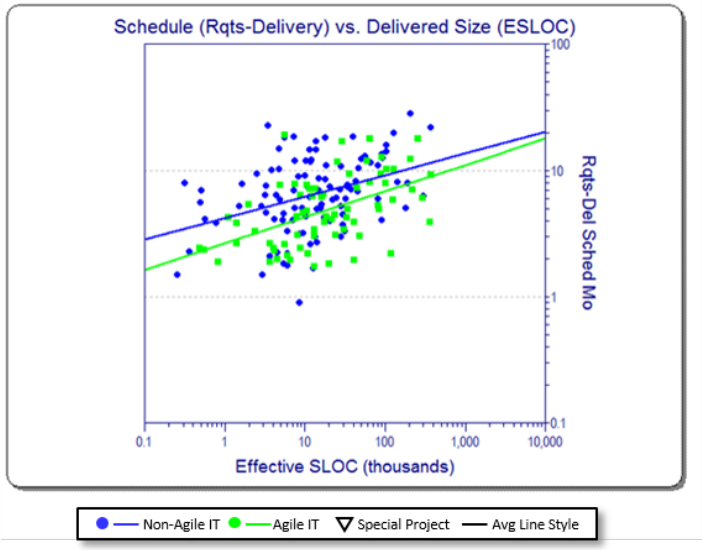


Figure 2. Schedule Efficiency

Bottom line: If schedule is your primary concern, large Agile projects are consistently more time-efficient than similarly sized non-Agile projects, but the Agile schedule advantage diminishes as project size grows.

Success Case #3: Balanced Productivity

To evaluate balanced schedule and cost productivity, we looked at a metric called the productivity index, or PI, which—unlike traditional productivity measures that examine resource or schedule efficiency, but not both—takes *both* time and cost into account (Figure 3).

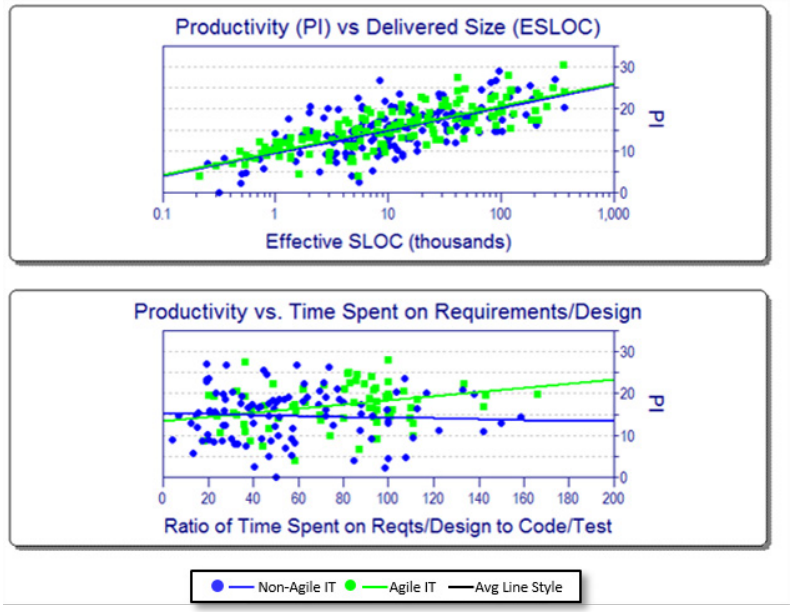


Figure 3. Balanced Productivity

According to the data, larger Agile projects enjoy a modest productivity edge over non-Agile projects, but again, the effect is quite subtle.

More interesting, perhaps, is what we see when plotting productivity against the proportion of time spent on design and story writing. Agile projects, it seems, become noticeably more productive as they spend a larger proportion of their time on requirements and design versus coding, testing, and packaging for delivery. This is consonant with findings in the Agile community in general that taking extra time "getting to ready" and ensuring that user stories are well thought out and communicated is critical to the success of Agile methods.

Bottom line: If process productivity is your primary consideration, large Agile projects benefit from increased time spent on requirements setting and high-level design.

Wait, That's a Paradox, Isn't It?

Not necessarily. It's true that one of the hallmarks of the Agile method is spending less time up front on requirements and design. But "less" is a relative term. Although the time spent on requirements is more spread out throughout the project with Agile methods, our data show that spending more time pays off in higher overall productivity.

Smaller projects are a more natural fit for "pure" Agile—after all, most small projects were using small teams and lighter processes even before Agile revolutionized software development. What we see from our data, however, is that as larger teams apply Agile methods to larger projects, they are wisely adapting those "pure" Agile precepts to the needs of larger, more complex systems, resulting in a tempered version of Agile.

What's most remarkable about this productivity chart, however, is that for non-Agile projects, spending additional time in the design phase does not appear to boost productivity at all.

Agile methods, it seems, help teams apply that additional time and effort more effectively—working smarter, not harder.

Hence, big companies can benefit from constructing their software in an "Agile-like," iterative fashion with frequent reprioritization of features and functionality, as long as their requirements and design work is sufficiently robust. In fact, this is precisely what a growing number of industry experts such as Dean Leffingwell, Steve McConnell, and Scott Ambler are recommending.

Perhaps "big Agile" is neither a savior nor an oxymoron; it's simply a compromise using the best of new methods and tried-and-true techniques. After all, one of the Agile tenets is allowing human judgment to trump rigid process guidelines.

4. PLANNING FOR SUCCESS

"If your only tool is a hammer, every problem looks like a nail."

– Abraham Maslow, American psychologist who was best known for creating Maslow's hierarchy of needs

"Whosoever desires constant success must change his conduct with the times."

– Niccolò Machiavelli, Italian historian, statesman, philosopher, political theorist, and writer

Using Metrics to Influence Enhanced Future Performance

Taylor Putnam

Assessing past performance is often an activity that is given low priority in software development. With time being valuable, it's natural to want to move onto the next project as soon as the pervious one is over. But what if understanding historical projects not only allowed for more accurate estimates, but also *improved* the performance of future projects?

The good news is that taking a little time upfront to conduct a benchmark assessment of your historical project portfolio can identify improvement areas for your organization, and this knowledge can help shape your process improvement plans. The easiest way to explain this concept is with an example.

The Benchmark.

After collecting data on completed projects from one company, a team of analysts plotted said data against the industry average trend lines (see Figure 1).

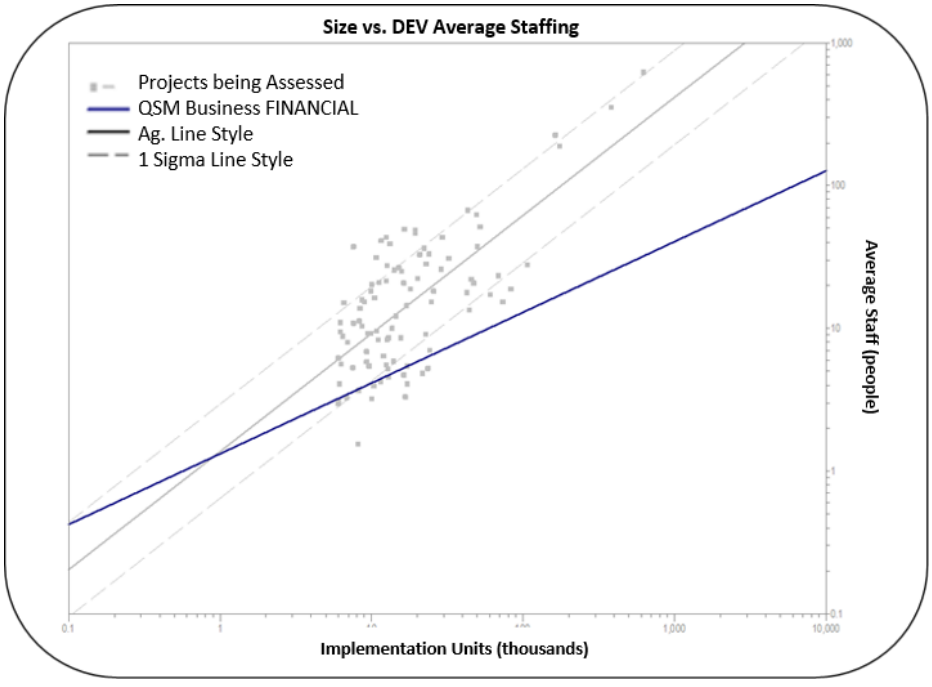


Figure 1. Marking the Overstaffed Projects

While analyzing the staffing data, it became obvious that this organization used *way* more people than was necessary to bring the project to production. Nearly all the projects in the dataset had higher staffing values than the industry average, and a majority of those were more than one standard deviation above the mean.

What impact does that have on schedule and cost? Popular belief would dictate that adding more staff to a project would decrease its overall schedule. After all, many hands make light work, right? However, an examination of the data would indicate otherwise.

In Figure 2, we tagged all the projects that were more than one standard deviation above the company average staffing, and then examined where those projects fell relative to the mean for schedule and effort.

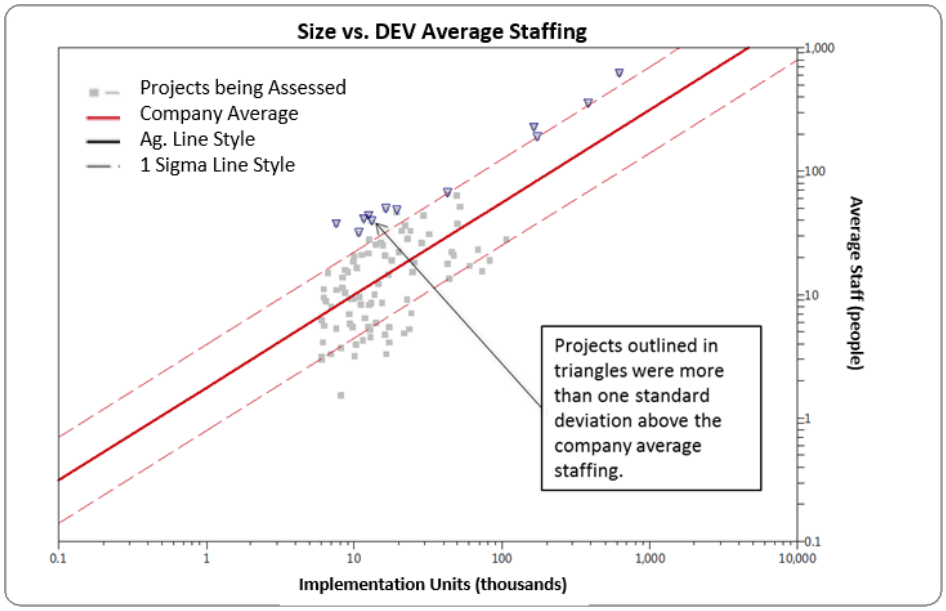


Figure 2. Marking the Overstaffed Projects

What we saw was interesting (see Figure 3). About 70% of the projects were able to achieve a minimal schedule compression of about 1-3 months on average while the other 30% did not realize any schedule compression at all. Upon examining the effort and cost, each overstaffed project expended an average of 15,000 more person hours than the company average. At a normalized labor rate of $100/hour, that results in $1.5 million in additional costs per project.

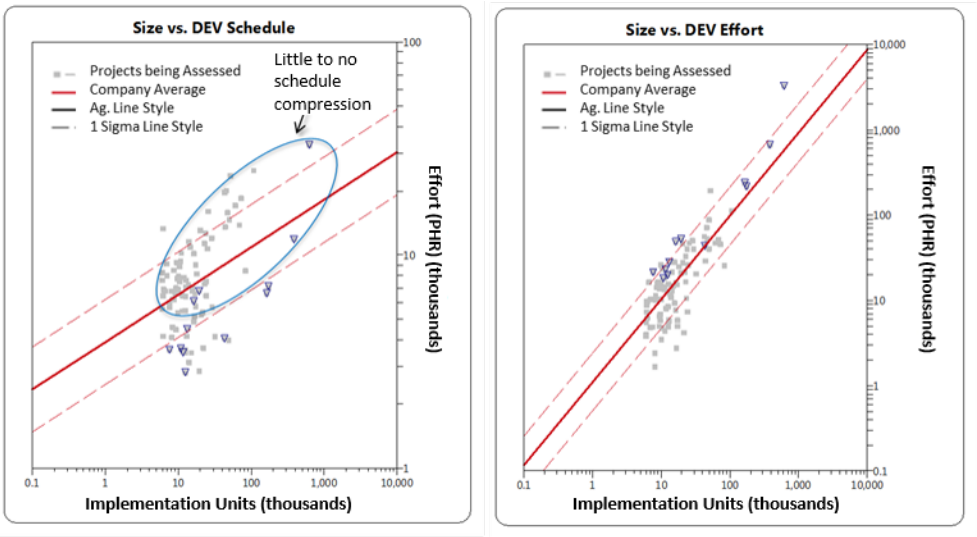


Figure 3. Schedule and Effort for Overstaffed Projects

Is three months of schedule compression worth $1.5 million? If there is a compelling business reason for compressing the schedule, then perhaps there is some value to staffing up. However, if the schedule was decided by arbitrary means and there is actually some room for schedule extension, than it's unlikely that the added cost will be worthwhile.

Knowing this, we wondered what the schedule penalty would be if projects reduced their staffing size. This time we tagged all the projects that were more than one standard deviation below the mean – indicating a lower than average staffing – and examined where they fell relative to the mean in terms of schedule and effort (see Figure 4).

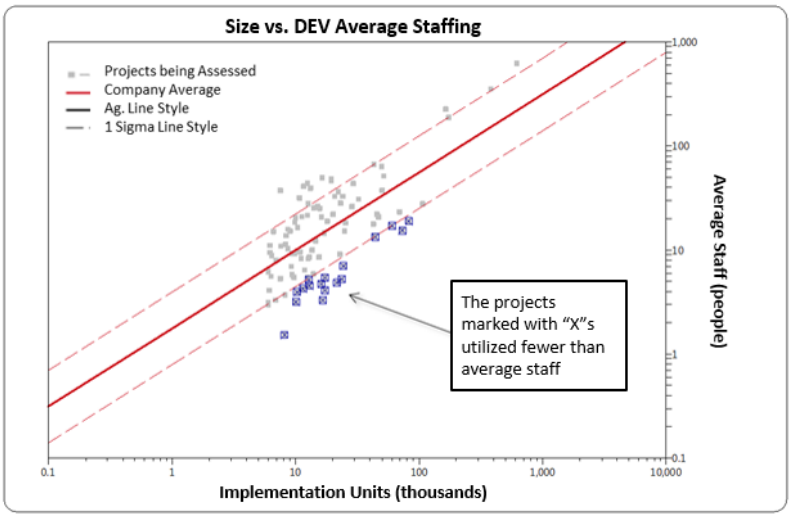


Figure 4. Building Projects with Fewer than Average Staff

This time, we saw slightly different results (see Figure 5). About 35% of the projects had average or better than average schedules, and the other 65% only increased their schedules by about 2 months. Moreover, all the projects expended less effort, averaging about 7,000 fewer person hours per project or a cost savings of $700,000 each.

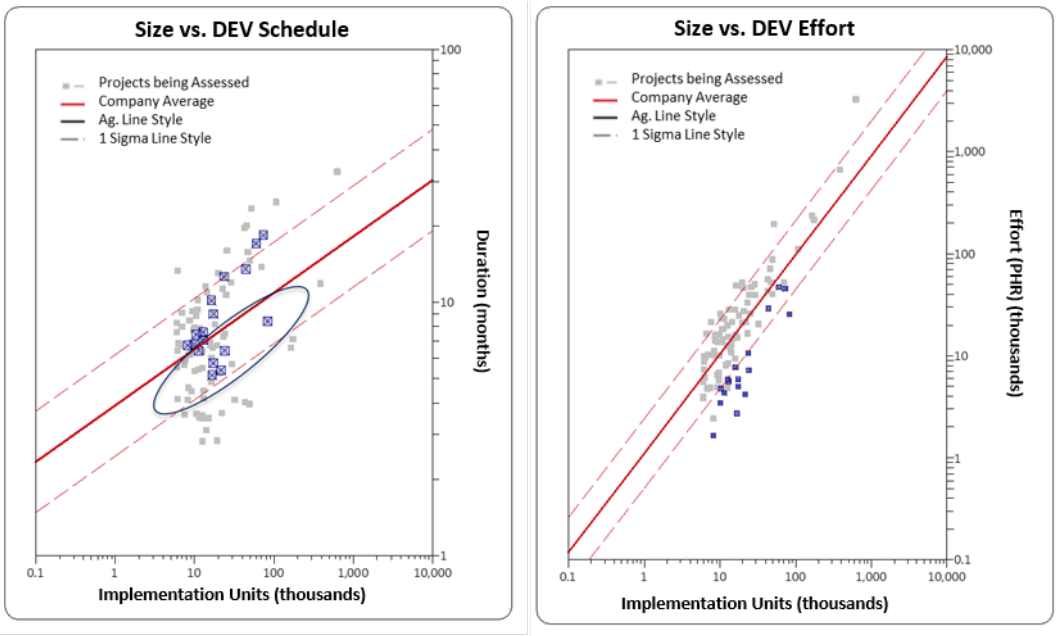


Figure 5. Schedule and Effort for Projects Using Fewer Staff

Findings from this study and previous studies (QSM, Inc.), confirm that overstaffing a project does little reduce the schedule duration, but exponentially increases the cost and also decreases the project's quality. Therefore, our recommendation for this company would be to reduce the number of Full Time Equivalents (FTE) that staff the projects.

Improving Performance by Manipulating Trends.

While best practices would suggest using the industry trend lines to estimate for improved future performance, which was definitely not an option for this company. Since the industry trend line was more than a standard deviation below current practices, making such a dramatic reduction in staffing would be nearly impossible to implement in real life. Such drastic changes could result in chaos for the organization and likely cause detrimental outcomes. In such situations, making a 10% incremental reduction in staffing would produce far more favorable results.

To do this, we had to create and plot a series of trend lines on a chart. We first created the Company Average trend line, which was the average of the dataset before outliers were eliminated. This trend line marks the company baseline from which all improvements can be measured (see Figure 6).

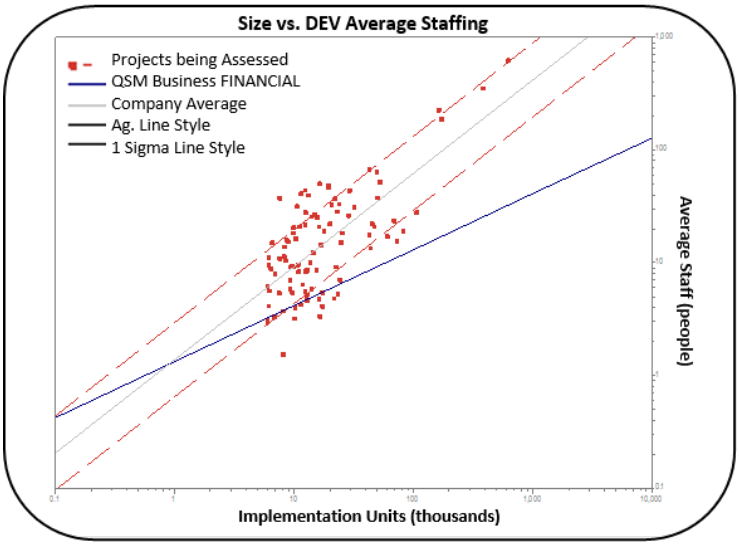


Figure 6. Newly Created Average Trend line

You'll notice that there are now three trend lines on the chart: the industry average (blue), the dataset average (red), and the Company Average (grey). For now, the Company Average trend line is superimposed on the dataset average trend line, temporarily hiding it from view.

Now we're ready to start manipulating the trend line.

The goal is to make it such that the new trend line will be somewhere between the Company Average trend line and the Industry Average Trend line, reflecting a 10% reduction in staffing. Using similar techniques to the analysis done earlier, we tagged the projects that were negatively influencing the trend line away from the industry average – those that were one standard deviation or more above the mean (see Figure 7).

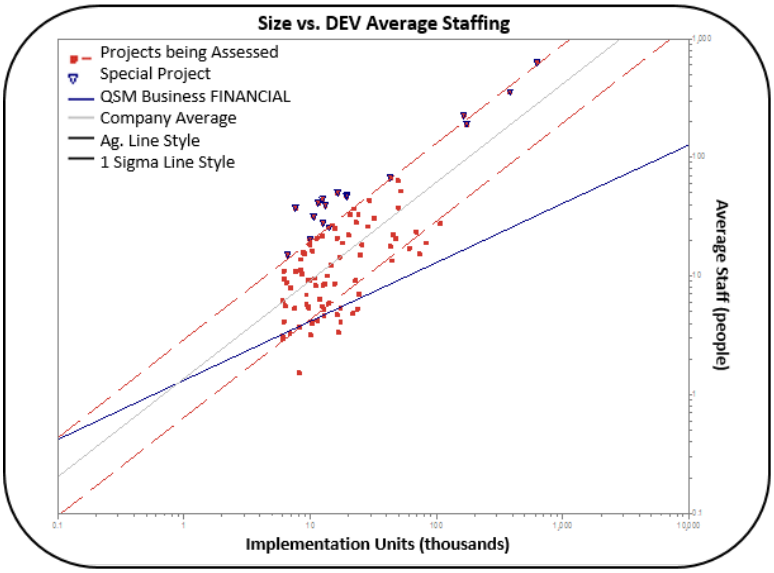


Figure 7. Overstaffed Projects Tagged

Once the overstaffed projects have been identified, they should be excluded from the trend (see the Resources section for detailed instructions). Notice how that affects the new average trend line (see Figure 8). Since it no longer includes the outliers that were skewing the data, it has emerged from behind the Company Average trend line and has moved closer to the industry trend line.

Figure 8. Newly Updated Trend line Excluding Overstaffed Projects

While the new average trend line reflects a decreased staffing size, the reduction is not so drastic that it will inhibit the company's ability to adopt the new behavior. Instead of cutting teams in half, each project team has been reduced by about 10%, a much more manageable change. In exchange, companies can than take their surplus of staff and apply them to other projects, thus *increasing* their overall productivity ability to manage product demand.

Based on the manipulations made to the Company Average trend line, the final step in this exercise would be to create a new Estimation trend line and then import it into a SLIM-Estimate® template. Once imported, this trend line can be used like any other custom trend for estimating future projects. The only difference is that this trend has systematically manipulated the project data so that it will model a more desirable behavior and encourage *improved* performance. Applying these techniques within your own organization will not only provide you with insights into areas for improvement, but also help you achieve your desired goals.

Set the Stage for Success

Donald Beckett

Troubled software projects quickly devolve into a blame game. Management blames the project manager. The project manager blames the developers (and, secretly, management). Developers have a wide array of blame options: senior management, project leaders, suppliers, customers, partners, sales — even each other. More important than assigning responsibility is determining what measures can be taken to reduce the incidence of software projects that miss their schedule, exceed their budget, deliver a mediocre product, or any combination of these.

The fact is management decisions made before the project is underway are a significant determining factor on whether a software project succeeds or fails. Management choices can either handicap a project before it begins or create the environment in which it can succeed.

Software development as a business or practice is 50 or 60 years old and ample data exist to distinguish between measures that promote success from those that contribute to failure. While adhering to the following seven principles will not guarantee that every software project will be successful, they will reduce the incidence of problem projects. As a corollary, ignoring them practically guarantees failure.

1. Your project is bigger than you think

This is a perception issue and can be very subtle. Simply stated, no matter how thoroughly the proposed software is analyzed, there remain issues that will only surface as the actual work is being done. If this were not the case, if software development were a straight forward deductive process, there would be no need for developers at all. We would simply feed the results of our analysis and design into a Case tool which would generate the code to support the logic. Unfortunately, Case tools did not prove to be a panacea and did little to improve overall productivity. The track record is that software projects take longer to develop, require more effort (i.e., cost more), and create more functionality than were originally planned. A study conducted by QSM found that, on average, projects exceed their schedules by 8%, cost 16% more than planned, and develop 15% more functionality than anticipated. Software consultant Capers Jones has stated that projects grow, on average, 1.5% per month.

Skilled project managers intuitively recognize that there are unknown factors and try to account for these by buffering the budget and schedule in their project plans. Often, these are the first things to be cut during plan review, since they cannot be identified and the project plan becomes a best case scenario, yet seldom plays out in reality.

This is a point at which business leaders can make a critical and positive contribution to project success. If the software is important enough to the enterprise to develop, make sure that it has sufficient schedule and budget to succeed. Since it contains more functionality than can be seen, and will require more time and effort to be developed, plan for this to happen. This is emphatically not padding an estimate; it is recognizing reality and planning accordingly. If you are using a parametric estimation tool to determine budget and schedule, increasing the size of the functionality to be developed by 15% will assist you in your planning. Remember, the additional cost, schedule, and functionality are already there. You just can't see them. Refuse to acknowledge their existence, and the project is well on its way to exceeding both budget and schedule.

2. Schedule and cost/effort are not interchangeable

Our concept of productivity comes from manufacturing. However, there is a key difference between software development and manufacturing. In a factory, if you have one assembly line and want to double production or reduce by half the time required to produce the same quantity, you add a second assembly line or a second shift. In essence, you double the effort to double your output or reduce the schedule by 50%. The relationship is linear.

This does not work for software development. In software, the relationship between schedule and effort is non-linear: one unit of schedule reduction is purchased at the cost of many additional units of effort. This is not theory; it is supported by 40 years of research. Figures 1 and 2, below, illustrate this concept. Figure 1 represents a business financial project that lasts 9 months and is average for productivity, effort, and schedule when compared to the QSM database. Figure 2 illustrates what occurs when the project is forced to complete in 8 months.

In essence, an 11% schedule reduction is purchased by a 56% increase in cost/effort. It is beyond the scope of this paper to go into detail why this occurs; but the fact remains that this is the wisdom derived from thousands of completed software projects. Neither of the solutions illustrated above is impossible; both are well within the normal range of variability observed for projects. Business leaders concerned with spending their IT budget judiciously can accomplish a great deal simply by relaxing the schedule modestly. (This will also have a positive impact on product quality, since testing won't have to be shortchanged to meet an arbitrary end date, and may improve team morale which could reduce staff turnover.) It is important to note that the relaxed schedule needs to be planned into the project, not added when the project is about to miss its scheduled end date. Planning a more relaxed schedule allows the project to take advantage of the non-linear relationship between schedule and cost/effort and use a smaller and less expensive development team.

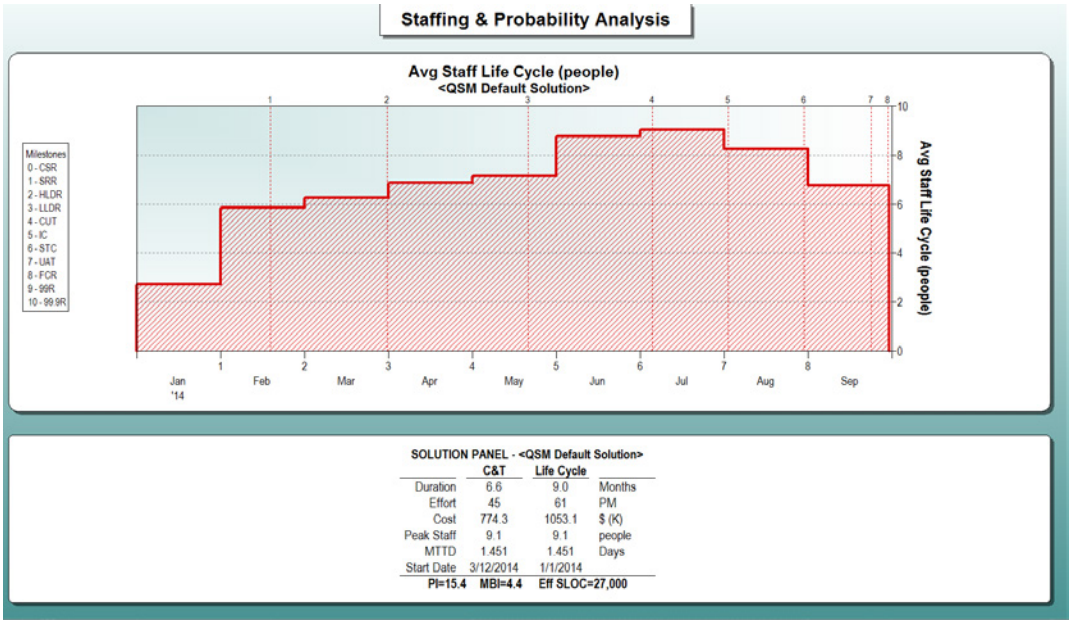


Figure 1. 9-Month Business Financial Project with Average Productivity, Effort, and Schedule

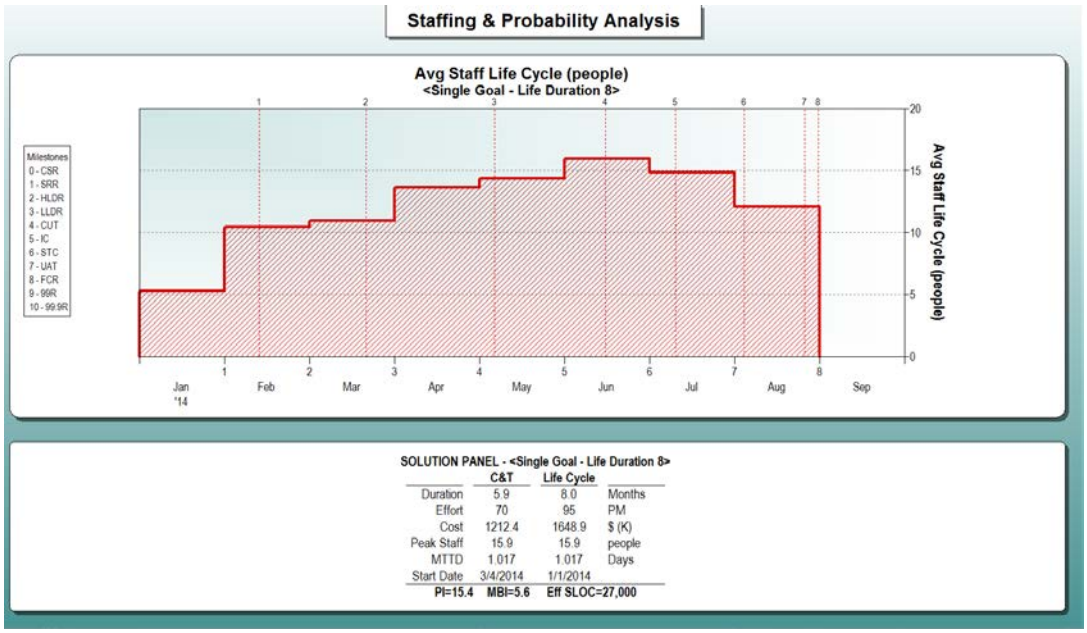


Figure 2. Results for Same Project Compressed from 9 Months to 8 Months

3. Know your capabilities

Organizations are creatures of habit. What this signifies for software is that within a company or a company division, there are identifiable patterns for how projects are staffed, their productivity, quality, and whether schedule or cost is optimized. These patterns are not

immutable; but they cannot be changed until they are identified and effort is consciously directed to alter them.

Do you know what your organization's patterns are? If not, the old adage applies that those who ignore history are condemned to repeat it (and any haphazard improvement efforts that are attempted will flounder). Every software project has the following features:

- It creates something (which can be quantified)
- It requires time and effort to do this and has an associated cost
- Issues arise throughout this process (defects)

These form the basis of the data that should be collected for every software project. With them productivity, schedule, and quality analysis can be done quantitatively. Impossible schedules (for upcoming projects) can be identified. Tradeoff analysis between cost and schedule becomes possible (and empirically based). Are you collecting these as every software project completes? Are they maintained in a database where they can be analyzed? These data are the byproduct of all software projects: size, schedule, effort, and defects. If you are not currently collecting and analyzing these, this is where you need to begin.

4. Keep the team small

There are compelling reasons to staff software projects sparingly:

- It increases productivity – significantly
- It *does not* affect the schedule (make projects take longer to complete)
- It costs less

Here is the supporting evidence taken from a study of over 2,100 completed software projects. In Table 1, the projects were sorted into size categories. Each size category was then sorted by the average project staff. For each staffing quartile within a size category the median (and average) productivity was calculated. A comparison was done between the productivities of the highest and lowest staffing quartiles for each size category. Without exception, the lowest staffing quartile was more productive with a range between 277% and 804%.

Productivity Rates (FP/PM) Smallest to Largest Staffing Quartiles					
Size Range (FP)	Lowest Staffing Quartile		Highest Staffing Quartile		Productivity Ratio
	Productivity (FP/PM)	Median Staff (FTE)	Productivity (FP/PM)	Median Staff (FTE)	
1-100	7.17	0.86	2.57	2.53	2.77 to 1
101-200	13.68	1.19	2.83	4.41	4.83 to 1
201-300	17.44	1.59	3.15	6.62	5.54 to 1
301-500	27.15	1.73	3.96	7.47	6.86 to 1
501-1000	34.96	1.76	4.35	10.95	8.04 to 1
>1000	45.29	2.86	5.76	15.04	7.86 to 1

Table 1. Comparison between Productivities of Highest and Lowest Staffing Quartiles by Size Category

The most frequent rejoinder to information like this is "I don't have time to use a small team. I've got a deadline to meet." The following figure (Figure 3) graphs the median schedule for each quartile and every size category, and shows that larger team sizes do not necessarily complete projects sooner.

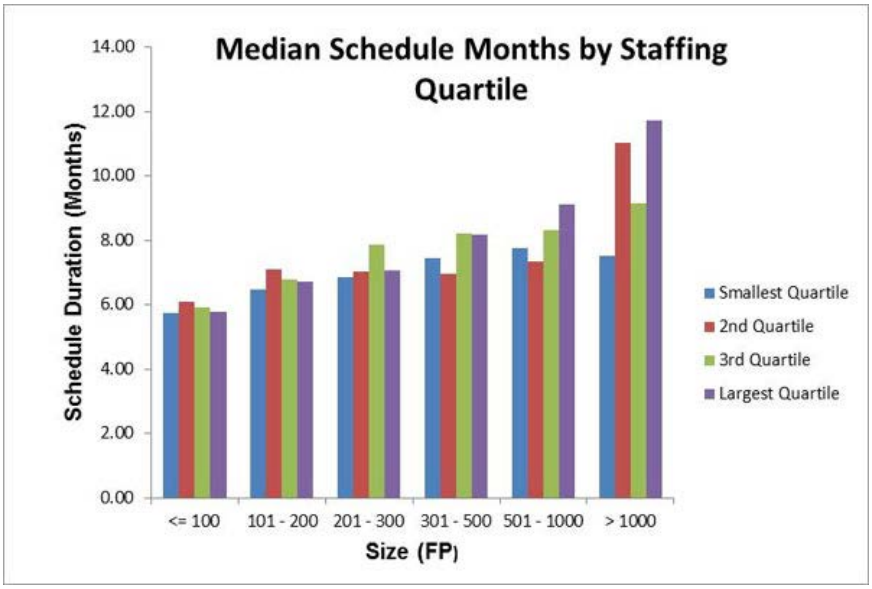


Figure 3. Median Schedule Months by Staffing Quartile

Deadlines are real; attempting to meet them by using a large staff is ineffective and, for larger projects, counterproductive.

5. Get an independent evaluation of your plans

The process of determining the schedule and budget for an upcoming software project may be subjected to intense pressure from persons or groups with conflicting interests and opinions. What comes out of this process may or may not be practical or even possible. In most cases the result is not empirically based. This is where an independent assessment by an experienced estimator using a parametric estimation tool becomes extremely valuable. As an "outsider" with no skin in the game, this person can generate likely outcomes that can be compared to the desired ones.

If your organization has been collecting project history, the estimator's models can be based on proven capabilities. A project with an inaccurate budget, schedule, or staffing plan is well on its way to failure before it gets underway. Getting an independent estimate from a person or team whose job is to produce good estimates will identify potential problems before they manifest. By insisting that independent tool based estimates be done for upcoming projects, business leaders can help avoid potential pitfalls and help create a plan that can succeed.

6. Track performance to plan and re-plan as needed

There is an old saying that no battle plan survives first contact with the enemy. Software projects share this trait and their plans must be able to adapt to what is transpiring. Attempting to keep to the original budget/schedule plan, when the assumptions it was based on are no longer valid, flies in the face of reality.

Where leadership can make a positive difference here is by establishing sound processes for monitoring projects. This has traditionally been done using financial measures. Unfortunately, by the time financial monitoring indicates that a project is in trouble, it is usually too late to take effective corrective action.

There is a better way. Every project plan has a schedule with milestones and a plan for expending effort (and money). Deliverables are created and tested. Using a parametric monitoring tool, these metrics (milestones, effort, and software deliverables) can be used to track progress and forecast likely outcomes. The advantage of these over strictly financial monitoring is that this can be done much earlier in the project when there may still be time to consider alternatives and take corrective action.

Identifying a problem does not resolve it. But, knowing that it exists sooner rather than later is a clear advantage and may allow alternative possibilities to be explored.

7. Allow time for planning

A consequence of short schedules is that both planning and testing do not receive the time and effort that they require. The results aren't pretty. Inadequate planning leads to software that does not meet the business requirements. Abbreviated testing allows defects to be discovered in production where they are more expensive to fix and visible to customers. QSM has twice conducted studies that compared projects that spent more than average effort for analysis and design (planning) with those that used less than average. Both times, the results were striking. Projects that spent above average effort in analysis and design completed sooner; had higher productivity; used less effort (cost less); and produced fewer defects. Table 2 below summarizes the results for the more recent QSM study, "An Analysis of Function Point Trends:"

Comparison at 20% Design Effort		
Medians		**% Difference**
PI <= 20%	**11.04**	
PI > 20%	**14.19**	**29%**
FP/PM <= 20%	**6.20**	
FP/PM > 20%	**7.93**	**28%**
Duration <= 20%	**7.23**	
Duration >20%	**6.20**	**-17%**
Total Effort <=20%	**22.59**	
Total Effort > 20%	**20.29**	**-11%**
Average staff <= 20%	**2.34**	
Average staff > 20%	**2.50**	**7%**
FP size <= 20%	**157.00**	
FP size > 20%	**171.00**	**9%**
Defects <= 20%	**20.00**	
Defects > 20%	**19.50**	**-3%**

Table 2. Comparison at 20% Design Effort

This is another area is which business leaders can exercise a positive influence on software development by insisting that sufficient time and effort are allocated to planning. The projects will complete sooner, cost less, and have fewer defects.

Bringing It All Together

While business leaders do not directly manage software projects, they exercise a profound influence on them through the decisions they make. In summary, here are the decisions leaders can make that create the environment in which projects can succeed.

- Plan for growth. Projects are larger than they seem at the outset and will require more time and effort to complete.
- Give projects the time they need to succeed. Setting an aggressive schedule is the single worst thing that can be done to a project.
- Collect and use project history to make empirical decisions.
- Use the smallest staff possible to complete the task. It won't hurt the schedule and costs a lot less.
- Get an independent evaluation (estimate) of every project – and pay attention to them.
- Establish and enforce procedures for ongoing monitoring of software projects.
- Spend time and effort up front determining what to do before starting to do it.

Traits of Successful Software Development Projects

Larry Putnam, Jr.

*This article originally appeared in the **Government Computer News**, June 26, 2014, and is reprinted here with permission.*

Enough already with Healthcare.gov and its embattled IT cousins; let's talk about government software projects that actually worked. Specifically, what do successful projects have in common, and how might forward-looking agencies replicate those conditions for success?

It's a difficult question to answer, but if we use the QSM database, which holds carefully vetted information on over 10,000 completed software projects (including thousands of government projects), we can uncover common traits that can predict success for government IT projects.

Defining Success

Because "successful" and "embattled" are relative terms, we'll use the designation "best-in-class" – for projects that performed at least one standard deviation better than average in time to market, effort (or cost) expended and quality – and "worst-in-class" for the exact opposite.

To put this in perspective, using a sample of over 500 business IT systems, best-in-class projects were (on average) 3.5 times faster to market than worst-in-class projects, and required 8.1 times less effort. When narrowing the sample to government-only business systems, the best-in-class projects were, on average, 3 times faster to market than what was typical for the IT industry, and required 6 times less effort. Meanwhile, the worst-in-class projects, on average, took twice as long to complete and required 5 times more effort than the typical industry standard.

In software development, effort and time to market often work against each other. (When schedules are compressed, we typically pay a premium in effort.) Hence, projects with outstanding marks in both effort and time to market are quite rare and indicative of

uncommon success. So given that background, what are the most common success factors for a best in class project?

Observation #1: Smaller Is Faster

The first and most prominent commonality among our best-in-class projects was a relatively modest team size. Contrary to the widely held belief that one can expedite a project by adding people to it, precisely the opposite is true. Larger teams often take longer to develop the same functionality as smaller teams.

Statistically speaking, the most efficient project teams rarely exceed 7-10 people (except on some of the largest, most complex systems greater than 200,000 lines of code). This represents an 18-39% reduction from industry averages. Furthermore, teams falling within this "optimal" size range completed projects with 28-69% greater productivity than larger teams with comparable tasks.

Based on this data, it stands to reason that government agencies should keep their IT project teams as small as possible – both for reasons of efficiency and for the added flexibility to take on new projects and reduce backlog.

Observation #2: Requirements Required

The second commonality pertains to the allocation of time and effort. Statistically, we found that best-in-class projects invested a much greater proportion of total effort in pre-coding activities – requirements analysis, architecture, and high-level design – than worst-in-class projects (28.0% vs. 7.6% of the total effort).

These findings were even more pronounced for government projects (36.3% vs. 4.4% of the total).

This data appears to support a "pay up-front or pay later with interest" concept, whereby best-in-class projects invest time and effort up-front through analysis in order to bypass greater time and effort expenditures later through maintenance and repair.

It's important to note that QSM's data set included both Agile and non-Agile projects. Regardless of whether or not the project is following an Agile approach, sufficient requirements analysis, architecture and high-level design work needs to be done to ensure the backlog of required functionality has been clearly identified, sized and prioritized prior to construction. Otherwise, there will be extensive rework that can quickly turn an otherwise "best-in-class" project into a "worst-in-class" project.

For government agencies, the lesson would seem to be one of patience and wise investment. More often than not, the QSM data shows that management decisions to overstaff or to begin coding prematurely in an attempt to achieve an aggressive schedule backfire and ultimately produce inferior results at greater cost in the same or longer time frames.

The Case for Quantitative Data

Of course, individual projects will vary, and factors like team size and percentage of effort spent in analysis cannot guarantee the success of any single initiative. But the principle of using past software project data to predict future project outcomes is crucial to modern government.

And considering government's propensity for outsourcing, the ability to perform feasibility estimates (based on past project data) holds greater value for contracting officers who use those estimates to spot unrealistic bids up-front and, on occasion, to defend the procurement process from unhappy vendor protests.

These analyses are typically called Independent Government Cost Estimates, and they're critical weapons in the war against project inefficiency.

But if the data tells us anything, it's that estimation shouldn't stop when development starts. Project managers should be continuously tracking, reforecasting and reassessing their project decisions based on shifting requirements and budgets.

Perhaps, in addition to small team size and robust analysis, we should add a third common trait among best-in-class projects: active, dedicated project managers.

After all, what use is data without the right people to interpret it?

Project Clairvoyance

Larry Putnam, Jr.

Bob Dale - standing in the corner of his Midtown office, face flushed with uncertainty, hands nervously cupping a re-gifted Magic 8-Ball® - is about to make the worst decision of his life.

A week ago, Bob's group was asked to bid on a fixed-price contract for a top financial firm's marquee software project - six million lines of code and no margin for error. Trouble is: Bob's never handled a project like this. So there's no way of knowing whether his estimate - six months, staff of 1,300 - is reasonable or a recipe for disaster. Nervously, our hero sketches a few equations on the back of an envelope and turns the 8-ball on its head. "All signs point to yes."

Six months from now when the project lies in shambles, beset with cost overruns and unending delays, Bob will remember the precise instant that his professional life took a nose-dive into the Bermuda triangle.

Or maybe not. In an alternate universe, Bob does things slightly differently. Before signing off on the fixed-price bid, our hero consults his estimation software.

He enters a few inputs and sees that his chances for success are only slightly higher than the chances of an impromptu alternate universe forming in the office suite next door. Quickly, Bob brings his original plan into alignment with the latest financial industry performance data. He adjusts the schedule to 20 months and reduces the staff to 250 people. The software validates his plan against a sample of recently completed financial sector projects. He factors in uncertainty and his firm's risk tolerances, and he generates a revised estimate.

The probability of successfully executing the revised (i.e., data-driven) plan increases to 90%. "Thank heavens for technology," he thinks to himself. Catastrophe averted; Magic 8-Ball® re-gifted once again.

Witchcraft, or the Science of Estimation?

"This is a story about the power of data," explains my colleague Kate Armel, Director of Research and Technical Support for project estimation authority at QSM. In other words, data-driven estimates are grounded in thousands of real project outcomes, not opinions

(which can be biased or inaccurate) or marketing hopes. If learning from experience is the key to success, imagine what someone could do with real-time access to three decades of research, thousands of projects, and over 600 industry trends.

"Having relevant data on tap helps our clients understand and account for the complex and counterintuitive interactions between staff, schedule, defects, and productivity," says Armel. "Estimators can factor in the effects of using different technologies and development methods and quickly counter unrealistic expectations. For example, they can show - empirically - that using large teams delivers only modest schedule reduction while increasing cost and defects by an average of 300-400 percent.

"We know that software development is a moving target. In the past decade alone, incremental development and Agile methods continue to shrink the scope of the projects in our database, reducing failure rates by over 10% and improving average time to market by nearly 20 percent. For firms determined to stay competitive in a rapidly changing environment, this kind of continual real-time feedback is invaluable" (see Figure 1).

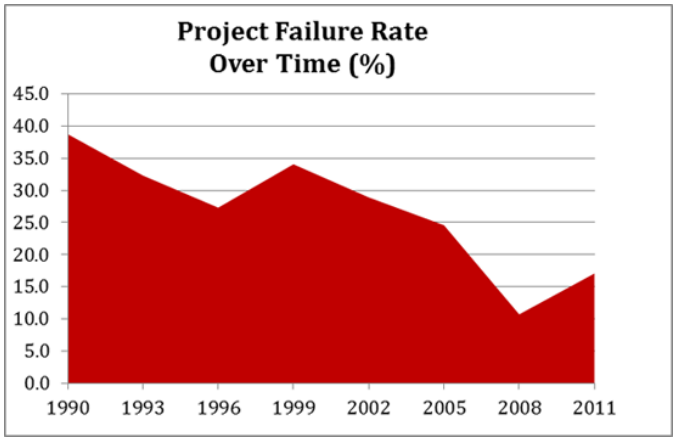


Figure 1. Project Failure Rate over Time

Clearly, there's no shortage of software stats and metrics for CIOs to ponder. The question is: how does access to accurate, timely information improve a company's bottom line?

Fear and Loathing in the C-Suite

In a 2007 survey, IBM's Scott Ambler found that 68.6% of IT professionals have been involved in projects that they knew would fail from the very start. Go ahead and read that sentence again.

"In other words, the people in a position to get the project on the right track, or at least in a position to influence the people who could do so, couldn't do anything about it," concluded Ambler.

A quick check of QSM's database of 10,000-plus completed projects reveals the impact of signing up for unachievable plans. A recent sample of more than 1400 projects that provided

overrun or slippage data showed that one third overran their planned schedules by at least 20 percent (Figure 2).

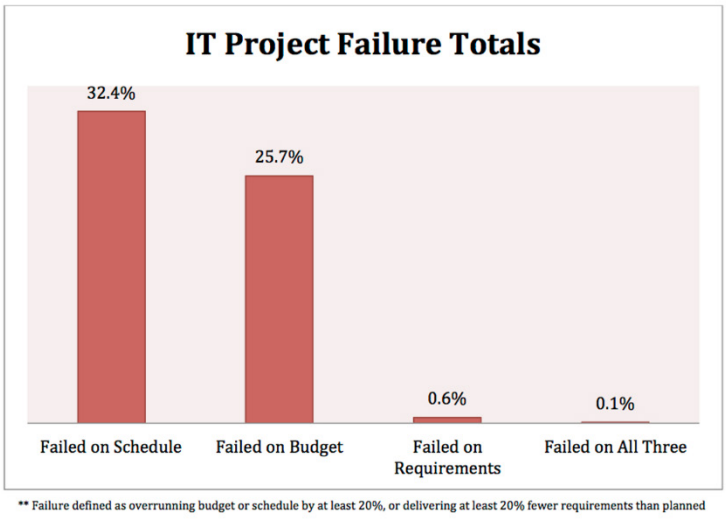


Figure 2. IT Project Failure Totals

How do project managers react when the schedule starts to slip? Typically, they pile on staff, driving up the cost and creating additional defects. (About 26 percent of projects exceeded their budgets by 20 percent or more.) As a last resort, stressed projects may cut features, delivering less functionality than planned. Only 0.6 percent of projects chose this (most drastic) option, probably because the "minimum releasable scope" constraint reared its ugly head.

Bad (or no) initial estimation forces projects to make unpalatable choices (a.k.a., the least bad option), often long after the window when timely intervention has the best chance of turning a project around. No matter how talented a team and its IT leadership may be, there's no painless way to combat faulty expectations or untenable parameters.

"We joke about the 8-Ball®, but that's really not so far from the truth," says my colleague Keith Ciocco, Vice President of Sales at QSM. "For instance, we're seeing a number of executives who confuse bottom-up planning with estimation. They'll divide a software project into its component tasks, and then try to match each task with the appropriate numbers."

So what's wrong with that?

"The problem is timing. Estimation has to come first, from the top-down. Schedules and budgets are usually locked-in early in the lifecycle - before we ever know what those detailed component tasks will look like."

In short, it's rearranging deck chairs on the Titanic, when the course for the iceberg has already been set.

What executives fail to realize, it seems, is that the majority of their IT challenges are tied to estimation - hitting profit margins, setting backlogs, utilizing staff. On top of that, the project data used to make accurate estimates can be essential for assessing productivity, comparing performance, and measuring improvement.

"CIOs know they need to do more with less, but they don't have any way to measure their progress," Ciocco says. "How does their productivity compare with competitors' teams'? Where are the bottlenecks happening?"

"When projects go awry, we tend to assume that the issue was our performance. But when you look at the data, it often shows that the team performed quite well. It's merely that the budget and the timeline were completely unrealistic."

How's that for music to a CIO's ears: You could drastically improve your company's performance and profitability without doing anything - just by making smarter, more informed decisions.

Estimation: Luxury or Necessity

CIOs are stressed, and with good reason. If you think making major decisions on massive IT projects is a nerve-wracking experience under normal circumstances, how about in a down economy?

"One of the biggest objections we hear to project estimation is that it's a luxury," Ciocco says. "Executives tend to think, 'Sure, we'd love to have access to all that great data. But we just can't afford it right now.'"

The irony is that they can't afford to go without it. In fact, a tough economic climate only magnifies the staggering financial risks associated with bad decision-making at the outset. About this, the data is unequivocal: That first decision - that initial estimation - is the key to everything -productivity, profitability, and ultimately project success.

"So do whatever you need to do to make that first decision wisely," advises Ciocco. "No matter what your 8-Ball® is telling you."

---

5. LONG TERM TRENDS

"The result of long-term relationships is better and better quality, and lower and lower costs."

– W.E. Deming, American engineer, statistician, professor, author, lecturer, and management consultant

"Maturity is achieved when a person postpones immediate pleasures for long-term values."

– Joshua L. Liebman, American rabbi and best-selling author

A View from Above

Katie Costantini

From its infancy, the software development industry has struggled to reduce costs, improve time to market, promote product quality and maintainability, and allocate resources to their most efficient uses. But because the landscape is constantly shifting, process improvement has not grown easier with time. An increasingly global marketplace and the accelerating pace of technological change present a continually evolving set of challenges and goals, and companies who can't adapt quickly don't survive.

One of the few constants is the ongoing need for practical measurement and metrics. To avoid costly overruns, software development firms must negotiate realistic schedules and budgets that reflect their actual ability to deliver software. This kind of reality driven planning only comes about when firms ask questions, gather data, and measure progress against an empirical baseline. Since 1978, QSM has provided data and industry trends from our 10,000+ database of completed software projects to support benchmarks of completed projects and estimates of future work. Our basic metric set focuses on size, time, effort, productivity, and defects, but these core metrics are supplemented by nearly 300 additional quantitative and qualitative measures.

In addition to assessing current performance, an industry database spanning over three decades can help answer broader questions like, "What kind of productivity gains can we realistically expect from adopting new technologies, or "Are we moving in the right direction?" In the 2006 QSM Software Almanac (IT Edition), we took a high level look at changes to software schedules, effort/cost, productivity, size, and reliability metrics from 1980 to 2004. Our latest study adds over 3000 new data points and some new measures and observations to our previous analysis.

Sample Demographics

The QSM database ("QSM Project Database") captures software lifecycle metrics from the initial feasibility study period through post implementation maintenance phase activities. Project types range from Real-time to Telecommunications to System and IT applications.

Like the IT Almanac, this paper focuses on a large and diverse sample of Business (IT) projects completed from January 1, 1980 to the present.

Projects in the Business domain typically automate common business functions such as payroll, financial transactions, personnel, order entry, inventory management, materials handling, warranty and maintenance products. Frequently they rely on distributed architectures and transaction processing supported by telecommunications infrastructure (LANs) and database back ends.

Though our database captures the full software lifecycle, this study focuses on schedule and effort from the start of Requirements (analysis and design) through the end of Build & Test when the system is put into production. Effort figures include all skilled labor needed to produce a viable product (analysis, design, coding, integration, testing, certification, documentation, and management).

Our long term trends sample contains:

- 8000+ Business projects completed and put into production since 1980.
- Over 600 million total source lines of code (SLOC).
- 2.6 million total function points.
- Over 100 million person hours of effort.
- 600+ programming languages.

Software projects can be classified in many ways: by country of origin, project type, technology (language) or industry. Projects in the QSM database hail from 5 continents with the majority developed in the United States, the United Kingdom, the Netherlands, Canada, and India (Figure 1):

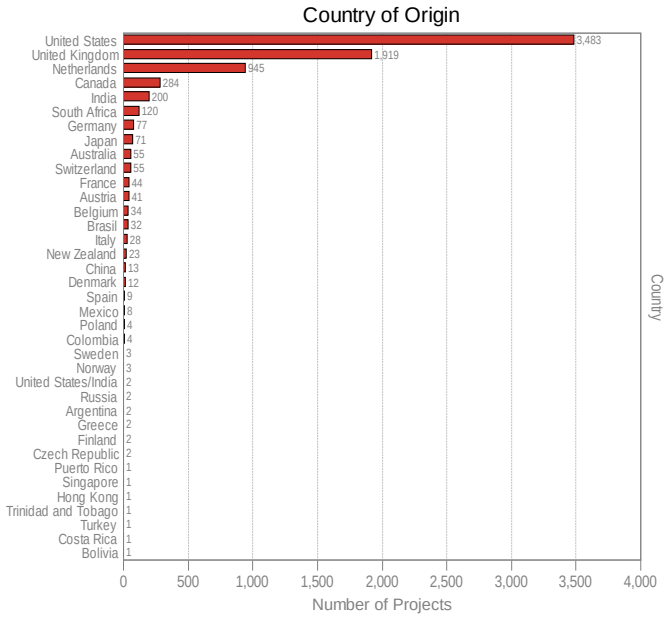


Figure 1. Projects vs. Country of Origin

Most projects in our IT sample were developed for the Financial, Utilities, Manufacturing, Government, and Health Care industries (Figure 2). In many cases, IT systems differ more by function than by industry but for highly regulated industry sectors like finance, government, and health care, complying with government standards required increases time to market and inflates cost . Due to the large sample size, bars with a 0% label contain between 2 and 26 projects each.

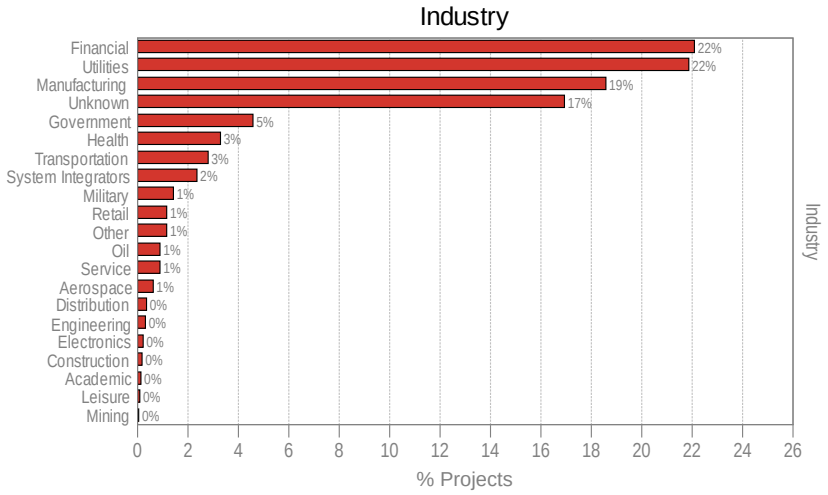


Figure 2. Percentage of Projects Grouped by Industry

Stratifying the overall Business sample by functional detail promotes apples to apples comparisons between systems that solve similar problems and perform the same functions. The majority of projects in our sample were Financial Management, Billing, E-commerce/Web systems, Facilities Management, Customer Care, and Sales applications (Figure 3).

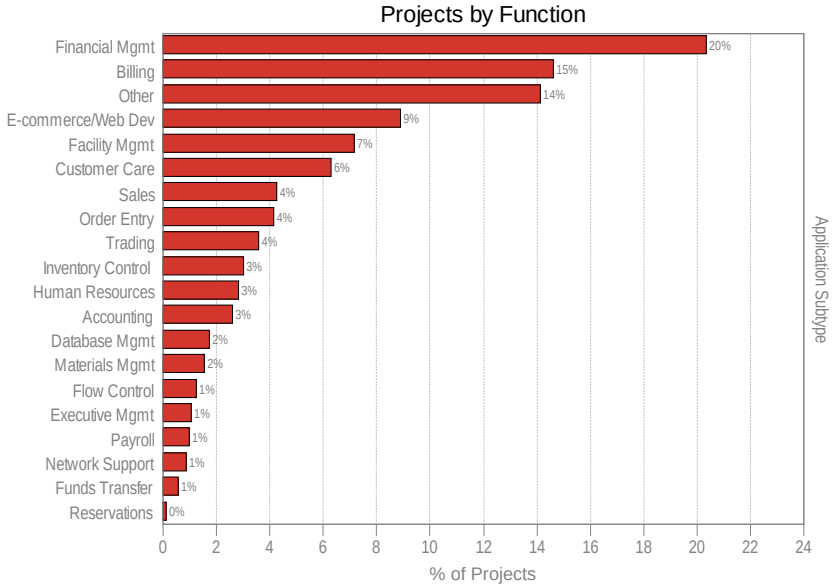


Figure 3. Percentage of Projects Grouped by Function

Figure 4 shows how the number of projects collected during each five-year period in our IT sample has increased over time. The growth of our database is a rough proxy for the increasing presence of software used to power everything from cars to computers to entertainment and communication devices.

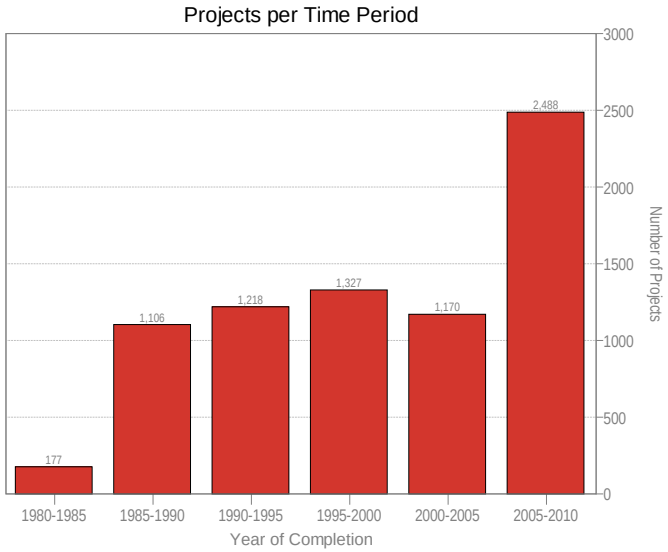


Figure 4. Projects Grouped by Completion Year

The "Typical Project" over Time

What does a typical project in the QSM database look like, and how has "what's typical" changed over time? To find out, we segmented our IT sample by decade and looked at the average schedule, effort, team size, new and modified code delivered, productivity, language, and target industry for each 10-year time period.

During the 1980s, the typical software project in our database delivered 154% more new and modified code, took 52% longer, and used 58% more effort than today's projects. The following figure (Table 1) captures these changes:

	1980-1989	1990-1999	2000-2009	2010-present
Schedule (months)	17.8	11.4	10.2	11.7
Effort (person hours)	18,019	13,541	8,658	11,414
Team Size (FTE staff)	7.3	6.7	6.7	6.9
New/Mod Code (KESLOC)	75.3	58.8	36.2	29.6
productivity index (PI)	14.7	16.2	12.9	13.2
Primary Language	COBOL	COBOL	Java	Java
Industry Sector	Financial	Utilities	Manufacturing	Financial

Table 1. Changes in Project Demographics over Time

- **Schedule**: Project schedules have decreased dramatically from a high of almost 18 months in the 1980s to 10 - 11 months after the year 2000.
- **Effort**: Average person hours per person month for Analysis through Build and Test started off high in the 1980s but decreased through the 2000s before increasing slightly from 2010 to the present. Overall, the trend is toward projects expending less effort in the Analysis through Build and Test phases.
- **Size**: from the 1980s to the present, average new and modified delivered code volume was reduced by about 65%. Later in this paper, we'll explore this reduction in size in more detail.
- **Team size**: Average team size changed only slightly since the 1980s, dropping by only half an FTE person. We suspect the influence of project size reduction has been offset by increases to architectural and algorithmic complexity. While smaller systems generally require fewer developers, technical complexity tends to increase the demand for team members with specialized skills and diverse subject matter expertise.
- **Primary Language**: For projects put into production during the 1980s and 1990s, COBOL was the dominant programming language. In the 2000s, Java eclipsed COBOL and has continued to be the most frequently used primary language. People are often surprised at the enduring presence of COBOL, but the majority of recent COBOL projects in our database represent maintenance releases of existing systems rather than new developments.

Programming Languages over Time

In 2006, the top three programming languages were COBOL, Visual Basic, and Java. One thing we did not look at was the use of single languages vs. multiple languages. Figure 5 below shows the relative proportion of projects developed in a single language to projects developed in multiple languages.

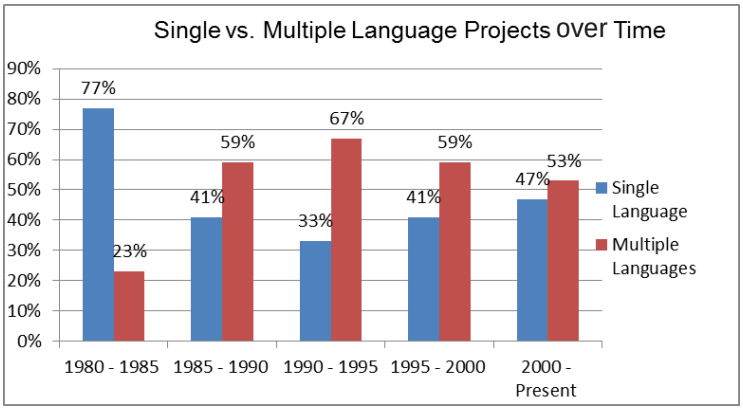


Figure 5. Language Trends over Time (Single vs. Multiple)

We expected the data to show that single language projects were on the decline, but our database had a surprise in store for us. We suspect that development class (whether a project contains entirely new functionality or enhances an existing system) may be influencing the results below. Over the last several time periods, enhancements to older

systems have increased relative to new developments. Enhancements to legacy systems that used a single language may use single languages as well.

The word cloud below (Figure 6) shows the top 25 primary language[2] for projects completed after 2008. The most popular language (largest font) is Java (26%) with COBOL (11%) as a close second.

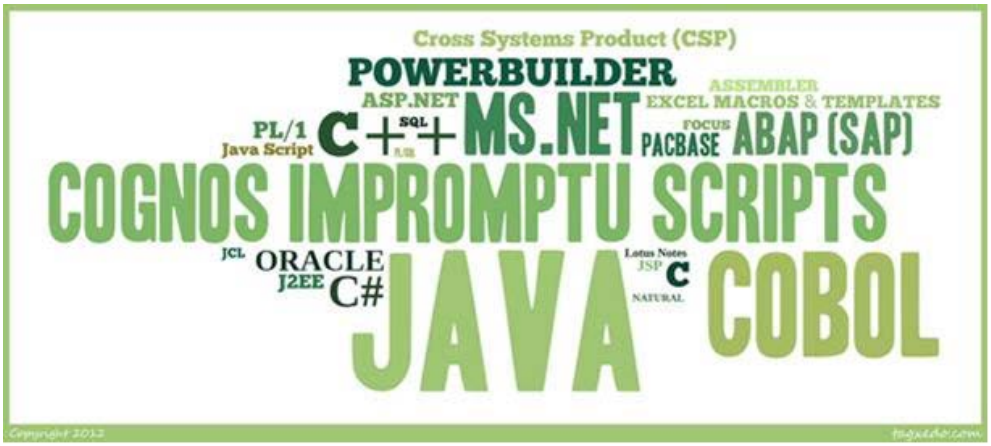


Figure 6. Programming Languages Word Cloud

Delivered Code Volume (Size) over Time

In the 2006 IT Almanac, we looked at how the average size (measured in new and modified code) of software projects had changed from approximately 85,000 new and modified SLOC in the early 1980s to about 28,000 by the early 2000s. The latest data shows that the long term trend towards smaller, more compact projects has continued (Figure 7):

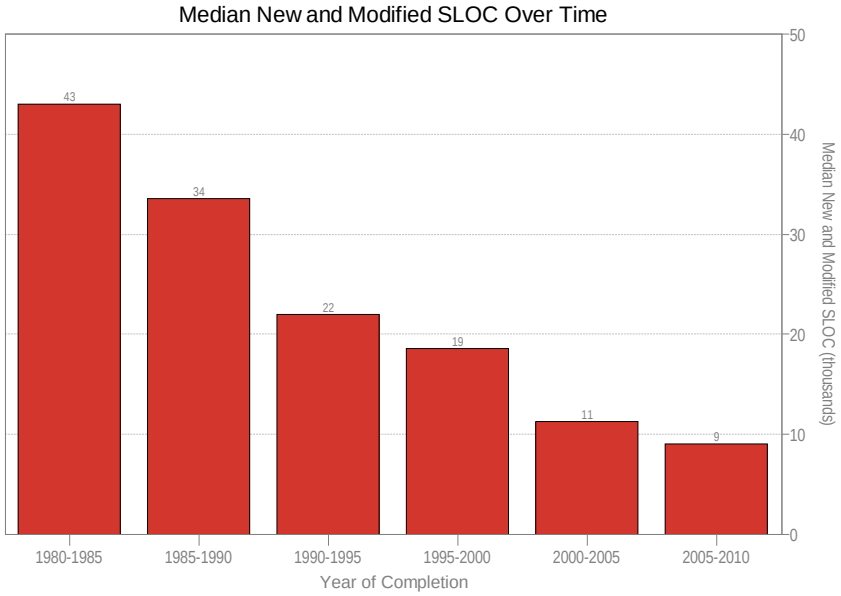


Figure 7. New and Modified Code Trends over Time

While the average is a useful measure of central tendency, it can be influenced by very large or very small values in the sample. The median (or "middle" value with half of the data above and below it) size values shown above demonstrate the trend toward smaller projects even more clearly.

In the early 1980s, the median new and modified code delivered was **four times larger** than median project sizes for systems completed after the year 2000.

Why Are Projects "Shrinking" over Time?

On average, today's developers deliver about one fourth as much new and modified code per project as they did 30 years ago. What is driving the steep and sustained decline in delivered code volume? This size reduction most likely reflects a combination of factors:

There is more unmodified code. Whether it takes the form of frameworks, reused/legacy code from existing applications, or generated code, reuse comprises an increasing portion of delivered applications. Since Figure 8 above reflects only new and modified code delivered during each time period, reuse is not reflected.

More efficient and powerful programming languages and methods. As technologies and development environments continue to evolve, each line of code delivers more "bang for the buck" per line of code in terms of functionality.

New lifecycle methodologies like Agile, RUP, and incremental builds attempt to manage scope creep by allotting smaller groups of features to predefined time boxes, sprints, or iterations.

Measurement inefficiency. Well established and defined sizing techniques like function points require trained practitioners and can be expensive to count, and they don't always capture all the work required to deliver the product. Organizations like IFPUG are developing techniques like SNAP (International Function Point Users Group, 2014) to account for this kind of nonfunctional work. Early, design-based size measures like requirements, stories, and use cases may be defined at too high a level to fully capture scope creep as the design evolves. As sizing techniques used in the industry become more refined, we'll be interested to see if the long term trend towards size reduction continues.

The scatter plot in Figure 8, below, provides another view of how average delivered project sizes have decreased over time. The blue trend line in the top tracks the average system size over the last three decades. While average size is going down, the range of system sizes observed seems to be increasing, with the greatest variability occurring from 2000 to 2006. In recent years, projects have stayed within 900 to 100,000 SLOC. There appears to be less variation than we saw in previous years, but this may change as projects are added to the database.

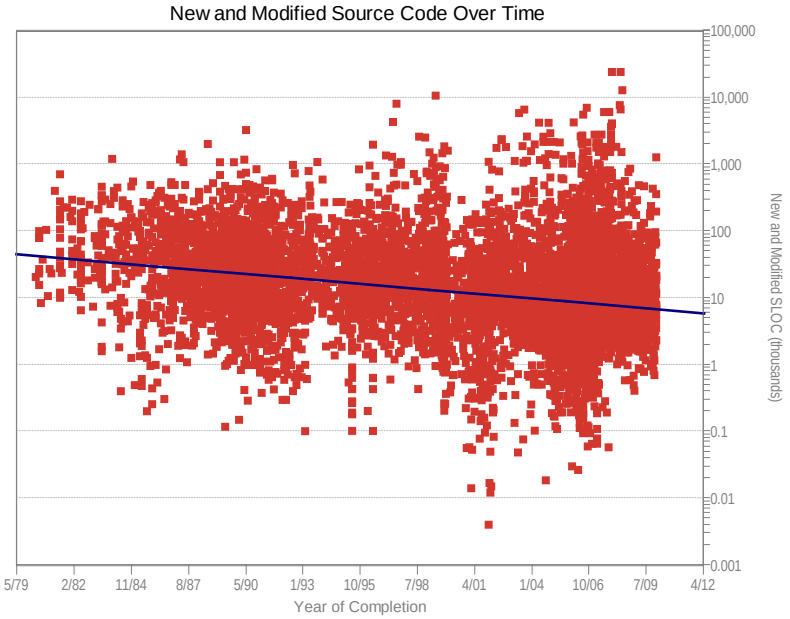


Figure 8. New and Modified Source Code over Time (Scatter Plot)

Code Reuse over Time

In Figures 9 and 10, below, we divided our sample into ten-year time buckets. The solid black line represents the average percentage of reused (unmodified) code. The percentage of reused code has declined steadily, from about 50% reused code, on average, for the 1980s, to about 45% reused code in the 1990s and about 35% in the 2000s.

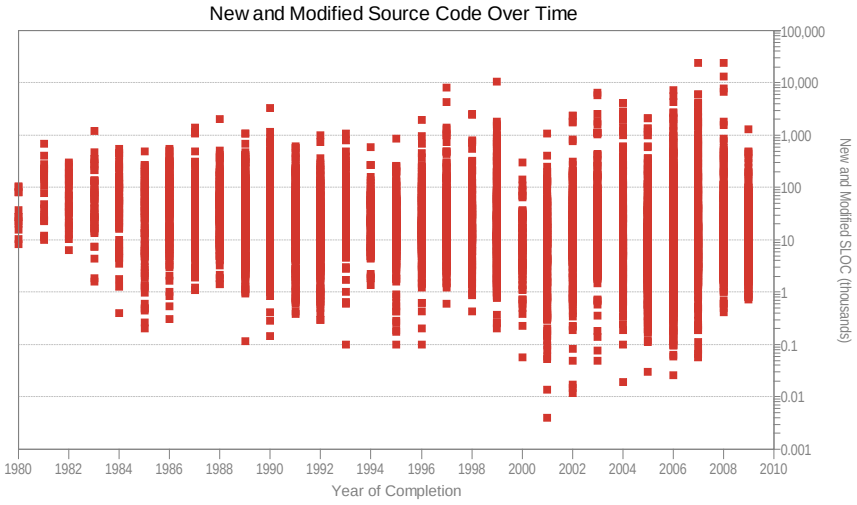


Figure 9. New and Modified Source Code over Time

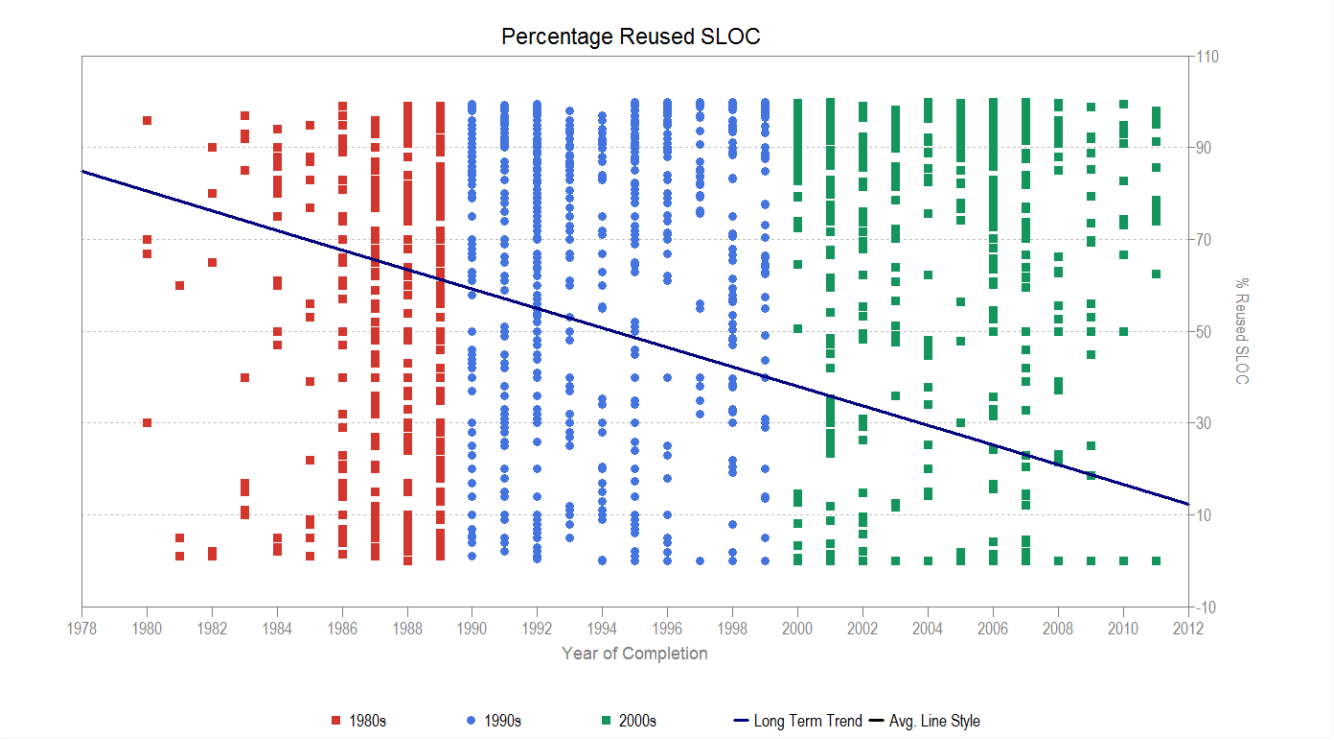


Figure 10. Percentage of Reused Source Code over Time

This decline in reuse may well reflect the realization that integrating existing frameworks and legacy code with newly written code is a significant complexity factor that can – depending on how well it is implemented – have dramatically different impacts on project productivity.

Development Class over Time

Out of 8,000 projects in our sample, about 69% provided development class information. Development classification describes the type of development effort undertaken by the project:

- **New Development**: 75% of the system is new functionality
- **Major Enhancement**: 25 to 75% of the system is new functionality
- **Minor Enhancement**: 5 to 25% of the system is new functionality
- **Conversion**: < 5% of the system is new functionality
- **Maintenance**

Figure 11 shows that about two thirds (63%) of the projects in our sample were Major or Minor Enhancements of existing systems. Under one third of the projects were New Developments.

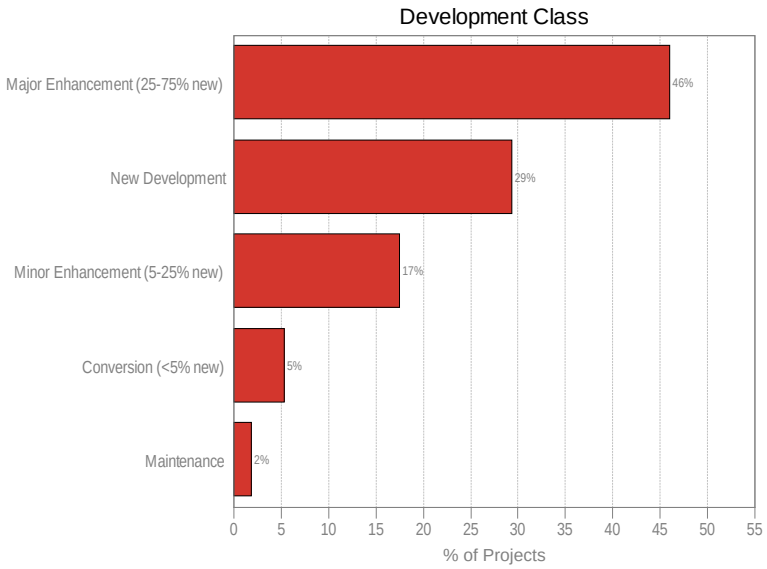


Figure 11. Development Classes over Time

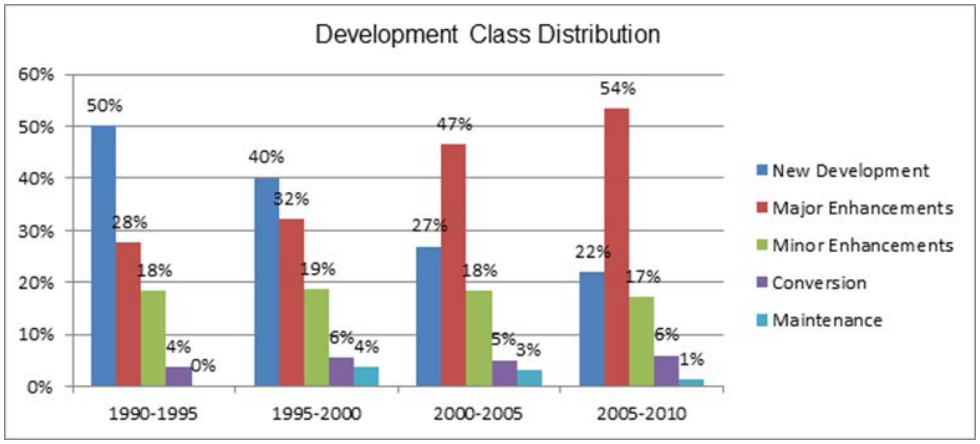


Figure 12. Development Class Distribution Percentage Grouped by Time (Year)

The bar chart at Figure 12, above, shows how the proportion of projects for each development class has changed over the last two decades:

- New developments have decreased by more than half since the 1990s (data was not available for earlier time periods).
- Major Enhancements have almost doubled, with Minor Enhancements holding relatively steady over time.
- In the early 1990s, enhancements to existing systems (major and minor enhancements combined) were roughly equal to new development. By the latest time period, enhancements outpaced new developments by a factor of three.

Project Schedule over Time

The average software project in the early 1980s took roughly two years to develop. Since the early 1990s, however, software development has stayed within the 10 to 11 month range (Figure 13). Median schedules show the same downward trend (Figure 14).

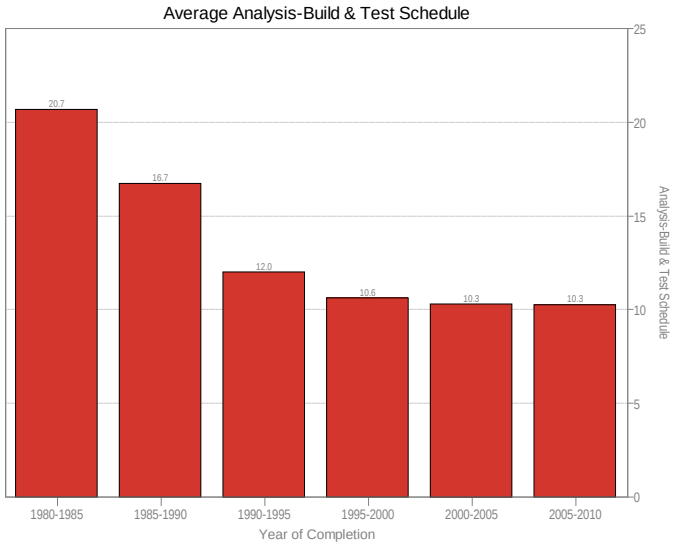


Figure 13. Average Analysis-Build and Test Schedule over Time

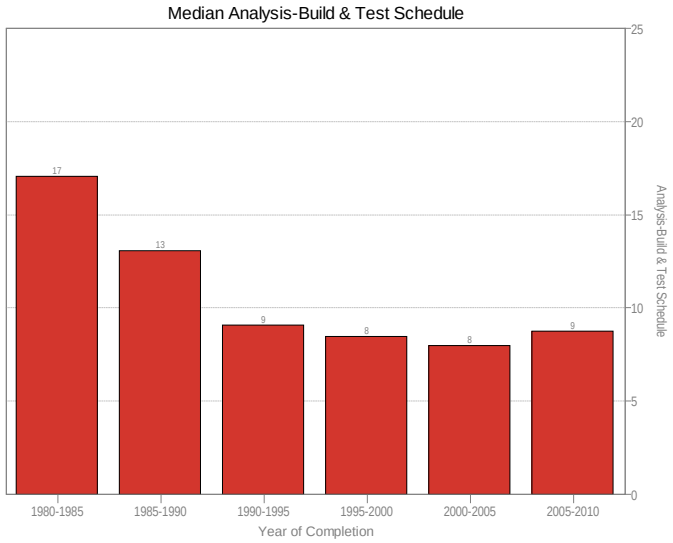


Figure 14. Median Analysis-Build and Test Schedule over Time

The individual data points shown in the scatter plot at Figure 15, below, provide some insight into the range of schedule outcomes. The portion of the graph from 2009 on shows less variability in project schedules than in the preceding years. This may reflect a corresponding decrease in size variation observed during the same time period earlier in this paper, or it

may reflect the increasing influence of time boxing methods used with by Agile and other iterative development methods.

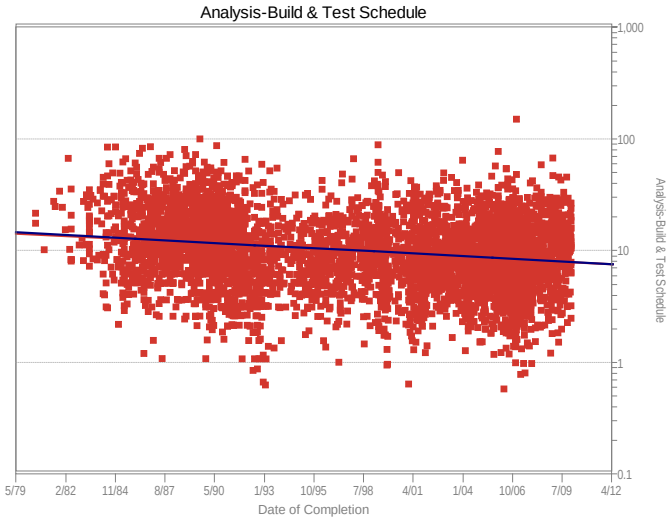


Figure 15. Analysis-Build and Test Scatter Plot with Average Trend line

Effort Performance over Time

The bar chart at Figure 16 shows average Analysis through Build and Test effort (in thousands of person hours) for five-year time buckets. Projects in the early 1980s took nearly twice as long as projects in the most recent time bucket. Effort expenditure is a function of staffing strategy (team size), productivity, and schedule. Over the same time period, project sizes have decreased by three fourths and schedules by half. As we saw in our *Typical Project* section, team sizes have remained relatively constant.

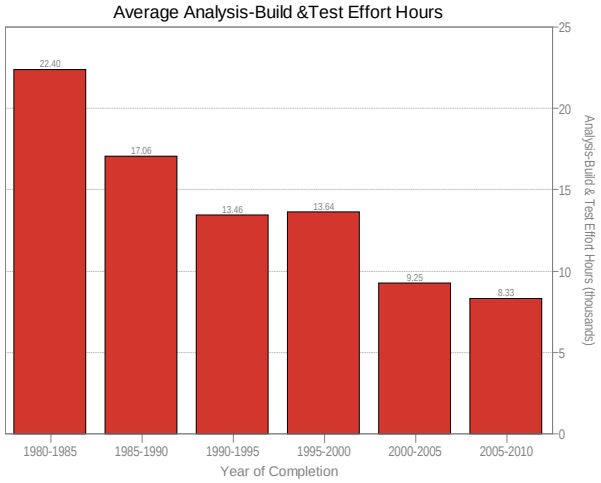


Figure 16. Average Analysis-Build and Test Effort Hours over Time

Why has effort reduction not kept pace with declines in average size and schedule? Application complexity (both algorithmic and architectural) may require more diverse skill

sets and thus, larger teams. More advanced and rigorous methods and practices may allow teams to work together more effectively. Finally, it may simply be that companies trying to meet aggressive time to market goals may be using larger teams than needed in the hopes of reducing the schedule.

The scatter plot of individual projects shown at Figure 17, below, reflects the overall trend toward lower effort expenditure. The average project in the early 1980s took about 8,000 person hours to complete, whereas the average project in 2010 used only 3,000 person hours. As we noted in the Typical Project section, average team size has remained fairly constant in recent years. Smaller projects with shorter schedules and roughly the same team sizes end up using less effort on average.

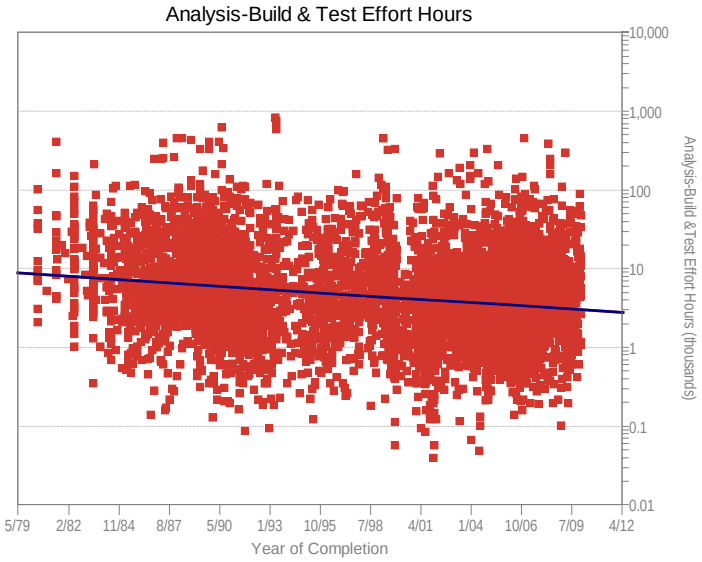


Figure 17. Average Analysis-Build and Test Effort Hours Scatter Plot with Average Trend line

Development Productivity over Time

QSM's productivity index is a measure of the total development environment. It embraces many factors in software development, including management influence, development methods, tools, techniques, skill and experience of the development team, computer availability, and application complexity. Values from .1 to 40 are adequate to describe the full range of projects. Low values are generally associated with poor environments and tools and complex systems. High values are associated with good environments, tools and management, and well-understood, straightforward projects.

A little known, but major productivity driver is application size. This is true regardless of the measure used; ratio based productivity measures like SLOC or Function Points per effort unit exhibit the same relationship to project size as QSM's productivity index (Armel) (which reflects not only size and effort but time to market as well).

The bar chart at Figure 18 shows that productivity peaked in the 1995-2000 time bucket and has decreased since then. We noted in the 2006 QSM Almanac that productivity in the 1995-2000 time period seemed artificially high due to Y2K projects that contained less original design work or new algorithms. We also noticed that projects in the 2000-2005 time period had a lower average productivity than projects in the 1990s. The downward trend in productivity we observed in 2006 has continued in the latest time period, though the effect is very slight (about half a PI).

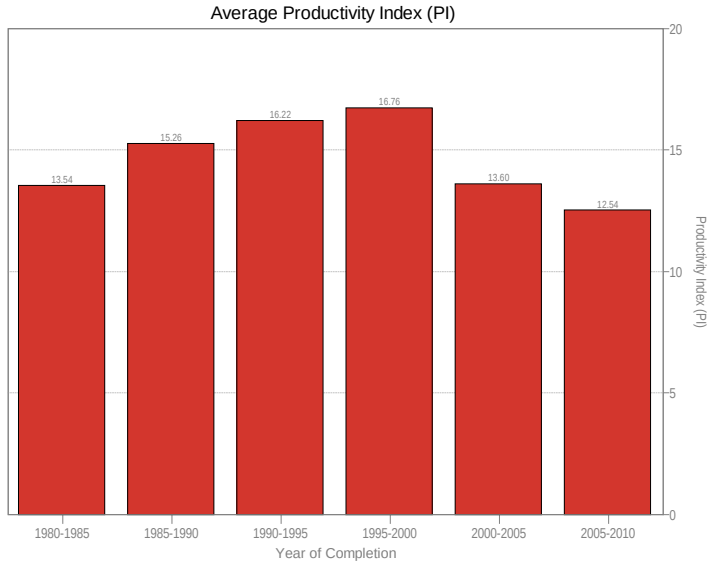


Figure 18. Average Productivity Index over Time

The scatter plot at Figure 19, below, shows productivity for individual projects in the sample over time.

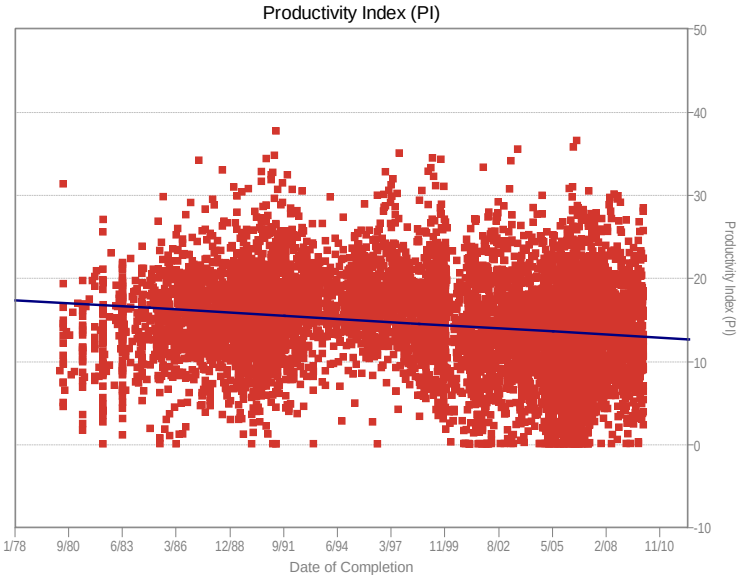


Figure 19. Productivity over Time Scatter Plot with Average Trend line

Note the increase in very low PIs starting around the year 2000. This may reflect the dramatic decline in project sizes (remember, productivity increases with project size) or possibly growing numbers of enhancement projects relative to new development.

Conclusions

In reviewing changes to project size, schedule, effort, and productivity over the past three decades, we thought it would be interesting to look back to the conclusions of our 2006 IT Almanac study to see how well our predictions and observations have held up:

	2006	2014
SIZE	From the early 1980s to 1997, the average size of software projects was cut by more than half. The reduction in median project size was even more dramatic: more than a 75% reduction in developed size. Developers are writing less new and modified code per release. We attribute this to more powerful languages and higher degrees of reuse.	We are still seeing a trend toward smaller projects. In fact, the median new and modified code delivered in the early 1980s is four times larger than the median for projects completed after 2000. We think that projects are smaller due to new lifecycle methodologies, such as Agile, which promote allocating smaller groups of features to a predefined time box, sprint, or iteration. We also think that programming languages are becoming more powerful, which allows each line of code to create more functionality.
SCOPE	A major shift in the way projects are managed is in the basic view of "what is a project"? It's entirely possible that *systems*, per se, aren't shrinking as fast as industry software size metrics would indicate. It may be, instead, that *it is the view of what makes up a project: a manageable unit of work that is tracked and charged as a separate entity* that has gotten smaller. In a very real sense, we may just be working smarter, not harder: breaking large, complicated, unmanageable systems into smaller, less complex, easier-to-handle projects that humans can manage and adapt to in an increasingly technical environment.	The definition of a project is still evolving. Since 2006, we've seen an uptick in Agile and RUP projects, which attempt to manage scope creep by allotting smaller groups of features to predefined time boxes, sprints, or iterations.

REUSE	Average reuse for the industry as a whole peaked at 65% in the mid-nineties and then declined to 50% in the early 2000s. Our research suggests that 60-70% reuse appears to be the practical upper limit for the average project portfolio. It appears to be easier for most projects to affect reuse on the small and large ends of the scale; i.e., when unmodified code is under 20% or over 70%. Code reuse has complexity and productivity implications. Developers don't get "credit" for unmodified code, which must still be tested and integrated with new and modified code. Reused code definitely has an impact on the project schedule and effort profile and on project productivity as well. It appears that reuse, along with the advent of complex n-tier client server applications and object-oriented programming, must be considered as another complexity factor which impacts productivity.	The percentage of reused code has continued to decline to about 35% for projects completed between 2005-2010. Integrating existing frameworks and legacy code with newly written code is a significant complexity factor that can have dramatically different effects on project productivity.
LANGUAGE	Programming language does not appear to be a strong influence on overall project productivity, but it may simply be that other factors are more important or that we cannot isolate the effects of language. Many projects these days are developed in multiple languages, making it difficult to gauge the effect of a single one. COBOL, Java, and Visual Basic remained the most popular development languages in 2004, as they did in 2001.	We expected to find increasing numbers of projects using multiple vs. single languages, but here the data surprised us. We suspect that the growing prevalence of enhancements to older systems (as opposed to new development) may be inflating the proportion of single language projects that come into the database.

Both our 2006 and 2010 studies showed steady improvement in most IT effectiveness measures over time. Projects have continued to become more compact, expend fewer resources, and take less time to complete even as they become more complex and the teams building them grow more diverse and distributed. Is there some natural limit, beyond which further attempts to reduce the size, effort, and schedules of what we call a project

become counterproductive? We're looking forward to seeing how these trends change (or stay the same) as we get more data.

Works Cited

Armel, Kate. "The Size-Productivity Paradox, Part I." *QSM Blog.* 29 March 2010. Web. <http://www.qsm.com/blog/2010/size-productivity-paradox-part-i>.

Costantini, Katie. "Top 25 Programming Languages since 2008." *QSM Blog.* 1 November 2012. Web. <http://www.qsm.com/blog/2012/top-25-programming-languages-2008>.

International Function Point Users Group. "About SNAP." *About IFPUG*. 2014. Web.

"The QSM Project Database." *QSM, Inc.* 2014. Web. <http://www.qsm.com/resources/qsm-database>.

RESOURCES

"Adapt what is useful, reject what is useless, and add what is specifically your own."

– Bruce Lee, arts instructor, filmmaker, and the founder of Jeet Kune Do

Function Point Table

The QSM Function Point Table provides industry averages, organized by programming language, for the source lines of code required to implement a function point (a unit of software functionality). The table is in its fifth version and, unlike similar industry references, is based on data from software projects that have been successfully completed and deployed.

The data for Release 5.0 came from more than 2,192 recently completed projects sized in function points. This sample included 126 different languages, of which 37 provided enough data to be included in the table. Data for three new languages, Brio, Cognos Impromptu Scripts, and Cross Systems Products (CSP), have been included. For each language, the table provides the average, median, and range (low-to-high values) to provide insight into the variance and central tendency of the data values.

Development teams and software estimators use the QSM table to estimate the level of effort and the corresponding time and budget required to achieve a set of software requirements. This estimation is a critical part of the software development process; QSM data suggests that approximately one-third of completed projects overrun their planned schedules or budgets by at least 20% (based on a 2012 sample of more than 3,300 completed projects in the QSM database that provided overrun or slippage data).

"Overall, the range of gearing factors (minimum and maximum) for each language in the table has grown smaller with each release," said Larry Putnam, Jr., Co-Chief Executive Officer for QSM. "Average and median values for most languages have also decreased since the last update. We attribute these changes to better programming practices and increases in the quality and quantity of data available for analysis. We hope Release 5.0 of our Function Point Languages Table will help us better serve IT project managers, steering them toward more successful and cost-effective IT implementations through our function point consulting services."

Function Point Languages Table, Version 5.0

The QSM Function Points Languages Table contains updated function point language gearing factors for 37 distinct programming languages/technologies. The data supporting release 5.0 was drawn from 2192 recently completed function point projects from the QSM database. The sample included 126 languages, 37 of which had sufficient data to be included in the table.

Release 5 features and observations:

- 3 new technologies added
- 32 gearing factors updated
- The range (minimum/maximum values) for each language has grown smaller with each release of the table. This trend continued with release 5.
- Average and median values for most languages have decreased since the last update.

Environmental factors can result in significant variation in the source statements per function point. For this reason, QSM recommends that organizations collect both code counts and final function point counts for completed software projects and incorporate this data into project estimates. Where there is no completed project data available for estimation, we provide the following industry gearing factor information (where sufficient project data exists):

- Average
- Median
- Range (low - high)

These three measures should allow software estimators to assess the amount of variation, the central tendency, and any skew to the distribution of gearing factors for each language.

See http://www.qsm.com/resources/function-point-languages-table#MoreInfo for additional information on gearing factors and recommendations on using this table.

Go to http://www.qsm.com/resources/qsm-sme-contact-information to request gearing factors for languages not found in the table.

* Languages with updated gearing factors.
+ New languages for which gearing factor data was not previously reported.

Language	QSM SLOC/FP Data			
	Avg	Median	Low	High
ABAP (SAP) *	28	18	16	60
ASP*	51	54	15	69
Assembler *	119	98	25	320
Brio +	14	14	13	16
C *	97	99	39	333
C++ *	50	53	25	80
C# *	54	59	29	70
COBOL *	61	55	23	297
Cognos Impromptu Scripts +	47	42	30	100
Cross System Products (CSP) +	20	18	10	38
Cool:Gen/IEF *	32	24	10	82
Datastage	71	65	31	157
Excel *	209	191	131	315
Focus *	43	45	45	45
FoxPro	36	35	34	38
HTML *	34	40	14	48
J2EE *	46	49	15	67
Java *	53	53	14	134
JavaScript *	47	53	31	63
JCL *	62	48	25	221
LINC II	29	30	22	38
Lotus Notes *	23	21	19	40
Natural *	40	34	34	53
.NET *	57	60	53	60
Oracle *	37	40	17	60
PACBASE *	35	32	22	60
Perl *	24	15	15	60
PL/1 *	64	80	16	80
PL/SQL *	37	35	13	60
Powerbuilder *	26	28	7	40
REXX *	77	80	50	80
Sabretalk *	70	66	45	109
SAS *	38	37	22	55
Siebel *	59	60	51	60
SLOGAN *	75	75	74	75
SQL *	21	21	13	37
VB.NET *	52	60	26	60
Visual Basic *	42	44	20	60

Table 1. Gearing Factor Table

More Information on Using Gearing Factors

What is a gearing factor? The gearing factor is simply the average number of new plus modified (Effective) Source Lines of Code per function point in the completed project. Gearing factors are calculated by dividing the effective code count for a completed project by the final function point count. SLOC counts represent logical, not physical line counts.

What if the language I am using is not in the table? If you do not see the language you need in the table, you may substitute a gearing factor from a comparable language. The uncertainty range for the estimated gearing factor may be increased to allow for any additional risk introduced by using a substitute. You may also contact QSM to see if revised information is available.

Should I use the average or the median? In a perfectly symmetrical distribution of gearing factors, the average and the median will be identical or very close. The average is obtained by summing the gearing factors and then dividing by the number of gearing factors included in that sum. Although its purpose is to measure "central tendency," the average can be pulled up or down by extreme data values (or outliers). The median, on the other hand, is simply the data point that lies in the center of an ordered list of gearing factors. One half of the data points will lie above (and one half below) the median. When the data set is skewed (biased either toward the high or low end by extreme data values), the median may be a more accurate indicator of the central tendency.

How should I use the range? The range simply shows lowest and highest gearing factors for each language. The range can be combined with the average and median, to choose a "most likely" gearing factor for estimation. The range can be useful as a starting point for choosing an uncertainty range around your "most likely" estimate of the gearing factor.

Where does the data come from? The gearing factors in this table were drawn from 2192 recently completed function point projects in the QSM database. As mixed-language projects are not a reliable source of gearing factors, only single-language projects are used.

Performance Benchmark Tables

The QSM Benchmark Tables provide a high-level reference for benchmarking and estimating IT, Engineering, and Real-time Systems. They display industry average duration, effort, staff, and SLOC (or FP) per Person Month for the full range of project sizes encompassed by each trend group.

The results were analyzed from a database of 1,115 high or moderate confidence projects completed between 2008 and 2012. Sixteen countries and 52 different languages were represented in this sample. In addition to the industry average, minimum and maximum values were also provided for each metric to help give a range of possible results.

The project sizes differed somewhat from the previous version to accommodate the new range of sizes present in the data. Rather than using the same project sizes across trend groups, we selected project sizes specific to each trend. Since Business projects are typically smaller than Engineering or Real-time projects, this allows readers to select a size relevant to the type of project they're estimating or benchmarking.

This tool can be particularly useful to developers and/ or project managers who are new to estimation or do not have historical project data.

"We wanted to give people an opportunity to get a quick and dirty comparison for productivity and schedule performance," said Doug Putnam, Co-Chief Executive Officer for QSM. "We think it can be a really good resource for early project negotiations to identify wildly unrealistic expectations."

While the Performance Benchmark Tables are provided free of charge, QSM also provides custom benchmark services to give a more in-depth analysis of your organization's projects and how they compare to the industry.

Average Project Performance

The following reference tables are updated versions of those produced in 2009, which provide summary performance data (schedule, effort, staff, and SLOC/PM) for typical projects from QSM's Business, Engineering, and Real-time software databases. This

information provides a high level, quick reference for benchmarking both completed projects and software estimates.

All data supplied in these tables comes from QSM's historical database of over 10,000 completed projects. The QSM database is a cornerstone asset utilized in all of our consulting service engagements. It represents the largest and most complete set of validated and completed software project data in the world.

- Business Systems (Source Lines of Code)
- Business Systems (Function Points)
- Engineering Systems
- Time Systems
- Measures

Business Systems (Source Lines of Code benchmarks)

Size: New & Modified SLOC	Duration (Months)	Effort (PM)	Average Staff (FTE)	SLOC/PM
2,500	5.9	11.3	1.9	430.0
10,000	7.1	26.1	3.5	650.0
25,000	8.5	45.8	5.2	876.0
50,000	9.5	70.0	7.2	1,088.0
100,000	10.4	102.7	9.3	1,300.0
Min: 600	*1.6*	*1.63*	*0.9*	*38.5*
Max: 7,920,000	*42.2*	*2,219.65*	*202.8*	*12,403.4*

Table 1. Business Systems (Source Lines of Code Benchmarks)

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

Business Systems: Function Point Benchmarks

Size: FP	Duration (Months)	Effort (PM)	Average Staff (FTE)	FP/PM
50	5.7	11.9	2.0	7.1
100	6.4	17.9	2.7	8.6
250	7.6	31.6	4.1	10.8
500	8.5	46.4	5.6	13.1
1,000	9.0	71.0	7.4	15.5
Min: 10	*1.6*	*1.6*	*0.4*	*1.1*
Max: 5,000	*42.2*	*1,705.0*	*121.4*	*234.0*

Table 2. Business Systems: Function Point Benchmarks

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

Engineering Systems

Size: New & Modified SLOC	Duration (Months)	Effort (PM)	Average Staff (FTE)	SLOC/PM
2,500	6.7	22.0	3.2	192.2
10,000	9.7	53.0	5.4	294.5
25,000	12.0	92.4	7.1	394.0
50,000	14.2	143.0	9.3	497.3
100,000	16.8	225.7	12.2	621.0
300,000	23.8	453.4	19.3	887.7
Min: 32	*1.8*	*0.83*	*< 1*	*7.1*
Max: 2,573,612	*55.0*	*10,037.00*	*339.6*	*13,514.9*

Table 3. Engineering Systems

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

Real-time Systems

Size: New & Modified Code	Duration (Months)	Effort (PM)	Average Staff (FTE)	SLOC/PM
2,500	9.6	12.30	1.4	211.8
10,000	13.6	55.1	4.0	223.0
25,000	17.2	143.0	8.3	244.1
50,000	20.6	281.0	14.3	250.5
100,000	25.1	596.9	23.6	259.3
300,000	33.7	1,850.3	54.4	274.3
Min: 344	*4.5*	*2.04*	*< 1*	*21.0*
Max: 2,141,000	*94.1*	*43,221.28*	*760.9*	*4,598.7*

Table 4. Real-time Systems

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

Measures:

- Schedule: elapsed time (in months) from Requirements Determination (Phase 2) through the Initial Release (end of Phase 3)
  - Schedule = (P2 Duration + P3 Duration) – P2 Overlap
- Effort: the number of Person Months expended during Requirements Determination (Phase 2) and Construct & Test (Phase 3)
  - Effort = P2 PM + P3 PM
- Average Staff: the number of Full Time Equivalent employees for Phases 2 - 3
  - Average Staff = (P2 + P3 Effort)/ (P2+ P3 Duration)
- SLOC/ PM: the number of Source Lines of Code produced per Person Month of effort during Phase 3
- FP/ PM: the number of Function Points produced per Person Month during Phase 3

INDEX

A

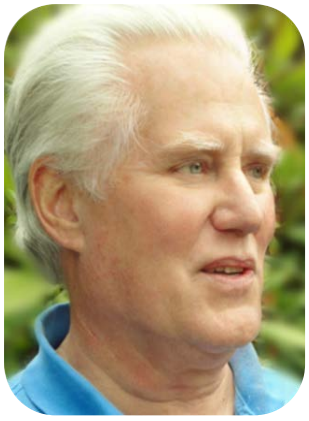
B

C

Return to Table of Contents

CONTRIBUTING AUTHORS

Kate Armel is the Director of Research & Technical Support at QSM. She has 15 years of experience providing technical and consultative support in the areas of software estimation, project tracking and forecasting, and industry benchmarking. She oversees collection, validation, and analysis of completed project data for the QSM database; development of over 900 industry regression trends; QSM client, internal, and technical support services; software testing and quality assurance; documentation and online help for SLIM-Suite®, SLIM-WebServices® applications, APIs, and utilities; and technical writing, research, and analysis to support QSM product development, research, and consulting services. Ms. Armel was the Chief Editor and analyst/co-author of the 2006 QSM IT Software Almanac, and has authored several published articles.



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, function point analysis, and has also 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 US Patent.

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, a Ph.D. in Mathematics from the University of Wisconsin, Madison, and has seven US Patents.

Katie Costantini worked her way up from Summer Intern to Testing Manager. She graduated from Virginia Commonwealth University *cum laude* with a Bachelor of Science degree in Economics and a minor in Latin and Roman Studies. Katie handles database queries, database validation, trend line creation, and helps with documentation for SLIM-Suite® and SLIM-WebServices®. She is a Certified Tester, Foundation Level, through the American Software Testing Qualifications Board, and is responsible for test case design, tracking, and management for SLIM-Suite® and SLIM-WebServices®.

Carol Dekkers, PMP, CFPS, P.Eng (Canada) is a consultant, author and speaker at international conferences. 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.) 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.

Larry H. Putnam, Jr., has 27 years of experience using the Putnam-SLIM® methodology. He has participated in hundreds of estimation and oversight service engagements, and is responsible for product management of the SLIM-Suite® of measurement tools and customer care programs. Since becoming Co-CEO, Larry has built QSM's capabilities in sales, customer support, product requirements, and, most recently, in the creation of a world class consulting organization. He has been instrumental in getting QSM product integrations validated as "Ready for IBM Rational" as an IBM Business partner. Larry has delivered numerous speeches at conferences on software estimation and measurement, and has trained more than 1,000 software professionals on industry best practice measurement, estimation and control techniques, and the use of the QSM SLIM® tools and methods.

Taylor Putnam is a Consulting Analyst at QSM and has over seven 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 about 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 holds a bachelor's degree from Dickinson College.

Return to Table of Contents