Data-driven Estimation, Management Lead to High Quality

KATE ARMEL
Quantitative Software Management, Inc.

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.

Key words

defects, estimation accuracy, MTTD, productivity, quality assurance, reliability, risk management, team sizes, uncertainty

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 2009). 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 2002). 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 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:

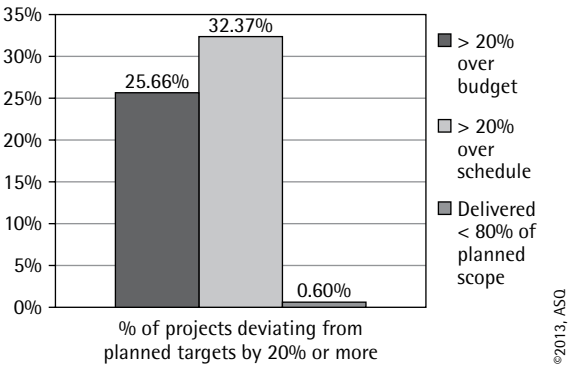
- Capture and use "failure" metrics to improve future estimates rather than punishing estimators and teams
- Keep team sizes small
- Study their best performers to identify best practices
- Choose practical defect metrics and models
- Match reliability targets to the mission profile

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 (Standish 2009). A more current study of recently completed software projects (Beckett 2013) points to one problem: While nearly all of the projects in the QSM database report actual schedule, effort, and size

FIGURE 1 Project failure



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.

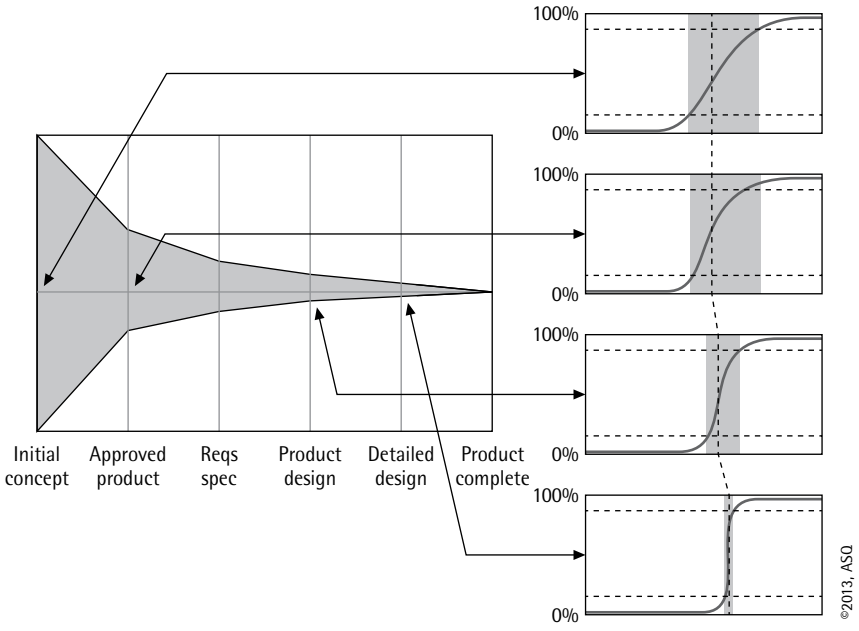
It's surprising—and a bit disconcerting—to see so many projects failing to meet their goals. These are companies that have formal metrics programs in place. They have invested in state-of-the-art tools, processes, and training. So why aren't they using project actuals to improve the accuracy of future 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 2008).

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.

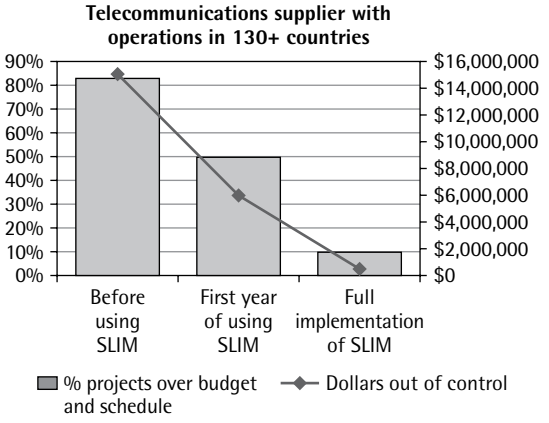
Does that mean one shouldn't track overruns and slippages at all? Absolutely not. In fact, it's vital that projects capture deviations between estimated and actual project outcomes *because this information allows them to quantify a crucial estimation input (risk) and account for it explicitly in future 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

FIGURE 3 Case study source: *Five Core Metrics: The Intelligence Behind Successful Software Management*



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 2003)

> "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."

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 2012). 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.

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 Figure 5 (Armel 2012).

The relative magnitude of the tradeoffs between team size and schedule, effort, and quality is easily visible by inspection of Figure 6. Large teams achieve only modest schedule compression (note the large overlap between large and small team projects in the top left 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.

FIGURE 4 Regression fits for average staff vs. system size (large and small team samples)

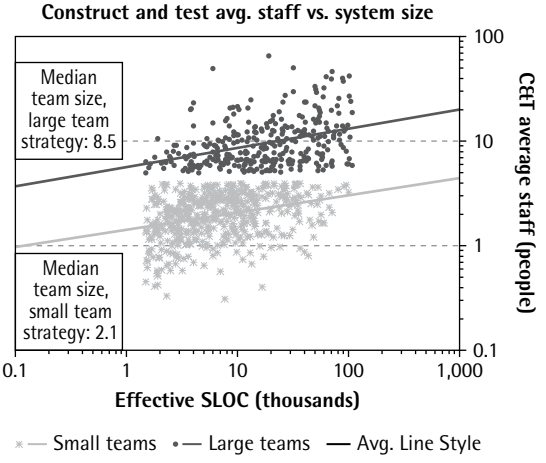
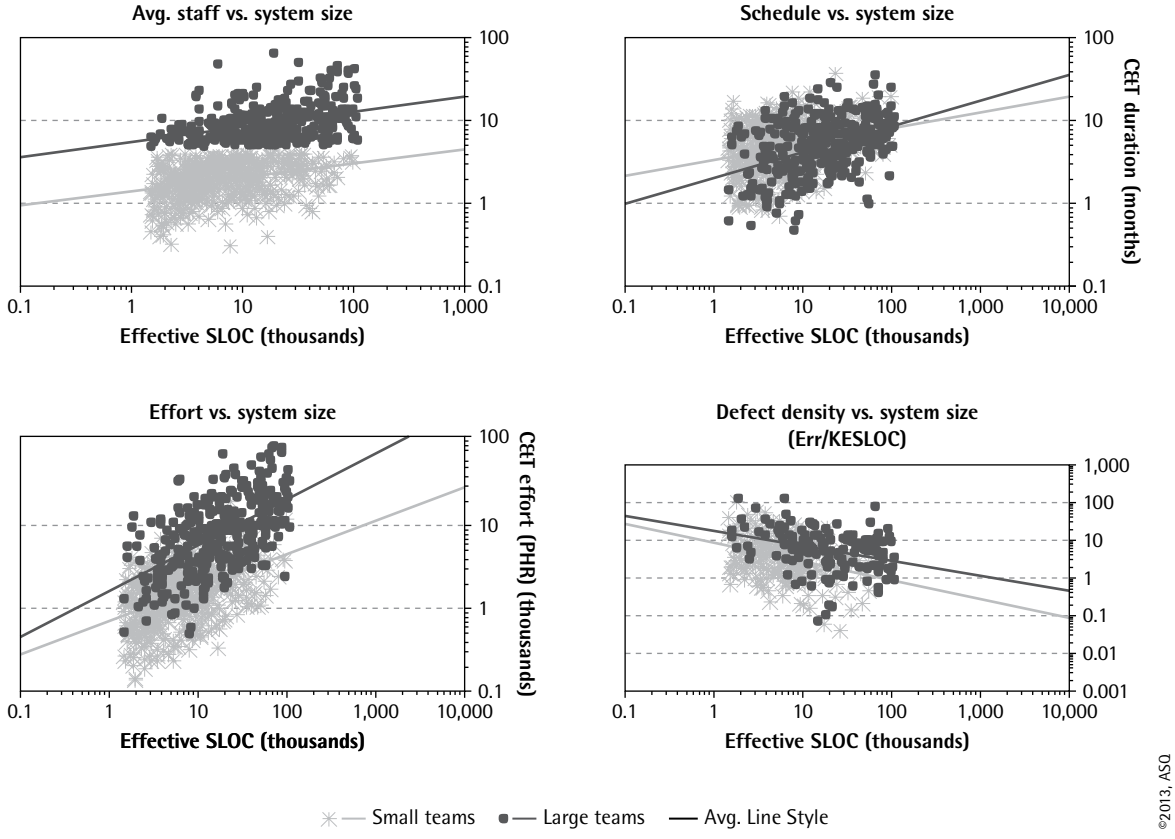


FIGURE 5 The large team strategy increased effort expenditure (334-441%) and defect creation (249-325%) while yielding only marginal reductions to schedule

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%*

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

FIGURE 6 Using larger teams had little impact on schedule but a large impact on effort expenditure and defect density



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 2006)

> "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 7 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

FIGURE 7 Worst-in-class projects used an average of 4.25 as many staff at the same project size as best-in-class projects

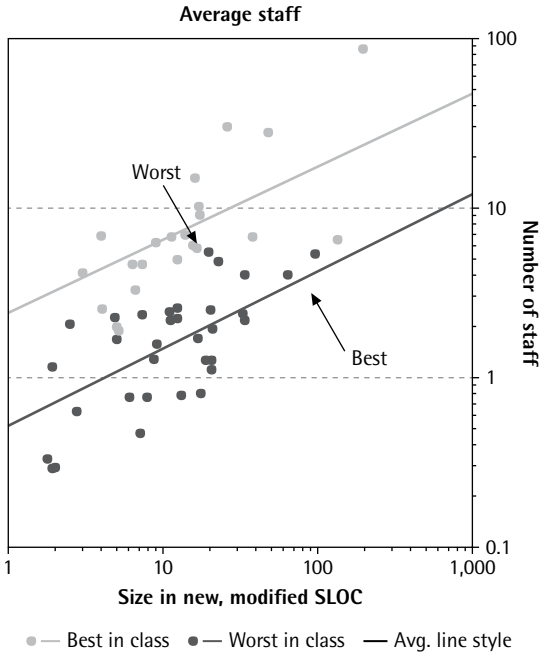
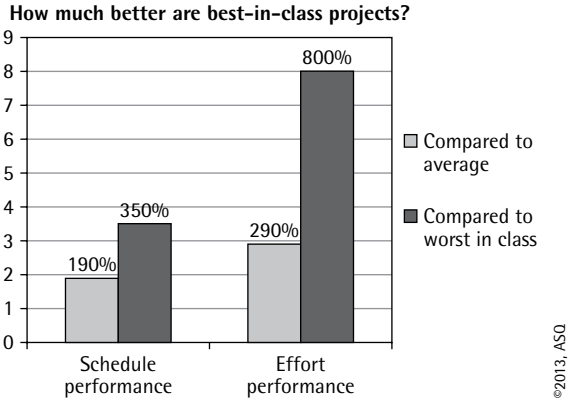


FIGURE 8 Performance of best-in-class projects against average and worst-in-class projects



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 the chart 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!"

How much better did best-in-class projects perform against both average and the worst performers? The differences, shown in Figure 8, are striking.

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 predelivery 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 2006). 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 2013)

"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 1992) 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) (see Figure 9).

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.

FIGURE 9 Defect rates over time follow a Rayleigh distribution

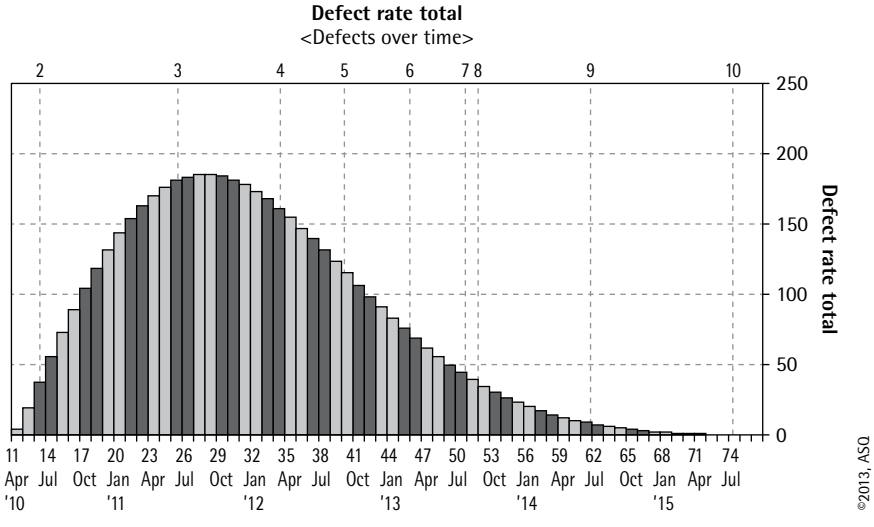
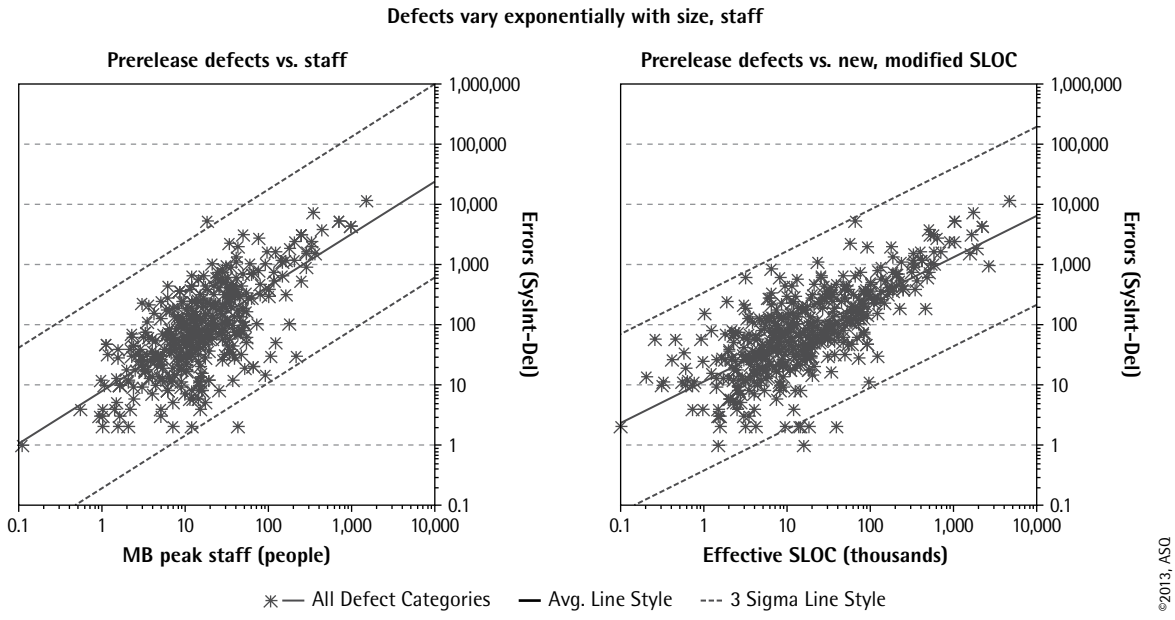


FIGURE 10 Defects increase exponentially with FTE staff size and system size



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 2006)

> "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).

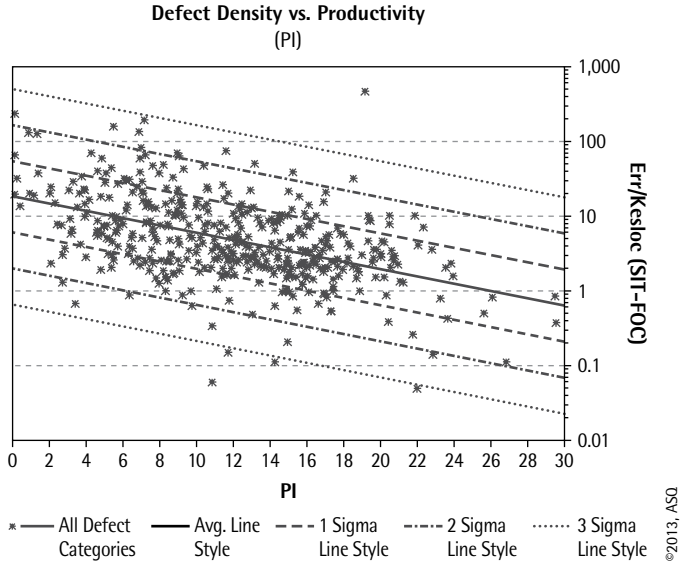
Though software development does not produce the identical widgets described in the manufacturing example, even software projects share certain characteristics and behaviors. Over the past three decades

these similarities have proven useful in managing and improving the software development process. Larry Putnam's (Putnam and Myers 1992) 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 10 shows, defects vary exponentially as staffing and code volume increase. 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 11 Defect density declines as project productivity increases



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

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.

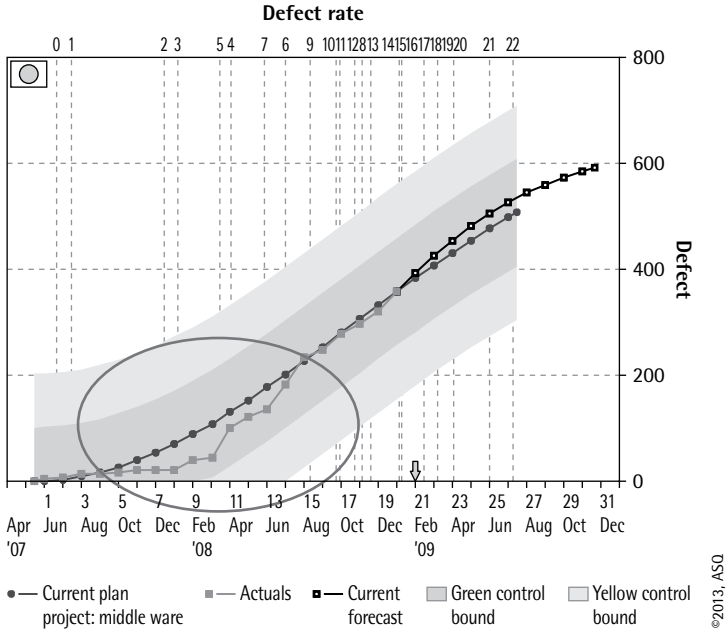
CHOOSING EFFECTIVE DEFECT METRICS

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 12 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.

FIGURE 12 Prior to system integration and test, actual defect data is often erratic, displaying jagged "find and fix" cycles. As the project nears delivery, defect discovery typically settles down and tracks the smooth Rayleigh curve closely.



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.

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.

FIGURE 13 The appropriate defect rate depends on the desired mission profile

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

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 Figure 13, 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 table 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.

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.

REFERENCES

Armel, K. 2012. History is the key to estimation success. *Journal of Software Technology* 15, no. 1 (February):16-22.

Armour, P. G. 2008. The inaccurate conception. *Communications of the ACM* 51, no. 3 (March): 13-16.

Beckett, D. 2013. Engineering best and worst in class systems. QSM. Available at: http://www.qsm.com/resources/research/research-articles-papers/.

NIST. 2002. The economic impacts of inadequate infrastructure for software testing. National Institute of Standards & Technology, May, 36.

Putnam, L., and W. Myers. 2003. *Five core metrics: The intelligence behind successful software management.* New York: Dorset House Publishing.

Putnam, L., and W. Myers. 1992. *Measures for excellence: Reliable software on time, within budget.* Upper Saddle River, NJ: P.T.R. Prentice-Hall, Inc.

QSM Software Almanac. 2006. 2006 IT Metrics Edition, Quantitative Software Management, Inc.

Seapine Software. 2009. Identifying the cost of poor quality. Seapine Software. Available at: http://downloads.seapine.com/pub/papers/CostPoorQuality.pdf.

The Standish Group. 2009. New Standish Group report shows more project failing and less successful projects. The Standish Group. Available at: http://www1.standishgroup.com/newsroom/chaos_2009.php.

BIOGRAPHY

Kate Armel is the director of research and technical support at Quantitative Software Management, Inc. She has 13 years of experience in technical writing and metrics research and analysis and provides technical and consultative support for Fortune 1000 firms in the areas of software estimation, tracking, and benchmarking. Armel was the chief editor and a researcher and co-author of the QSM Software Almanac. She also manages the QSM database of more than 10,000 completed software projects. Armel can be reached by email at kate.armel@qsm.com.