Int J Software Informatics, Volume 5, Issue 3 (2011), pp. 415–433
International Journal of Software and Informatics, ISSN 1673-7288

E-mail: ijsi@iscas.ac.cn
http://www.ijsi.org
Tel: +86-10-62661040

Measuring Agility and Architectural Integrity

Walker Royce

(Chief Software Economist, IBM, Boston, Massachusetts, USA)

Abstract Most organizations that depend on software are pursuing more flexible architectures and more agile life-cycle processes to increase business flexibility. What does agility look like, and how do we measure it? A truly agile project or organization should experience changes that are more straightforward and more predictable. Consequently, improvements are best measured by gauging the change trends in software baselines. A well-accepted tenet of software engineering states, "The later you are in the life cycle, the more expensive things are to fix." This iron law, an artifact of a waterfall culture, should not apply if you have transformed to agile software delivery with a well-architected system. This bold assertion is the root of the metric patterns presented in this paper.

Key words: agility; architecture; earned value; econometrics; honest communications; integration; measured improvement; metrics; steering; software management; software economics; trust

Royce W. Measuring agility and architectural integrity. *Int J Software Informatics*, **Vol.5, No.3 (2011): 415–433. http://www.ijsi.org/1673-7288/5/i92.htm**

Three recurring themes are bubbling to the top of business priorities in most organizations that derive value from their software delivery capability: integration, agility, and measured improvement.

The integration of systems, products, applications, and organizations provides most of the differentiated value in today's competitive information marketplace. Integration challenges also represent the primary sources of uncertainty, complexity, and cost of developing and maintaining systems. In the systems and software development life cycle, resolving the significant uncertainties early through continuous integration is a well-established best practice that improves economic outcomes. Yet most organizations still address the easy, unambiguous activities first to show early progress. To realize breakthrough economic gains, integration must take on a higher priority. A natural extrapolation of this principle is that integration testing should precede unit testing. This counterintuitive statement cuts against the grain of conventional software delivery culture. Most "mature" projects and organizations work in reverse: Unit testing precedes integration testing. Not only is this sort of conventional, waterfall-model thinking mature; in many cases it is geriatric.

Integration testing activities ensure that components, services, and data interoperate properly so that a system's functional behaviors can be assessed in an executable configuration. In practice, integration has proven to be the most enlightening activity in exposing the architecturally significant risks in software development. Resolving

Corresponding author: Walker Royce, Email: weroyce@us.ibm.com
Paper received on 2011-01-21; Revised paper received on 2011-03-09; Paper accepted on 2011-03-10; Published online 2011-03-17.

requirements and architectural uncertainties earlier is paramount to reducing the burdens of the typical late-in-the-life-cycle rework that stifles agility. Performing integration testing prior to investing in unit test completion will improve agility by addressing the far-reaching, potentially malignant, changes earlier. Resolving the more benign, finer-grained adjustments exposed in unit testing is necessary, but these activities overemphasize completeness and coverage measurements that should be secondary concerns until the system composition challenges have been largely resolved. In more blunt terms: Don't address precision in the components until you have achieved accuracy in the architecture (composition, relationships, and behavior of components).

Measured improvement is a key theme. Why is measurement so important? What should we measure and why? Six metric patterns – three progress perspectives and three quality perspectives – are proposed to measure software outcomes. These six patterns illustrate the improvement trends possible by transforming from conventional engineering governance to economic governance. This transformation is enabled by a "steering" style of leadership focused on measured improvement. Some of the metric patterns are unorthodox and are best understood when examined from an integration-first perspective. The resulting change trends show how breakthrough agility can result in significant business flexibility by allowing more change freedom late in the life cycle and post-delivery.

When changes are easy to implement, a project is more likely to increase the number of changes, thereby increasing quality. Organizations and projects can balance their resources used on defensive necessities (such as defect resolution and meeting schedule commitments on feature content) with offensive thrusts (such as new integrations, new innovations, improved performance, earlier releases, and higher quality). More time playing offense offers opportunities for improving economic outcomes and market differentiation.

1 The Importance of Measurement

Software professionals work in a domain with a high degree of uncertainty and complexity. So our measurement challenges are severe, but therein is an opportunity for competitive advantage. How does a systems and software development organization make improved productivity claims more credible and demonstrable, thereby earning more trust among their stakeholders?

As more and more businesses differentiate their products, systems, and services through their software delivery capability, more attention to software economics becomes vital. Measured improvement is a best practice for improving software economics. This applies to teams, projects, and organizations. In 1981, Barry Boehm laid the foundations for decades of work on improving the economics of software development[1,2]. Since then, software economics has evolved in many directions but has been mostly absorbed in process transformations to new methods[3−6]. Measured improvement involves four basic steps:

1. Define the current capability: the as-is measures.

2. Propose a target capability improvement: the to-be measures.

3. Develop a roadmap to get from the current capability to the target capability with measurable, incremental feedback at key checkpoints along the route.

4. Progress toward the targets while steering with real-time gauges of measured

progress and quality to adjust and balance the win conditions of all stakeholders.

Scientists define measurement as an observation that reduces uncertainty, where the result is expressed as a quantity[7]. We use measurement to advance our understanding and reduce uncertainties. Any significant reduction in uncertainty is enough to make a measurement valuable. One of the recurring attributes of most best practices for software delivery is that they reduce uncertainties earlier in the life cycle and thereby increase the probability of success, even if success is defined as cancelling a project earlier so that wasted cost is minimized.

Reducing uncertainty by measuring trends increases trust among stakeholders. Without quantified backup data, our software estimates, proposals, and plans look like long-shot propositions with no compelling evidence that we can deliver predictably or improve the status quo. Trust is earned when we combine integrity and competence. Measurements have integrity when there is an accepted basis of theory and practice. They exude competence when we demonstrate a track record of benchmarks (scales for judging appropriateness) and capture our experience in precedent results, references, and experience in the field. Since our industry does not have enough accepted theory and practice, we all suffer from an environment of distrust and skepticism surrounding our measured improvement claims.

I use the term *software econometrics* as the measurement foundation underlying improvements in software delivery productivity. The *econo* prefix has the right connotation for reinforcing these as measures for modern economic governance as opposed to traditional engineering governance and corresponding measures. The econometric patterns asserted here are grounded in successful practice that I have observed across some of the industry's most successful projects. They have been demonstrated in some larger scale, industrial-strength applications, as well as numerous projects at smaller scale. However, we still lack adequate empirical evidence and well-documented case studies. My hope is that this paper will provide some target patterns for others to validate or challenge.

Improving measurement discipline is important because it correlates strongly with better business performance. The data in Figure 1, compiled from hundreds of software organizations by Capers Jones[4], provides a compelling basis for three observations:

1. The difference between projects and organizations with strong measurement practices and those with weak practices is impressive. Measurement discipline enables more trustworthy communications among stakeholders. The value of that trust can only be quantified coarsely, but the impact is eye-opening.

2. The return on investment (ROI) for measurement is significant in the near term and improves over time.

3. Three of the top five reasons for legal proceedings in software are directly related to poor measurement practices.

2 Software Metrics History

Measurements in most software organizations are more like the sleight-of-hand statistics quoted by politicians than the matter-of-fact statistics quoted by engineers and scientists. Capers Jones and I agree on this harsh assessment. He says, "Software has perhaps the worst measurement practices of any 'engineering' field in human

Summary outcomes for projects with strong and weak measurement practices:

	Strong	Weak
On-time projects	75%	45%
Late projects	20%	40%
Cancelled projects	5%	15%
Defect removal	95%	< 85%
Resource estimates	Accurate	Optimistic
Client satisfaction	Higher	Lower
Staff morale	Higher	Lower

Fortune 500 firms with:	
Productivity measures	30%
Quality measures	45%
Complete measures	15%
Number of projects measured:	160,000
Number of projects not measured:	50,000,000

ROI for software measurement:

	Cost	Return
Year 1	5%	$ 4.50
Year 2	4%	$ 6.25
Year 3	4%	$ 8.75
Year 4	4%	$ 11.50
Year 5	3%	$ 15.00

Top reasons for software litigation:

1. Unstable, changing requirements	95%
2. Inadequate quality control measures	90%
3. Inadequate progress tracking	85%
4. Inadequate cost and schedule estimating	80%
5. False promises by sales personnel	80%
6. Optimistic schedule estimates	75%
7. Informal, unstructured development	70%
8. Poorly articulated requirements	60%
9. Inexperienced project managers	50%
10. Inadequate quality assurance tools	55%

Figure 1. How important is measurement?

history." Politicians and the software industry have similar track records for underdelivering on commitments. Most stakeholders in the software business are justifiably cynical because their experience with software productivity improvement is plagued by hyperbole and spin. Although many conventional software metrics approaches, such as earned value management (EVM), are well-founded, they are typically implemented in a way that is (unintentionally) dishonest. This statement may seem hyperbolic, but decades of first-hand experience across hundreds of projects have led me to this provocative conclusion.

A brief review of past measurement approaches, looking at three genres[1] of software governance, gives context for why transformations to better measurement are needed.

1. Engineering governance. The waterfall model[8] is still practiced by the majority of software development teams because it is the predominant legacy culture of traditional engineering governance. Waterfall management is simple, but it is overly simplistic for software where uncertainties dominate a project's timeline.

2. Hybrid governance. About 20 to 30 percent of system and software development teams practice iterative development techniques[5], where architectures are constructed first and evolved through a sequence of executable releases. Although iterative development is more complex to manage, therefore requiring more project management savvy, it succeeds more frequently.

3. Economic governance. Perhaps 10 to 20 percent of industry teams practice agile or lean software delivery techniques[9], where project teams focus on early reduction of uncertainty, continuous integration, asset-based development, increased stakeholder interaction, and smarter, collaborative environments. Management savvy,

[1] While "genre" might seem like the wrong word, it is used intentionally to signify that software is much more of a collaborative creation than engineering production.

combined with meaningful measurement and instrumentation, results in higher levels of agility and business flexibility.

Consider the levels of uncertainty shown in Figure 2. Most engineering disciplines have evolved into the lower three levels, where uncertainty is relatively manageable. High-uncertainty engineering efforts are still attempted, such as capping the Gulf of Mexico oil leak in 2010, but these are the exceptions. Software delivery, by comparison, is a discipline dominated by human creativity, market forecasting, value judgments, and uncertainty levels commensurate with other economic endeavors like movie production and venture capital management. Some of the software development life cycle can be managed like engineering endeavors, but most software projects include higher levels of uncertainty.

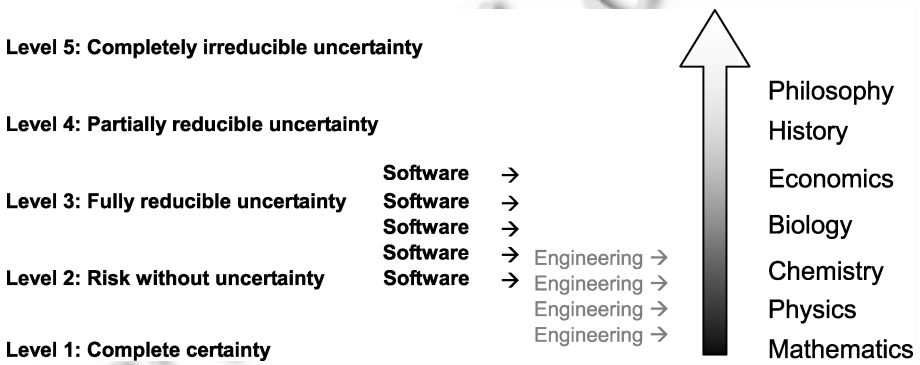
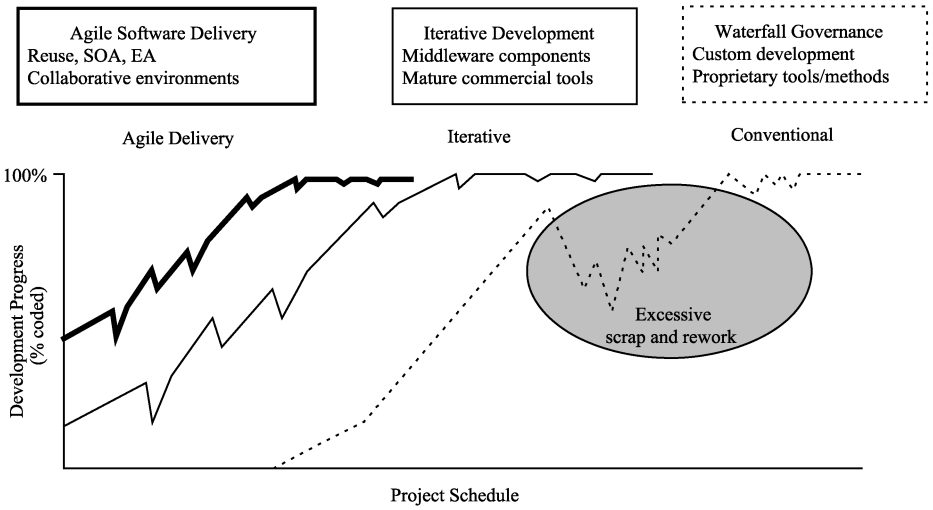


Figure 2. Lo and Mueller's taxonomy of uncertainty[10]

Conventional engineering project management techniques assume little uncertainty in their requirements and exploit mature precedents for production and deployment. Software delivery is much more of a creative economic discipline, akin to making movies[11,12]. Engineering discipline has its place, but software governance techniques must steer through far greater levels of uncertainty to deliver better economic outcomes. For decades, the software industry has been marching toward better process models for attacking uncertainty. Barry Boehm's introduction of the spiral model[13] in 1988 shone one of the brightest spotlights on the importance of managing uncertainties better. This theme has dominated the software industry's evolution of best practices ever since.

Figure 3 shows a project manager's view of the process transition that the industry has been marching toward for decades. Project profiles representing each of the three genres plot development progress versus time, where progress is defined as *percent executable*, that is, demonstrable in its target form. Progress correlates to tangible intermediate outcomes and is measured through executable demonstrations. The term *executable* does not imply that a baseline configuration is complete, compliant, nor up to specifications; it does imply that the software is integrated, testable, and measurable.

The table at the bottom of the figure describes the primary measures used to govern waterfall projects (right column), the measurement framework best suited for transforming to iterative development (middle column), and the six econometrics described later in this paper for measuring agile software delivery (left column).



Agile econometrics	Iterative measures	Conventional measures
Demonstrable user stories	Honest earned value	Inaccurate earned value
Demonstrable code base	Release content	Activity/milestone completion
Variance in estimates to complete	Release quality	Code/test production
Defects and maturity	Prioritized risk lists	Requirements-design-code traceability
Scrap and changeability	Change management trends	Inspection coverage
Rework and adaptability	Actuals vs dynamic plans	Actuals vs static plan

Figure 3. Improved project profiles and measures

Waterfall governance typically results in a project profile with protracted integration activities and excessive late design breakage, as highlighted in the shaded oval of Figure 3[5]. More agile software delivery approaches look like the middle and left-hand profiles in Figure 3. These methods start projects with more existing assets, architectures, components, and services. Modern best practices, combined with a supporting platform, enable earlier integration testing and more effective team collaboration. These two goals are tightly coupled. Effective collaboration is a prerequisite for continuous integration.

Assessing functional integrity is important at all levels of a software system under development. However, ensuring that all parts interoperate holistically – that they integrate into executable, testable configurations – is higher priority than making sure the parts themselves are 100% complete as early as possible. This distinction can be understood most easily in terms of testing. Features, behaviors, usage models, and performance of the integrated system or product must be substantially tested prior to investing in complete unit testing. Otherwise, the project will be exposed to stifling levels of rework.

The riskiest requirements, design issues, and test issues are attacked and resolved first, with the malignant (i.e., architecturally significant) changes addressed earlier, thereby improving agility at each stage in the life cycle. Measurable progress and quality insight are accelerated, and projects can converge on deliverable products

that can be released to users and testers more predictably. Life-cycle scrap and rework is reduced considerably, and architectural integrity is maintained through early refactoring and elimination of late, suboptimal fixes made under budget or schedule duress.

Projects that have transitioned to a more agile "steering" leadership style based on effective measurement can optimize scope, design, and plans, reducing unnecessary scrap and rework. *Steering* implies active management involvement and frequent course corrections to produce better results. Effective steering eliminates uncertainties earlier and significantly improves the probability of win-win outcomes for all stakeholders. Scrap and rework rates are not driven to zero, but to a level that corresponds to healthy discovery, experimentation, and production commensurate with resolving the uncertainty of the product being developed (perhaps 15% of total effort spent in reworking the code and test base, as opposed to the 40% of effort more typical with conventional governance.)

To transform successfully from conventional engineering governance to modern agile governance requires a significant cultural transformation. This is best achieved through the pursuit of one simple change theme: Integration testing should precede unit testing. In practice, this theme is overly simplistic: Integration and unit testing actually proceed in parallel. However, to accelerate the transformation to increased agility, it is best to simplify and clarify that the highest priority is to achieve intermediate milestones of executable test cases of integrated functionality.

This is not a new idea. The principles of iterative development[5], the spiral model[13], risk management[14], and the foundations of modern agile methods[9], such as test-driven development all exemplify a process spirit that emphasizes continuous and early integration. To translate these principles into better outcomes, project teams need to plan their activities and early releases to drive integration testing targets prior to unit testing targets. In the most standout software success stories that I have observed, where software product releases are straightforward to maintain over a long period of time, teams prioritize continuous integration testing of the forest over detailed unit testing of the trees. The two strongest, industrial-strength examples of integration-first results are well documented[5,15]. This integration-first spirit has a natural parallel in managing systems and software development teams: Optimizing collaborative teamwork is more important than optimizing individual productivity.

Keeping this integration-first theme in mind will help increase the agility of the larger project team, no matter what role you play in a software project.

• Project managers. Lay out plans, resources, and measures that prioritize integration testing of key usage scenarios in multiple checkpoints for stakeholder steering.

• Analysts. Analyze the business context or system context to define first the integrated behaviors, qualities, and usage scenarios that represent the holistic value and whose elaboration will reduce the most uncertainty.

• Architects. Elaborate the architecture of the solution and the evaluation criteria for incremental demonstration of the most important behaviors and attributes, such as changeability, performance, integrity, security, usability, and reliability.

• Designers/developers. Develop units, services, and components that are always executable and testable, evolving from initial versions that permit execution within their usage context (that is, satisfy their interface) in a trivial way and then progress

toward more complete components that meet quality expectations across their entire operational spectrum.

• Testers. Identify testing infrastructure, data sets, sequences, harnesses, drivers, and test cases that permit automated regression testing and integration testing to proceed without reliance on completely tested units and components.

Emphasizing integration tests as the intermediate outcomes will lead to more team collaboration across roles and earlier reconciliation of the significant uncertainties. Better measurement across the industry will provide stronger evidence to back up that last assertion. Our industry needs more experimental results and measured case studies.

3 Managing Uncertainty

Successfully delivering software products in a predictable and profitable manner requires a steering leadership style that expects an evolving mixture of discovery, production, and assessment. All stakeholders must collaborate to converge on moving targets and manage uncertainties, as illustrated in Figure 4.

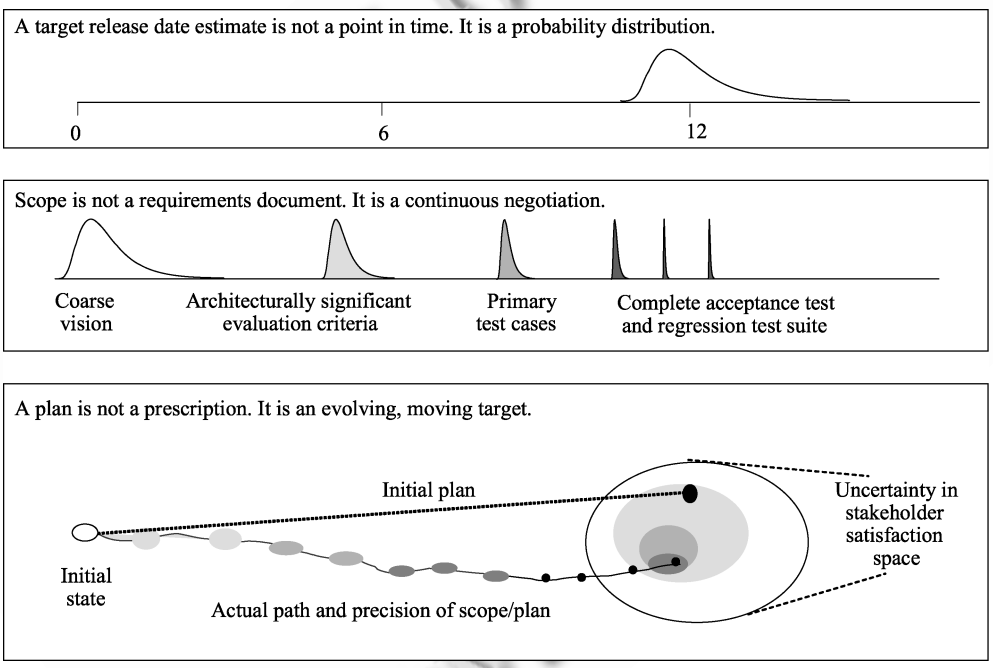


Figure 4. Important steering perspectives to manage uncertainty

Uncertainty is best quantified by measuring the reduction in variance in the distribution of resource estimates to complete. These estimates are random variables and should be represented by their probability distributions, not just the mean values[16]. In a healthy software project, each phase of development produces an increased level of understanding by reducing uncertainty in the evolving plans, specifications, and demonstrable releases. At any point in the life cycle, the precision of the subordinate artifacts, especially the code and test base, should be in balance with the evolving precision in understanding and at compatible levels of detail.

Iterative and agile development processes have evolved into more successful delivery processes by improving the navigation through uncertainty. This steering requires measured improvement with dynamic controls, instrumentation, and intermediate checkpoints that permit stakeholders to assess what they have achieved so far (their as-is situation), what perturbations they should make to the target objectives (their to-be situation), and how to refactor what they have achieved to adjust and deliver those targets in the most economical way (the roadmap forward). The key outcome of these modern agile delivery principles is increased flexibility throughout the life cycle. This flexibility enables the continuous negotiation of scope, plans, and solutions for effective economic governance. Precision in the life-cycle artifacts is added as uncertainties are resolved.

The difference between precision and accuracy is an enlightening lens for focusing on the crux of software management. Accuracy is a measure of truth and freedom from error. Precision identifies the degree of accuracy and implies repeatability or elimination of uncertainty. Unjustified early precision — in requirements or plans — has proved to be a substantial, yet subtle, recurring obstacle to success. Most of the time, the pursuit of early precision is alluring but serves only to provide a counterproductive façade for portraying illusory progress and quality. Trust among stakeholders erodes as the divergence between reported progress and true progress inevitably reveals itself. Unfortunately, many stakeholders demand early precision and detail because it gives them (false) comfort in the progress achieved. Software management is full of gray areas, situation dependencies, and ambiguous tradeoffs. Understanding the difference between precision and accuracy is a fundamental skill of good software managers, who must accurately forecast resource estimates, risks, and the effects of change. There are several ways to present estimates, and trust is established with honest communications. Emphasizing accurate estimates with honest qualified precision will help stakeholders engage in a more fruitful discussion of the uncertainties remaining.

The definition of truth by the American justice system provides an illuminating metaphor for understanding the difference between accuracy and precision. Consider these three familiar components: the truth (be accurate), the whole truth (be precise *enough* and include everything relevant), and nothing but the truth (don't be *overly* precise and add irrelevant detail).

In the software development world, providing an accurate estimate is the main target dimension of the truth. Providing a credible measure for telling the whole truth requires backup evidence such as precedent experience, a prototype, or a resource estimate from an empirical model. Using honest precision (such as ranges or probability distributions) ensures that you are focused on nothing but the truth and are not misleading anyone by implying more certainty than is justified. This is best accomplished by openly admitting the uncertainties in your forecasts, providing some quantitative indication of the variance of your estimates. We have all seen proposals for software projects that forecast the cost with incredible precision, such as $20,320,541. Can we really forecast the cost of a project that we plan to deliver nine months from now down to the dollar? This is dishonest precision. Although precise commitments are frequently necessary in the business world, accurate representations of the uncertainty will establish more trust among stakeholders.

4 Six Econometric Patterns

IBM's Rational Division has been compiling best practices and economic improvement experiences for decades. We are in the continuing process of synthesizing this experience into more consumable advice and valuable intellectual property in the form of value traceability trees, metrics patterns, benchmarks of performance, and instrumentation tools to provide a closed-loop feedback control system for improved insight and management. Measurement of software progress and quality is a complex undertaking, given the large number of product, project, and people contexts that have an impact on software development efforts. However, several aspects of software measurement are generally applicable to almost all software projects.

The six metrics patterns are presented here from two perspectives of project governance. These perspectives illustrate the difference in outcomes between a conventional governance approach (waterfall model management using traditional plan-and-track project management techniques) and an economic governance approach (the measure-and-steer leadership style more pervasive in modern software delivery approaches).

I could provide hundreds of examples that support the patterns presented as typical of conventional governance. They are all too common. However, I can only provide about 10 projects that have demonstrated the measured patterns presented as the modern economic governance targets. These standout project successes all had a recurring attribute: They all implemented a process that drove integration testing earlier in the life cycle to assess the architecturally significant design and requirements tradeoffs. In other words, they attacked the larger uncertainties first, and they achieved cost-of-change trends that were counter to conventional wisdom. In most organizations, including IBM, these economic governance patterns are still aspirations more than expectations.

Reasoning through these metric patterns and reflecting on the impact of integration testing preceding unit testing will help to explain why the target metric patterns are so strikingly different from conventional wisdom.

There are two dimensions of metrics needed for effective steering: progress and quality. Progress metrics are indicators of how much work has been accomplished. Quality metrics provide indicators of how well that work has been accomplished. With these two perspectives, stakeholders can more accurately assess whether a project is likely to deliver successfully and predictably.

Although financial metrics are also needed, financial status is well understood. We know exactly how much money has been spent and how much time has elapsed. The challenge with earned value management (EVM) systems is to assess how much technical progress has been accomplished so that it can be compared with cost expended and time expended[2].[5] A reliable measure of earned value (or percent complete) is necessary to forecast accurately the estimates to complete. Traditional EVM methods measure against static targets. However, there is no reason EVM cannot be used with dynamically changing targets, which would result in more honest assessments of software delivery. I have only seen such EVM practices in a few industrial software projects where there was very high trust among stakeholders. The metric

[2]Wikipedia also has a concise summary of earned value management.

patterns that follow need to be combined with conventional cost and schedule tracking to provide more insightful EVM.

Conventional projects whose intermediate products were mostly paper documents relied on subjective assessments of technical progress or measured the number of documents completed and their level of detail. While these documents did reflect progress in expending energy, they were not very indicative of useful work being accomplished. Hence my assertion that such activity-based EVM techniques were inaccurate. They do not provide metrics that correlate with true progress.

Each proposed metric has two dimensions: the static value and the dynamic trend. While a discrete value provides one dimension of insight, how these values change over time provides the important perspective for assessing the current situation, forecasting expectations, and steering a project or organization. The change trends provide insight into how the products are evolving, whether situations are getting better or worse, and by how much. Agile software delivery and a steering management style are focused on managing change. Measuring change is therefore a necessity.

In modern software delivery projects or organizations focused on a software line of business, the historical values of previous iterations and projects provide precedent data for planning subsequent iterations and projects. Consequently, once metrics collection is ingrained, a project or organization can measurably improve its ability to predict the cost, schedule, or quality performance of future work activities. With the agile framework known as "Scrum," for example, estimation revolves around the product backlog. If the team velocity is well understood from previous sprints, and even if there are no significant impediments in sight, Scrum estimation still involves some guesswork. However, planning and expected timeframes become more trustworthy over time since they have a defendable, measurable basis of estimate.

4.1 Progress econometrics

Progress metrics are indicators of how much work has been accomplished. Figure 5 illustrates three core metrics for measuring progress. The presentation format for each metric provides two perspectives. The left-hand graphics are the anti-pattern, namely the metric trends expected if you follow conventional waterfall model processes and use engineering governance. The right-hand graphics are the target pattern of outcomes you would expect from healthy projects with true agility and robust architectural solutions.

4.1.1 Planning progress: demonstrable capability over time

Planning progress with conventional engineering governance (Figure 5, upper left) is measured through milestone achievement and earned value by producing all supporting artifacts, mostly documents. Planning progress is better measured with an anatomy of planned features, user stories, and capabilities for demonstrating progress to the user community (Figure 5, upper right). Reasoning about how you will demonstrate product or system features to your users will change the focus of planning and scope management from precise documents and models (namely, the anti-pattern of getting all the requirements right up front) to the important and incremental sequence of executable results needed to best resolve uncertainties. Planning progress is defined

here as the key measure of requirements management and keeps the team focused on user-perceived quality and progress.

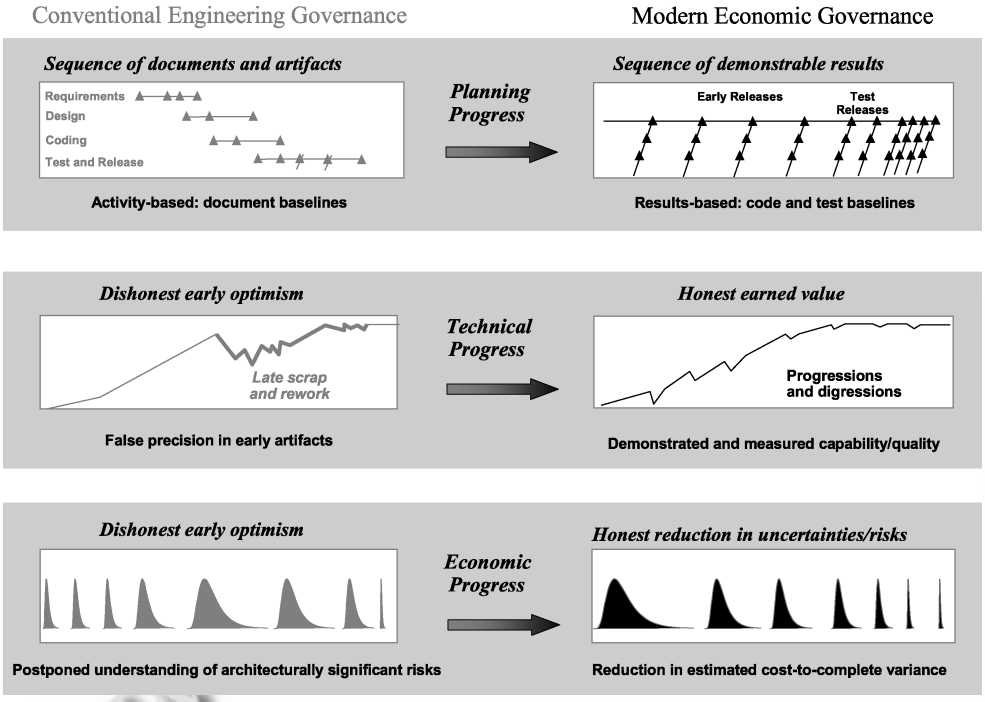


Figure 5. Progress metrics for better economic governance

An initial vision statement evolves into interim evaluation criteria for early demonstrable releases, which evolve into test cases for later release baselines and finally detailed acceptance criteria for releases to the user community. Scope evolves from abstract and accurate representations into precise and detailed representations as stakeholder understanding evolves (that is, uncertainty is reduced).

In modern economic governance, scope management is a discipline of asset-based development, where solutions evolve from stakeholder needs and stakeholder needs evolve from available solution assets. This equal and opposite interaction between the requirements space (user need) and solution space (assets, architectural patterns, and design efforts) is the engine for iteration. We don't build many applications dominated by custom code development anymore; they are neither economically feasible nor competitive.

Many organizations have adopted some form of measuring the burn-down rate and project "velocity" as a more agile measure of progress. Especially as projects get into the production stages of executable releases, the rate of change of remaining release content in units of user visible capabilities (features, user stories, scenarios, functions, services, or whatever you call them) becomes important. The big difference here is that you are measuring executable results rather than expended effort.

4.1.2 Technical progress: executable code and test base over time

The conventional profile for technical progress (Figure 5, middle left) is all too

familiar to most software development organizations. With few exceptions, waterfall-managed projects are mired in inefficient integration and late discovery of substantial design issues. They expend roughly 40% or more of their total resources in integration and test activities, with much of this effort consumed in excessive scrap and rework during the late stages of the planned project. By following a linear sequence of activities from requirements to design to code, to unit test and then to integration and test, projects effectively delay their tangible understanding of the significant uncertainties. The early false precision built into requirements documents, design documents, and project plans translates into late scrap and rework. Project management typically reports a linear progression of earned value up to 90% complete before reporting a major increase in the estimated cost of completion as they suffer through late integration testing where the real need to resolve architecturally significant uncertainties manifests itself.

Software earned value systems based on conventional activity, document completion, and milestone completion are not credible. They ignore integrated progress and quality of the evolving system architecture, where the value and uncertainties dominate, and focus instead on measuring piecemeal progress of activities and the quality of intermediate artifacts and components (the easy parts). The result of conventional engineering governance applied to software projects is that the end-game of most software projects is consumed completely in playing defense and fighting off excessive scrap and rework to avoid over-running cost and schedule targets.

The target profile for technical progress in agile software delivery (Figure 5, middle right) is a result of performing integration testing earlier through a progression of demonstrable releases, thereby exposing the architecturally significant uncertainties to be addressed earlier where they can be resolved efficiently in the context of life-cycle goals. A greater reliance on more standardized architectures and reuse of commercial components and other middleware are equally important. This reuse and architectural conformity contribute significantly to reducing uncertainty through less custom development and precedent patterns of construction. The downstream scrap and rework tar pit is avoidable, along with late shoe-horned fixes that degrade system changeability. Architectural changes are addressed earlier, resulting in a significant reduction in late, malignant changes. When software changes are more benign, projects have increased flexibility and can play offense in the later phases: adding features, adding quality, or delivering earlier.

A demonstration-driven life cycle where integration test coverage precedes unit test coverage results in a different project profile. Rather than a linear progression of earned value (which is usually inaccurate and misleading), a healthy project will exhibit an honest sequence of progressions and digressions as they resolve uncertainties, refactor architectures and scope, and converge on an economically governed solution. This technical progress metric becomes the key measure of architecture management, helping you to reason about how you will elaborate the architecture and refactor it as you flesh it out.

4.1.3 Economic progress: uncertainty in estimates-to-complete over time

The last progress metric is a measure of the variance in the cost- or time-to-complete estimate. This is a measure of the uncertainty remaining and requires

that project management periodically quantify an estimate-to-complete and explicitly address the error sources in that evolving sequence of estimates.

The conventional profile for economic progress (Figure 5, lower left) is characterized by optimistic early assessments of low uncertainty as stakeholders enjoy false comfort in the façade of detailed artifacts, plans, models, and early coded units. Relatively few design flaws and requirements tradeoffs are uncovered by the quality assurance techniques of human inspection, design review meetings, and requirements traceability analyses. Documents, models, and even source code are "speculative" representations: They are guesses. Only executable code provides tangible facts about behavior, performance, and quality. Although unit testing provides factual feedback on components, this only resolves the simpler bugs and local fixes. Once integration testing starts making the integrated software qualities tangible, we can move from speculative adjustments to factual steering on the most important (and usually most uncertain) system-level requirements tradeoffs and architectural behaviors and attributes. As these integration tests are executed later in the life cycle, the real uncertainties in the estimate-to-complete start to emerge.

The target profile for economic progress (Figure 5, lower right) illustrates the outcome of honest communications among stakeholders, where the uncertainties (and hence the variance in the estimates-to-complete) start off higher and the project leadership prioritizes activities, artifacts, demonstrations, and releases to steer the project systematically toward reduced uncertainty. Projects can be much more effective at steering toward better outcomes by prioritizing integration testing and the most challenging design tradeoffs ahead of unit test coverage, completeness of requirements, and completeness of design details in (falsely) precise artifacts. In the worst case, stakeholders realize that projects should be cancelled earlier because they are not converging on success. In the best case, stakeholders can make tradeoffs continuously among cost, value, and time. Both of these situations represent positive economic governance outcomes.

Modern agile governance of software delivery means managing uncertainty through steering. In a healthy software project, each successive phase of development produces an increased level of understanding in the evolving plans, specifications, and completed solution, because each phase furthers a sequence of executable capabilities and the team's knowledge of competing objectives. At any point in the life cycle, the precision of the subordinate artifacts should be in balance with the evolving precision in understanding, at compatible levels of detail and reasonably traceable to each other.

4.2 Quality econometrics

The defining characteristic of software is that it is "soft": The easier the software is to change, the easier it is to achieve any of its other required characteristics. The most important quality metrics are therefore centered on measurements of change trends (defects, scrap and rework) in the software release baselines throughout the life cycle.

The three progress metrics in Figure 5 help to quantify how much has been accomplished, but this is not enough information to steer systems and products to a state where they are suitable for release to their users. You need insight into how close

the current release is to meeting user expectations of both quality and content. You also need insight into the agility of your process and architecture in accommodating changes in the future if you are going to improve your economic forecasting.

Once software is placed in a controlled baseline, all changes can be instrumented. A distinction must be made for the cause of change. Change categories typically include:

• Critical failures, which are defects that are nearly always fixed prior to any external release. In general, these sorts of changes represent show-stoppers that have an impact on the usability of the software in its primary use cases.

• Non-critical failures, which are bugs or defects that either do not impair the utility of the system or can be worked around. Such errors tend to correlate to nuisances in primary use cases or serious defects in secondary use cases.

• A change that is an enhancement rather than a response to a defect. Its purpose is typically to improve performance, testability, usability, or some other aspect of quality that represents good value engineering.

• A change caused by an update to the current system or product, including new features or capabilities that are outside the scope of the current business case.

• Some other change, such as a version upgrade to a commercial component.

With the conventional process and custom architectures, change was more expensive to incorporate later in the life cycle. With waterfall projects that measured such trends, the cost of change tended to increase as they transitioned from testing individual units of software to testing the larger, integrated system. This is intuitively easy to understand, since unit changes, typically related to implementation issues or coding errors, were relatively easy to debug and resolve, and integration changes such as design issues, interface errors, or performance issues were relatively complicated to resolve. Furthermore, unit changes tend to be the responsibility of a single person, whereas integration changes required collaboration among multiple people.

A discriminating result of modern economic governance is that the more expensive changes are discovered earlier, when they can be efficiently resolved. Changes get simpler and more predictable later in the life cycle. This is the result of attacking the uncertainties in architecturally significant requirements tradeoffs and design decisions earlier in the life cycle. The big process change required is that integration activities mostly precede unit test activities, thereby resolving the riskier architectural and design challenges prior to investing in unit test coverage and complete implementations.

Quality metrics are indicators of how well work has been accomplished. Figure 6 illustrates three core metrics for measuring quality. The presentation format for each metric provides two perspectives. The left-hand graphics are the anti-pattern, namely, the metric trends expected if you follow conventional waterfall model processes and use engineering governance. The right-hand graphics are the target pattern of outcomes you would expect from healthy projects with true agility and robust architectural solutions.

4.2.1 Defects and maturity

Defects are software errors logged against a change-controlled baseline release. Maturity is a context-specific measure of the suitability for a system or software baseline to be released to its user community. Tracking defect statistics and backlogs are

well-understood best practices in most organizations. There are usually multiple levels of defect classification, and while implementations vary across the broad spectrum of software contexts, the principles and patterns are well established.

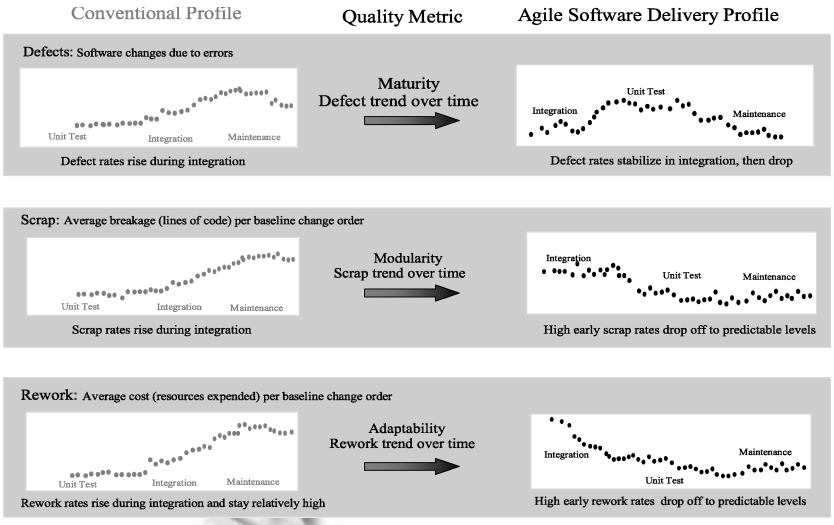


Figure 6. Quality metrics for better economic governance

4.2.2 Scrap and modularity

Scrap is a measure of the average breakage per software change. The units of measurement for scrap can be source lines of code, function points, object points, files, components, or some other measure of software size. Source-line-of-code (SLOC) metrics are probably the easiest measure to automate, but without savvy tooling and context-dependent insight, they can also lead to overly simplistic conclusions. Tacking the average scrap per change over time gives you a measure of modularity. This indicator provides insight into the benign or malignant character of software change.

4.2.3 Rework and adaptability

Rework is defined as the average cost of change, which is the effort to analyze, resolve, and retest all changes to software baselines. The units of measurement for rework are staff-hours per change. Adaptability is defined as the rework trend over time. Not all changes are created equal. Some changes can be made in a staff-hour, while others take staff-weeks. The overall trends in effort per change provide the most valuable insight into process effectiveness (that is, agility) and architecture qualities (such as changeability, resilience, and modularity).

Discrete change traffic needs to be tracked. This is simply a measure of the number of software change orders opened and closed over the life cycle. This metric can be collected by change type, by release, across all releases, or by team, by components, by subsystem, and so forth. Coupled with other progress and quality metrics, discrete change traffic provides insight into the stability of the software and its convergence towards stability (or divergence towards instability)[5].

Consistency of application is important for accurate interpretation, just as it is with cost estimation techniques. Software cost estimation has mostly subjective inputs and objective outputs. These three quality metrics are objective indicators that require subjective interpretation by stakeholders within the context of a specific project situation. For example, the amount of rework following the first configuration baseline during development is an ambiguous value without further context. Zero rework might be interpreted as a perfect baseline (unlikely), an inadequate test program, or an unambitious first build.

In general, being "mature enough to release" is a judgment call that requires a fair amount of history and context. Most organizations understand how to do this, but their processes and architectures develop, test, and mature software in a very suboptimal manner that wastes tremendous amounts of resources. The conventional profile for defect rates (Figure 6, upper left) typically rises as more software is tested and testing covers more of the usage models. Resolution of defects keeps up with defects identified for the early phases of unit testing, where most of the changes are relatively benign. But as integration testing ramps up, the errors uncovered tend to affect many units and be much more difficult to resolve. Escalating scrap rates (Figure 6, middle left) and escalating rework rates (Figure 6, lower left) compel the entire project team to play defense and struggle to keep up.

With such late discovery of requirements and design issues in waterfall model governance, there is no time to redesign or renegotiate requirements. Consequently, project teams tend to shoehorn in suboptimal fixes, resulting in malignant changes and further introduction of defects. As these software releases are transitioned to users, the defect rate remains high, architectural integrity suffers, and the iron law of software development becomes inevitable: The later you are in the life cycle, the more expensive things are to fix.

The profiles for defect rate and maturity can be very different if you can drive integration testing earlier (Figure 6, upper right). Initial defect rates may start off rather low. The scrap rate (modularity, Figure 6, middle right) and the rework rate (adaptability, Figure 6, lower right) will start off relatively high. As the architecture is prototyped, demonstrated, tested, and refactored across the significant usage models, behaviors, and performance scenarios, these early changes will resolve the significant uncertainties and accelerate stakeholder understanding of the requirements and design tradeoffs. As the defect rate drops in integration testing, stakeholders can be more confident that the architecture baseline is mature, that further malignant changes will be less likely, and that future changes will be mostly benign. When early and continuous integration is executed effectively, the later you are in the life cycle, the more predictable and straightforward things are to change.

5 Conclusions

True agility with a track record of measured improvement is the aspiration of every business that depends on software delivery capability. To accelerate this improvement, projects and organizations should set tangible, measureable targets for becoming more agile and channel that agility into business flexibility:

1. True agility means that life-cycle changes become more predictable and straightforward over time.

2. To achieve true agility, the completion and coverage of integration tests must take precedence over the completeness and coverage within unit tests.

3. Measurement is a cornerstone of agility, and measuring trends in executable software baselines illuminates the progress and quality indicators needed to steer projects with economic governance to more successful outcomes.

A strong foundation for measurable improvement is a powerful by-product of agile methods and metrics extraction from the release baselines undergoing continuous integration testing. This is the basis for honest reduction of uncertainty and honest assessment of progress and quality.

The six econometrics described here are based on field experience with both successful and unsuccessful metrics programs. Their attributes include the following:

• They are simple, objective, easy to collect, easy to interpret, and hard to misinterpret.

• Collection can be automated and nonintrusive.

• They provide for consistent assessments throughout the life cycle and are derived from the evolving product baselines rather than some subjective assessment.

• They are useful by both management and engineering personnel in communicating progress and quality.

• They accurately portray the agility of a process and architecture with precision that improves across the life cycle commensurate with the understanding of the user need, design solution, and forecasted plans.

The last attribute needs some emphasis. Metrics applied to early life-cycle phases, dominated by high-variance human creativity and risk resolution, should be accurate but far less precise than those applied to the later phases in the life cycle, dominated by low variance production activities and rigorous change management.

True agility translates into business flexibility. Organizations and projects that can transform to truly agile development with economic governance and strong measurement disciplines have a huge advantage. When changes are easy to implement, a project is more likely to increase the number of changes, thereby increasing quality. When architectures are resilient, processes embrace change, and platforms support change automation and measurement, then projects can balance their resource investments between defensive efforts (such as bug fixes, feature commitments, and schedule commitments) and offensive efforts (such as new integrations, new innovations, improved performance, earlier releases, and higher quality).

Well-executed measured improvement efforts will speed up a sales cycle or a delivery cycle because they establish more trust among stakeholders. More trust translates directly into less overhead and less waste – key themes within lean management principles and agile methods. Measured improvement in software delivery is still far from a mature discipline. We have plenty of anecdotal experience, expert opinions on best practices, and subjective judgments on patterns of success. We have relatively few objective case studies, benchmarks, and quantified experience reports. The software industry needs to strengthen the credibility of measured improvement benchmarks and guidance.

Acknowledgments

Scott Ambler, Dave Bernstein, Murray Cantor, Capers Jones, Joe Marasco, Mar-

tin Nally, Mike Perrow, Mirek Rzadkowski, Danny Sabbah, and Richard Solely reviewed and improved early drafts of this paper. Karen Ailor helped sharpen the presentation and content.

Finally, my academic and professional interests in improving software economics were enabled significantly by Barry Boehm. His insight and guidance helped shape my perspective and values. Barry stands out as one of the pre-eminent contributors to one of the world's most powerful and useful endeavors: software development. We all owe him our gratitude, and I would like to express my thanks for his vast contributions and mentorship over the past three decades.

References

[1] Boehm BW. Software engineering economics. IEEE Trans. on Software Engineering, SE-10, January 1984, 4–21.
[2] Boehm BW. Software Engineering Economics. Englewood Cliffs, New Jersey: Prentice-Hall Inc, 1981.
[3] Boehm BW. Software Cost Estimation with COCOMO II. Upper Saddle River, New Jersey: Prentice Hall PTR, 2000.
[4] Jones C. Software Engineering Best Practices. McGraw Hill, 2010.
[5] Royce WE. Software Project Management. Addison-Wesley, 1998.
[6] Royce WE, Kurt B, Michael P. The Economics of Software Development. Reading, Massachusetts: Addison-Wesley, 2009.
[7] Hubbard D. How to Measure Anything: Finding the Value of Intangibles in Business. Hoboken, New Jersey: Wiley, 2010.
[8] Royce WW. Managing the Development of Large Software Systems. Proc. of IEEE Wescon, August 1970: 1–9.
[9] Kennaley M. SDLC 3.0, Beyond a Tacit Understanding of Agile. Fourth Medium Press, 2010.
[10] Lo A, Mark M. Moody's/NYU 6th Annual Credit Risk Conference. New York, March 2010.
[11] Austin R, Lee D. Artful Making. Prentice Hall, 2003.
[12] Royce W. Successful software management style: steering and balance. IEEE Software, September/October 2005, 22(5): 40–47.
[13] Boehm B. A spiral model of software development and enhancement. IEEE Computer, May 1988, 21(5): 61–72.
[14] Boehm B. Software Risk Management. IEEE Computer Society Press, 1989.
[15] Berggren C, Jack J, Jonas S. Lagomizing, organic integration, and systems emergency wards: innovative practices in managing complex systems development projects. Project Management Journal, 2008, 39 (Supplement): S111–S122.
[16] Cantor M. Estimation Variance and Governance. The Rational Edge, March 2006.