Improving software economics

White paper

May 2009

Rational. software

Improving Software Economics

Top 10 Principles of Achieving Agility at Scale

by Walker Royce

Vice President, IBM Software Services, Rational

Contents

From software development to software delivery

The world is becoming more dependent on software delivery efficiency and world economies are becoming more dependent on producing software with improved economic outcomes. What we have learned over decades of advancing software development best practice is that software production involves more of an economics than an engineering discipline. This paper provides a provocative perspective on achieving agile software delivery and the economic foundations of modern best practices.

Improvement in software life-cycle models and software best practices has been a long slog that accelerated in the 1980s as the engineering roots of software management methods continued to fail in delivering acceptable software project performance. IBM's Rational team has partnered with hundreds of software organizations and participated in thousands of software projects over the last 25 years. Our mission has been twofold: first, to bring software best practices to our customers, and second, to participate directly on their diverse projects to learn the patterns of success and failure so that we could differentiate which practices were best, and why. The Rational team didn't invent iterative development, object-oriented design, UML, agile methods, or the best practices captured in the IBM® Rational® Unified Process. The industry evolved these techniques, and we built a business out of synthesizing the industry's experience and packaging lessons learned into modern processes, methods, tools, and training. This paper provides a short history of this transition by looking at the evolution of our management principles. It presents our view of the Top 10 principles in managing an industrial-strength software organization and achieving agility at any scale of business challenge.

Most organizations that depend on software are struggling to transform their life-cycle model from a development focus to a delivery focus. This subtle distinction in wording represents a dramatic change in the principles that are driving the management philosophy and the governance models. Namely, a "software development" orientation focuses on the various activities required in the development process, while a "software delivery" orientation focuses on the results of that process. Organizations that have successfully made this transition—perhaps 30-40% by our estimate—have recognized that engineering discipline is trumped by economics discipline in most software-intensive endeavors.

Table 1 provides a few differentiating indicators of successfully making the transformation from conventional engineering governance to more economically driven governance.

Table 1: Differentiating conventional engineering governance from economically driven governance

Software Development: Engineering Driven	**Software Delivery: Economics Driven**
Distinct development phases	Continuously evolving systems
Distinct handoff from development team to maintenance team	Common process, platform, and team for development and maintenance
Distinct and sequential activities from requirements to design to code to test to operations	Sequence of usable capabilities with ever-increasing value
Phase and role specific processes and tools	Collaborative platform of integrated tools and process enactment
Collocated teams	Geographically distributed, Web-based collaboration
Early precision in complete plans and requirements	Evolving precision as uncertainties are resolved
Governance through measurement of artifact production and activity completion	Governance through measurement of incremental outcomes, and progress/quality trends
Engineering discipline: precisely define requirements/plans completely, then track progress against static plans and scope	Economic discipline: reduce uncertainties, manage variance, measure trends, adapt and steer with continuous negotiation of targets

Success rates in applying engineering governance (a.k.a. waterfall model management) have been very low; most industry studies assess the success rate at 10-20%. Where waterfall model projects do succeed, one usually finds that the project has been managed with two sets of books. The front-office books satisfy the external stakeholders that the engineering governance model is being followed and the back-office books, where more agile techniques are employed with economic governance, satisfy the development team that they can predictably deliver results in the face of the uncertainties. The results of the back-office work gets fed back to meet the deliverables and milestones required for the front-office books. "Managing two sets of books" has been expensive, but it is frequently the only way for developers to deliver a satisfactory product while adhering to the stakeholder demand for engineering governance. Advanced organizations have transitioned to more efficiently managing only one set of honest plans, measures, and outcomes. Most organizations still manage some mixture of engineering governance and economic governance to succeed.

Let's take a minute to think about engineering vs. economics governance -- i.e., precise up-front planning vs. continual course correction toward a target goal -- in terms even those outside the software industry can relate to. This may be a thought-provoking hypothesis: Software project managers are more likely to succeed if they use techniques similar to those used in movie production, compared to those used conventional engineering projects, like bridge construction.[1,2] Consider this:

- Most software professionals have no laws of physics, or properties of materials, to constrain their problems or solutions. They are bound only by human imagination, economic constraints, and platform performance once they get something executable.
- Quality metrics for software products have few accepted atomic units. With the possible exception of reliability, most aspects of quality are very subjective, such as responsiveness, maintainability and usability. Quality is best measured through the eyes of the audience.
- In a software project, you can seemingly change almost anything at any time: plans, people, funding, requirements, designs, and tests. Requirements—probably the most misused word in our industry—rarely describe anything that is truly required. Nearly everything is negotiable.

These three observations are equally applicable to software project managers and movie producers. These are professionals that regularly create a unique and complex web of intellectual property bounded only by a vision and human creativity. Both industries experience a very low success rate relative to mature engineering enterprises.

The last point above is worth a deeper look. The best thing about software is that it is soft (i.e., relatively easy to change) but this is also its riskiest attribute. In most systems, the software is where we try to capture and anticipate human behavior, including abstractions and business rules. Most software does not deal with natural phenomena where laws of physics or materials provide a well-understood framework. Hence, most software is constrained only by human imagination; the quality of software is judged more like a beauty contest than by precise mathematics and physical tolerances. If we don't carefully manage software production, we can lull ourselves into malignant cycles of change that result in massive amounts of scrap, rework, and wasted resources.

With the changeability of software being its greatest asset and greatest risk, it is imperative that we measure software change costs and qualities and understand the trends therein. The measure of scrap and rework is an economic concern that has long been understood as a costly variable in traditional engineering, as in the construction industry. While in the software industry we commonly blow up a product late in the lifecycle and incur tremendous scrap and rework to rebuild its architecture, we rarely do this in the construction industry. The costs are so tangibly large, and the economic ramifications are dire. In software, we need to get an equally tangible understanding of the probable economic outcomes.

A lesson that the construction industry learned long ago was to eliminate the risk of reinventing the laws of construction on every project. Consequently, they enforced standards in building codes, materials, and techniques, particularly for the architectural engineering aspects of structure, power, plumbing, and foundation. This resulted in much more straightforward (i.e., predictable) construction with innovation mostly confined to the design touches sensed by its human users. This led to guided economic governance for the design/style/usability aspects with standardization and engineering governance driving most of the architecture, materials, and labor. When we innovate during the course of planned construction projects with new materials, new technology, or significant architectural deviations, it leads to the same sorts of overruns and rework that we see in software projects. For most products, systems, and services, you want to standardize where you can and not reinvent.

Economic discipline and governance is needed to measure the risk and variance of the uncertain outcomes associated with innovation. Most software organizations undertake a new software project by permitting their most trusted craftsmen to reinvent software capabilities over and over. Each project and each line of business defend the reasons why their application is different, thereby requiring a custom solution without being precise about what is different. Encumbered with more custom developed architectures and components than reused ones, they end up falling back on the waterfall model, which is easy to understand. But this approach is demonstrably too simplistic for uncertain endeavors like software.

The software industry has characterized new and improved software life-cycle models using many different terms, such as: spiral development, incremental development, evolutionary development, iterative development, and agile development. In spirit, these models have many things in common, and, as a class, they represent a common theme: anti-waterfall development. However, after 20-30 years of improvement and transition, the waterfall model mindset is still the predominant governance process in most industrial-strength software development organizations. By my estimation, more than half of the software projects in our industry still govern with a waterfall process, particularly organizations with mature processes. Perhaps geriatric could be used as an explicit level of process maturity, one that should be recognized in software maturity models to help organizations identify when their process has become too mature and in need of a major overhaul.

The move to agility

We have learned many best practices as we evolved toward modern agile delivery methods. Most of them we discovered years ago as we worked with forward-looking organizations. At IBM, we have been advancing techniques largely from the perspective of industrial strength software engineering, where scale and criticality of applications dominate our governance and management methods. We were one of the pioneers of agile techniques like pair programming[3] and extreme programming,[4] and IBM now has a vibrant technical community with thousands of practitioners engaged in agile practices in our own development efforts and our professional services. Many pioneering teams inside and outside of IBM have advanced these best practices from smaller scale techniques, commonly referred to as "agile methods," and these contributions were developed separately in numerous instances across the diverse spectrum of software domains, scales, and applications.

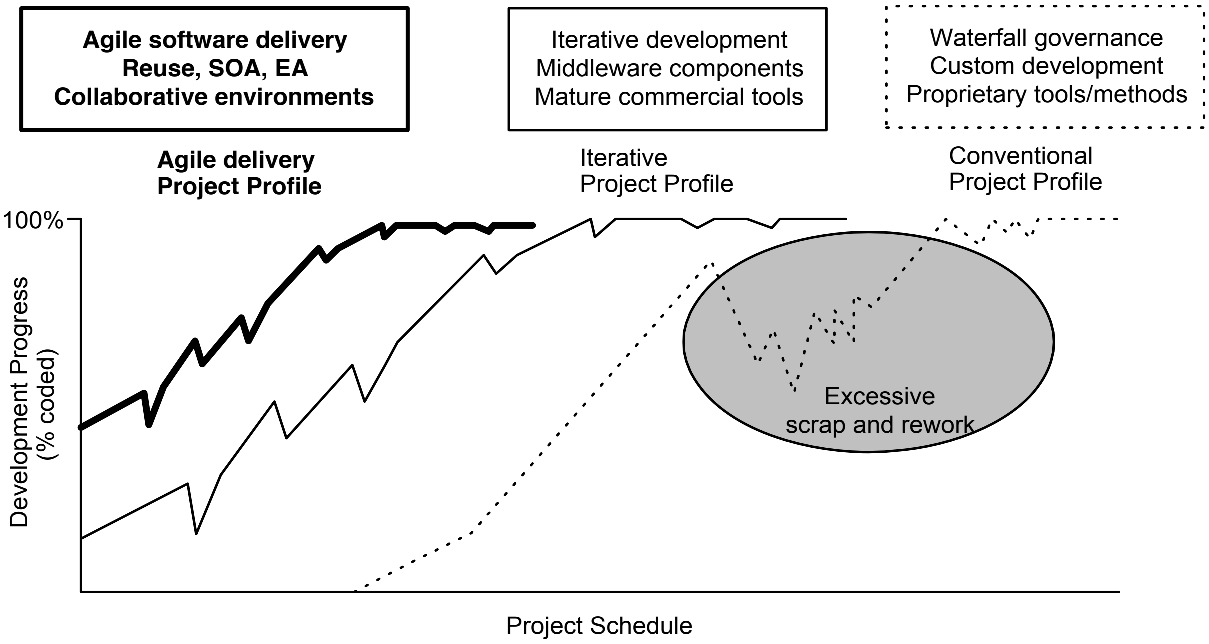
For years, we have worked to unite the agile consultants (i.e., small scale development camps) with the process maturity consultants (i.e., industrial strength software development camps). While these camps have been somewhat adversarial and wary of endorsing one another, both sides have valid techniques and a common spirit, but approach common problems with a different jargon and bias. There is no clear right or wrong prescription for the range of solutions needed. Context and scale are important, and every nontrivial project or organization needs a mix of techniques, a family of process variants, common sense, and domain experience to be successful.

In *Software Project Management,*[5] I introduced my Top 10 Principles of managing a modern software process. I will use that framework to summarize the history of best-practice evolution. The sections that follow describe three discrete eras of software life-cycle models by capturing the evolution of their top 10 principles. I will denote these three stages as 1) conventional waterfall development, 2) transitional iterative development, and 3) modern agile delivery. I will only describe the first two eras briefly since they have been covered elsewhere in greater detail and their description here is only to provide benchmarks for comparison to the top 10 principles of a modern agile delivery approach.

Figure 1 provides 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 eras plot development progress versus time, where progress is defined as percent executable—that is, demonstrable in its target form. Progress in this sense correlates to tangible intermediate outcomes, and is best measured through executable demonstrations. The term "executable" does not imply complete, compliant, nor up to specifications; but it does imply that the software is testable. The figure also describes the primary measures that were used to govern projects in these eras and introduces the measures that we find to be most important moving forward to achieve agile software delivery success.

Conventional waterfall projects are represented by the dotted line profile in Figure 1. The typical sequence for the conventional waterfall management style when measured this way is:

1. Early success via paper designs and overly precise artifacts,
2. Commitment to executable code late in the life cycle,
3. Integration nightmares due to unforeseen implementation issues and interface ambiguities,
4. Heavy budget and schedule pressure to get the system working,
5. Late shoe-horning of suboptimal fixes, with no time for redesign, and
6. A very fragile, expensive-to-maintain product, delivered late.



Agile Econometrics	**Iterative Trends**	**Waterfall measures**
Accurate net present value	Honest earned value	Dishonest earned values
Reuse/custom asset trends	Release content over time	Activity/milestone completion
Release quality over time	Release quality over time	Code/test production
Variance in estimate to complete	Prioritized risk management	Requirements-design-code traceability
Release content/quality over time	Scrap/rework/defect trends	Inspection coverage
Actuals vs dynamic plans	Actuals vs dynamic plans	Actuals vs static plan

Figure 1: Improved project profiles and measures in transitioning to agile delivery processes

Most waterfall projects are mired in inefficient integration and late discovery of substantial design issues, and 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, when project management had imagined shipping or deploying the software. 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 the late scrap and rework. In retrospect, software earned value systems based on conventional activity, document, and milestone completion are not credible since they ignore the uncertainties inherent in the completed work. Here is a situation for which I have never seen a counter-example: A software project that has a consistently increasing progress profile is certain to have a pending cataclysmic regression.

The iterative management approach represented by the middle profile in Figure 1 forces integration into the design phase 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. Equally as critical to the process improvements are a greater reliance on more standardized architectures and reuse of operating systems, data management systems, graphical user interfaces, networking protocols, and other middleware. This reuse and architectural conformity contributes significantly to reducing uncertainty through less custom development and precedent patterns of construction. The downstream scrap and rework tarpit is avoidable, along with late patches and malignant software fixes. The result is a more robust and maintainable product delivered more predictably with a higher probability of economic success. Iterative projects can deliver a product with about half the scrap and rework activities as waterfall projects by re-factoring architecturally significant changes far earlier in the lifecycle.

Agile software delivery approaches start projects with an ever increasing amount of the product coming from existing assets, architectures, and services, as represented in the left hand profile. Integrating modern best practices and a supporting platform that enables advanced collaboration allows the team to iterate more effectively and efficiently. Measurable progress and quality are accelerated and projects can converge on deliverable products that can be released to users and testers earlier. Agile delivery projects that have fully transitioned to a steering leadership style based on effective measurement can optimize scope, design, and plans to reduce this waste of unnecessary scrap and rework further, eliminate uncertainties earlier, and significantly improve the probability of win-win outcomes for all stakeholders. Note that we don't expect scrap and rework rates to be driven to zero, but rather to a level that corresponds to healthy discovery, experimentation, and production levels commensurate with resolving the uncertainty of the product being developed.

Table 2 provides one indicative benchmark of this transition. The resource expenditure trends become more balanced across the primary workflows of a software project as a result of less human-generated stuff, more efficient processes (less scrap and rework), more efficient people (more creative work, less overhead), and more automation.

Table 2: Resource expenditure profiles in transitioning to agile delivery processes

Lifecycle Activity	**Conventional**	**Iterative**	**Agile**
Management	5%	10%	**10-15%**
Scoping	5%	10%	**10-15%**
Design/demonstration	10%	15%	**10-20%**
Implementation/coding	30%	25%	**15-20%**
Test and assessment	40%	25%	**15-25%**
Release and Deployment	5%	5%	**10%**
Environment/tooling	5%	10%	**10%**

Top 10 Principles of Conventional Software Management

Most software engineering references present the waterfall model[6] as the source of the "conventional" software management process, and I use these terms interchangeably. Years ago, I asserted the top 10 principles of the conventional software process to capture its spirit and provide a benchmark for comparison with modern methods. The interpretation of these principles and their order of importance are judgments that I made based on experiences from 100s of project evaluations, project diagnoses performed by the Rational team, and discussions with Winston Royce, one of the pioneers in software management processes. My father is well-known for his work on the waterfall model, but he was always more passionate about iterative and agile techniques well before they became popular.[7]

Top 10 Management Principles of Waterfall Development

1. Freeze requirements before design.
2. Forbid coding prior to detailed design review.
3. Use a higher order programming language.
4. Complete unit testing before integration.
5. Maintain detailed traceability among all artifacts.
6. Thoroughly document each stage of the design.
7. Assess quality with an independent team.
8. Inspect everything.
9. Plan everything early with high fidelity.
10. Control source code baselines rigorously.

Conventional software management techniques typically follow a sequential transition from requirements to design to code to test with extensive paper-based artifacts that attempt to capture complete intermediate representations at every stage. Requirements are first captured in complete detail in ad hoc text and then design documents are fully elaborated in ad hoc notations. After coding and unit testing individual code units, they are integrated together into a complete system. This integration activity is the first time that significant inconsistencies among components (their interfaces and behavior) can be tangibly exposed, and many of them are extremely difficult to resolve. Integration — getting the software to operate reliably enough to test its usefulness — almost always takes much longer than planned. Budget and schedule pressures drive teams to shoehorn in the quickest fixes. Re-factoring the design or reconsideration of requirements is usually out of the question. Testing of system threads, operational usefulness, and requirements compliance gets performed through a series of releases until the software is judged adequate for the user. More than 80% of the time, the end result is a late, over-budget, fragile, and expensive-to-maintain software system.

Hindsight from thousands of software project post-mortems has revealed a common symptom of governing a software project with an engineering management style: the project's integration and test activities require an excessive expenditure of resources in time and effort. This excessive rework is predominantly a result of postponing the resolution of architecturally significant issues (i.e., resolving the more serious requirements and design uncertainties) until the integration and test phase. We observed that better performing projects would be completed with about 40% of their effort spent in integration and test. Unsuccessful projects spent even more. With less than one in five projects succeeding, better governance methods were imperative.

One of the most common failure patterns in the software industry is to develop a five-digits-of-precision version of a requirement specification (or plan) when you have only a one-digit-of-precision understanding of the problem. A prolonged effort to build precise requirements or a detailed plan only delays a more thorough understanding of the architecturally significant issues — that is, the essential structure of a system and its primary behaviors, interfaces, and design trade-offs. How many frighteningly thick requirements documents or highly precise plans (i.e., inchstones rather than milestones) have you worked on, perfected, and painstakingly reviewed, only to completely overhaul these documents months later?

The single most important lesson learned in managing software projects with the waterfall model was that software projects contain much more uncertainty than can be accommodated with an engineering governance approach. This traditional approach presumes well-understood requirements and straightforward production activities based on mature engineering precedent.

Top 10 Principles of Iterative Software Management

In the 1990s, Rational Software Corporation began evolving a modern process framework to more formally capture the best practices of iterative development. The primary goal was to help the industry transition from a "plan and track" management style (the waterfall model) to a "steering" leadership style that admitted uncertainties in the requirements, design, and plans.

The software management approach we evolved led to producing the architecture first, then usable increments of partial capability, then you worry about completeness. Requirements and design flaws are detected and resolved earlier in the life cycle, avoiding the big-bang integration at the end of a project by integrating in stages throughout the project life cycle. Modern, iterative development enables better insight into quality because system characteristics that are largely inherent in the architecture (e.g., performance, fault tolerance, adaptability, interoperability, maintainability) are tangible earlier in the process where issues are still correctable without jeopardizing target costs and schedules. These techniques attacked major uncertainties far earlier and more effectively. Here are my top 10 principles of iterative development[8] from the 1990s and early 2000s era:

Top 10 Management Principles of Iterative Development

1. Base the process on an architecture-first approach.
2. Establish an iterative life-cycle process that confronts risk early.
3. Transition design methods to emphasize component-based development.
4. Establish a change management environment.
5. Enhance change freedom through tools that support round-trip engineering.
6. Capture design artifacts in rigorous, model-based notation.
7. Instrument the process for objective quality control and progress assessment.
8. Use a demonstration-based approach to assess intermediate artifacts.
9. Plan intermediate releases in groups of usage scenarios with evolving levels of detail.
10. Establish a configurable process that is economically scalable.

Whereas conventional principles drove software development activities to overexpend in integration activities, these modern principles resulted in less total scrap and rework through relatively more emphasis in early life-cycle engineering and a more balanced expenditure of resources across the core workflows of a modern process.

The architecture-first approach forces integration into the design phase, where the most significant uncertainties can be exposed and resolved. The early demonstrations do not eliminate the design breakage; they just make it happen when it can be addressed effectively. The downstream scrap and rework is significantly reduced along with late patches and sub-optimal software fixes, resulting in a more robust and maintainable design.

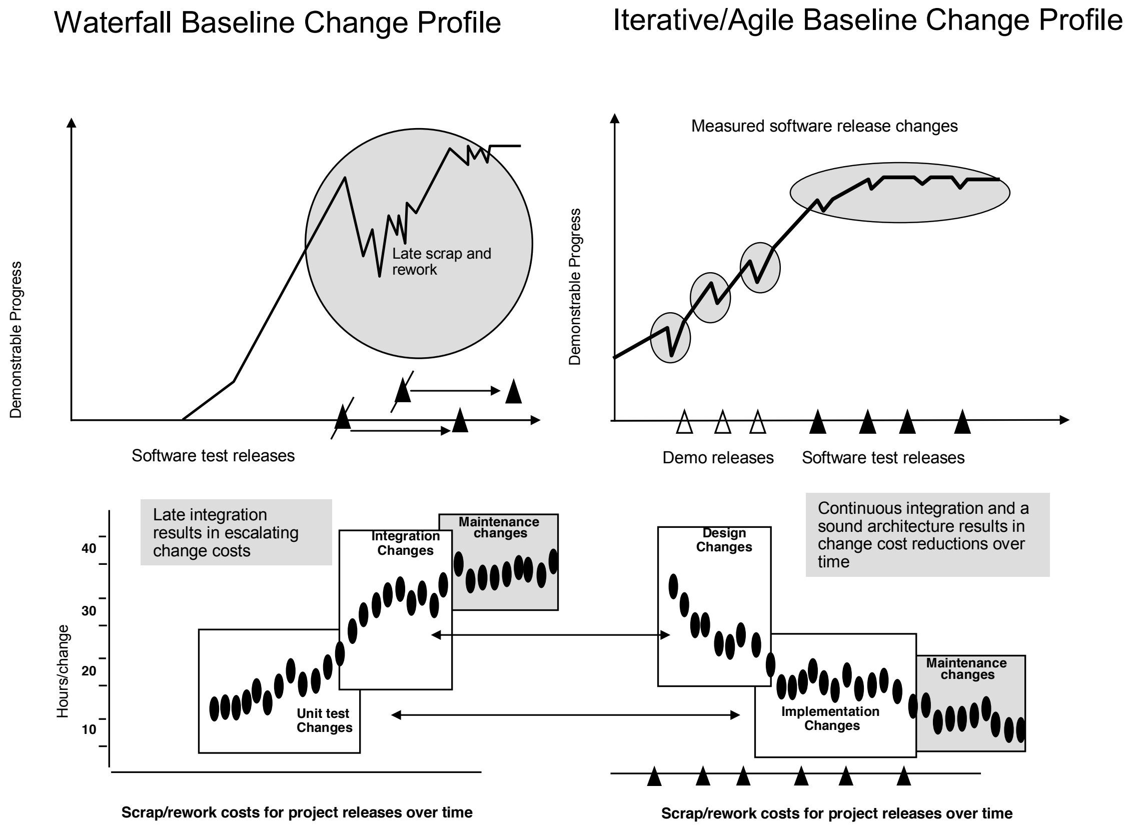
Interim milestones provide tangible results. Designs are now "guilty until proven innocent." The project does not move forward until the objectives of the demonstration have been achieved. This does not preclude the renegotiation of objectives once the milestone results permit further refactoring and understanding of the tradeoffs inherent in the requirements, design, and plans.

Figure 1 illustrates the change in measurement mindset when moving from waterfall model measures of activities to iterative measures of scrap and rework trends in executable releases. The trends in *cost of change*[9] can be observed through measuring the complexity of change This requires a project to quantify the rework (effort required for resolution) and number of instances of rework. In simple terms, adaptability quantifies the ease of changing a software baseline, with a lower value being better. When changes are easy to implement, a project is more likely to increase the number of changes, thereby increasing quality. With the conventional process and custom architectures, change was more expensive to incorporate as we proceeded later into the life cycle. For waterfall projects that measured such trends, they tended to see the cost of change 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 implementation issues or coding errors) were relatively easy to debug and resolve and integration changes (design issues, interface errors or performance issues) were relatively complicated to resolve.

A discriminating result of a successful transition to a modern iterative process with an architecture first approach is that the more expensive changes are discovered earlier when they can be efficiently resolved and get simpler and more predictable as we progress later into the life cycle. This is the result of attacking the uncertainties in architecturally significant requirements tradeoffs and design decisions earlier. The big change in an iterative approach 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.

This is the single most important measure of software project health. If you have a good architecture and an efficient process, the long-accepted adage: "*The later you are in the life cycle, the more expensive things are to fix*" does NOT apply. [10]

Successful steering in iterative development is based on improved measurement and metrics extracted directly from the evolving sequence of executable releases. These measures, and the focus on building the architecture first, allow the team to explicitly assess trends in progress and quality and systematically address the primary sources of uncertainty. The absolute measures are useful, but the relative measures (or trends) of how progress and quality change over time are the real discriminators in improved steering, governance, and predictability.



Waterfall Baseline Change Profile

Iterative/Agile Baseline Change Profile

Demonstrable Progress
Late scrap and rework
Software test releases
Measured software release changes
Demonstrable Progress
Demo releases
Software test releases
Late integration results in escalating change costs
Hours/change
40
30
20
10
Unit test Changes
Integration Changes
Maintenance changes
Design Changes
Implementation Changes
Maintenance changes
Continuous integration and a sound architecture results in change cost reductions over time
Scrap/rework costs for project releases over time
Scrap/rework costs for project releases over time

Figure 1: The discriminating improvement measure: change cost trends

Balancing innovation with standardization is critical to governing the cost of iterating, as well as governing the extent to which you can reuse assets versus developing more custom components. Standardization through reuse can take on many forms including:

- **Product assets**: architectures, patterns, services, applications, models, commercial components, legacy systems, legacy components…
- **Process assets**: methods, processes, practices, measures, plans, estimation models, artifact templates …

- **People assets**: existing staff skills, partners, roles, ramp-up plans, training …
- **Platform assets**: schemas, commercial tools, custom tools, data sets, tool integrations, scripts, portals, test suites, metrics experience databases …

While this paper is primarily concerned with the practice of reducing uncertainty, there is an equally important practice of reusing assets based on standardization. The value of standardizing and reusing existing architectural patterns, components, data, and services lies in the reduction in uncertainty that comes from using elements whose function, behavior, constraints, performance, and quality are all known. The cost of standardizing and reuse is that it can constrain innovation. It is therefore important to balance innovation and standardization, which requires emphasis on economic governance to reduce uncertainty; but that practice is outside the scope of this paper.

Reducing uncertainty: The basis of best practice

The top 10 principles of iterative development resulted in many best practices, which are documented in the Rational Unified Process.[11] The Rational Unified Process includes practices for requirements management, project management, change management, architecture, design and construction, quality management, documentation, metrics, defect tracking, and many more. These best practices are also context dependent. For example, a specific best practice used by a small research and development team at an ISV is not necessarily a best practice for an embedded application built to military standards. After several years of deploying these principles and capturing a framework of

best practices, we began to ask a simple question: "Why are these best? And what makes them better?"
IBM research and the IBM Rational organization have been analyzing these questions for over a decade, and we have concluded that *reducing uncertainty* is THE recurring theme that ties together techniques that we call best practices. Here is a simple story that Murray Cantor composed to illustrate this conclusion.

Suppose you are the assigned project manager for a software product that your organization needs to be delivered in 12 months to satisfy a critical business need. You analyze the project scope and develop an initial plan and mobilize the project resources estimated by your team. They come back after running their empirical cost/schedule estimation models and tell you that the project should take 11 months. Excellent! What do you do with that information? As a savvy and scarred software manager, you know that the model's output is just a point estimate and simply the expected value of a more complex random variable, and you would like to understand the variability among all the input parameters and see the full distribution of possible outcomes. You want to go into this project with a 95% chance of delivering within 12 months. Your team comes back and shows you the complete distribution illustrated as the "baseline estimate" at the top of Figure 2. I'll describe the three options shown in a moment.

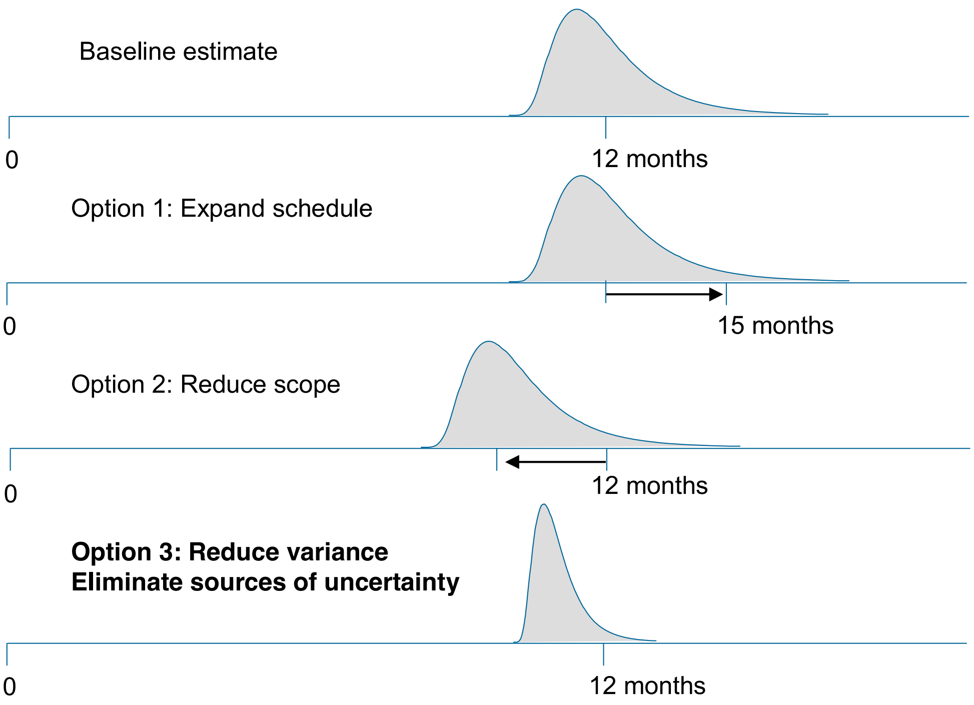


Figure 2: A baseline estimate and alternatives in dealing with project management constraints.

Examining the baseline estimate, you realize that about half of the outcomes will take longer than 12 months and you have only about a 50% chance of delivering on time. The reason for this dispersion is the significant uncertainty in the various input parameters reflecting your team's lack of knowledge about the scope, the design, the plan, and the team capability. Consequently, the variance of the distribution is rather wide.[12] Now, as a project manager there are essentially three paths that you can take; these are also depicted in Figure 2:

1. Option 1: Ask the business to move out the target delivery date to 15 months to ensure that 95% of the outcomes complete in less time than that.

2. Option 2: Ask the business to re-scope the work, eliminating some of the required features or backing off on quality so that the median schedule estimate moves up by a couple of months. This ensures that 95% of the outcomes complete in 12 months.

3. Option 3: This is the usual place we all end up and the project managers that succeed work with their team to shrink the variance of the distribution. You must address and reduce the uncertainties in the scope, the design, the plans, the team, the platform, and the process. The effect of eliminating uncertainty is less dispersion in the distribution and consequently a higher probability of delivering within the target date.

The first two options are usually deemed unacceptable, leaving the third option as the only alternative—and the foundation of most of the iterative and agile delivery best practices that have evolved in the software industry. If you examine the best practices for requirements management, use case modeling, architectural modeling, automated code production, change management, test management, project management, architectural patterns, reuse, and team collaboration, you will find methods and techniques to reduce uncertainty earlier in the life cycle. If we retrospectively examine my top 10 principles of iterative development, one can easily conclude that many of them (specifically 1, 2, 3, 6, 8, and 9) make a significant contribution to addressing uncertainties earlier. The others (4, 5, 7 and 10) are more concerned with establishing feedback control environments for measurement and reporting. It was not obvious to me that the purpose of these principles was also to reduce uncertainty until I read Douglass Hubbard's book *How to Measure Anything,*[13] where I rediscovered the following definition:

> Measurement: A set of observations that reduce uncertainty where the result is expressed as a quantity.

Voila! The scientific community does not look at measurement as completely eliminating uncertainty. Any significant reduction in uncertainty is enough to make a measurement valuable. With that context, I concluded that the primary discriminator of software delivery best practices was that they effectively reduce uncertainty and thereby increase the probability of success—even if success is defined as cancelling a project earlier so that wasted cost was minimized. What remains to be assessed are how much better these practices work in various domains and how do we best instrument them. IBM research continues to invest in these important questions.

Achieving "Agility at Scale": Top 10 principles of Agile software delivery

After ten years of experience with iterative development projects, we have experience from 100s of projects to update our management principles. The transitional mix of disciplines promoted in iterative development needs to be updated to the more advanced economic disciplines of agile software delivery. What follows is my proposed top ten principles for achieving agile software delivery success.

Top 10 Management Principles of Agile Software Delivery

1. Reduce uncertainties by addressing architecturally significant decisions first.
2. Establish an adaptive life-cycle process that accelerates variance reduction.
3. Reduce the amount of custom development through asset reuse and middleware.

4. Instrument the process to measure cost of change, quality trends, and progress trends.
5. Communicate honest progressions and digressions with all stakeholders.
6. Collaborate regularly with stakeholders to renegotiate priorities, scope, resources, and plans.
7. Continuously integrate releases and test usage scenarios with evolving breadth and depth.
8. Establish a collaboration platform that enhances teamwork among potentially distributed teams.
9. Enhance the freedom to change plans, scope and code releases through automation.
10. Establish a governance model that guarantees creative freedoms to practitioners.

Successfully delivering software products in a predictable and profitable manner requires an evolving mixture of discovery, production, assessment, and a steering leadership style. The word "steering" implies active management involvement and frequent course-correction to produce better results. All stakeholders must collaborate to converge on moving targets, and the principles above delineate the economic foundations necessary to achieve good steering mechanisms. Three important conclusions that can be derived from these principles and practical experience are illustrated in Figure 4.

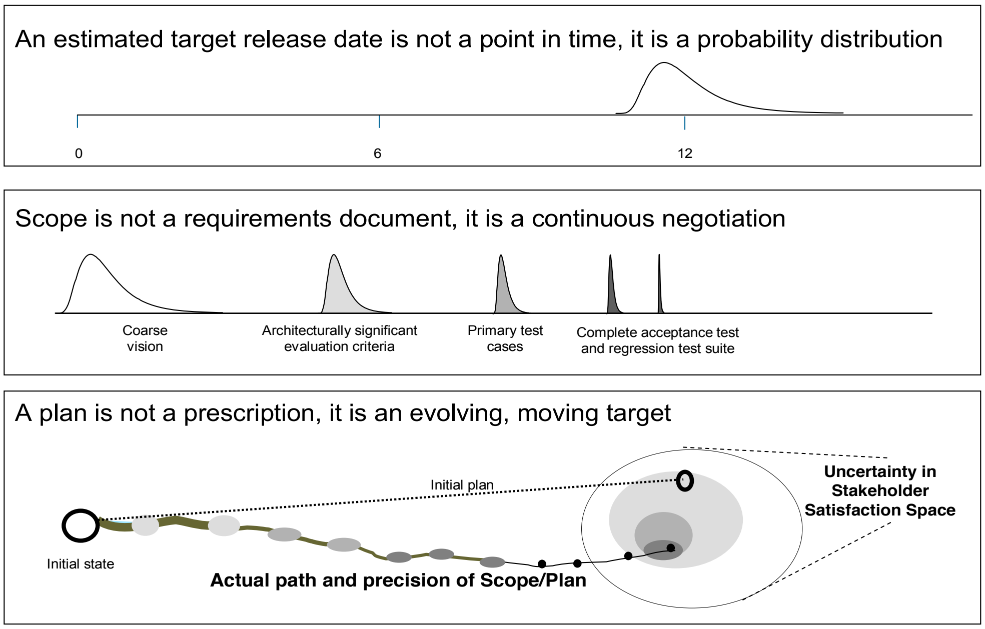

An estimated target release date is not a point in time, it is a probability distribution
0
6
12
Scope is not a requirements document, it is a continuous negotiation
Coarse vision
Architecturally significant evaluation criteria
Primary test cases
Complete acceptance test and regression test suite
A plan is not a prescription, it is an evolving, moving target
Initial plan
Uncertainty in Stakeholder Satisfaction Space
Initial state
Actual path and precision of Scope/Plan

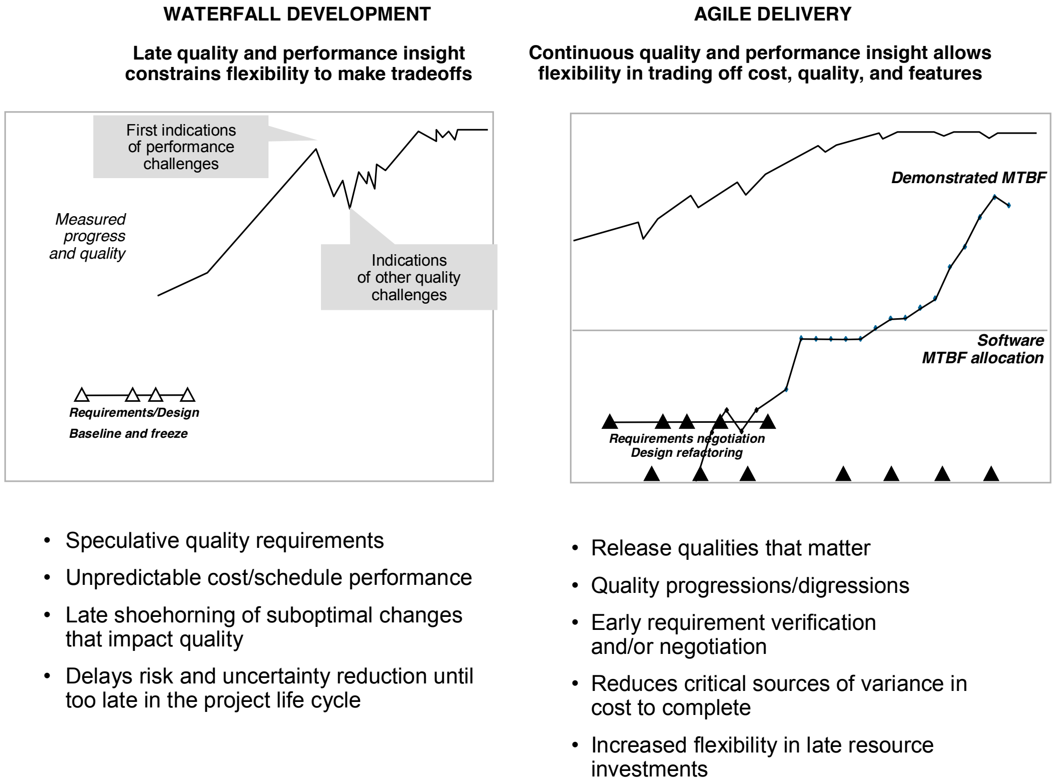
Figure 4: The governance of Agile software delivery means managing uncertainty and variance through steering

In a healthy software project, each 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.

The difference between precision and accuracy in the context of software management is not trivial. 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 estimates, risks, and the effects of change. Precision implies repeatability or elimination of uncertainty. Unjustified precision — in requirements or plans — has proved to be a substantial yet subtle recurring obstacle to success. Most of the time, this early precision is just plain dishonest and serves to provide a counter-productive façade for portraying illusory progress and quality. Unfortunately, many sponsors and stakeholders demand this early precision and detail because it gives them (false) comfort of the progress achieved.

Iterative development processes have evolved into more successful agile delivery processes by improving the navigation through uncertainty with balanced precision. This steering requires dynamic controls and intermediate checkpoints, whereby stakeholders can assess what they have achieved so far, what perturbations they should make to the target objectives, and how to re-factor what they have achieved to adjust and deliver those targets in the most economical way. The key outcome of these modern agile delivery principles is increased flexibility, which enables the continuous negotiation of scope, plans, and solutions for effective economic governance.

Figure 5 provides another example of this important metric pattern. What this figure illustrates is the tangible evolution of a quality metric (in this case, the demonstrated mean time between failure for the software embedded in a large scale command and control system[14]). Whereas, the conventional process would have to deal speculatively with this critical performance requirement for most of the lifecycle, the project that employs a modern agile delivery approach eliminates the uncertainty in achieving this requirement early enough in the project's schedule that the team can effectively trade-off remaining resources to invest in more run-time performance, added functionality, or improved profit on system delivery. This sort of reduction in uncertainty has significant economic leverage to all stakeholders.



WATERFALL DEVELOPMENT

Late quality and performance insight constrains flexibility to make tradeoffs

- Speculative quality requirements
- Unpredictable cost/schedule performance
- Late shoehorning of suboptimal changes that impact quality
- Delays risk and uncertainty reduction until too late in the project life cycle

AGILE DELIVERY

Continuous quality and performance insight allows flexibility in trading off cost, quality, and features

- Release qualities that matter
- Quality progressions/digressions
- Early requirement verification and/or negotiation
- Reduces critical sources of variance in cost to complete
- Increased flexibility in late resource investments

Figure 5: Reduced uncertainty in critical quality requirements improves the variance in the cost to complete and adds flexibility in downstream resource investments.

I have observed four discriminating patterns that are characteristic of successful agile delivery projects. These patterns represent a few “abstract gauges” that help the steering process to assess scope management, process management, progress management, and quality management. My hunch is that most project managers certified in traditional engineering project management will react negatively to these notions, because they run somewhat counter to conventional wisdom.

1. **Scope evolves:** Solutions evolve from stakeholder needs, and stakeholder needs evolve from available solutions assets. [Anti-pattern: Get all the requirements right up front.] This equal and opposite interaction between user need and solution is the engine for iteration that is driving more and more asset-based development. We just don't build many applications dominated by custom code development anymore. A vision statement evolves into interim evaluation criteria which evolve into test cases and finally detailed acceptance criteria. Scope evolves from abstract and accurate representations into precise and detailed representations as stakeholder understanding evolves (i.e., uncertainty is reduced).

2. **Process rigor evolves**: Process and instrumentation evolve from flexible to rigorous as the life-cycle activities evolve from early, creative tasks to later production tasks. [Anti-pattern: Define the entire project's life-cycle process as light or heavy.] Process rigor should be much like the force of gravity: the closer you are to a product release, the stronger the influence of process, tools, and instrumentation on the day-to-day activities of the workforce. The farther you are from a release date, the weaker the influence. This is a key requirement to be fulfilled by the development platform with automation support for process enactment if practitioners are to perceive a life-cycle process that delivers 'painless governance'.

3. **Progress assessment is honest**: Healthy projects display a sequence of progressions and digressions. [Anti-pattern: consistently progressing to 100% earned value as the original plan is executed, without any noticeable digression until late in the life cycle.] The transition to a demonstration-driven life cycle results in a very different project profile. Rather than a linear progression (often dishonest) of earned value, a healthy project will exhibit an honest sequence of progressions and digressions as they resolve

uncertainties, re-factor architectures and scope, and converge on an economically governed solution.

4. **Testing is the steering mechanism**: Testing of demonstrable releases is a full life-cycle activity and the cost of change in software releases improves or stabilizes over time. [Anti-pattern: testing is a subordinate, bureaucratic, late life-cycle activity and the cost of change increases over time]. Testing demands objective evaluation through *execution* of software releases under a controlled scenario with an expected outcome. In an agile delivery process that is risk-driven, integration testing will mostly precede unit testing and result in more flexibility in steering with more favorable cost of change trends.

With immature metrics and measures, software project managers are still overly focused on playing defense and struggling with subjective risk management. With further advances in software measurement and collaborative platforms that support process enactment of best practices and integrated metrics collection and reporting, we can manage uncertainty more objectively. Software project managers can invest more in playing offense through balancing risks with opportunities, and organizations can better exploit the value of software to deliver better economic results in their business.

A framework for reasoning about improving software economics

Today's empirical software cost estimation models (like COCOMO II, SEER, QSM Slim and others) allow users to estimate costs to within 25-30 percent, on three out of four projects[15]. This level of unpredictability in the outcome of software projects is a strong indication that software delivery and governance clearly requires an economics discipline that can accommodate high levels of uncertainty. These cost models include dozens of parameters and techniques for estimating a wide variety of software development projects. For the purposes of this discussion, I will simplify these estimation models into a function of four basic parameters:

1. **Complexity**. The complexity of the software is typically quantified in units of human-generated stuff and its quality. Quantities may be assessed in lines of source code, function points, use-case points, or other measures. Qualities like performance, reuse, reliability, and feature richness are also captured in the complexity value. Simpler and more straightforward applications will result in a lower complexity value.

2. **Process**. This process exponent typically varies in the range 1.0 to 1.25 and characterizes the governance methods, techniques, maturity, appropriateness, and effectiveness in converging on wins for all stakeholders. Better processes will result in a lower exponent.

3. **Teamwork**. This parameter captures the skills, experience, motivations and know-how of the team along with its ability to collaborate toward well-understood and shared goals. More effective teams will result in a lower multiplier.

4. **Tools**. The tools parameter captures the extent of process automation, process enactment, instrumentation and team synchronization. Better tools will result in a lower multiplier.

The relationships among these parameters in modeling the estimated effort can be expressed as follows:

$$\text{Resources} = (\text{Complexity})^{(\text{Process})} * (\text{Teamwork}) * (\text{Tools})$$

By examining the mathematical form of this equation and the empirical data in the various models and their practical application across thousands of industry projects, one can easily demonstrate that these four parameters are in priority order when it comes to the potential economic leverage. In other words, a 10% reduction in complexity is worth more than a 10% improvement in the process, which is worth more than a 10% more capable team, which is worth more than a 10% increase in automation. In practice, this is exactly what IBM services teams have learned over the last twenty-five years of helping software organizations improve their software development and delivery capability.

We have been compiling best practices and economic improvement experiences for years. 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 of the econometrics introduced earlier. Figure 6 summarizes the rough ranges of productivity impact and timeframes associated with many of the more common initiatives that IBM is investing in and delivering every day across the software industry. The impact on productivity typically affects only a subset of project and organization populations—they require savvy tailoring to put them into a specific context. As the scale of an organization grows, the impacts dampen predominantly because of standard inertia -- i.e., resistance to change.

We have been careful to present ranges and probability distributions to ensure that it is clear that "your mileage may vary." The key message from Figure 6 is that there is a range of incremental improvements that can be achieved and there is a general hierarchy of impact. The more significant improvements, like systematic reduction in complexity and major process transformations, also require the more significant investments and time to implement. These tend to be broader organizational initiatives. The more incremental process improvements, skill improvements, and automation improvements targeted at individual teams, projects, or smaller organizations are more predictable and straightforward to deploy.

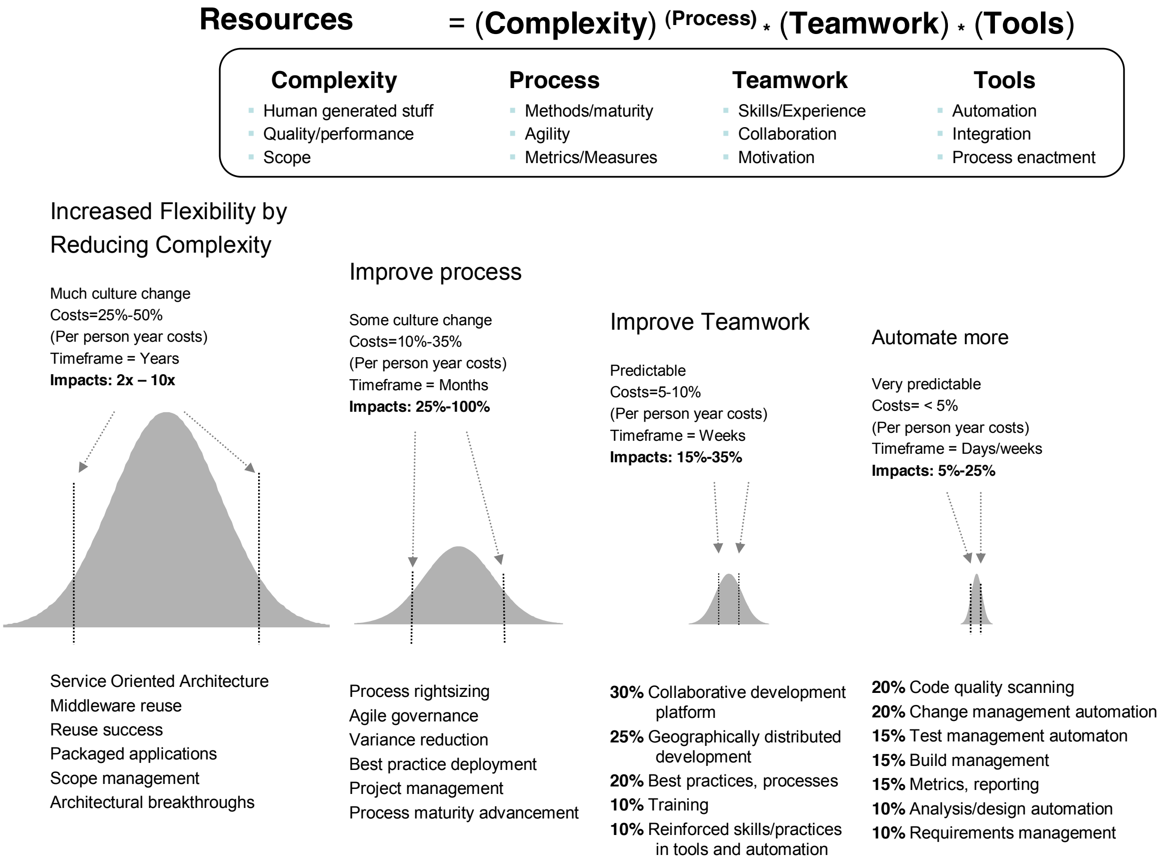


Figure 6: A rough overview of expected improvements for some best practices

The main conclusion that one can draw from the experience captured in Figure 6 is that ***improvements in each dimension have significant returns on investment.*** The key to substantial improvement in business performance is a balanced attack across the four basic parameters of the simplified software cost model: reduce complexity, streamline processes, optimize team contributions, and automate with tools. There are significant dependencies among these four dimensions of improvement. For example, new tools enable complexity reduction and process improvements; size reduction leads to process changes; collaborative platforms enable more effective teamwork; and process improvements drive tool advances. At IBM, and in our broad customer base of software development organizations, we have found that the key to achieving higher levels of improvements in teamwork, process improvement, and complexity reduction lies in supporting and reinforcing tooling and automation. Deploying best practices and changing cultures is more straightforward when you can systematically transform ways of working. This is done through deployment of tangible tools, which automate and streamline the best practices and are embraced by the practitioners, because these tools *increase* the practitioner’s creative time spent in planning, analysis, prototyping, design, refactoring, coding, testing and deploying, while these tools *decrease* the time spent on unproductive activities such as unnecessary rework, change propagation, traceability, progress reporting, metrics collection, documentation, and training.

I realize that listing training among the unproductive activities will raise the eyebrows of some people. Training is an organizational responsibility, not a project responsibility. Any project manager who bears the burden of training people in processes, technologies, or tools is worse off than a project manager with a fully trained work force. A fully trained work force on every project is almost never possible, but employing trained people is always better than employing untrained people, other things being equal. In this sense, training is considered a non-value-added activity.

This is one of the fundamental dilemmas that organizations face as they try to improve in any one of the four dimensions. The overhead cost of training their teams on new things is a significant inhibitor to project success; this cost explains many managers' resistance to any new change initiative, whether it regards new tools, practices, or people.

In making the transition to new techniques and technologies, there is always apprehension and concern about failing, particularly by project managers who are asked to make significant changes in the face of tremendous uncertainty. Maintaining the status quo and relying on existing methods is usually considered the safest path. In the software industry, where most organizations succeed on less than half of their software projects, maintaining the status quo is not a safe bet. When an organization does decide to make a transition, two pieces of conventional wisdom are usually offered by both internal champions and external change agents: (1) Pioneer any new techniques on a small pilot program. (2) Be prepared to spend more resources – money and time – on the first project that makes the transition. In my experience, both of these recommendations are counterproductive.

Small pilot programs have their place, but they rarely achieve any paradigm shift within an organization. Trying out a new little technique, tool, or method on a very rapid, small-scale effort – less than three months, say, and with just a few people – can frequently show good results, initial momentum, or proof of concept. The problem with pilot programs is that they are almost never considered on the critical path of the organization. Consequently, they do not merit "A" players, adequate resources, or management attention. If a new method, tool, or technology is expected to have an adverse impact on the results of the trailblazing project, that expectation is almost certain to come true. Why? Because software projects almost never do better than planned. Unless there is a very significant incentive to deliver early (which is very uncommon), projects will at best steer their way toward a target date. Therefore, the trailblazing project will be a non-critical project, staffed with non-critical personnel of whom less is expected. This adverse impact ends up being a self-fulfilling prophecy.

The most successful organizational paradigm shifts I have seen resulted from similar sets of circumstances: the organizations took their most critical project and highest caliber personnel, gave them adequate resources, and demanded better results on that first critical project.

Conclusion

Day-to-day decisions in software projects have always been, and continue to be, dominated by decisions rooted in the tradition of economics discipline, namely: value judgments, cost tradeoffs, human factors, macro-economic trends, technology trends, market circumstances, and timing. Software project activities are rarely concerned with engineering disciplines such as mathematics, material properties, laws of physics, or established and mature engineering models. The primary difference between economics and engineering governance is the amount of uncertainty inherent in the product under development. The honest treatment of uncertainty is the foundation of today's best practices; we have learned over and over that what makes a software practice better or best is that the practice reduces uncertainty in the target outcome.

Here are four concluding thoughts that summarize the main themes of this paper:

1. Agile software delivery is better served by economic governance principles. With software delivery becoming a more dominant business process in most product, systems, and services companies, the predictability and track record of applying conventional engineering principles to managing software won't be competitive.

2. Our top ten principles of agile software delivery have a common theme: They describe "economic governance" approaches that attack uncertainties and reduce the variance in the estimate to complete.

3. The primary metric for demonstrating that an organization or project has transitioned to effective agile delivery is the trend in the cost of change. This measure of the adaptability inherent in software releases is a key indicator of the flexibility required to continuously navigate uncertainties and steer projects toward success.

4. The next wave of technological advances to improve the predictability and outcomes of software economics needs to be in measurement and instrumentation that supports better economic governance.

IBM, and the Rational organization in particular, will continue to invest in research, practices, measures, instrumentation, and tools to advance our knowledge and practice of software economic governance, so that our customers can exploit a mature business process for agile software delivery.

References

1. Royce, Bittner, Perrow, *The Economics of Software Development*, Addison-Wesley, 2009.
2. Royce, Walker, "Successful Software Management Style: Steering and Balance," IEEE Software, Vol. 22, No. 5, September/October 2005
3. Royce, Winston W., "Managing the Development of Large Software Systems," IEEE Wescon, 1970.
4. Kruchten, Philippe, *The Rational Unified Process: An Introduction*, Addison-Wesley, 1999, 2003.
5. Kruchten, Philippe, Kroll, Per, *The Rational Unified Process Made Easy: A Practitioner's Guide to the RUP*, Addison-Wesley, 2003.
6. Hubbard, Douglass W., *How to Measure Anything*, John Wiley and Sons, 2007.
7. Williams, Laurie, Kessler, Robert. *Pair Programming Illuminated*, Addison Wesley, 2003.
8. Williams, L., Krebs, W., Layman, L., Anton, A., "Toward a Framework for Evaluating Extreme Programming, Empirical Assessment in Software Engineering," (EASE), 2004.
9. Austin, Robert, Devin, Lee, *Artful Making*, FT Press, 2003.

Endnotes

[1] Royce, Bittner, Perrow, *The Economics of Software Development*, Addison-Wesley, 2009.

[2] Royce, Walker, "Successful Software Management Style: Steering and Balance," IEEE Software, Vol. 22, No. 5, September/October 2005

[3] Williams, Laurie, Kessler, Robert. *Pair Programming Illuminated*, Addison Wesley, 2003.

[4] Williams, L., Krebs, W., Layman, L., Anton, A., "Toward a Framework for Evaluating Extreme Programming, Empirical Assessment in Software Engineering," (EASE), 2004.

[5] Royce, Walker E., *Software Project Management,* Addison Wesley, 1998.

[6] Royce, Winston W., "Managing the Development of Large Software Systems," *IEEE Wescon*, 1970.

[7] Royce, Op cit.

[8] Royce, Op cit.

[9] Royce, Op cit.

[10] Appendix D in my book *Software Project Management* provides a large scale case study of a DoD project that achieved the cost of change pattern on the right side of Figure 2.

[11] Kruchten, Philippe, *The Rational Unified Process: An Introduction*, Addison-Wesley, 1999, 2003.

[12] The variance of a random variable (i.e., a probability distribution or sample) is a measure of statistical dispersion. Technically, variance is defined as the average of the squared distance of all values from the mean. The mean describes the expected value and the variance represents a measure of uncertainty in that expectation. The square root of the variance is called the standard deviation and is a more accepted measure since it has the same units as the random variable.

[13] Hubbard, Douglass W., *How to Measure Anything*, John Wiley and sons, 2007.

[14] Royce, Op cit.

[15] Boehm, Barry. *Software Cost Estimation with COCOMO II*, Prentice Hall PTR, 2000.

IBM Corporation
Software Group
Route 100
Somers, NY 10589
U.S.A.

Produced in the United States of America
May 2009

References in this publication to IBM products or services do not imply that IBM intends to make them available in all countries in which IBM operates.

The information contained in this document is provided for informational purposes only. While efforts were made to verify the completeness and accuracy of the information contained in this documentation, it is provided "as is" without warranty of any kind, express or implied. In addition, this information is based on IBM's current product plans and strategy, which are subject to change by IBM without notice. IBM shall not be responsible for any damages arising out the use of, or otherwise related to, this documentation or any other documentation. Nothing contained in this document is intended for, nor shall have the effect of, creating any warranties or representations from IBM (or its suppliers or licensors), or altering the terms and conditions of the applicable license agreement governing the use of IBM software.