from The Rational Edge: This article introduces some comparisons between managing a software production and managing a movie production, and then discusses four software management practices observed from successful projects.

Managing software projects successfully has proven to be very failure prone when using the traditional engineering management discipline. Comparing the challenge of software management to that of producing a major motion picture exposes some interesting perspectives. Both management problems are concerned with developing a complex piece of integrated intellectual property with constraints that are predominantly economic. This article introduces some comparisons between managing a software production and managing a movie production, then elaborates four software management practices observed from successful projects. The overall recommendation is to use a steering leadership style rather than the detailed plan-and-track leadership style encouraged by conventional wisdom.

Over the course of my professional career, I have had the opportunity to observe and assess hundreds of software projects across a wide swath of industries and applications. A significant determinant that separates the successes from the failures is the project management style used to move a project forward from inception to user acceptance.

From my experience, software project managers are more likely to succeed if they use techniques that are more like managing a movie production than an engineering production. "Heresy!" some may shout. "Software projects need more disciplined engineering management, not less." Before you dismiss my claim as an insult to the profession, consider these observations:
• 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. Some developers of embedded software are the obvious exception.

• In a software project, you can change almost anything at any time: plans, people, funding, milestones, requirements, designs, tests. Requirements -- probably the most misused word in our industry -- rarely describe anything that is truly required. Nearly everything is negotiable.

• Metrics and measures for software products have no atomic units. Economic performance more typical in service industries (value as observed by the users vs. cost of production) has proven to be the best measure of success. Most aspects of quality are very subjective, such as maintainability, reliability, and usability.

These observations are also quite applicable to movie producers, professionals who regularly create a unique and complex web of intellectual property bounded only by a vision and human creativity. I hypothesize that the economic performance of movie production looks pretty similar to the economic performance of software projects: Since 2000, about one in three delivers on budget and on schedule with any sort of predictability.¹ I'm sure the observations sound countercultural to project managers who use engineering mindsets to produce airplanes, bridges, heart transplant valves, nuclear reactors, skyscrapers, and satellites (unless these engineering projects include significant software content or are unprecedented, first-of-a-kind systems).

OK, one might say, software is simply an immature engineering discipline. In my professional career, the technologies underlying software projects have turned over about every five years. This includes languages, architectural patterns, applications, user interfaces, computing platforms, networking, environments, processes, and tools. Is this rapid, continuing evolution going to slow down? Not in the foreseeable future. The human imagination and market forces are far too strong.

Like the movie industry, we do need qualified architects (directors), analysts (scriptwriters, designers), software engineers (production crews, editors, special effects producers, actors, stunt doubles), and project managers (producers). Also like the movie industry, we must get increments of software into executable form (get it on film) to make things tangible enough to assess progress and quality. There is a lot of scrap and rework in this process as we discover what works and what does not, and we synthesize the contributions of many people into one cohesive piece of integrated intellectual property.

What I have learned is that software management is more accurately described as a discipline of software economics rather than software engineering. Day-to-day decisions by software managers (like those of movie producers) are dominated by value judgments, cost tradeoffs, human factors, macro-economic trends, technology trends, market strength, and timing. Software projects are rarely concerned with precise mathematics, material properties, laws of physics, or established and mature engineering tenets.

I hope this software-movie comparison has hit a nerve and motivated you to read on. For more elaboration on such comparisons see the essays of Paul Graham.² The remainder of this article focuses on a few assertions derived from seeing handfuls of successful software projects and
boatloads of unsuccessful software projects during my 25+ years of following, leading, practicing, and advising in the software project management trenches.

The iterative approach

One reason for the low success rate of software-intensive projects is that traditional project management approaches do not encourage the steering and adjustment needed to reconcile significant levels of uncertainty in:

- The problem space (what the user really wants or needs)
- The solution space (what architecture and technology mix is most appropriate)
- The planning space (cost and time constraints, team composition and productivity, stakeholder communication, incremental result sequences, etc.)

We need a more dynamic and adaptive way of thinking about software progress and quality management, one that accommodates patterns of successful projects. Today's modern software management approaches are generally known as iterative development methods. Rather than tracking against a precise, long-term plan, the iterative method steers software projects through the minefield of uncertainties inherent in developing today's software applications, products, and services. Successfully delivering software products on schedule and on budget 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.

The IBM Rational Unified Process, a well-accepted benchmark of a modern iterative development process, provides a framework for a more balanced evolution that encourages the management of uncertainty and risk. Its life cycle includes four phases, each with a demonstrable result:

1. Inception: definition and prototype of the vision and business case
2. Elaboration: synthesis, demonstration, and assessment of an architecture baseline
3. Construction: development, demonstration, and assessment of useful increments
4. Transition: usability assessment, final production, and deployment

The phase names represent the state of the project rather than a sequential-activity-based progression from requirements to design to code to test to delivery.

We call this iterative management style results-based rather than activity-based. In the world of software, real results are executable programs. Everything else (requirements documents, use case models, design models, test cases, plans, processes, documentation, inspections) is secondary, and simply part of the means to the end: an executable software program. Think back to your programmer days: When you were building a model, sketching a flowchart, reasoning through logic of a state machine, or composing source code, you knew you were simply speculating and synthesizing an abstract solution. It wasn't very tangible until you got it to compile, link, and execute; then you could truly reason about its quality, performance, usefulness, and completeness. Project managers should feel the same way. As long as you are assessing the beauty or goodness of a plan, model, document, or some other nonexecutable representation,
you are only speculating about quality and progress. Movie producers feel the same way about scripts, storyboards, set mockups, and costume designs. They commit scenes to film to make the presentation tangible enough that they can judge its overall integrated effect.

**Precision vs. accuracy**

In a successful 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 this 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 as trivial as it may seem. 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. Unjustified precision -- in requirements or plans -- has proven to be a substantial yet subtle recurring obstacle to success. Most of the time, early precision is just plain dishonest and serves to provide a facade for more progress or more quality than actually exists. Unfortunately, many sponsors and stakeholders demand this early precision and detail because it gives them (false) comfort with respect to progress achieved.

One of the most common failure patterns I have observed in the software industry is developing a five-digits-of-precision specification when the stakeholders have only a one-digit-of-precision understanding of the problem, solution, or plan. A prolonged effort to build a precise requirements understanding or a detailed plan only delays a more thorough understanding of the architecturally significant issues. How many frighteningly thick requirements documents or micromanaged inch-stone plans have you worked on, perfected, and painstakingly reviewed, only to totally overhaul them months later after the project achieved a meaningful milestone of demonstrable capability that accelerated stakeholder understanding of the real tradeoffs? This common practice is aptly known in our trade as turd polishing.

**Some patterns of successful steering**

Iterative processes have evolved mostly from the need to navigate better through uncertainty and manage software risks. This steering requires dynamic controls and intermediate checkpoints where the stakeholders can assess what they have achieved so far, what perturbations they should make to the target objectives, and how to refactor what they have achieved to get to those targets in the most economical way. I have observed four patterns that are characteristic of successful software-intensive projects. These patterns represent a few "abstract gauges" that help the steering process to assess scope management, process control, progress, and quality control. My hunch is that most project managers certified in project management will react negatively to these notions, because they run somewhat counter to conventional wisdom.

1. **Scope management:** Solutions evolve from user specifications, and user specifications evolve from candidate solutions. As opposed to getting all the requirements right up front.
2. Process control: Process and instrumentation rigor evolves from light to heavy. **As opposed to defining the entire project's life-cycle process as light or heavy.**

3. Progress: Healthy projects display a sequence of progressions and digressions. **As opposed to progressing to 100% earned value as the predicted plan becomes fulfilled, without any noticeable digression.**

4. Quality control: Testing of demonstrable releases is a first-class, full life-cycle activity. **As opposed to a subordinate, later life-cycle activity.**

I admit that each is easier said than done on a software project of substance, and certainly they need to be applied differently across domains. Web application development teams will implement these patterns differently than embedded application development teams, but the pattern still applies. Writing books and papers about methods and patterns and techniques, which the industry calls **thought leadership**, is relatively easy. However, most of us are not looking for best thoughts, we are looking for best practices. Managing real projects requires **practice leadership** where applying and executing these ideas under game conditions is relatively difficult. We need to cherish proven project managers who understand practice leadership: They are probably the scarcest resource in every company. Good project managers, like good movie producers, not only create good products time after time, they also serve as mentors for younger team members who can learn effective techniques and develop their own skills in multi-dimensional tradeoffs, continuous negotiation, risk management, pattern recognition, and a steering leadership style. Project management training courses are good catalysts for learning, but apprenticeship is a necessity.

**Scope management**

One of the more subtle challenges in the conventional software process has been in presenting the life cycle as a sequential thread of activities: from requirements analysis to design, to code, to test, and finally to delivery. In an abstract way, successful projects do implement this progression, but the boundaries between the activities are fuzzy and are accepted as such by collaborative stakeholders. Unsuccessful projects, on the other hand, have typically gotten mired in striving for crisp boundaries between activities. For example, the pursuit of a 100% frozen requirements baseline before the transition to design, or for fully detailed design documentation before transitioning to coding, are strict interim goals that resulted in wasteful attention to minutia while progress on the important engineering decisions slowed or even stopped.

When we build software solutions that are 100% new stuff, the flow of specifications from requirements to design in successive levels of detail makes some sense. But today, we are usually upgrading an existing system or building new systems out of existing components (operating systems, web services, networks, GUIs, data management, packaged applications, middleware, computing platforms, legacy systems). The economic benefits of adapting or reusing existing assets force us to think about specifying the user need within the context and constraints of these existing assets.

Earlier, I said that **requirements** is probably the most misused word in our industry. **Required** means non-negotiable, yet we see requirements being changed and bartered and negotiated on almost every successful project. Changing a requirement receives tremendous scrutiny because
it usually has an impact on the contract between stakeholders. I propose that we use the word specification instead. Specifications are changeable and are understood as the current state of our understanding.

From what I have seen, there are two important forms of user specifications. The first is the vision statement (or user need), which captures the contract between the development group and the buyer or user. This information should be represented in a format that is understandable to the user (an ad hoc format that may include text, mockups, use cases, spreadsheets, or whatever). A use case model that supports the vision statement serves to capture the operational concept and expected behaviors in terms the user or buyer will understand.

The second form of specification is very different from requirements. I prefer the term evaluation criteria, which are inherently understood as transient snapshots of objectives for a given intermediate life-cycle checkpoint such as a demonstrable release. Evaluation criteria are interim steering targets derived from the vision statement as well as from many other sources (make/buy/reuse analyses, risk management concerns, architectural considerations, shots in the dark, implementation constraints, quality thresholds, and so on). They, along with use case models, provide a better framework for early testing than do conventional requirements representations. An initial proposal for the sequence of planned release content and planned evaluation criteria serves as a great format for a risk management plan.

**Process rigor**

For years, I have tried to reconcile the zealous arguments of the agile camps (the liberal left of software process opinions) and the process maturity camps (the conservative right of software process opinions). They both have useful perspectives and appropriate techniques. But as with most sophisticated endeavors, there is no clear right or wrong prescription for the range of solutions needed. Context is extremely important, and almost any nontrivial project or organization needs to combine technique, common sense, and domain experience to be successful. Most project managers would agree with the following:

<table>
<thead>
<tr>
<th>When is less process rigor appropriate?</th>
<th>When is more process rigor appropriate?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Co-located teams</td>
<td>Distributed teams</td>
</tr>
<tr>
<td>Smaller, simpler projects</td>
<td>Large projects (teams of teams)</td>
</tr>
<tr>
<td>Few stakeholders</td>
<td>Many stakeholders</td>
</tr>
<tr>
<td>Adaptable quality</td>
<td>Extreme quality specifications</td>
</tr>
<tr>
<td>Internally imposed constraints</td>
<td>Externally imposed constraints (Standards, liability, contracts, etc.)</td>
</tr>
<tr>
<td>We need to add one more item to the lists:</td>
<td></td>
</tr>
<tr>
<td>Early life-cycle phases</td>
<td>Later life-cycle phases</td>
</tr>
</tbody>
</table>

In my view, this last perspective describes the most critical determinant for deciding between the speed and freedom of agile methods and the quality and prescriptive guidance of rigorous methods. **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 axiom seems to be completely missing, or at least grossly underemphasized, by the process evangelists and literature, but it is usually very observable in successful software projects.

A discriminating characteristic of most successful software development processes is the well-defined transition from the creative phases (Inception and Elaboration) to the production phases (Construction and Transition.) When software projects do not succeed, a primary reason is an inappropriate emphasis (either too much or too little) on process rigor. This is true for both conventional and iterative processes. Most unsuccessful projects exhibit one of these characteristics:

- Over-engineering on the early phases (creative aspects) of the life cycle. You need maneuverable processes that easily adapt to discovery and accommodate a degree of uncertainty to attack a few major risk items, prototype solutions, and build early and coarse artifacts. What creative discipline can you think of in which more process rigor is considered beneficial in helping humans think?
- Under-engineering on the later phases (production aspects) of the life cycle. Extensive change-managed baselines of detailed and elaborate artifacts need engineering processes with insightful instrumentation and attention to detailed consistency and completeness to converge on a quality product.

Successful modern projects -- and successful projects developed under the conventional process -- tend to have a well-defined project milestone, a noticeable transition from a creative research attitude to a production attitude. Earlier phases focus on achieving demonstrable functionality. Later phases focus on achieving a product that can be shipped to a customer, with explicit attention to robustness, performance, fit, and finish.

Another important aspect is the effect on the overall team of the transition from a creative world into a production world. Good design teams are usually repelled by process rigor, details, and premature precision. Good production teams are usually offended by loose, fluid, and coarse results. Project managers need to manage the balance of the various teams so that the center of gravity for technical leadership evolves across the life cycle from the management team in inception, to the architecture team in elaboration, to the development team in construction, to the test/assessment team in transition. The human aspects of software project management are underappreciated, and the topic of team dynamics deserves a much more thorough treatment than most project management courses offer today.

**Progress**

Many aspects of the classic development process cause stakeholder relationships to degenerate into mutual distrust. Trust is essential to steering and negotiating a balance among user needs, product features, and plans. A more iterative model, with more effective communications between stakeholders (enabled by a sequence of demonstrable releases), allows tradeoffs to be based on a more objective understanding by all. This requires customers, users, and monitors to become focused on the delivery of a usable system rather than religiously enforcing standards and contract terms. It also requires a development organization to be focused on delivering value in a profitable manner.
An iterative process requires sequential construction of a progressively more complete system that demonstrates the architecture, enables objective requirements negotiations, validates the technical approach, and addresses resolution of key risks. Ideally, all stakeholders are focused on these milestones as incremental deliveries of useful functionality, as opposed to speculative paper descriptions of a final vision. 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.

Here are two related observations for which I have never seen counter-examples:

1. A software project that has a consistently increasing earned value profile is certain to have a pending cataclysmic regression.
2. Healthy software projects demonstrate a sequence of increasing progressions and decreasing digressions as they resolve uncertainties and converge on an acceptable solution.

Ambitious demonstrations are excellent milestones on a healthy project's path. The purpose of early life-cycle demonstrations is to expose design flaws, not to put up a facade. Stakeholders must not overreact to early mistakes, digressions, or immature designs. If early engineering phases are overconstrained, development organizations will set up intermediate checkpoints that are less ambitious. Early increments will be immature. External stakeholders such as customers and users cannot expect initial deliveries to perform up to final delivery specifications -- that is, to be complete, fully reliable, or have end-target levels of quality or performance.

On the other hand, development organizations must be held accountable for, and demonstrate, tangible improvements over successive increments. Objectively quantifying changes, fixes, and upgrades will provide honest indicators of progress and quality. Open and attentive followthrough is necessary to resolve issues. Good and bad project performance is much more obvious earlier in the life cycle. With a steering leadership style, success breeds success. After posting a sequence of demonstrable results, you can predict the outcome pretty well. A persistent lack of progress or a stagnant sequence of results is a sign that a project needs a serious reconsideration of resources, scope, or project worthiness. Software project experience has shown time and again that the early phases make or break a project. This is why a small, highly competent start-up team should handle the planning and architecture phases. If these early phases are done right, projects can be completed successfully, with teams of average competency evolving the applications into the final product. If the planning and architecture phases are not performed adequately, all the expert programmers and testers in the world probably will not achieve success over the course of the later phases.

**Quality control**

If you are successfully managing a project in the spirit of iterative development, most integration testing will precede component testing. Stop and think about that statement. While there is always a mixture of both activities going on at any time, you should consider initial component development and testing to be primarily a means of exercising a component's interface and function in an integrated, system-level thread or behavior. Once the interface and integrated behaviors are successfully tested, then component completeness can be tested. Driving
integration test first accelerates the resolution of architecturally significant issues into earlier phases of the life cycle. It also provides an evolving testbed for continuous assessment of system- and component-level progress and performance.

A key byproduct of the integration-first approach is that testing and testers become first-class citizens in the process. In conventional approaches, testers create speculative plans, procedures, and paper that are subordinate to the analysis and design artifacts. Their jobs and early life-cycle artifacts are insignificant indicators of project success and tend to attract the B-players in most organizations (namely, the folks who didn't cut it as first-rate analysts and designers). In healthy iterative projects, the early-life-cycle demonstrations require significant test perspectives and products. Many test teams are responsible for some of the most effective "analysis" activities and results. Too many analysts work solely in abstract model-land with limited constraints to drive their analysis. But testers are faced with building "test cases" -- real-world representations of use cases or evaluation criteria or expected behaviors. They ask a whole different set of questions and look at the world from a different perspective because their job is translating abstract things into testable things.

Here is an example. Many projects today are confronted with the make/buy tradeoffs associated with commercially available components and applications. If the first result-oriented milestone of a project is to resolve those make/buy decisions through demonstration, you would task your teams as follows:

- Analysis team: Work with the users to capture the key use cases that drive the worst-case performance conditions, such as the peak data load or most critical control scenario.
- Design team: Configure a prototype capable of exercising the candidate commercial components.
- Test team: Construct test cases (for example, a message set, a test driver, a smart stub, a populated database, a sequence of GUI actions) that reflect the key use cases and can drive the prototype and capture its response.

In achieving this first milestone, your teams may concern themselves only with two of the critical use cases (perhaps 10% of the user need), a few of the key components, and a few of the critical test cases, but they and the users will have resolved perhaps 30% of the risk very early in the life cycle. By including the testing perspective as an equal partner in the early phases of the process, you will be able to attract better people who contribute to better analysis because the work is more interesting and more effective at contributing to success.

Conventional software testing approaches follow the same document-driven approach that was applied to software development. Development teams build requirements documents, top-level design documents, and detailed design documents before constructing any source files or executables. Similarly, test teams build system test plan documents, system test procedure documents, integration test plan documents, unit test plan documents, and unit test procedure documents before building any test drivers, stubs, or instrumentation. This document-driven approach causes the same problems for the test activities that it does for the development activities: lots of turd polishing that ends up as future scrap and rework.
To drive integration testing earlier in the life cycle, the testing sequence should be organized by iteration rather than by component. Typically, it should be captured by a set of use cases and other textually represented objectives that can be meaningfully demonstrated to a user. Here is an abstract description:

- **Inception iterations**: Perhaps five to ten evaluation criteria capturing the driving issues associated with the primary use cases that have an impact on architecture alternatives and the overall business case.
- **Elaboration iterations**: Perhaps dozens of evaluation criteria that, when demonstrated against the candidate architecture, verify a solid framework for the primary use cases and demonstrate that the critical risks have been resolved.
- **Construction iterations**: Perhaps hundreds of evaluation criteria associated with some meaningful set of use cases that, when passed, constitute useful subsets of the product that can be transitioned to alpha or beta releases.
- **Transition iterations**: The complete set of use cases and associated evaluation criteria (perhaps thousands) that constitute the acceptance test criteria associated with deploying a version into operation.

A modern process also uses the same basic tools, languages, notations, and artifacts for the products of test activities that are used for product development. Testing refers to the explicit evaluation through execution of some set of components under a controlled scenario with an expected and objective outcome. The success of a test can be determined by comparing the expected outcome to the actual outcome using generally well-defined metrics of success. Tests are exactly the forms of assessment that can be largely automated and instrumented.

**Conclusions**

Figure 1 provides a project manager’s view of the improving time to value transition that we are all trying to achieve. It provides a good abstract perspective for summarizing the results of effectively implementing the steering leadership style implied by my four recommendations. I have presented three project profiles by plotting development progress versus time, where progress is defined as percent executable, that is, demonstrable in its target form. Progress in this sense really correlates to results, as I described earlier, and is best measured through executable demonstrations. Executable does not imply complete, compliant, nor up to specifications; it does imply that the software is testable.
The typical sequence for the conventional engineering project management style when measured this way is (1) early success via paper designs and thorough (often too thorough) 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, unmaintainable product, delivered late.

The modern management approach I have touched on here forces integration into the design phase through a progression of demonstrable releases, thereby forcing the architecturally significant breakage to happen earlier where it can be resolved in the context of life-cycle goals. The downstream integration nightmare is avoided, along with late patches and malignant software fixes. The result is a more robust and maintainable product delivered predictably with a higher probability of economic success.

Conventionally managed projects, mired in inefficient integration and late discovery of substantial design issues, expend roughly 40% of their total resources in integration and test activities, with much of this effort consumed in excessive scrap and rework. Modern projects with an iterative process and steering leadership style can deliver a product with about 25% of the total budget consumed by these activities.

I have discussed four key success patterns for projects that are managed with the true spirit of iterative development. Each pattern represents a dimension of balance that can help a team steer its way to the product and economic efficiencies implied by the modern and future profiles represented in the illustration:

- User needs balanced with design assets
- Creative process freedom balanced with production process rigor
- Production progressions balanced with experimental discovery digressions
- Abstract vision balanced with tangible assessment through testing
From my experience, the conventional profile in the illustration is still the norm and is characteristic of more than half the projects we see today. While most of these projects use the traditional engineering management approach, some claim to be using modern iterative development. However, without practicing a steering leadership style, they fail to deliver the business results expected. Perhaps one out of four of today’s projects is delivering on the modern profile, while one out of eight manages to operate on the target profile. It is from these more fluid profiles and successful outcomes that I have observed consistent usage of the styles discussed in this article.

Is software project management really more like managing a movie production than it is like managing the construction of a bridge? Probably not, especially in the later phases of production. But I hope the analogy provokes readers to look at software project management techniques from a different frame of reference. These patterns are not new. They have been practiced (although infrequently) in many organizations and to varying degrees across a broad set of domains. If you look deeply into all the subtle dimensions of making the patterns work in practice, you will see that they all deal with the human and teamwork aspects of management, with little science, engineering, or manufacturing bias. I think organizations that adopt a steering style of management are more likely to achieve economic success -- perhaps even a blockbuster.

Notes

*To be published in IEEE Software, September 2005. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE. This material is presented to ensure timely dissemination of scholarly and technical work. Copyright and all rights therein are retained by authors or by other copyright holders. All persons copying this information are expected to adhere to the terms and constraints invoked by each author’s copyright. In most cases, these works may not be reposted without the explicit permission of the copyright holder.

Successful software management style: Steering and balance*
About the author

Walker Royce

Walker Royce is the vice president of IBM's Worldwide Rational Brand Services. He has managed large software engineering projects and consulted with a broad spectrum of software development organizations. He is the author of *Software Project Management, A Unified Framework* (Addison-Wesley Longman, 1998) and a principal contributor to the management philosophy inherent in the Rational Unified Process. Before joining Rational and IBM, he spent sixteen years in software project development, software technology development, and software management roles at TRW Electronics & Defense. He was a recipient of TRW's Chairman's Award for Innovation for distributed architecture middleware and iterative software processes and was a TRW Technical Fellow. He received his BA in physics from the University of California, and his MS in computer information and control engineering from the University of Michigan.

© Copyright IBM Corporation 2005
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)