

Successful Software Management Style: Steering and Balance

Walker Royce, IBM Software Group

Project management style is a significant determinant separating successful projects from failures. Contrary to conventional wisdom, steering leadership is better than detailed plan-and-track leadership.

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 might shout. “Software projects need more disciplined engineering management, not less.”

Before you dismiss my claim as an insult to the profession, consider these observations (a related discussion appears elsewhere¹):

- 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. (Some embedded-software developers are an occasional exception.)
- In a software project, you can change almost anything at any time: plans, people, funding, milestones, requirements, designs, and 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 and are highly subjective. Economic performance more typical in service industries (measured by a user’s perceived value rather than the cost of production) has proven to be the best measure of success for software.

These observations probably sound counter-cultural to project managers who use engineering mindsets to produce airplanes, bridges, heart transplant valves, nuclear reactors, skyscrapers, and satellites (unless these projects include significant software content or are first-of-a-kind systems). However, they do apply to movie producers—professionals who regularly create a unique and complex web of intellectual property limited only by vision and creativity.

I prefer to describe software management as a discipline of software *economics* rather than software *engineering*. Software projects are rarely concerned with established and mature engineering tenets. Rather, a software manager’s day-to-day decisions (like those of movie producers) are dominated by value judgments, cost trade-offs, human factors, macro-economic trends, technology trends, market strength, and timing. To deal with these more subjective decisions, I recommend using a *steering* leadership

style that comprises active management involvement and frequent course correction.

An iterative approach

The economic performance of software projects is difficult to capture in one simple metric, but in the last five years, approximately one in three has delivered on budget and on schedule with any sort of predictability.² I suspect the economic performance of movie productions looks pretty similar.

Traditional project management approaches in software-intensive projects don't encourage the steering and adjustment needed to reconcile significant levels of uncertainty in the

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

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.³⁻⁵ 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 within budget requires an evolving mixture of discovery, production, assessment, and a steering leadership style. This implies active management involvement, frequent course correction to produce better results, and having all stakeholders collaborate to converge on moving targets.

The IBM Rational Unified Process, an accepted benchmark of a modern iterative development process, provides a framework for a more balanced evolution that encourages the management of uncertainty and risk.⁶ Its life cycle comprises four phases, each with a demonstrable result:

- *Inception*. Define and prototype the vision and business case.

- *Elaboration*. Synthesize, demonstrate, and assess an architecture baseline.
- *Construction*. Develop, demonstrate, and assess useful increments.
- *Transition*. Assess usability and produce and deploy product.

Each phase represents a project state rather than a sequential-activity-based progression from requirements to design to code to test to delivery.

This iterative management style is *results-*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, and inspections) is secondary—simply part of the means to the end. Just as the movie industry gets action on film, we too must get increments of software into executable form to make things tangible enough to assess progress and quality. A lot of scrap and rework exists in this process as we discover what works and synthesize the contributions of many people into one cohesive piece of integrated intellectual property.

Think back to your programmer days: When building a model, sketching a flowchart, reasoning through a state machine's logic, or composing source code, you knew you were speculating about and synthesizing an abstract solution. The solution wasn't tangible until you got it to compile, link, and execute; then you could reason about its quality, performance, usefulness, and completeness. Project managers should feel the same way. As long as you're assessing the merits of a plan, model, document, or some other nonexecutable representation, you're only speculating about quality and progress. Movie producers view scripts, storyboards, set mockups, and costume designs in the same way. They commit scenes to film to make the presentation tangible enough to judge its overall integrated effect.

Precision vs. accuracy

In a successful software project, each phase increases the stakeholders' understanding of the evolving plans, specifications, and completed solution, because each furthers a sequence of executable capabilities as well as the team's knowledge of competing objectives. At any point in the life cycle, the subordinate artifacts' precision should reflect this understanding, so it should evolve as the understanding evolves.

We need a more dynamic and adaptive way of thinking about software progress and quality management.

The difference between precision and accuracy in the context of software management isn't as trivial as it may seem.

The difference between precision and accuracy in the context of software management isn't as trivial as it may seem. Software management is full of gray areas, situation dependencies, and ambiguous trade-offs, and software managers must accurately forecast estimates, risks, and the effects of change. Unjustified precision—in requirements or plans—is a substantial yet subtle recurring obstacle. Most of the time, early precision is dishonest, providing a facade for more progress or quality than actually exists. Unfortunately, many sponsors and stakeholders demand this early precision and detail because it gives them (false) comfort regarding the progress achieved.

A common failure pattern 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 overhaul them months later after the project achieved a meaningful milestone of demonstrable capability that accelerated stakeholder understanding of the real trade-offs? This common practice is aptly known in our trade as “turd polishing.”

Four patterns for successful steering

Iterative processes have evolved mostly from the need to better navigate through uncertainty and to better manage software risks. This steering requires dynamic controls and intermediate checkpoints where the stakeholders can assess achievements, identify perturbations that should become target objectives, and refactor what they've achieved to obtain those objectives in the most economical way.

Following are four patterns (and antipatterns) characteristic of successful (and unsuccessful) software-intensive projects, which help create such checkpoints:

- *Scope management.* Solutions evolve from user specifications, and user specifications evolve from candidate solutions (antipattern: the requirements are precisely and thoroughly defined up front).
 - *Process rigor.* Process and instrumentation rigor evolves from light to heavy (antipattern: the entire project's lifecycle process is defined as light or heavy).
 - *Progress honesty.* Healthy projects display a sequence of progressions and digressions (antipattern: without any noticeable digression, projects progress to 90 percent earned value as the predicted plan is blindly executed.)
 - *Quality control.* Testing demonstrable releases is a first-class, full lifecycle activity (antipattern: testing is a subordinate, later lifecycle activity.)
- My hunch is that most conventionally certified project managers will react negatively to these notions, because they run somewhat counter to conventional wisdom. I admit that they're easier said than done on a software project of substance, and certainly we must apply each differently across domains. Web-application development teams will implement these patterns differently from 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 aren't looking for best thoughts; we're 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're probably the scarcest resource in every company. Like the movie industry, we need qualified architects (directors), analysts (scriptwriters and designers), software engineers (production crews, editors, special effects producers, actors, and stunt doubles), and, in particular, project managers (producers). Good project managers, like good movie producers, not only create good products but also serve as mentors for less experienced team members. They teach effective techniques for multidimensional trade-offs, continuous negotiation, risk management, pattern recognition, and steering leadership. 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 pre-

senting 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 implement this progression, but the boundaries between the activities are fuzzy, and collaborative stakeholders accept them as such. Unsuccessful projects, on the other hand, typically strive for crisp boundaries between activities. For example, pursuing a completely frozen requirements baseline before transitioning to design or fully detailed design documentation before transitioning to coding results in wasteful attention to minutia, while progress on the important engineering decisions slows or even stops.

When we build software solutions comprising entirely new stuff, the flow of specifications from requirements to design in successive levels of detail makes some sense. But we're 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, and so on). The economic benefits of adapting or reusing existing assets force us to consider 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 nonnegotiable, yet in almost every successful project we see changed, bartered, and negotiated requirements. A changed requirement receives tremendous scrutiny because it usually affects the contract between stakeholders. I propose using the word "specification" instead. Specifications are changeable and are understood as the current state of our understanding.

Two important forms of user specifications exist. The first is the *vision statement* (or user need), which captures the contract between the development group and the buyer or user. Developers should represent this information in a format that the user can understand (an ad hoc format that might include text, mockups, use cases, or spreadsheets). A use-case model that supports the vision statement captures 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 phrase *evaluation criteria*, which are inherently understood as transient snapshots of objectives for a

given intermediate lifecycle 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, or 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've 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). Both have useful perspectives and appropriate techniques, but a clear prescription for the range of solutions needed doesn't exist. Context is extremely important, and almost any nontrivial project or organization must combine technique, common sense, and domain experience.

Most project managers would agree that more rigorous processes are appropriate when

- the teams are distributed,
- the project is large (comprising teams of teams),
- many stakeholders are involved,
- the quality specifications are extreme,
- the constraints are externally imposed (standards, liability, contracts, and so forth), and
- the project is later in the lifecycle phases.

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 act like gravity: the closer you are to a product release, the stronger the influence of process, tools, and instrumentation on day-to-day activities. The farther you are from a release date, the weaker the influence. The literature and most process evangelists grossly underemphasize this axiom—or miss it entirely.

Successful software development processes exhibit a well-defined transition from creative to production phases. Earlier phases focus on achieving demonstrable functionality; later

Required means nonnegotiable, yet almost every successful project includes changed, bartered, and negotiated requirements.

Successful software development processes exhibit a well-defined transition from creative to production phases.

phases realize the product and thus focus on robustness and performance. When software projects fail, a primary reason (for both conventional and iterative processes) is an inappropriate emphasis on process rigor.

Over-engineering often occurs early in a software project's life cycle. However, you need maneuverable processes that can easily adapt to developers' discoveries and can accommodate a degree of uncertainty when developers attack risk items, create prototypes of solutions, and build early and coarse artifacts. Can you think of a creative discipline in which more process rigor is considered beneficial in helping humans think? Under-engineering, on the other hand, is usually a problem during the production phase. 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.

Another important aspect of a successful transition is its effect on the team. Process rigor, details, and premature precision usually repel good design teams, and loose, fluid, and coarse results usually offend good production teams. Project managers must balance the various teams so that the center of gravity for technical leadership evolves throughout the life cycle from the management team in inception, to the architecture team in elaboration, to the development team in construction, to the test and assessment team in transition. The human aspects of software project management are underappreciated, and the topic of team dynamics deserves more thorough treatment than most project management courses offer today.

Progress honesty

Many aspects of the classic development process cause stakeholder relationships to degenerate into mutual distrust. Trust is essential to steering and to negotiating a balance among user needs, product features, and plans. A more iterative model, with effective communication between stakeholders (enabled by a sequence of demonstrable releases), lets developers base trade-offs on an objective understanding by all stakeholders. So, customers, users, and contract monitors can focus on delivering a usable system rather than religiously enforcing standards and contract terms. Also, the development organization must focus on delivering value in a profitable manner.

An iterative process requires sequentially constructing a progressively more complete system that demonstrates the architecture, enables objective requirements negotiations, validates the technical approach, and resolves key risks. Ideally, all stakeholders focus 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.

Two related observations I've made are that, first of all, a software project that has a consistently increasing earned value profile is certain to have a pending cataclysmic regression. Second, 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 shouldn't 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 can't 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.

Yet development organizations must be held accountable for, and demonstrate, tangible improvements over successive increments. Objectively quantifying changes, fixes, and upgrades provides honest indicators of progress and quality. Open and attentive follow-through is necessary to resolve issues. Project performance is more obvious earlier in the life cycle. With a steering leadership style, success breeds success. After posting a sequence of demonstrable results, you can usually predict the outcome more accurately. A persistent lack of progress or a stagnant sequence of results is a sign that a project needs to reconsider its resources, scope, or worthiness. Because the early phases can make or break a project, a small, highly competent start-up team should handle the planning and architecture phases. If these early phases are

done right, projects are successful, with teams of average competency evolving the applications into the final product. If the planning and architecture phases aren't performed adequately, all the expert programmers and testers in the world probably won't succeed over the course of the later phases.

Quality control

If you're successfully managing a project in the spirit of iterative development, most integration testing will precede component testing. Although a mixture of both activities occurs throughout the life cycle, 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 you've successfully tested the interface and integrated behaviors, you can test component completeness. Early integration testing helps resolve architecturally significant issues early in the life cycle. It also provides an evolving test bed 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 papers that are subordinate to the analysis and design artifacts. Their jobs and early lifecycle 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-lifecycle 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 must build test cases—real-world representations of use cases, evaluation criteria, or expected behaviors. They ask different questions and view the world from a different perspective because they're translating abstract things into testable things.

For example, many projects today are confronted with the make-or-buy decision associated with commercially available components and applications. If a project's first result-oriented milestone is to decide through demonstration whether to make or buy, you would task your teams as follows:

- The analysis team would work with users to capture key use cases driving the worst-case performance conditions, such as the peak data load or most critical control scenario.
- The design team would configure a prototype capable of exercising the candidate commercial components.
- The test team would construct test cases (for example, a message set, test driver, smart stub, populated database, and 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 might concern themselves only with two of the critical use cases (perhaps 10 percent 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 percent of the risk early in the life cycle. By including the testing perspective as an equal partner early in the process, you can attract better testers and thus produce a better analysis, because the work is more interesting and directly relates to the project's success.

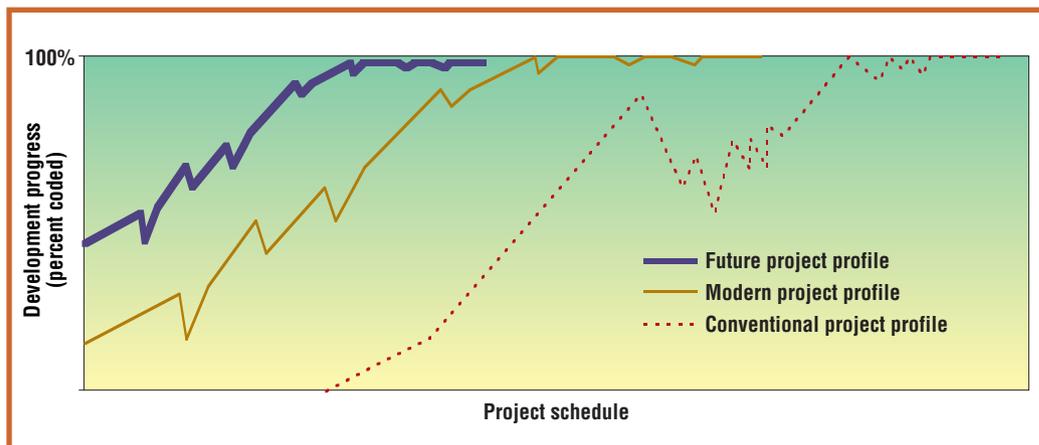
Conventional software testing approaches follow the same document-driven approach applied to software development. Development teams build requirements documents, top-level design documents, and detailed design documents before constructing any source files or executables. Similarly, test teams build 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 encourage integration testing earlier in the life cycle, organize the testing sequence by iteration rather than by component. Capture it using a set of use cases and other textually represented objectives that can be meaningfully demonstrated to a user, including

- *inception iterations*—five to 10 evaluation criteria capturing the driving issues associated with the primary use cases that affect architecture alternatives and the overall business case.

A key byproduct of the integration-first approach is that testing and testers become first-class citizens in the process.

Figure 1. Three project profiles transitioning from a plan-and-track management style (right) to a steering leadership style (left). (A conventional profile includes conventional processes, stovepipe architectures, and proprietary tools and methods; a modern profile includes iterative processes, middleware components, and mature commercial tools; and a future profile includes right-sized processes, enterprise architectures, and integrated environments.)



- *elaboration iterations*—dozens of evaluation criteria that, when demonstrated against the candidate architecture, verify a solid framework for the primary use cases and show that the critical risks have been resolved.
- *construction iterations*—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 deployment.

An iterative process also uses the same basic tools, languages, notations, and artifacts for the products of test activities used for product development. Testing refers to the explicit evaluation by executing some set of components under a controlled scenario with an expected and objective outcome. You determine a test's success by comparing the expected outcome to the actual outcome using generally well-defined metrics of success. Furthermore, tests can be largely automated and instrumented.

The economic benefits of a leadership style

Figure 1 provides a project manager's view of the *improving time-to-value transition* we're all trying to achieve. This view helps summarize the results of effectively implementing the steering leadership style. It pres-

ents three project profiles plotted according to development progress versus time, where progress is a percentage of executed code. Executable doesn't imply complete, compliant, or up to specifications; it implies that the software is testable.

The typical sequence for the conventional engineering project management style is

- early success via paper designs and thorough (often too thorough) artifacts;
- commitment to executable code late in the life cycle;
- integration nightmares due to unforeseen implementation issues and interface ambiguities;
- heavy budget and schedule pressure to get the system working;
- late shoe-horning of suboptimal fixes, with no time for redesign; and
- a fragile, unmaintainable product, delivered late.

Modern management pushes integration into the design phase through a progression of demonstrable releases, which forces the architecturally significant breakage to happen earlier, so developers can resolve it in the context of lifecycle goals. This avoids the downstream integration nightmare, 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 expend roughly 40 percent of their total resources in integration and test activities, with much of this effort consumed in excessive scrap and re-

work. Modern projects with an iterative process and steering leadership style can deliver a product with these activities consuming about 25 percent of the budget.

From my experience, the conventional profile in figure 1 is still the norm and is characteristic of more than half of today's projects. Although most of these projects use the traditional engineering management approach, some claim to be using modern iterative development. However, without practicing steering leadership, they fail to deliver the business results expected. Perhaps one of four projects delivers the modern profile, while one of eight manages to operate on the target profile. It's from these more fluid profiles and successful outcomes that I've observed consistent usage of the styles I've discussed here.

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

References

1. P. Graham, *Hackers and Painters: Big Ideas from the Computer Age*, O'Reilly, 2004.
2. Standish Group International, *CHAOS Chronicles*, 2004.
3. W.E. Royce, *Software Project Management: A Unified Framework*, Addison-Wesley Longman, 1998.
4. M. Cantor, *Software Leadership*, Addison-Wesley, 2002.
5. J. Marasco, *The Software Development Edge: Essays on Managing Successful Projects*, Addison-Wesley, 2005.
6. P. Kroll and P. Kruchten, *The Rational Unified Process Made Easy: A Practitioner's Guide*, Addison-Wesley Longman, 2003.

For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/publications/dlib.

About the Author



Walker Royce is the vice president of IBM's Worldwide Rational Services Organization. He has managed large software engineering projects and consulted with a broad spectrum of software development organizations. He received his BA in physics from the University of California, Berkeley. He's the author of *Software Project Management: A Unified Framework* (Addison-Wesley Longman, 1998) and a principal contributor to the management philosophy inherent in Rational's Unified Process. Contact him at weroyce@us.ibm.com.

THE IEEE'S 1ST ONLINE-ONLY MAGAZINE



IEEE Distributed Systems Online brings you peer-reviewed articles, detailed tutorials, expert-managed topic areas, and diverse departments covering the latest news and developments in this fast-growing field.

Log on for **free access** to such topic areas as

**Grid Computing • Middleware
Cluster Computing • Security
Peer-to-Peer • Operating Systems
Web Systems • Parallel Processing
Mobile & Pervasive
and More!**

To receive monthly updates, email
dsonline@computer.org

<http://dsonline.computer.org>