UNIT - V

CCPDS-R Case Study and Future Software Project Management Practices
Modern Project Profiles, Next-Generation software Economics, Modern Process Transitions.

CCPDS-R Case Study

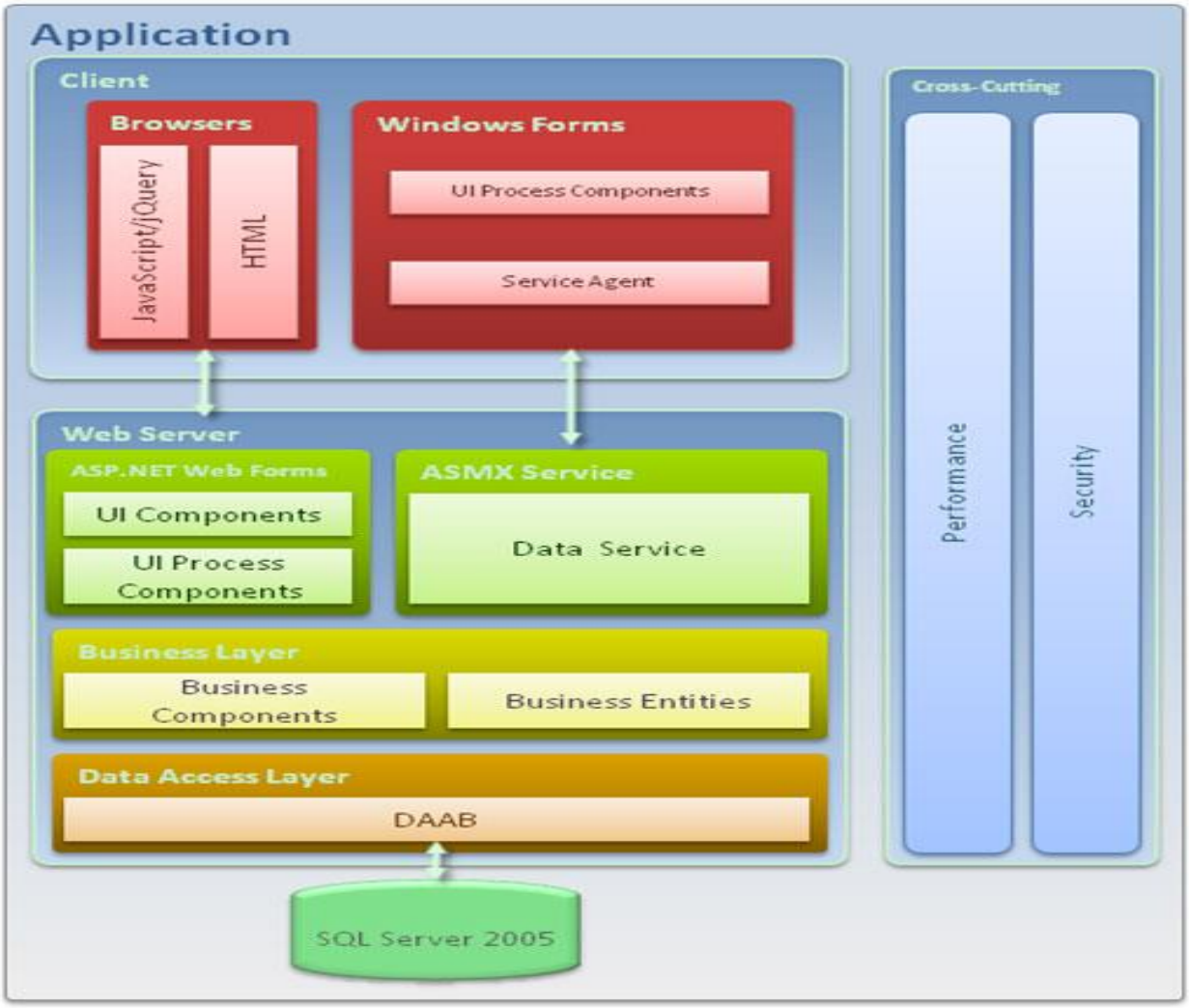


Figure 5-1: CCPDS-R Case Study Model

Detailed case study of a successful software project that followed many of the techniques presented in this book. Successful here means on budget, on schedule, and satisfactory to the customer. The Command Center Processing and Display System - Replacement (CCPDS-R) project was performed for the U.S. Air Force by TRW Space and Defense in Redondo Beach, California. The entire project included systems engineering, hardware procurement, and software development, with each of these three major activities consuming about one-third of the total cost. The schedule spanned 1987 through 1994.

The software effort included the development of three distinct software systems totaling more than one million source lines of code. This case study focuses on the initial software development, called the Common Subsystem, for which about 355,000 source lines were developed. The Common Subsystem

effort also produced a reusable architecture, a mature process, and an integrated environment for efficient development of the two software subsystems of roughly similar size that followed. This case study therefore represents about one-sixth of the overall CCPDS-R project effort.

An objective case study is a true indicator of a mature organization and a mature project process. The software industry needs more case studies like CCPDS-R.

The metrics histories were all derived directly from the artifacts of the project's process. These data were used to manage the project and were embraced by practitioners, managers, and stakeholders.

CCPDS-R was one of the pioneering projects that practiced many modern management approaches. This appendix provides a practical context that is relevant to the techniques, disciplines, and opinions provided throughout this book.

The system on budget and on schedule, and the users got more than they expected. TRW was awarded the Space and Missile Warning Systems Award for Excellence in 1991 for "continued, sustained performance in overall systems engineering and project execution." A project like CCPDS-R could be developed far more efficiently today. By incorporating current technologies and improved processes, environments, and levels of automation, this project could probably be built today with equal quality in half the time and at a quarter of the cost.

Responsibilities included managing the development of the foundation technologies, developing the technical and cost proposals, conducting the software engineering exercise, and managing the software engineering activities through the early operational capability milestone.

Although the software industry can claim many successful projects (not enough, but many), good case studies are lacking. There are very few well-documented projects with objective descriptions of what worked, what didn't, and why. This was one of my primary motivations for providing the level of detail contained in this appendix. It is heavy in project-specific details, approaches, and results, for three reasons:

1. Generating the case study wasn't much work. CCPDS-R is unique in its detailed and automated metrics approach. All the data were derived directly from the historical artifacts of the project's process.

2. This sort of objective case study is a true indicator of a mature organization and a mature project process. The absolute values of this historical perspective are only marginally useful. However, the trends, lessons learned, and relative priorities are distinguishing characteristics of successful software development.

3. Throughout previous chapters, many management and technical approaches are discussed generically. This appendix provides in a real-world example at least one relevant benchmark of performance.

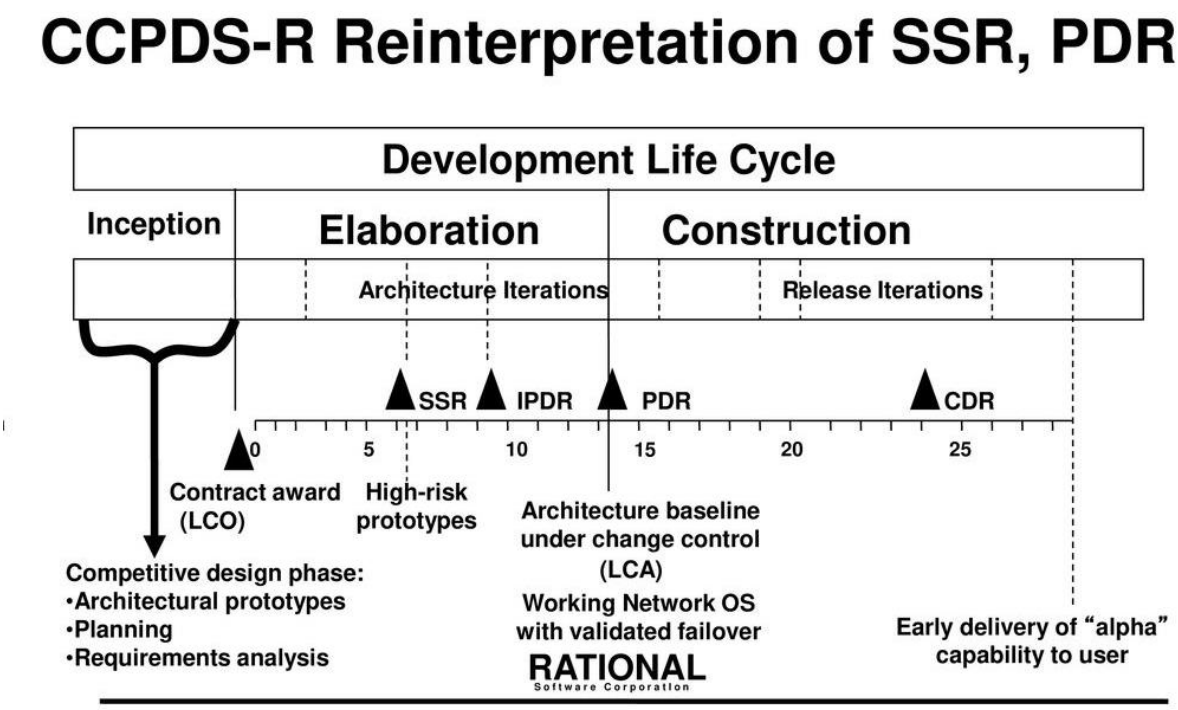


Figure 5-2: CCPDS-R Reinterpretation

Common Subsystem Overview

The CCPDS-R contract called for the development of three subsystems:
1. The Common Subsystem was the primary missile warning system within the Cheyenne Mountain Upgrade program. It required about 355,000 source lines of code, had a 48-month software development schedule, and laid the foundations for the subsystems that followed (reusable components, tools, environment, process, procedures). The Common Subsystem included a primary installation in Cheyenne Mountain, with a backup system deployed at Offutt Air Force Base, Nebraska.
2. The Processing and Display Subsystem (PDS) was a scaled-down missile warning display system for all nuclear-capable commanders-in-chief. The PDS software (about 250,000 SLOC) was fielded on remote, read-only workstations that were distributed worldwide.
3. The STRATCOM Subsystem (about 450,000 SLOC) provided both missile warning and force management capability for the backup missile warning center at the command center of the Strategic Command.

CCPDS-R was also a very large software development activity and was one of the first projects to use the Ada programming language. There was serious concern that the Ada development environments, contractor processes, and contractor training programs might not be mature enough to use on a full-scale development effort. The purpose of the software engineering exercise was to demonstrate that the contractor's proposed software process, Ada environment, and software team were in place, were mature, and were demonstrable.

The software engineering exercise occurred immediately after the FSD proposals were submitted. The customer provided both bidders with a simple two-page specification of a "missile warning simulator." This simulator had some of the same fundamental requirements as the CCPDS-R full-scale system, including a distributed architecture, a flexible user interface, and the basic processing scenarios of a simple CCPDS-R missile warning thread. The exercise requirements included the following:

• Use the proposed software team.
• Use the proposed software development techniques and tools.
• Use the FSD-proposed software development plan.

Future Software Project Management Practices

Rapid technology improvement ensures increased efficiency and higher productivity in every business whether it is small or large. Here we will see how technology is scaling our business and project management in an altered way and what the future plans are.
Basically, project management is an applied science that doesn't have any dimension. But yes, it has a dimension that is carried by factors of computational and managerial skills.
But the question is what the future of project management holds?
Essentially, a project is a process that is meant to complete a service, a goal or product within a fixed budget, time, and resource.
In earlier days, projects had to be managed at any cost to be completed. But there was no certain management process that was discovered to manage the process and projects.
This was the biggest issue for which the managers and the entire organization had to face a lot.
What a project management process exactly looks like? Here's the answer in a pictorial form;

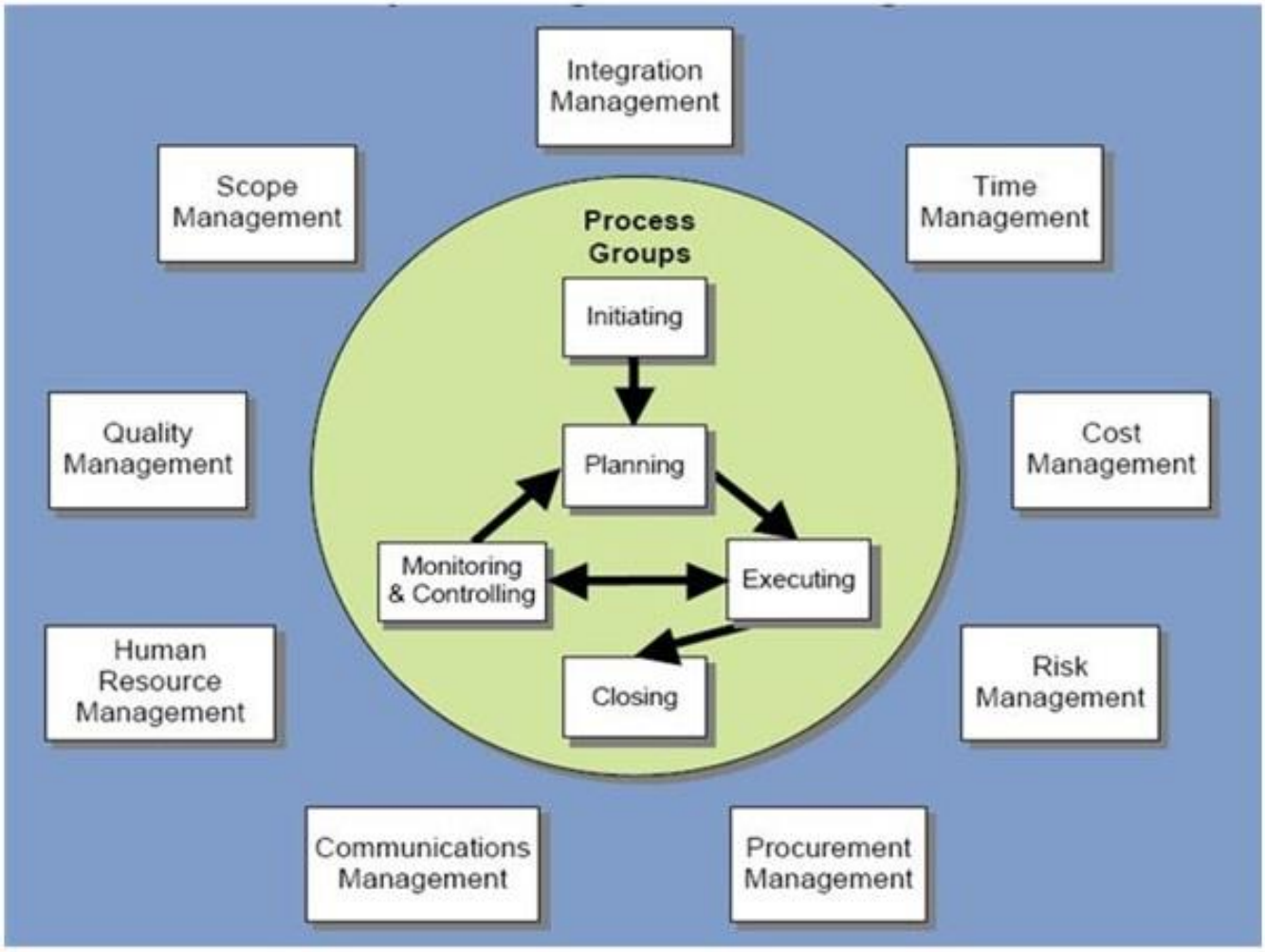


Figure 5-3: Future Software Project Management Practices

If we consider project management as a science, then it can be identified and applied as a science.

All the time, its application needs to improvise which has led to the quick development of a project. Its impact will definitely increase with time.

Project management is now available for those activities which don't have a physical dimension. You can apply techniques of project management if you are not producing or carrying goods that have a physical existence.

Future of project management look like

Every manager's aim is to complete a project or task within the given time period and available resources that too in his/her control.
Technology is playing and will play a big role in project monitoring and execution. Through project management technology, liability within each team member will increase while executing a project and its delivery.
If you are handling a project with a team, then it is important that every team member is skilled enough according to the requirements of the project.
So that it will go smoothly and you can finish your project in the proper time span without compromising the quality. If the team members are not capable to handle some work, then the manager has to change the members.
Project management is carried out by specialized teams whose skills are owned by their knowledge of certain computational skills.
Today, the software has become more user-friendly. So it has become a mandatory one to be known to the skills and aspects of software so that you can take out the project in an advanced way.
Latest project management tools and methods are emerging. Many tools will need to undergo major changes to fulfill the new demands. Seeing the competition in the market, everyone needs updated software or method so that they can beat the market.
As per the project requirements, responsibilities of every team member will increase. Every team member will need to be the best in their roles and responsibilities in their projects.
Future projects are hoping to be totally paperless with the required information being shared over the internet. PM Software will be advanced that will gather information automatically.

Modern project management

Modern project management serves unique projects. These could be one-time projects, such as website builds. Often, a services provider specializes in delivering unique projects repeatedly, such as a creative website builder. Because of its customized results, modern project management requires more flexible processes and scopes as well as highly specialized resources. It's common to find services firms tapping outside resources through contract tasks that roll up into a larger project within the provider's core services offering.
Project management, in a basic understanding, is the way people plan, lead, organize, and control (PLOC) work to arrive at a result, such as a product or service.
In modern project management, there have been several shifts related to planning, leading, organizing, and controlling work.

1. Plan

Project planning has become more flexible. Once a deal closes, project managers must find the appropriate resources, budget, timeframe, and tools (i.e., technology) required to complete the work in a timely manner and for a cost to which the client has agreed.

2. Lead

Today's project managers need a more widespread understanding of specific resource skills, since they will be leading a variety of people through specialized tasks that roll-up into the successful deliverable. They must speak many languages, from programming and creative to project financials, sales, and accounting. This knowledge helps them gain group consensus, solicit specific contributions, and create the best environment for individuals to contribute to the group project.

3. Organize

Organization requires greater flexibility as well. Project managers are increasingly relying on technology to help conduct their work. They are looking for solutions that pull them out of time-consuming spreadsheet calculations, data exports, manual report building, and manual monitoring in favor of real-time integrations and tracking tools now available. This helps them optimize resource allocation, budget tracking, report building, and on-time delivery.

4. Control

With more flexible tools comes greater control. Project managers are able to shift resources mid-project in order to address an issue that has arisen, such as potentially not meeting a deadline or going over budget. Today's software creates this environment of greater control, with early adopters of platform innovations pulling ahead of their competitors.

Next-Generation Software Economics

Next-generation software economics is being practiced by some advanced software organizations. Many of the techniques, processes, and methods described in this book's process framework have been practiced for several years. However, a mature, modern process is nowhere near the state of the practice for the average software organization. This chapter introduces several provocative hypotheses about the future of software economics. A general structure is proposed for a cost estimation model that would be better suited to the process framework in this book. I think this new approach would improve the accuracy and precision of software cost estimates, and would accommodate dramatic improvements in software economies of scale. Such improvements will be enabled by advances in software development environments. Finally, I look again at Boehm's benchmarks of conventional software project performance and describe, in objective terms, how the process framework should improve the overall software economics achieved by a project or organization.

Key Points

▲ Next-generation software economics should reflect better economies of scale and improved return on investment profiles. These are the real indicators of a mature industry.

▲ Further technology advances in round-trip engineering are critical to making the next quantum leap in software economics.

▲ Future cost estimation models need to be based on better primitive units defined from well-understood software engineering notations such as the Unified Modeling Language.

Next-Generation Cost Model

A next-generation software cost model should explicitly separate architectural engineering from application production, just as an architecture-first process does. The cost of designing, producing, testing, and maintaining the architecture baseline is a function of scale, quality, technology, process, and team skill.

The architecture cost model because it is inherently driven by research and development-oriented concerns. When an organization achieves a stable architecture, the production costs should be an exponential function of size, quality, and complexity, with a much more stable range of process and

personnel influence. The production stage cost model should reflect an economy of scale (exponent less than 1.0) similar to that of conventional economic models for bulk production of commodities.

Next-generation software cost models should estimate large-scale architectures with economy of scale. This implies that the process exponent during the production stage will be less than 1.0. My reasoning is that the larger the system, the more opportunity there is to exploit automation and to reuse common processes, components, and architectures.

Effort = (personnel)(environment)(quality)(size Process)
(Note: effort is exponentially related to size….)

Time = F(PArch, EffortArch) + F(PApp, EffortApp)

where:

T = technology parameter (environment automation support), S = scale parameter (such as use cases, function points, source lines of code), Q= quality parameter (such as portability, reliability, performance), P = process parameter (such as maturity, domain experience)

- Quantifying the scale of the software architecture in the engineering stage is an area ripe for research. Over the next decade, two breakthroughs in the software process seem possible, both of them realized through technology advances in the supporting environment.
- The first breakthrough would be the availability of integrated tools that automate the transition of information between requirements, design, implementation, and deployment elements. These tools would allow more comprehensive

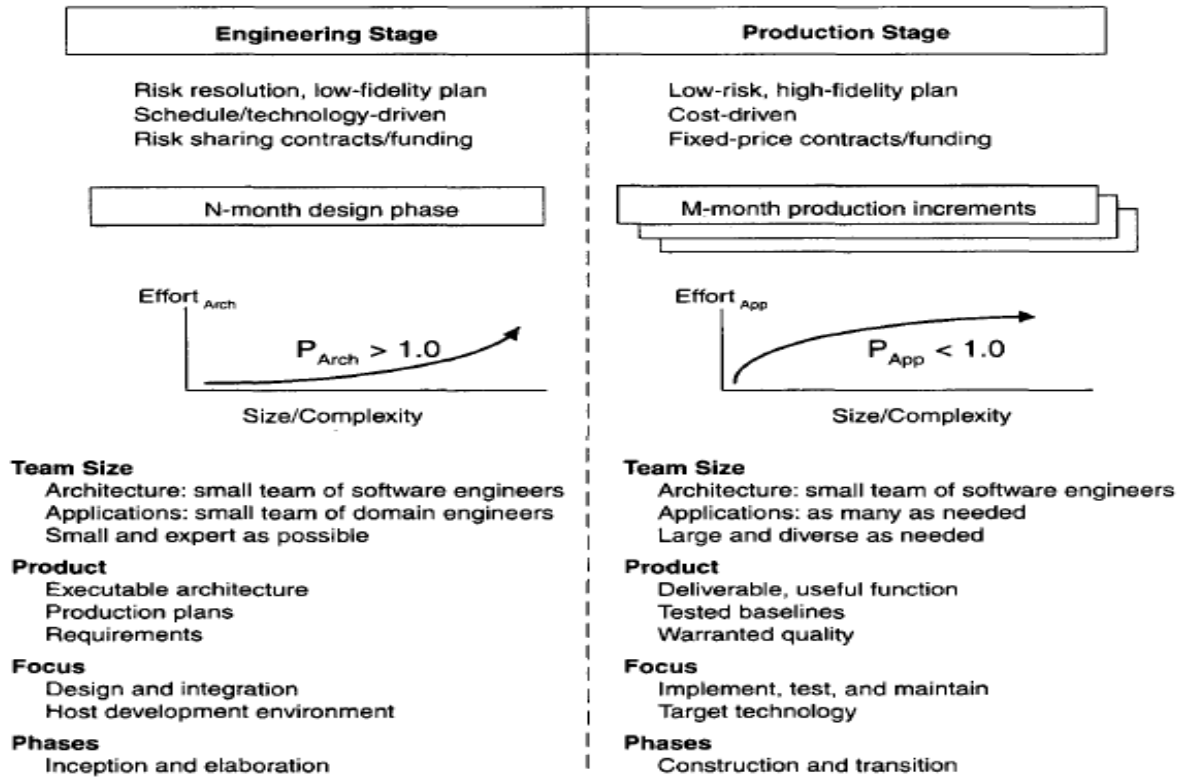


Figure 5-4: Next Generation Cost Model

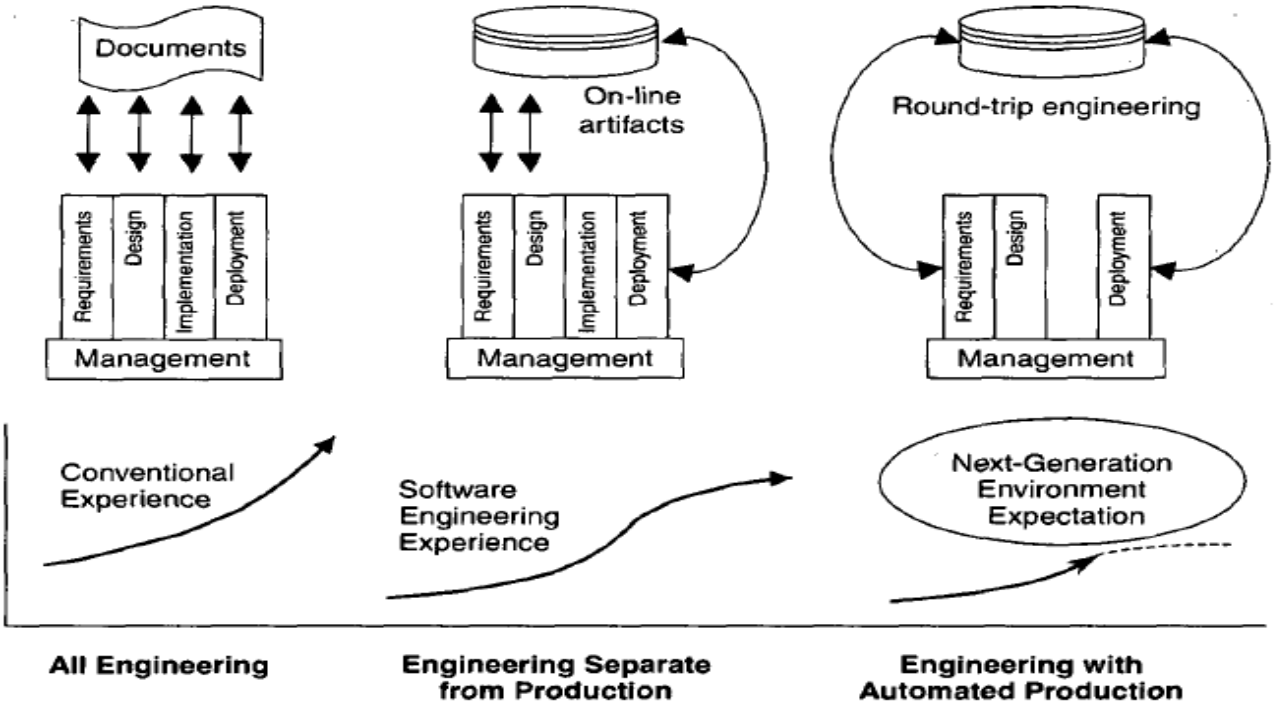


Figure 5-5: Next Generation Construction Process

Automation of the construction process in next-generation environments round-trip engineering among the engineering artifacts.

 The second breakthrough would focus on collapsing today's four sets of fundamental technical artifacts into three sets by automating the activities associated with human- generated source code, thereby eliminating the need for a separate implementation set.

 This technology advance, illustrated in Figure would allow executable programs to be produced directly from UML representations without human intervention. Visual modeling tools can already produce code subsets from UML models, but producing complete subsets is still in the future.

 Some of today's popular software cost models are not well matched to an iterative software process focused an architecture-first approach

 Many cost estimators are still using a conventional process experience base to estimate a modern project profile

A next-generation software cost model should explicitly separate architectural engineering from application production, just as an architecture-first process does.

Two major improvements in next-generation software cost estimation models:

 Separation of the engineering stage from the production stage will force estimators to differentiate between architectural scale and implementation size.

 Rigorous design notations such as UML will offer an opportunity to define units of measure for scale that are more standardized and therefore can be automated and tracked.

Modern Process Transitions

Successful software management is hard work. Technical breakthroughs, process breakthroughs, and new tools will make it easier, but management discipline will continue to be the crux of software project success. New technological advances will be accompanied by new opportunities for software applications, new dimensions of complexity, new avenues of automation, and new customers with different priorities. Accommodating these changes will perturb many of our ingrained software management values and priorities. However, striking a balance among requirements, designs, and plans will remain the underlying objective of future software management endeavors, just as it is today.

The software management framework I have presented in this book is not revolutionary; numerous projects have been practicing some of these disciplines for years. However, many of the techniques and disciplines suggested herein will necessitate a significant paradigm shift. Some of these changes will be resisted by certain stakeholders or by certain factions within a project or organization. It is not always easy to separate cultural resistance from objective resistance. This chapter summarizes some of the important culture shifts to be prepared for in order to avoid as many sources of friction as possible in transitioning successfully to a modern process.

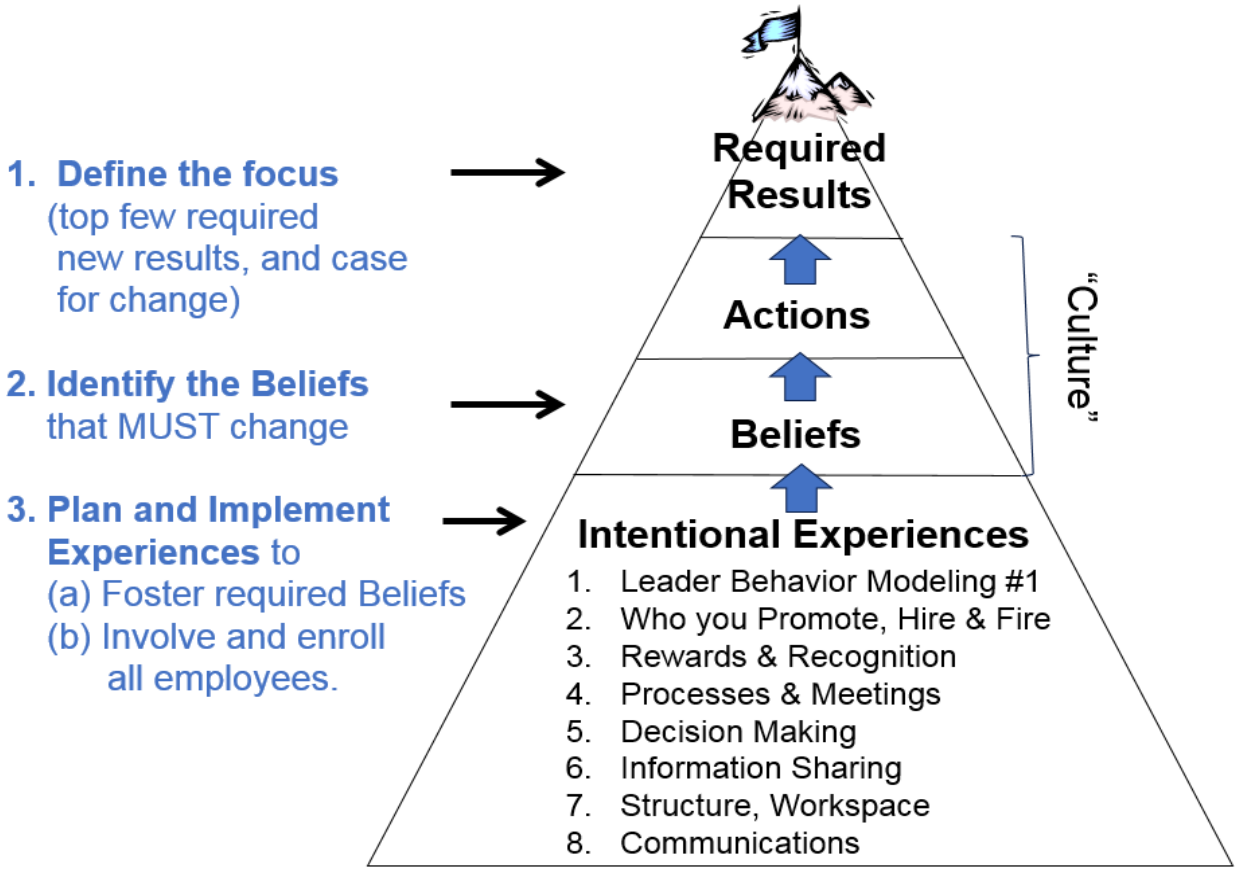
Key Points

▲ The transition to modern software processes and technologies necessitates i several culture shifts that will not ; always be easy to achieve.

▲ Lessons learned in transitioning organizations to a modern process have exposed several recurring themes of success that represent important culture shifts from conventional practice.

▲ A significant transition should be attempted on a significant project. Pilot i projects do not generally attract top tal-; ent, and top talent is crucial to the success of any significant transition.

Culture shifts



Holders must not overreact to early mistakes, digressions, or immature designs. Evaluation criteria in early release plans are goals, not requirements. If early engineering obstacles are overemphasized, development organizations will set up future iterations to be less ambitious. On the other hand, stakeholders should not tolerate lack of follow-through in resolving issues. If negative trends are not addressed with vigor, they can cause serious downstream perturbations. Open and attentive follow-through is necessary to resolve

issues. The management team is most likely to resist this transition (especially if the project was oversold), because it will expose any engineering or process issues that were easy to hide using the conventional process. Customers, users, and the engineering team will embrace this transition for exactly the same reason.

• Good and bad project performance is much more obvious earlier in the life cycle. In an iterative development, success breeds success, and early failures are extremely risky to turn around. Real-world project experience has shown time and again that it is the early phases that make or break a project. It is therefore of paramount importance to ensure that the very best team possible performs the planning and architecture phases. If these phases are done right and with good teams, projects can be completed successfully by average teams 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. No one should resist early staffing with the right team. However, most organizations have scarce resources for these sorts of early life-cycle roles and are hesitant to make the necessary staff allocations.

• Early increments will be immature. External stakeholders, such as customers and users, cannot expect initial deliveries to perform up to specification, to be complete, to be fully reliable, or to have end-target levels of quality or performance. On the other hand, development organizations must be held accountable for, and demonstrate, tangible improvements in successive increments. The trends will indicate convergence toward specification. Objectively quantifying changes, fixes, and upgrades will indicate the quality of the process and environment for future activities. Customers and users will have difficulty accepting the flaws of early releases, although they should be impressed by later increments. Management and the development team will accept immaturity as a natural part of the process.

• Artifacts are less important early, more important later. It is a waste of time to worry about the details (traceability, thoroughness, and completeness) of the artifact sets until a baseline is achieved that is useful enough and stable enough to warrant time-consuming analyses of these quality factors.

Otherwise, a project will squander early engineering cycles and precious resources adding content and quality to artifacts that may quickly become obsolete. While the development team will embrace this transition wholeheartedly, traditional contract monitors will resist the early de-emphasis on completeness.

• Real issues are surfaced and resolved systematically. Successful projects recognize that requirements and designs evolve together, with continuous negotiation, trade-off, and bartering toward best value, rather than blindly adhering to an ambiguous contract statement. On a healthy project that is making progress, it should be easy to differentiate between real and apparent issues. Depending on the situation, this culture shift could affect almost any team.

• Quality assurance is everyone's job, not a separate discipline. Many organizations have a separate group called quality assurance. I am generally against the concept of separate quality assurance activities, teams, or artifacts. Quality assurance should be woven into every role, every activity, every artifact. True quality assurance is measured by tangible progress and objective data, not by checklists, meetings, and human inspections. The software project manager or designee should assume the role of ensuring that quality assurance is properly integrated into the process. The traditional policing by a separate team of inspectors is replaced by the self-policing teamwork of an organization with a mature process, common objectives, and common incentives. Traditional managers and quality assurance personnel will resist this transition. Engineering teams will embrace it.

• Performance issues arise early in the life cycle. Early performance issues have surfaced on almost every successful project I know of. These issues are a sign of an immature design but a mature design process. Stakeholders will usually be concerned over early performance issues. Development engineers will embrace the emphasis on early demonstrations and the ability to assess and evaluate performance trade-offs in subsequent releases.

• Investments in automation are necessary. Because iterative development projects require extensive automation, it is important not to underinvest in the capital environment. It is also important for stakeholders to acquire an integrated environment that permits efficient participation in an iterative development. Otherwise, interactions with the development organization will degenerate to paper exchange and many of the issues of the conventional process. These investments may be opposed by organization managers overly focused on near-term financial results or by project personnel who favor the preference of the individual project over the global solution that serves both the project and the organization goals.

• Good software organizations should be more profitable. In the commercial software domain, this is not an issue. In most of the software contracting domain, especially government contracts, it is definitely an issue. As part of the adversarial nature of the acquisition and contracting process, there is considerable focus on ensuring that contractor profits are within a certain acceptable range (typically 5% to 15%). Occasionally, excellent contractor performance, good value engineering, or significant reuse results in potential contractor profit margins in excess of the acceptable initial bid. As soon as customers (or their users or engineering monitors) become aware of such a trend, it is inevitable that substantial pressure will be exerted to apply these "excess" resources to out-of-scope changes until the margin is back in the acceptable range.

As a consequence, the simple profit motive that underlies commercial transactions and incentivizes efficiency is replaced by complex contractual incentives (and producer-consumer conflicts) that are usually suboptimal. Very frequently, contractors see no economic incentive to implement major cost savings, and certainly there is little incentive to take risks that may have a large return. On the other hand, contractors can easily consume large amounts of money (usually at a small profit margin) without producing results and with little accountability for poor performance.

For the software industry to prosper, good contractors should be rewarded (more profit) and bad contractors should be punished (less profit). A customer who gets a good product at a reasonable price should be happy that the contractor also made a good profit. Allowing contractors who perform poorly to continue doing so is not good for anyone. This is one area in which the commercial domain is far more effective than the government contracting domain.