Software Management Renaissance
Walker Royce

The waterfall model of conventional software management, which is still prevalent in many mature software organizations, has served its purpose. The ever-increasing market demands on software development performance continue. The increasing breadth of Internet applications has further accelerated the transition to a more modern management process known as spiral, incremental, evolutionary, or iterative development. A comparison of conventional and modern software development models illustrates some of the critical discriminators in this transition.

Top 10 Principles of Conventional Software Management

Most software engineering texts present the waterfall model as the source of the conventional software management process. Conventional software management techniques work well for custom-developed software where the requirements are fixed when development begins. The life cycle typically follows a sequential transition from requirements to design to code to testing, with ad hoc documentation that attempts to capture complete intermediate representations at every stage. After coding and unit testing of individual components, the components are compiled and linked together (integrated) into a complete system.

In my view, the top 10 principles of conventional software management are:

Freeze requirements before design,
Forbid coding before detailed design review,
Use a higher-order programming language,
Complete unit testing before integration,
Maintain detailed traceability among all artifacts,
Thoroughly document each stage of the design,
Assess quality with an independent team,
Inspect everything,
Plan everything early with high fidelity, and
Rigorously control source-code baselines.
Significant inconsistencies among component interfaces and behavior, which can be extremely difficult to resolve, cannot be identified until integration, which almost always takes much longer than planned. Budget and schedule pressures drive teams to shoehorn in the quickest fixes. Redesign usually is out of the question. Testing of system threads, operational usefulness, and requirements compliance is performed through a series of releases until the software is judged adequate for the user. About 90% of the time, the process results in a late, over-budget, fragile, and expensive-to-maintain software system.

A typical result of following the waterfall model is that integration and testing consume too much time and effort in major software development workflows. For successful projects, about 40% of resources go to integration and testing. The percentage is even higher for unsuccessful projects. With such a low success rate, better risk management is imperative.

Top 10 Principles of Modern Software Management

While the software industry has been evolving the management process for many years, the Internet has accelerated the transition from the waterfall model to iterative development. First, the Internet offers a powerful set of mechanisms for multi-site collaboration and electronic information exchange that support iterative processes.

Second, distributed infrastructures for common architectural patterns support executable architectures of Internet-based applications. Web-based applications consist of many moving parts that are constantly updated, making iterative development the process of choice. Finally, by introducing a new set of business models, projects, and organizations, the Internet has created a demand for incredibly rapid development of quality applications. Iterative development processes are necessary to meet challenging project performance goals.

Modern software development produces the architecture first, followed by usable increments of partial capability, and then completeness. Requirements and design flaws are detected and resolved earlier in the life cycle, avoiding the big-bang integration at the end of a project. Quality control improves because system characteristics inherent in the architecture (such as performance, fault tolerance, interoperability, and maintainability) are identifiable earlier in the process where problems can be corrected without jeopardizing target costs and schedules.

My top 10 principles of modern software management are:

Base the process on an architecture-first approach,
Establish an iterative life-cycle process that confronts risk early,
Transition design methods to emphasize component-based development,
Establish a change-management environment,
Enhance change freedom through tools that support round-trip engineering,
Capture design artifacts in rigorous, model-based notation,
Instrument the process for objective quality control and progress assessment,
Use a demonstration-based approach to assess intermediate artifacts,
Plan intermediate releases in groups of usage scenarios with evolving levels of detail, and
Establish an economically-scalable, configurable process.
Where conventional approaches mire software development in integration activities, these modern principles should result in less scrap and rework through a greater emphasis on early life-cycle engineering and a more balanced expenditure of resources across the core workflows of a modern process.

Demonstrations, enabled by the architecture-first approach, force integration into the design phase. They do not eliminate design breakage, but they make it happen when it can be addressed effectively. By avoiding the downstream integration nightmare (along with late patches and suboptimal software fixes), a more robust and maintainable design results. Interim milestones provide tangible results. The project does not move forward until it meets the demonstration objectives. This process does not preclude the renegotiation of objectives once the interim findings permit further understanding of the trade-offs inherent in the requirements, design, and plans.

The Rational Unified Process, a well-accepted benchmark of a modern iterative development process, embodies my top 10 principles. Its life cycle has four phases:

Inception: definition and assessment of the vision and business case
Elaboration: synthesis, demonstration, and assessment of an architecture baseline
Construction: development, demonstration, and assessment of useful increments
Transition: usability assessment, productization, and deployment
Each phase of development produces a certain amount of precision in the product/system description called software artifacts. Life-cycle software artifacts are organized into five sets that are roughly partitioned by the underlying language of:

requirements (organized text and UML models of the problem space);

design (UML models of the solution space);
implementation (human-readable programming language and associated source files);
deployment (machine-processable languages and associated files); and
management (ad hoc textual formats such as plans, schedules, and spreadsheets). At any point in the life cycle, the different artifact sets should be in balance, at compatible detail levels, and traceable to each other. As development proceeds, each part evolves in more detail. When the system is complete, all five sets are fully elaborated and consistent with each other. Unlike the conventional practice, the modern process does not specify the requirements, then develop the design, then write code, then execute. Instead, the entire system evolves throughout the process.

Principles That Didn't Make the Cut

A comparison of my top 10 principles with other lists, such as the Software Project Management Network's Best Practice Initiative or the SEI Capability Maturity Model's key process areas, reveals several notable omissions.

Requirements-first emphasis. The most obvious difference is my apparent under-emphasis on requirements. Requirements are a means, not an end. Conventional wisdom has over-prescribed "better requirements" as the cure for recurring project woes. Requirements, designs, and plans should evolve together.

Detailed planning and "inch-stones." Overplanning, another misapplied practice, is different from evolutionary planning. Early, false precision is a recurring source of downstream scrap and rework.

Inspections. Inspections are overhyped and overused. While properly focused inspections help to resolve known issues, inspections too often are used to identify issues and provide quality coverage. Human inspections are inefficient, labor-intensive, and expensive. In my experience, inspections can uncover many cosmetic errors, but they rarely uncover architecturally-significant defects.

Separate testing. Testing is not covered by a separate principle; it is covered by all of them. A modern process integrates testing activities throughout the life cycle with homogeneous methods, tools, and notations. The integration of interfaces, behaviors, and structures should be emphasized before concentrating on completeness testing and requirements compliance.

Separate quality assurance. The much-touted concept of a separate quality-assurance reporting chain has resulted in projects that isolate "quality police." A better approach is to work quality assessment into every activity through the checks and balances of organizational teams focused on architecture, components, and usability. Quality is every team's job, not one team's job.

Requirements traceability to design. Demanding rigorous problem-to-solution traceability is frequently counterproductive, forcing the design to be structured in the same manner as the requirements. Good component-based architectures have chaotic traceability to their requirements. Tight problem-to-solution traceability might have been productive when 100% custom software was the norm; those days are gone.

Predicting the Future

Planning and expenditure allocations will continue to shift as modern project management methods, architectural infrastructures (such as Java 2 Enterprise Edition and Microsoft Windows DNA), and software development processes and technology mature. Resource expenditure trends will lead to more balance across the primary workflows as a result of increased exploitation of standard architectural patterns and infrastructure components (less human-generated stuff), more efficient processes (less scrap and rework), more proficient people (in smaller teams), and more automation. The resource allocations in Table 1 reflect my experience in waterfall process projects and several successful iterative process projects. These values are deliberately imprecise; their purpose is to relate the relative trends over time. Table 1's "future" column provides my view on major trends that

will surface in the coming years.

Table 1. Expenditure allocations.

Life-cycle activity	Conventional	Modern	Future
Management	5%	10%	12%
Requirements	5%	10%	12%
Design	10%	15%	20%
Implementation	30%	25%	14%
Test and assessment	40%	25%	18%
Deployment	5%	5%	12%
Environment	5%	10%	12%
Totals	100%	100%	100%

Several major trends will surface in the coming years:

More automation of implementation activities and reuse of commercial components will reduce implementation activities, resulting in relatively more burden on requirements and design activities and environments.
More mature iterative development methods and Web-based architectures will drive deployment activities into a larger role within the life cycle.
More mature iterative development environments (process and tooling) will enable further reduction of life-cycle scrap and rework.
Because iterative development is more challenging than the simple management paradigm presented by the waterfall model, disciplined software management and common sense will remain one of the paramount discriminators of software engineering success or failure.
In many software domains, a distinct line divides development and maintenance. Future software projects (legacy system upgrades, new developments, or some combination of the two) probably will not differentiate much between development and maintenance.
Iterative development and the Internet are driving software engineering toward a more homogeneous software-management framework. With most of the software industry focusing on iterative-process frameworks, advanced requirements and design notations, and Web-based architectural patterns, we should see dramatic improvements in software project performance and higher returns on organizational software technology investments.

Ten years ago, about one in 10 software projects succeeded. Consequently, software project managers spent too much time playing defense and worrying about risk management. Today, that ratio has improved to about 1:4, still as challenging as batting against a major league pitcher. As modern iterative development and supporting environments advance, the success ratio for delivering a software project 10 years from now could improve to 1:2. Software project managers should invest more time playing offense through success management, and organizations should continue to accelerate software development leverage to deliver more value and new features faster and more profitably.

Walker Royce is the vice president and general manager of strategic services for Rational Software Corporation. He is the author of Software Project Management, A Unified Framework (1998, Addison-Wesley). Contact him at mailto:wroyce@rational.com