

Improving Software Development Economics Part II: Reducing Software Product Complexity and Improving Software Processes

by [Walker Royce](#)

Vice President and General
Manager
Strategic Services
Rational Software



In [last month's issue](#) of The Rational Edge, I began a four-part series of articles that summarize our experience and discuss the key approaches that have enabled our customers to make substantial improvements in software development economics. In that article, I introduced a framework for reasoning about economic impacts and introduced four approaches to improvement:

- 1. Reduce the size or complexity of what needs to be developed.*
- 2. Improve the development process.*
- 3. Use more proficient teams.*
- 4. Use integrated tools that exploit more automation.*

This month, I will discuss some of the discriminating techniques involved with the first two approaches: reducing the size or complexity of what needs to be developed, and improving the development process.

Reducing Software Product Size or Complexity

The most significant way to improve economic results is usually to achieve a software solution with the minimum amount of human-generated source material. Our experience shows that managing scope, raising the level of abstraction through component-based technology, and using visual modeling notations are the highest leverage techniques that make a difference in reducing complexity.

- ▶ [subscribe](#)
- ▶ [contact us](#)
- ▶ [submit an article](#)
- ▶ [rational.com](#)
- ▶ [issue contents](#)
- ▶ [archives](#)
- ▶ [mission statement](#)
- ▶ [editorial staff](#)

Managing Scope

The scope of a software product is usually defined by a set of features, use cases, or requirements that specify what the product needs to do and how well it needs to do it. Managing scope requires an understanding of the economic tradeoffs inherent in a set of features or requirements. At a minimum, you need to understand the relative value and cost of achieving each unit of scope. Typically, the development cost and user value associated with the required product features are fuzzy in the early phases of a project, but they evolve and become clearer with time.

Consider a trivial product with four features, as represented in Figure 1. This illustration abstracts the units of cost, value, and economic leverage to simplify a very complex and context-dependent discipline. In most applications, representations of cost, value, and economic leverage assessments do not need to be precise to manage scope. They could be as simple as "low, moderate, high," or they could be represented by an integer ranging from 1 to 10, as in the illustration. The main point is that all requirements are not created equal. To manage scope, the units of scope need to be objectively differentiated. Simple coarse-grained assessments are usually good enough to make most of the significant decisions. Obvious conclusions to be drawn from the Figure 1 tradeoffs are that delivering a subset of the features faster may be a highly desirable option for the end product, or that a certain subset of features constitutes the best target for the first increment of capability demonstration.

Required Feature	Cost	Value
A	1	2
B	3	7
C	4	5
D	5	4

Product Option 1: Produce All Features A, B, C, and D
Development Cost = 13, User Value = 18
Development Time = 14 months
Economic Leverage: $18/13 = 1.4$

Product Option 2: Produce Only Features B and C
Development Cost = 7, User Value = 12
Development Time = 8 Months
Economic Leverage: $12/7 = 1.7$

Figure 1: Tradeoffs in Managing Scope

Most products are not so trivial. A product might consist of tens, hundreds, or even thousands of requirements that are somewhat interrelated and cannot simply be added to or subtracted from the scope. Nevertheless, if you recognize the relative importance of different requirements and define some simple objective measures of cost and value for each unit of scope,

you will succeed in managing scope and evolving a product toward a solution with more optimal economic leverage.

Reducing the Size of Human-Generated Code

Component-based technology is a general term for reducing the size and complexity of human-generated code necessary to achieve a software solution. Commercial components, domain-specific reuse, architectural patterns, and higher-order programming languages are all elements of component-based approaches focused on achieving a given system with fewer lines of human-specified source directives (statements). For example, to achieve a certain application with a fixed number of features, we could use any of the following potential solutions:

- Develop 1,000,000 lines of custom assembly language.
- Develop 400,000 lines of custom C++.
- Develop 100,000 lines of custom C++, integrate 200,000 lines of existing reusable components, and purchase a commercial middleware product.
- Develop 50,000 lines of custom Visual Basic, and purchase and integrate several commercial components on a WinDNA platform.
- Develop 5,000 lines of custom Java, develop 10,000 lines of custom HTML, and purchase and integrate several commercial components on a J2EE platform.

Each of these solutions represents a step up in exploiting component-based technology and a commensurate reduction in the total amount of human-developed code, which in turn reduces the time and the team size needed for development. Since the difference between large and small projects has a greater than linear impact on the life-cycle cost, the use of the highest-level language and appropriate commercial components has the highest potential cost impact. Furthermore, simpler is generally better. Reducing the size of custom-developed software usually increases understandability, reliability, and the ease of making changes.

Managing Complexity Through Visual Modeling

Object-oriented technology and visual modeling achieved rapid acceptance during the 1990s. The fundamental impact of object-oriented technology has been in reducing the overall size and complexity of what needs to be developed through more formalized notations for capturing and visualizing software abstractions. A model is a simplification of reality that provides a complete description of a system. We build models of complex systems because we cannot comprehend any such system in its entirety. Modeling is important because it helps the development team visualize, specify, construct, and communicate the structure and behavior of a system's architecture.

Using a standard modeling notation such as the Unified Modeling Language (UML), different members of the development team can communicate their decisions unambiguously to each other. Using visual

modeling tools facilitates the management of these models, letting you hide or expose details as necessary. Visual modeling also helps maintain consistency among a system's artifacts: its requirements, designs, and implementations. In short, visual modeling helps improve a team's ability to manage software complexity.

Improving the Development Process

In order to achieve success, real-world software projects require an incredibly complex web of sequential and parallel steps. As the scale of the project increases, more overhead steps must be included just to manage the complexity of this web. All project processes consist of productive activities and overhead activities.

- *Productive activities* result in tangible progress toward the end product. For software efforts, these activities include prototyping, modeling, coding, integration, debugging, and user documentation.
- *Overhead activities* have an intangible impact on the end product. They include plan preparation, requirements management, documentation, progress monitoring, risk assessment, financial assessment, configuration control, quality assessment, testing, late scrap and rework, management, personnel training, business administration, and other tasks. Although overhead activities include many value-added efforts, in general, when less effort is devoted to these activities, more effort can be expended on productive activities.

The main thrust of process improvement is to improve the results of productive activities and minimize the impact of overhead activities on personnel and schedule. Based on our observations, these are the three most discriminating approaches for achieving significant process improvements:

1. Transitioning to an iterative process.
2. Attacking the significant risks first through a component-based, architecture-first focus.
3. Using key software engineering best practices, from the outset, in requirements management, visual modeling, change management, and assessing quality throughout the life-cycle.

Using an Iterative Process

The key discriminator in significant process improvement is making the transition from the conventional (waterfall) approach to a modern, iterative approach. The conventional software process was characterized by transitioning through sequential phases, from requirements to design to code to test, achieving 100 percent completion of each artifact at each life-cycle stage. All requirements, artifacts, components, and activities were treated as equals. The goal was to achieve high-fidelity traceability among all artifacts at each stage in the life-cycle.

In practice, the conventional process resulted in:

- Protracted integration and late design breakage.
- Late risk resolution.
- Requirements-driven functional decomposition.
- Adversarial stakeholder relationships.
- Focus on documents and review meetings.

These symptoms almost always led to a significant diseconomy of scale, especially for larger projects involving many developers. By contrast, a modern (iterative) development process framework is characterized by:

1. Continuous round-trip engineering from requirements to test, at evolving levels of abstraction.
2. Achieving high-fidelity understanding of the architecturally significant decisions as early as practical.
3. Evolving the artifacts in breadth and depth based on risk management priorities.
4. Postponing completeness and consistency analyses until later in the life cycle.

A modern process framework attacks the primary sources of the diseconomy of scale inherent in the conventional software process.

Figure 2 provides an objective perspective of the difference between the conventional waterfall process and a modern iterative process. It graphs development progress versus time, where progress is defined as percent coded, that is, demonstrable in its target form. At that point, the software is compilable and executable. It is not necessarily complete, compliant, nor up to specifications.

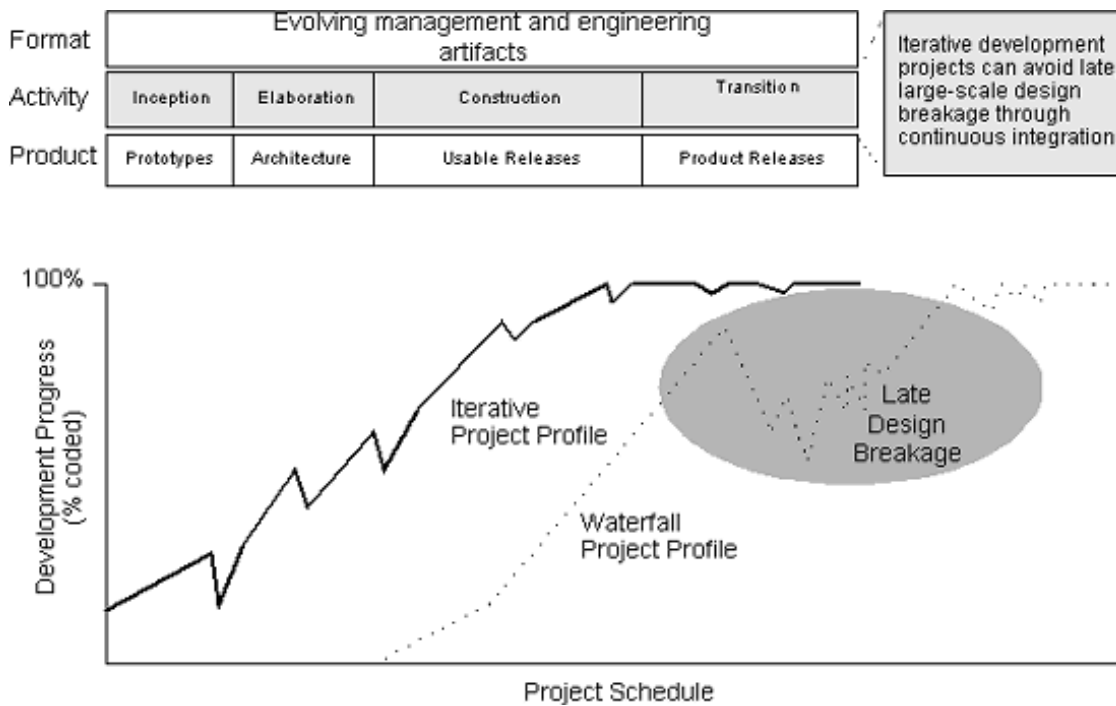


Figure 2: Project Profiles for the Waterfall and Iterative Processes

In the waterfall project life cycle, software development typically progressed without issue until the integration phase. Requirements were first captured in complete detail in *ad hoc* text. Design documents were then fully elaborated in *ad hoc* notations. Coding and unit testing of individual components followed. Finally, the components were compiled and linked together into a complete system. This integration activity was the first time that significant inconsistencies among components (their interfaces and behavior) could be tangibly recognized. These inconsistencies, some of which were extremely difficult to uncover, resulted from using ambiguous formats for the early life-cycle artifacts. Getting the software to operate reliably enough to test its usefulness took much longer than planned. Budget and schedule pressure drove teams to shoehorn in the quickest fixes; redesign was usually out of the question. Then the testing of system threads, usefulness, requirements compliance, and quality was performed through a series of releases until the software was judged adequate for the user. About 90 percent of the time, the end result was a software system that was late, over budget, fragile, and expensive to maintain.

A review of numerous conventional projects that followed a waterfall model shows a recurring symptom: Although it was not usually planned this way, the resources expended in the major software development workflows resulted in an excessive allocation of resources (time or effort) to accomplish integration and test. Successfully completed projects consumed 40 percent of their effort in these activities; the percentage was even higher for unsuccessful projects. The overriding reason was that the effort associated with the late scrap and rework of design flaws was collected and implemented during the integration and test phases. Integration is a non-value-added activity, and most integration and test

organizations spent 60 percent of their time integrating (that is, getting the software to work by resolving the design flaws and the frequently malignant scrap and rework associated with these resolutions). It is preferable for integration to take little time and little effort so the integration and test team can focus on demonstrating and testing the software, which are value-added efforts.

Attacking Significant Risks Early

Using an iterative development process, the software development team produces the architecture first, allowing integration to occur as the "verification" activity of the design phase and allowing design flaws to be detected and resolved earlier in the life-cycle. This replaces the big-bang integration at the end of a project with continuous integration throughout the project. Getting the architecturally important things to be well understood and stable before worrying about the complete breadth and depth of the artifacts should result in scrap and rework rates that decrease or remain stable over the project life-cycle.

The architecture-first approach forces integration into the design phase and demonstrations provide the forcing function for progress. The demonstrations do not eliminate the design breakage; they just make it happen in the design phase where it can be fixed correctly. In an iterative process, the system is "grown" from an immature prototype to a baseline architectural skeleton to increments of useful capabilities to, finally, complete product releases. The downstream integration nightmare is avoided, and a more robust and maintainable design is produced.

Major milestones provide very tangible results. Designs are now guilty until proven innocent: The project does not move forward until the objectives of the demonstration have been achieved. Results of the demonstration and major milestones contribute to an understanding of the tradeoffs among the requirements, design, plans, technology, and other factors. Based on this understanding, changes to stakeholder expectations can still be renegotiated.

The early phases of the iterative life cycle (inception and elaboration) focus on confronting and resolving the risks before making the big resource commitments required in later phases. Managers of conventional projects tend to do the easy stuff first, thereby demonstrating early progress. A modern process, as shown in Figure 3, needs to attack the architecturally significant stuff first, the important 20 percent of the requirements: use cases, components, and risks.

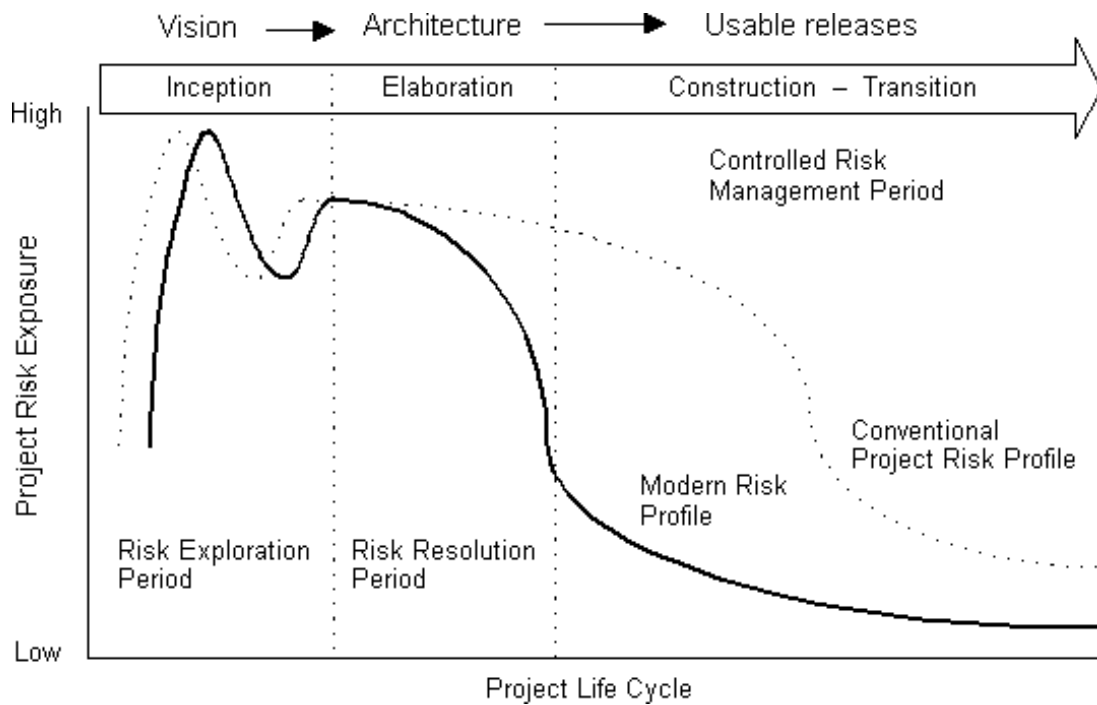


Figure 3: Architecture First, Then Production

The "80/20" lessons learned during the past 30 years of software management experience provide a useful perspective for identifying some of the key features of an iterative development philosophy.

- **80% of the engineering is consumed by 20% of the requirements.** Do not strive prematurely for high fidelity and full traceability of the complete requirements set. Instead, strive to understand the driving requirements completely before committing resources to full-scale development.
- **80% of the software cost is consumed by 20% of the components.** Elaborate the cost-critical components first so that planning and control of cost drivers are well understood early in the life-cycle.
- **80% of the errors are caused by 20% of the components.** Elaborate the reliability-critical components first so that assessment activities have enough time to achieve the necessary level of maturity.
- **80% of software scrap and rework is caused by 20% of the changes.** Elaborate the change-critical components first so that broad-impact changes occur when the project is nimble.
- **80% of the resource consumption (execution time, disk space, memory) is consumed by 20% of the components.** Elaborate the performance-critical components first so that engineering tradeoffs with reliability, changeability, and cost effectiveness can be resolved as early in the life-cycle as possible.
- **80% of the progress is made by 20% of the people.** Make

sure the initial team that plans the project and designs the architecture is of the highest quality. An adequate plan and adequate architecture can then succeed with an average construction team. An inadequate plan or inadequate architecture will probably not succeed, even with an expert construction team.

Using Software Best Practices

Best practices are a set of commercially proven approaches to software development that, when used together, strike at the root causes of software development problems. The Rational Unified Process integrates six industry best practices into one process framework:

1. Develop iteratively
2. Manage requirements
3. Use component architectures
4. Model visually
5. Continuously verify quality
6. Manage change

The techniques and technologies inherent in these best practices are discussed in detail in the [Rational Unified Process](#). One way to view the impact of these best practices on the economics of software projects is through the differences in resource expenditure profiles between conventional projects and modern iterative projects.

Conventional principles drove software development activities to overexpend during implementation and integration activities. A healthy iterative process, an architecture-first focus, and incorporation of software best practices should result in less total scrap and rework through relatively more emphasis on the high-value activities of management planning, requirements analysis, and design. This results in a more balanced expenditure of resources across the core workflows of a modern process (see Table 1).

Table 1. Resource Expenditures

Life-Cycle Activity	Conventional	Modern
Management	5%	10%
Requirements	5%	10%
Design	10%	15%
Implementation	30%	25%
Test and assessment	40%	25%
Deployment	5%	5%
Environment	<u>5%</u>	<u>10%</u>
	100%	100%

One critical lesson learned from successful iterative development projects is that they start out with a planning profile different from the standard profile of a conventional project. If you plan modern iterative projects with the old waterfall planning profile, the chance of success is significantly diminished. By planning a modern project with a more appropriate resource profile derived from successful iterative projects, there is much more flexibility in optimizing the project performance for improvements in productivity, quality, or cycle time, whichever is the business driver.

During the past decade, Rational has participated in the software process improvement efforts of numerous companies, including most of the leading software development organizations in the Fortune 500 companies. Typical goals are to achieve a 2X, 3X, or 10X increase in productivity, quality, time to market, or some combination of all three, where X corresponds to how well the organization does now. The funny thing is that most of these organizations have only a coarse grasp on what X is, in objective terms.

Table 1 characterizes the impact on project expenditure profiles associated with making about a 3X reduction in scrap and rework. This improvement is the primary goal of transitioning from the conventional waterfall software development process to a modern iterative software development process.

Standardizing on a common process is a courageous undertaking for a software organization, and there is a wide spectrum of implementations. I have seen organizations attempt to do less (too little standardization, or none) and more (too much standardization) with little success in improving software return on investment. Process standardization requires a very balanced approach.

In next month's issue of The Rational Edge, I will explore two more critical techniques for improving the economics of software development: how software organizations and projects create and manage more efficient teams, and how software teams can improve automation through integrated software environments.



For more information on the products or services discussed in this article, please click [here](#) and follow the instructions provided. Thank you!