Leading Modern Systems Development

by Murray Cantor, Ph.D.
Principal Consultant
Rational Software Corporation

Systems development has become increasingly complex over the last decades; project teams must respond not only to the concerns of the classical engineering disciplines (such as electrical and mechanical engineering) and software engineering, but also to stakeholder demands for more capable, integrated, distributed, and optimized systems. Those who manage the development of these new systems face new and difficult challenges that we will explore in this article, suggesting strategies and leadership approaches for meeting them.

Modern Systems Challenges

The launch of the first imaging satellite represented a great technical accomplishment. First, the development team for that project had to analyze what capabilities were required for this unprecedented system to meet user needs. Then, they had to figure out how to use existing technology to image a structure on the ground, store the image long enough to download it to a ground station, and then make it accessible from an analyst's workstation. At that time, it was of little concern that the solution used dedicated resources. Providing the capability itself was enough to meet stakeholder needs.

Today, however, the situation is very different.

Larger Problem Space

Today, those designing satellite imaging systems are concerned less with

providing the basic capability and more with optimizing the system to meet a broader set of stakeholder needs. For example, in addition to wanting information the system gathers as soon as possible, analysts want the ability to integrate that information with data from other sources. Also, systems owners want to minimize dedicated hardware (and software) use; they can't afford to maintain one system per capability and want greater reuse of existing resources. Further, they are very concerned with lowering ongoing operation and maintenance costs.

On top of this, it is certain that the mission and enabling technologies for a system will change several times over the program's lifespan. And those who invest in the system want it to evolve in response to these changes at minimal cost and with minimal disruption. In addition, they expect their investment to result in reusable intellectual property.

Larger Solution Space

Today's technologies can help designers meet these complex challenges because they provide more ways to approach systems design. For example, whereas the designers of the first satellite image system had limited choices and had to be clever about overcoming technical constraints, today's systems designers actually have excess processing capacity and can reallocate subsystem responsibilities in order to optimize the system. This phenomenon is true for a number of domains, such as telecommunications, avionics, information technology, and so on.

Increased Software Size and Complexity

A third change in the systems development landscape is that the size and complexity of software applications continues to grow. With the advent of object technology, component frameworks, and software development automation, software development productivity has increased three-fold since 1970.[1] At the same time, there have been large gains in processing power, computer memory, available data storage, and network bandwidth. These, in turn, have led to ever more sophisticated operating environments. The software and systems industries have used all of this improved capability to develop increasingly large, more complex programs. In fact, competitive pressures have led to a ten-fold increase in the size of an average software application, as measured by function points.[2] And, presumably, as technology continues to improve, this trend will continue.

Modern Leadership Challenges

Given this growth in the problem space, solution space, and application size, it is not surprising that some classical systems development methods have proven less than effective in recent projects. Even if a systems effort is dominated by software development, the leadership challenge now is not so much to maximize team members' individual productivity as it is to ensure that the right system gets built -- one that provides the needed services and best meets (possibly conflicting) stakeholder needs. Below, we will explore some approaches to dealing with this modern challenge.

Two Perspectives

There are many ways to conceptualize a system. Two of them are:

- As a physical entity governed by the laws of physics and classical disciplines such as mechanical, electrical, and civil engineering.
- As a large state machine governed by the insights of computer science and software engineering.

Some systems, such as rocket engines, are perhaps best thought of from the first perspective. Others, such as most information technology systems, are dominated by the second perspective. In fact, however, all systems fit some aspects of both perspectives. Rocket engines have software-driven embedded controllers. Information systems are governed by the physical laws that constrain their performance, latency, and reliability.

In the past, project leaders would typically adopt one of these points of view, depending on the kind of system, and manage accordingly. If they thought of the system primarily as a physical thing, they viewed the software as a necessary evil that enabled the hardware to do its job. If they viewed the system from a software perspective, they thought of the hardware as a hosting mechanism -- and usually treated it as an afterthought.

When you focus on only one of these two perspectives, you also tend to focus on different quality concerns:

- Physics -- Reliability (mean time to failure, mean time to recovery), responsiveness, field supportability, and so on.
- Software -- Maintainability (ease of isolating and removing defects), extensibility (ease of adding capability), and so on.

The management approaches for each of these perspectives have evolved in different ways. In general, I believe it is fair to say that the physics-based camp has taken a more serialized activity approach, whereas the software camp has moved toward a more iterative approach.

As technology continues to evolve, the need to consider both perspectives is becoming more pressing. The economic value of the software in physical systems has increased significantly. For example, a modern automobile engine's performance is primarily determined by the large amount of software instantiated by the engine controller. In fact, most automobile systems are now software-dependent, and this trend will only grow with the emergence of hybrid and fuel-cell based systems. The same can be said for telecommunications, medical, aircraft, and other equipment. A purely physics-based approach does not recognize these major dependencies. A pure software approach, on the other hand, does not adequately address the high costs-of-ownership associated with the increase in large, distributed information systems. Modern systems require a balanced approach to deal with quality concerns.

Failure to balance the two management perspectives results in a set of symptoms that plague development projects, including:

- Poor communication between hardware and software teams.
- Overall lack of systems perspective and unifying architecture.
- Late breakage, with considerable scrap and rework expense.
- Poor correlation between actual project status and tracking metrics.

Over the last few years, progressive systems development teams and organizations have successfully adjusted their leadership and management approaches to achieve a balance between the physics and software perspectives. Many also realize that the lessons learned over the last decades for managing large software projects, appropriately modified, should be applied to systems development. We'll review some of these lessons below.

Planning for Creativity and Other Uncertainties

One of the lessons software development experts have learned is that detailed plans are rarely followed. According to the Standish Group,[3] 49 percent of software projects eventually are completed, but not according to initial plan. In the end, most projects miss on either schedule or budget -- or, as Table 1 shows, it's even more likely that projects will deliver content different from the content originally planned. (Note, however, that the percent of projects that fail completely is on the decline.)

Table 1: Software Project Outcomes[4]

Year	Delivered on time, budget, original content	Delivered, but not according to original plan	Failed to deliver
2000	28%	49%	23%
1998	26%	46%	28%
1996	27%	33%	40%
1994	16%	53%	31%

In order to have detailed plans, a manager must have precise estimates of the effort and duration of development. If you have enough experience to provide good code size estimates, then you can make tight activity and delivery estimates. However, if your system requires exploration of the problem and solution space, then estimation is more difficult. For example, according to the authors of the very sophisticated COCOMO estimation model,

> Over those 161 data points, the '98 (*COCOMO*) release demonstrates an accuracy of within 30% of actuals 75% of the time (and within 30% of the actuals 80% of the time after stratification by organization) for effort, and within 30% of actuals 72% of the time (within 30% of the actuals 81% of the time after stratification by organization) for a nonincremental development schedule.[5]

Classical systems development processes require that each activity, be it project planning, requirements analysis, or design, must have a highly precise, detailed solution before the team proceeds to the next activity. If, over time, the artifacts are found to have flaws or gaps, these are considered to be team failures. However, because of the large problem and solution spaces, many modern projects have more uncertainty at their onset than did more traditional systems. There is only a general understanding of the stakeholders and their needs. Further, given the challenge and opportunities involved in meeting stakeholder needs, it is likely that the solution will take some creativity, which introduces even more uncertainty into the process.

Finally, as the project team grows, how the project proceeds is the result of thousands of individual decisions made by team members as they struggle to work together to develop the system. It is hardly surprising that it is tough to predict how the project will proceed.

To take a quantitative approach, let us consider the probability of completing a project plan specified as a set of activities with durations, as in a Gantt chart. Using a simple model, the probability of completing the project as planned is the joint probability of completing all N of the tasks, as in the equation:

$$P(Completing_Plan) = \prod_{n=1}^{N} P(Completing_task_n)$$

Given the uncertainties and time pressures inherent in software and systems development, let's say the manager sets the task durations so that there is a 95 percent chance of completion (I suspect most plans lack even this level of certainty at the task level.). In this case, the probability of completing the plan for a project of N activities is given in Table 2.

Table 2: Probability of Completing Initial Plan

Number of Tasks	P (completing plan)
10	.59
25	.27
50	.07

100	.006
1500	10^{-23}

Clearly, the greater the number of tasks, the lower the chances of completing the plan. By making your initial plan less detailed, you can give yourself more flexibility to deal with the project's inherent uncertainties and increase the likelihood that you'll actually follow the plan.

Meeting the Challenges

Given the impossibility of making accurate detailed plans for large, creative software projects, logically, the goal of modern systems management should be to better manage programs with this inherent uncertainty. On the other hand, building physical systems seems to *demand* detailed plans and early commitments in order to deal with the lead times required for developing -- or even acquiring -- software. In fact, business processes that involve systems or product development may require detailed planning and commitments that seem *impossible*. A leadership and management challenge is to deal with this tension.

Although there is no simple solution, there is a set of principles that project leaders have applied to successfully complete system projects. We'll discuss these in the remainder of this section.

Defining Project Success

The first key is to rethink what constitutes project success. If your notion of a successful project is one that lies in the first column of Table 1, then you must restrict your efforts to delivering those projects whose problem and solution spaces are so well understood that detailed estimates and plans are possible. Generally, this means repeatedly delivering the same system with small variations. However, if you define a successful project as one that requires creativity, then it is unreasonable to expect you can meet the criteria in that first column. Rather, you should strive to meet those in the second column -- to deliver a system that provides value and balances stakeholder (user, owner, and investor) needs. Then, the measure of success is not how closely you adhere to the plan, but rather the value of the resulting system.

Results-Based Development

If it is impossible to make detailed plans, then, by implication, you need to move from activity-based to results-based development. That is, the role of management shifts from seeing that planned activities are carried out to ensuring accomplishment of project goals. As the project proceeds, the focus is on reducing uncertainty rather than completing an activity.

The choice of lifecycle milestones should reflect the same focus. In the Rational Unified Process® lifecycle (Figure 1), the four phases focus on removing uncertainty and then driving stakeholder agreement that the

risks are removed.[6] The first phase, *Inception*, focuses on the problem space: determining what system needs to be delivered, its boundaries, interactions, services, and so forth. The phase is complete when all stakeholders agree with the system description. The second phase, *Elaboration*, focuses on developing a sufficiently robust architecture to meet stakeholder needs. The third phase, *Construction*, focuses on incrementally building the system to remove risks to successful delivery. Each of the iterations in this phase provides more capability than the last, and therefore can pass more of the system tests. The fourth phase, *Transition*, concludes with successful adoption in the field.

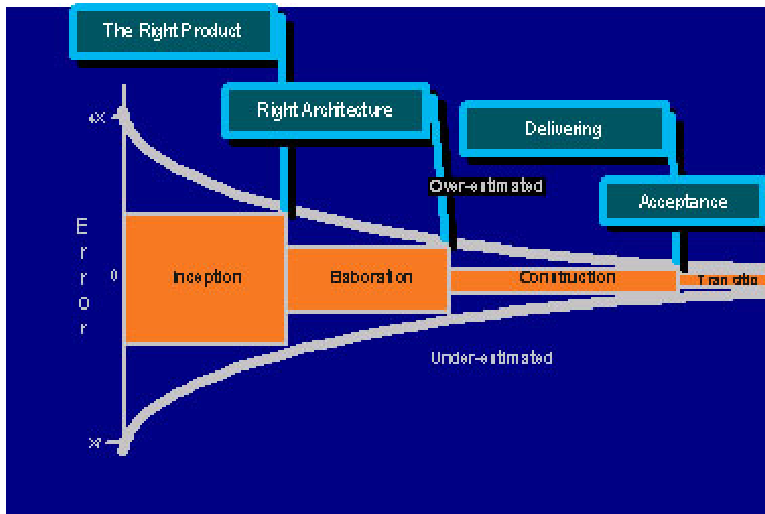

Figure 1: The Rational Unified Process Lifecycle Phases Focus on Risk Reduction

There are some important guidelines in applying this lifecycle model to achieve results-based development:

- Do what it takes.
- Breadth before depth.
- Steer the project.
- Manage results.

Let's explore each of these guidelines briefly.

Do what it takes. In this lifecycle model, during each phase the team is not restricted to a single activity, but instead carries out the needed mixture of requirements analysis, architecture, implementation, and test to meet stakeholder goals. The appropriate mix depends on the project. For example, in some projects, the team might build a version of the system during Inception to test acceptance of the human/system interaction model. During Elaboration, they might build a version of the system to show that the architecture is feasible and meets performance goals. During Construction, the system is delivered in increments, with ongoing integration and test. This practice provides a mechanism for removing technical risk as early as possible and prioritizing content.

Breadth before depth. The completion of each phase is not based on getting all details of the system requirements design correct. Trying to achieve 100 percent completeness and precision is expensive; it also flies in the face of Pareto's Law (the final 20 percent of the benefit takes 80 percent of the effort). In addition, trying to achieve 100 percent completion and precision creates a false impression of accuracy. It is likely that developers will introduce many errors that will be difficult and expensive to remove later in the process. In practice, most project documents are inconsistent in their level of detail: Often, they provide a lot of detail about what was initially understood but little about the overall scope of the project. To achieve process lifecycle goals, you need stakeholder agreement on the breadth of the project's scope and architecture; it is important to be as accurate as possible, but to stay at an appropriate level of precision.

Steer the project. As the project evolves, you remove uncertainty and gain information. The team better understands stakeholder needs and has identified and mitigated technical risks; their ability to deliver results is better calibrated. As this information accrues, the project manager can add details to the project activities, resetting tasks and assigning resources so that, in the end, the stakeholders' needs will be satisfied.

Manage results. As activities are adjusted as the project proceeds, there is little value in planning, tracking, and managing activities. Instead, project managers should be focused on achieving the desired results: working code; architecture that has been reviewed and approved; and stakeholder agreements. This focus should be reflected in all project management artifacts, including the work breakdown structure and earned value methodology.

Use Architecture to Manage Communications

Based on experience in the software management realm, we know that as a project grows in size and complexity, it becomes increasingly crucial to manage communications between stakeholders and the project team, as well as within the team. Generally, the effort associated with communications grows nonlinearly with the size of the project.[7] A primary concern is to maintain the right amount of communication; you want enough to make good decisions and keep everyone on the same page, but you don't want communication activities to dominate the effort.

The key is to develop and maintain a system architecture that provides a context for reasoning about ways to meet stakeholder needs and to document your decisions. Architecture is not a new idea, but there are some new wrinkles. In current practice, the architecture is dominated by either the physical or state machine perspective. In the first case, the architecture is seen as a set of physical resources or assemblies, and software is determined at the hardware boundaries. In the second case, the architecture is seen as either object classes or Unified Modeling Language (UML) subsystems whose code is then deployed on the hardware. In practice, both views (and others) provide insight into the system and a way to separate concerns. The different views can be used to address different quality issues and technical risks. There are ongoing

efforts to promote use of the UML as a basis for this unified architecture framework. Rational has been working with customers to develop an extension of the Rational Unified Process for systems engineering (RUP SE)[8] (Rational Software Corporation, 2002) that provides such a UML-based framework.

Once this architecture is developed, it is maintained and evolved throughout the development process. It serves many functions. For example, it might enable the team to partition the development, so they can work concurrently on design and engineering while maintaining overall design integrity. Further, it can provide intellectual property that enables another team to rehost the system or add capability. It can also serve as a basis for setting iterations plans to reduce technical and content risks early in the project.

Meet Systems Development Challenges with Software Development Best Practices

As we said earlier, modern problems require modern solutions. The problems facing systems development leaders have changed over the last decades, and today, systems development projects have many of the same inherent uncertainties that have long plagued software development. Accordingly, adapting management best practices learned from managing large software efforts can be effective in meeting current systems development challenges. These best practices are embodied in the Rational Unified Process, which provides support and guidance for applying them.

References

- Barry Boehm et al., *Software Cost Estimation with Cocomo II*. Prentice Hall, 2000.
- Fredrick Brooks, *The Mythical Man-Month: Essays on Software Engineering*, Anniversary Edition (2nd Edition). Addison-Wesley, 1995.
- COCOMO Web site, Center for Software Engineering (CSE): http://sunset.usc.edu/research/COCOMOII/
- Philippe Kruchten, *The Rational Unified Process: An Introduction*, 2nd Edition. Addison-Wesley, 2000.
- David Longstreet, "Software Productivity Since 1970," 2002. http://www.ifpug.com/Articles/history.htm
- Rational Software Corporation, "The Rational Unified Process for System Engineering (RUP SE)." Technical Paper 165. http://www.rational.com/media/whitepapers/TP165.pdf.

Notes

[1] David Longstreet, "Software Productivity Since 1970." http://www.ifpug.com/Articles/history.htm, 2002.

[2] *Ibid.*

[3] *CHAOS Chronicles II*. Standish Group, 2001.

[4] Data from Standish Group, 2001.

[5] Center for Software Engineering (CSE), http://sunset.usc.edu/research/COCOMOII/, University of Southern California, 2002.

[6] Philippe Kruchten, *The Rational Unified Process: An Introduction*, 2nd edition. Addison-Wesley, 2000.

[7] See Barry Boehm et al., *Software Cost Estimation with Cocomo II* (Prentice Hall, 2000) and Fredrick Brooks, *The Mythical Man-Month: Essays on Software Engineering*, Anniversary Edition (2nd Edition; Addison-Wesley, 1995).

[8] Rational Software Corporation, "The Rational Unified Process for System Engineering (RUP SE)." Technical Paper 165. http://www.rational.com/media/whitepapers/TP165.pdf.

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

Copyright Rational Software 2003 | Privacy/Legal Information