By I. Robert Chiang and Vijay S. Mookerjee

IMPROVING Software Team Productivity

Choosing the most effective quantitative process design approach will not only improve a firm's project management but, ultimately, the bottom line.

The widespread adoption of information technology (IT) over the last few decades has helped organizations reap numerous operational and strategic benefits. Consumers have also benefited from IT as it reduces market frictions caused by geographical separation, price opacity, and information latency. The resulting increase in demand for IT products and services has created new challenges for delivering IT solutions involving hardware, software, and networking components under ever-tightening deadlines.

While hardware speed and network capacity have made impressive strides through manufacturing automation and technological innovation, software development has not improved under the same order of magnitude. As a result, the software component of information systems chronically causes project delays, cost overruns, and customer dissatisfaction.

There are several factors that make software systems difficult to develop, notably the complexity of software that demands human intervention throughout its creation. As a result, massive automation of software development (for example, using automatic code generation) has not been realized. Furthermore, the innovative nature of software makes it difficult to leverage experience and team knowledge across projects. Finally, software—being intangible—complicates measurement and resists the quantitative analysis necessary for continuous productivity improvement.

It is widely recognized that improving software development productivity requires a balanced approach toward the three pillars [8] of software management: technology, people, and process. Much effort has been devoted to refining software-building technologies, with significant results. For example, sophisticated compilers, middleware, and scripting technologies have increased programming speed; robust utilities have allowed for easy bug tracking and configuration management; and communication applications and networking have made it relatively easy to keep project and system information transparent.

Due to the human-centric nature of developing software, however, the benefits of technological improvements cannot be fully realized without a capable work force. Thus, it is also critical to improve individual competency. However, investing in human capital requires long-term planning and commitment, and does not produce immediate payoffs. CMU/SEI's People Capability Maturity (P-CMM) framework[1] has provided recommendations on how organizational changes can be carried out to facilitate better management and development of the work force.

[1]For more details visit www.sei.cmu.edu/publications/documents/01.reports/01mm001.html.

A third way of increasing productivity is to refine the development process. The benefits of process improvement are not limited to accelerating the development work but also reducing the effort spent on corrective activities. Without a proper development process in place, a project team could operate in a chaotic manner, resulting in low productivity and poor system quality [11].

Previous literature (such as Brooks' incremental development [1] and Boehm's spiral model [2]) suggests that system development should be an act of gradual enhancement rather than a forced assembly of software components. Brooks has found that incremental development—interleaving development work with periods of testing and debugging—leads to easy backtracking and natural prototyping. Frameworks for incremental development, however, are often qualitative in nature and offer no precise guidance on how incremental development can be optimally implemented when faced with a vast array of system, personnel, and technical factors.

Here, we advance a variety of quantitative process models and policies to better manage incremental development. The main focus is on the *system construction* phase, that is, the stage of a project in which the system is actually being coded. The specific activities that occur during system construction are coding and unit testing, (software) module integration (integrating a module with the rest of the system), team communication (for example, walkthroughs, peer reviews, project meetings, and so on), and system integration (system-level debugging).

These activities are depicted in Figure 1 forming a *construction cycle.* At the end of each cycle, a *system baseline* (consisting of a collection of stable software modules) is established upon which future development can be based. Typically, several construction cycles are needed before the entire system is coded and debugged.

With the exception of Step 1, the activities depicted in Figure 1 are coordination related. Most real-world system construction requires coordination among project participants such as developers, testers, designers, and system users. In the context of software development, Kraut and Streeter [7] have described coordination to mean any activity that facilitates different people working on a project to have a common definition of what they are building, to share information, and to mesh their efforts with one another.

Coordination Problem and Approaches

Here, we provide examples of coordination decisions in real-world software organizations. Specifically, we discuss how organizations address the question:

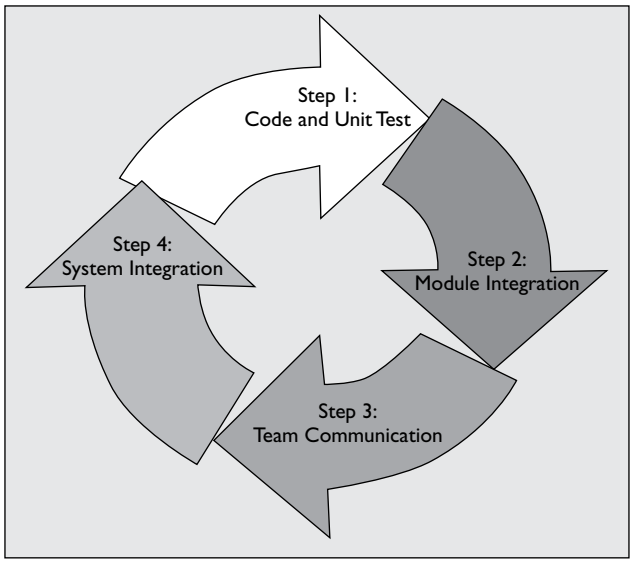


Figure 1. Activities in a construction cycle.

When is the best time to coordinate? This is important to answer because delaying coordination beyond a point can lead to costly rework. At the same time, premature coordination can be counterproductive because it can disrupt development work.

We describe three current approaches being used: Big Bang, frequent integration and periodic synchronization, and fault-driven. In each approach, there is a different coordination trigger.

Big Bang. The Big Bang approach is one where all coordination occurs at the end of the project. In this approach, Steps 2, 3, and 4 of Figure 1 are put on hold until Step 1 is completed. This approach follows the Waterfall process model and thus is not an incremental development technique per se. Since the timing of coordination is not actively managed, this approach incurs low project management overhead and works well for small and competent teams working on well-defined projects [9].

The main drawback of Big Bang coordination is scalability. Software components in a system can have intricate interactions; a fault, if not removed in a timely manner, either becomes more difficult to rectify and/or causes additional faults to occur later in the project. Such a downstream effect makes it very costly to use Big Bang for large projects. Also, this approach does not benefit from learning that could accrue if the team is allowed to coordinate periodically.

Frequent integration and periodic synchronization. Software organizations are now performing module integration (such as Step 2 in Figure 1) more frequently—often on a daily basis. A well-publicized approach adopted by Microsoft, the "Daily Build and Smoke Test," has been used in many of the

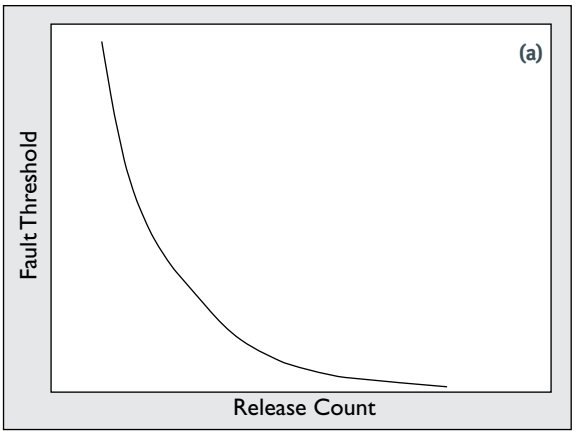
firm's high-profile development projects [4, 12]. In addition to frequent module integration, periodical team communication and system integration are performed to ensure product quality throughout its construction.[2]

While more frequent coordination on the module and system level help alleviate downstream effects, an important question remains unanswered: How long should development in each construction cycle last? We have noticed the timing of coordination is often ad hoc in many organizations. In this regard, there is some evidence of the use of a bell-shaped coordination policy where coordination is intense at the beginning of the project, relaxes at the middle, and becomes intense again near the end. A bell-shaped coordination policy can be explained as a result of two factors: team learning and system stability.

Fault-driven coordination. While the time-based coordination policy has clear administrative benefits, it does not appear to fully capture the dynamics of an evolving project. Specifically, coordination is more urgent when the system appears to be going out of synch, otherwise it may be appropriate to allow development work to continue. This implies the coordination decision should somehow be tied to the health of the system. Several firms that we contacted used a fault-driven coordination policy.

With advanced development and project management tools, it is now possible to obtain system fault data and other related metrics on a near continuous basis. Using current system fault (bug) count and severity metrics, project managers can schedule coordination whenever the average cost to fix a fault is expected to rise. Figure 2(a) shows a fault threshold curve that can be derived with this reasoning. The shape of a threshold curve depends on specific project factors; for example, the complexity of the system, team size and experience, the development environment, the project domain, and the available time to construct the system [3].

In a fault-driven policy, coordination occurs at the release of a module if the observed fault count exceeds the threshold associated with the release. The downward slope of the policy curve ensures that unless there are a large number of faults, coordination should not occur when relatively few modules have been released. On the other hand, thresholds decrease for high release counts to encourage coordination unless the development work is of extremely high quality. Figure 2(b) compares time-based and fault-driven coordination policies and shows that fault-driven coordination is consistently more effective. The use of system fault data for making coordination decisions is particularly beneficial for projects with a compressed schedule (that is, high ratio of work required to time available).



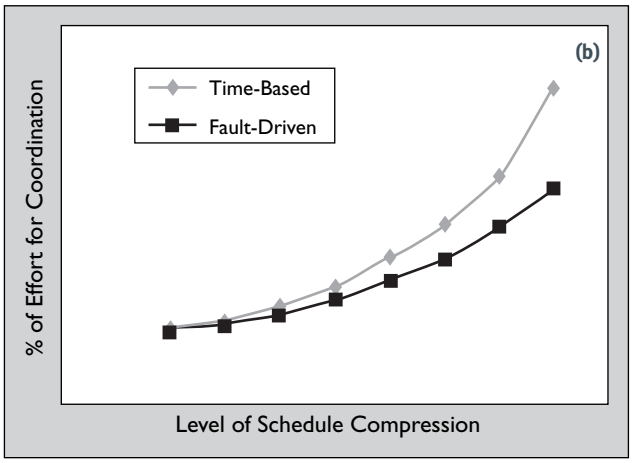


Figure 2. Fault-driven coordination. (a) Fault threshold curve. (b) Benefit of fault-driven coordination.

Factors Affecting Coordination

Conceptually, the intensity of coordination is the proportion of effort spent on coordination in one construction cycle. We have clustered the factors affecting coordination intensity into four groups: project, team, system, and technology.

A critical factor affecting team productivity is the allowable system construction time. For a given set of requirements, a shorter construction time should translate to a larger team and thus more expensive coordination among the team members; hence, increasing team size diminishes the productive output per member. This "mythical man-month" phenomenon implies that increasing team size beyond a point

[2]Our study on a NASA project revealed that about 10–15% of the system construction effort could have potentially been saved if the Big Bang approach had been replaced by an approach involving periodic synchronization [10].

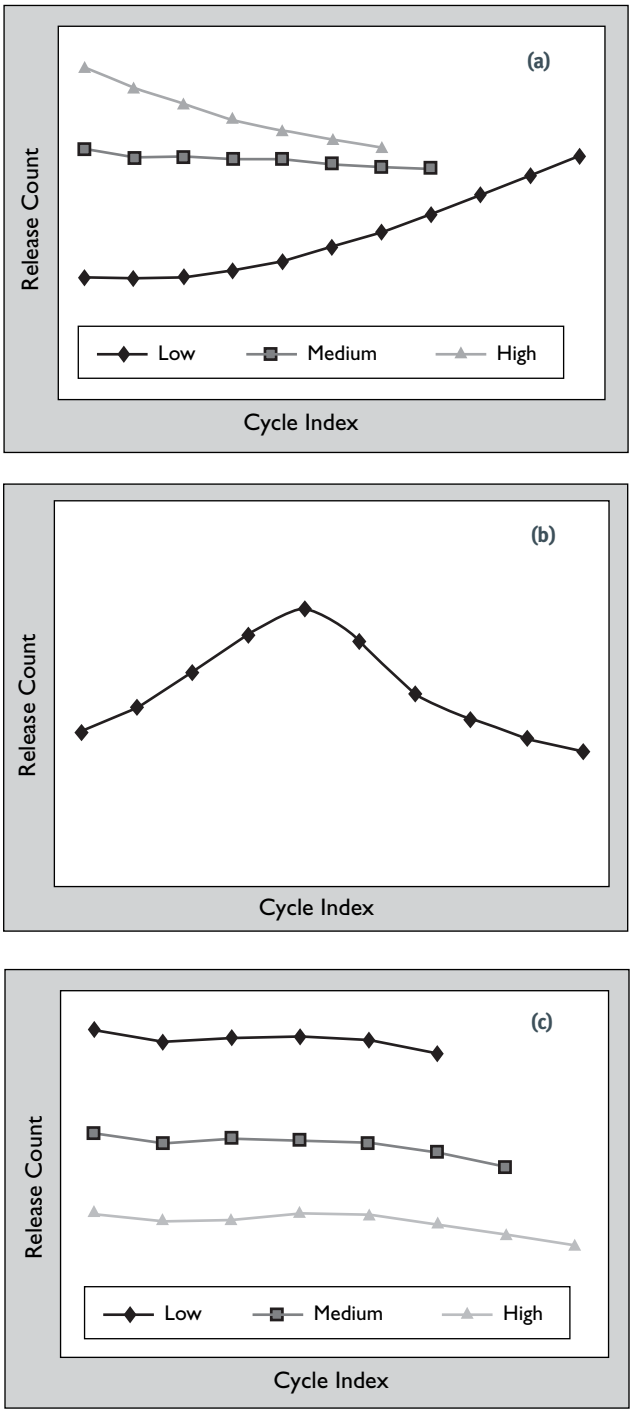


Figure 3. Change in coordination policy curve: (a) Effect of system stabilization rate; (b) Effect of team learning; (c) Effect of system complexity.

can actually lengthen the project duration. The remedy when faced with a large team is to establish a hierarchical communication structure so that relatively small groups of developers can focus on well-interfaced system components [10].

Team skill and learning effect. A notable advantage of using incremental development is the availability of several baseline versions of the system. Modules that are part of the system baseline are no longer subject to major revision, thereby reducing two-way module adaptation to one-way accommodation (new modules must be consistent with baseline modules). The policy curve in Figure 3(a) suggests that project teams that can stabilize modules fairly quickly can afford to coordinate less frequently early on in the project. On the other hand, with more volatility, frequent coordination is necessary until sufficient familiarity and stability can be achieved. For example, a project undertaken by a less experienced team can be expected to stabilize more slowly, thus the team should coordinate more frequently. An indication of slow stabilization is that core system components (such as system interfaces or driver modules) continue to undergo change late into the project.

Many studies have shown a tenfold productivity gap between novice and proficient developers. One trait of top-notch developers is their ability to acquire project-specific knowledge and avoid major rework—even in an unfamiliar project domain. Evidence of this learning process is the development team stabilizes (or "baselines") newly developed modules more efficiently as the project progresses. The policy curve in Figure 3(b) shows that relatively tight coordination is appropriate for such a team at the start of the project to foster learning. The team should then focus on programming tasks with less intervention until near the end, when tight control is again necessary to mitigate schedule risk. Thus, the coordination scheme in Figure 3(b) should be recommended to a skilled team assigned to a project in a new domain.

Software characteristics. Everything else being equal, the higher the system (architectural/design) complexity the more intense the coordination should become. In more complex systems, intermodule discrepancies are more prone to occur and are more difficult to rectify. Thus, more intense coordination helps to keep faults under check. Figure 3(c) shows the optimal coordination policy at different levels of system complexity. The total coordination cost is found to also increase with the system complexity. One implication of Figure 3(c) is that it provides us with a basis for design economics. Faced with limited resources or a tight schedule, project managers are often tempted to forego system design tasks for more productive coding tasks. Often, however, they may be better off investing in a good quality design to avoid costly "fire fighting" later during construction.

Technology and tools. Changeover efficiencies can be measured by how easily team members toggle their work environment between development and coordination. Another measure is the program comprehension support provided by the project environment. In

contrast to the effect of structural complexity shown in Figure 3(c), coordination should be more intense when project changeover is more efficient. For example, in an organization with sophisticated CASE support, the development team should be encouraged to coordinate more frequently. Technological innovations offered by new tools are often regarded as silver bullets for programming speed. However, if used properly, they can also reduce coordination overhead.

Implications and Next Steps

Software development organizations now enjoy a wide assortment of tools and technologies that promise to enhance development productivity. Yet in an industry that practices "creative destruction," speed and quality advantages gained from technology are being harshly challenged by elevated user expectations. We believe that only through a quantitative management of the project process will we achieve the best use of personnel and technology.

Release economics. An interesting area for future research is to develop models and theories of heterogeneous software systems. In previous studies, the software system is typically viewed as a collection of (more or less) identical modules. While the homogeneity assumption has obvious analytical benefits, there are many situations in which software modules may possess significant differences in size, complexity, and functionality. A possibility is to link the sequence in which modules are developed with business factors such as user functionality, revenue arrangements, and development cost. In other words, it may be possible to develop an economic basis for the release of software modules and then tie in coordination concerns based on the actual, rather than an average release pattern.

Coordination economics. A measuring stick for a software organization's maturity is whether it delivers consistent productivity in the long run. Most productivity studies focus on reducing the direct coding effort. However, a team's productivity is also profoundly influenced by how well coordination efforts are distributed. An analogy from Amdahl's Law indicates that unless coordination (dependency) is controlled, it is unlikely that quality software can be produced faster simply by adding more people. We argue here that proper process design is the key to better productivity.

Many organizations view the development process as a deterrent rather than a catalyst for productivity because the development process is often used to enforce the sequencing of project tasks without explicitly considering coordination activities. Another misperception is that establishing an infrastructure for development is a costly proposition. However, Glass [5] and the discussions here argue imposing discipline on the development process could have dramatic payoffs.

Managing expectation. Blindly seeking productivity improvements to meet customer expectation is wishful thinking if the expectations are set with little chance of achieving them [6]. Many project managers are unable to conduct sensitivity analyses for studying the consequences of change requests on the project cost or schedule. Project attributes such as system complexity, stabilization rate, and team learning provide guidelines on how to assess the impact of midcourse project adjustments on the project schedule.

This article attempts to bridge the chasm between analytical and empirical views of project management to provide guidelines on how software projects can be best managed. We are currently investigating the coordination design issues for massive customization (for example, SAP systems) and open source (decentralized) projects. C

References

1. Brooks F.P. *Mythical Man-Month: Essays on Software Engineering, Anniversary Edition,* Addison-Wesley, Reading, PA, 1995.
2. Boehm, B. A spiral model of software development and enhancement. *IEEE Computer* (1988), 61–72.
3. Chiang, I.R., Mookerjee, V.S. A fault threshold policy to manage software development projects. *Information Systems Research* (2004), Forthcoming.
4. Cusumano, M., and Selby, R. *Microsoft Secrets,* Free Press, 1998.
5. Glass, R.L. An embarrassing, yet rewarding, ending to a previous column. *Commun. ACM 44,* 1 (Jan. 2001), 11–13.
6. Glass, R.L. Evolving a new theory of project success. *Commun. ACM 42,* 11 (Nov. 1999), 17–19.
7. Kraut, R., and Streeter, L. Coordination in software development. *Commun. ACM 38*, 3 (Mar. 1995), 69–81.
8. Landis, L., McGarry, F., Waligora, S., et al. Manager's handbook for software development (Revision 1). NASA Software Engineering Laboratory, SEL-84-101, (Nov. 1990).
9. McConnell, S. *Rapid Development.* Microsoft Press, 1996.
10. Mookerjee, V.S., Chiang, I.R. A dynamic coordination policy for software system construction. *IEEE Trans. Software Engineering 28,* 7 (2002), 684–694.
11. Paulk, M.C., Curtis, B., Chrisses, M.B., Weber, C.V. Capability Maturity Model for software. CMU/Software Engineering Institute, Technical Report, CMU/SEI-93-TR-024, (Feb. 1993).
12. Zachery, G.P. *Showstopper!* Macmillan, New York, 1994.

I. Robert Chiang (Robert.chiang@business.uconn.edu) is an assistant professor of information management at the School of Business, University of Connecticut, Storrs, CT.
Vijay S. Mookerjee (vijaym@utdallas.edu) is a professor of information systems at the School of Management, University of Texas at Dallas, Richardson, TX.