Concurrent Software Development

Joseph Blackburn, Gary Scudder, and Luk N. Van Wassenhove

By necessity, software development has become a critical skill for many industrial firms. Software that captures the intellectual assets of the firm in its products and services increasingly defines the critical path in development and thus governs the firm's speed-to-market. When embedded in hardware, such as with a television or an office copier, software can be a particularly strong determinant of development cycle time.

What happens when software development exceeds its time targets and is late to market? One example, widely reported in the press, illustrates that dilatory software development can devastate the bottom line and affect the boardroom. In 1994, Novell purchased WordPerfect for over $855 million in an effort to create an integrated software product to compete with Microsoft's Office suite; Novell later sold WordPerfect (and Quattro) to Corel for $186 million. What caused the calamitous 80% drop in market value? Simply Novell's inability to keep apace with Microsoft in the race to bring new software features to market. Speed is obviously a key to retaining a competitive edge in these markets.

Software productivity is another key development performance metric with direct financial consequences. Firms such as Microsoft recognize that software development is a fixed cost business with virtually no variable production costs; higher productivity thus results in lower input costs, and higher profit margins [6].

The twin objectives of speed and productivity raise vexing issues for a software development manager. Cycle time and productivity are not perfectly correlated because a developer can achieve a shorter cycle time even with low productivity, by adding developers to the project. Brooks [4] and others have noted that the practice of adding bodies to a project to lower cycle time may have the opposite result, since coordination complexities make larger teams more difficult to manage. With low productivity, speed is achieved at high cost.

Fortunately, the pursuit of speed and productivity is not a zero-sum game. The

Gary D. Scudder (gary.scudder@owen.vanderbilt.edu) is a professor of Management at the Owen Graduate School of Management, Vanderbilt University, TN.
Joseph D. Blackburn (blackburn@owen.vanderbilt.edu) is Acting Dean and a professor of Management at the Owen Graduate School of Management, Vanderbilt University, TN.
Luk N. Van Wassenhove (Wassenhove@insead.fr) is Full Professor and Area Coordinator of the Technology Management Department at INSEAD, Fontainebleau, France.

empirical research described in this article outlines a clear path for development managers that leads to both higher productivity and shorter cycle times, eliminating the need for a trade-off. These results form a software development management framework, called Concurrent Software Engineering, that integrates people into a coherent, structured management process and makes software development a positive-sum game. Our framework reinforces and extends many of the principles captured in the Software Engineering Institute's Capability Maturity Model [9, 11].

Empirical Research

Our framework of people and process is derived from a four-year, empirical study in Japan, the U.S. and Western Europe, of speed and productivity in software development. In 1992, we began interviewing hardware and software developers in Europe to gain insight into the issues affecting lead-time and the differences between hardware and software processes. Intrigued by the growing importance of software and the complexities of hardware/software development, we decided to focus our study on software development and designed a survey instrument to highlight the management practices that best support shorter software development cycles and greater productivity. The first surveys were conducted in 1992–93 in the U.S. and Japan [3]. From these surveys we developed a sample of 49 responses from large, well-known firms that we believed represented best practice. A 1994 European survey resulted in 96 responses from managers of software projects; firms were similar in size to those sampled in the U.S. and Japan [2]. Except where noted, the results discussed in this article are based on the global sample of 145 responses.

We asked survey respondents to describe a recently completed software project, including the project type, language used, management procedures, and performance measures. Respondents also supplied quantitative information about the project in four areas: project size and productivity over time; allocation of time and effort among project phases; the effectiveness of different management tools and techniques for compression of the development process; and stages of the process in which time reductions were achieved. In the European sample we also collected input on the team size in various project phases and the newness of project (from minor revision to completely new).

To estimate increases in development speed, we asked respondents: "If you had developed the same software product line five years ago, how long would the project have taken?" To estimate productivity, we computed lines of code per person-month of coding and per total project person-months. Although lines-of-code (LOC) has major shortcomings as a productivity metric, it is the only measure in wide enough use to be understood with any consistency by respondents in a survey.

To learn about the importance of certain project management factors, we also asked respondents to identify factors that helped reduce overall software development time. Eleven factors were chosen that have been identified in the literature and our interviews (and were ranked on a 1–5 Likert scale):

1. The use of prototyping to demonstrate how the proposed software will work;
2. Better customer specifications initially;
3. The use of CASE tools and technology;
4. Concurrent development of stages or modules;
5. Less rework or recoding;
6. Improved project team management;
7. Better testing strategies;

8. Reuse of code or modules;
9. Changes in module size and/or linkages (smaller modules and standard interfaces speed coding and testing with parallel development);
10. Improvements in communication between team members; and
11. Better programmers or software engineers.

Our initial probes of the data were focused on cross-cultural differences in product development; these results are discussed in [3]. Somewhat to our surprise, we found an overall lack of global differences in software practices. Globally, firms allocate time and effort similarly, as well as weight the 11 project management factors similarly. The global sample weights are displayed in Figure 1.

But important differences emerged when we divided the data along lines of performance and best practice, rather than culture. When we ranked the respondents according to speed and productivity and correlated these measures with the project management descriptors, we found that fast firms and high-productivity firms, whatever their nationality, tend to have more in common than firms within a country.

Correlating speed and productivity with the 11 project management factors, team size, and allocation of time and effort in the different development stages, yielded counterintuitive results about time compression and software productivity that we call the Time and Productivity Paradoxes. To solve these conundra we employ Concurrent Software Engineering—our framework for managerial action.

The Time Paradox

The fastest firms in software development understand that to have the shortest cycle times, not every element of the cycle must be faster than the competition; some parts of the process need to be slow. This unconventional wisdom also extends to productivity. The key insight uncovered by our empirical research is that *more* time and effort (in person-hours) invested in the early stages of a software project yields faster cycle times and higher productivity.

Figure 2 displays the combined data from our global sample (U.S., Japan, and Europe) on the relationship between development speed and the allocation of time across project stages. We divided our sample into fast firms (those that reported more than a 25% decrease in development cycle time over the past five years) and slower firms (less than 25%). Figure 2 shows that the faster firms spend more time in the customer requirements stage and less in subsequent stages. (with the exception of the coding stage).

The data on coding productivity from the global sample shows a similar pattern. When the global sample is partitioned based on reported LOC productivity, it is clear that the high productivity firms devote more cycle time and person-hours effort to the customer requirements stage. Figure 3 displays the percentages for cycle time and effort in a combined chart with the percentage allocations for time and effort among the low productivity firms as a stacked column; an area chart is used to show the stacked percentages for the high productivity firms. This figure demonstrates that the high-productivity firms spend more time and effort in the initial, customer requirements stage and less in coding/implementation and testing/integration.

Statistics from the more recent European sample confirm these observations (details of the statistical analysis may be found in [2]). Development speed and productivity are significantly correlated with the amount of effort in the customer requirements stage, but for all other stages, the correlations are negative or insignificant. Taken together, the

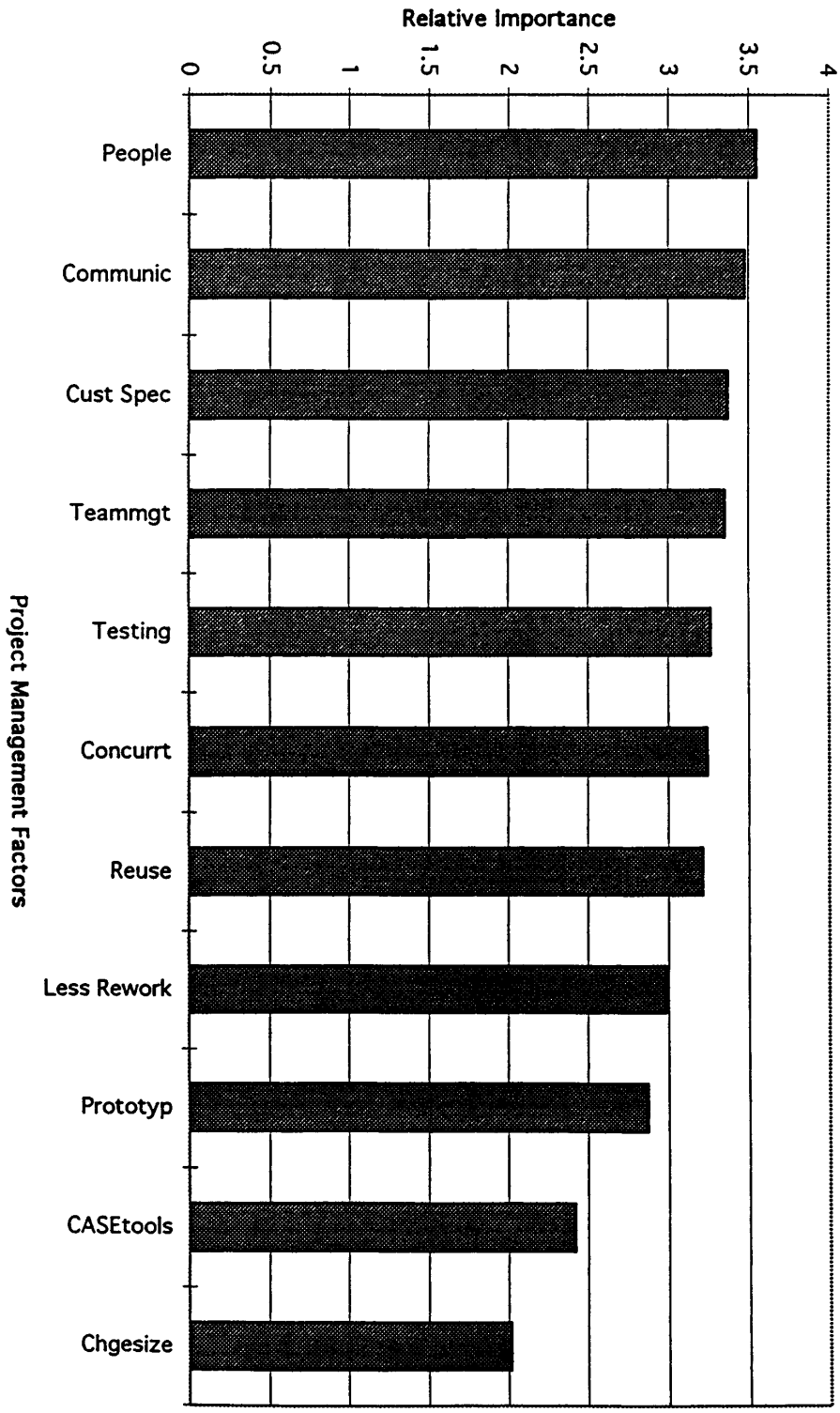

Figure 1. Importance rating: Project management factors (global sample).

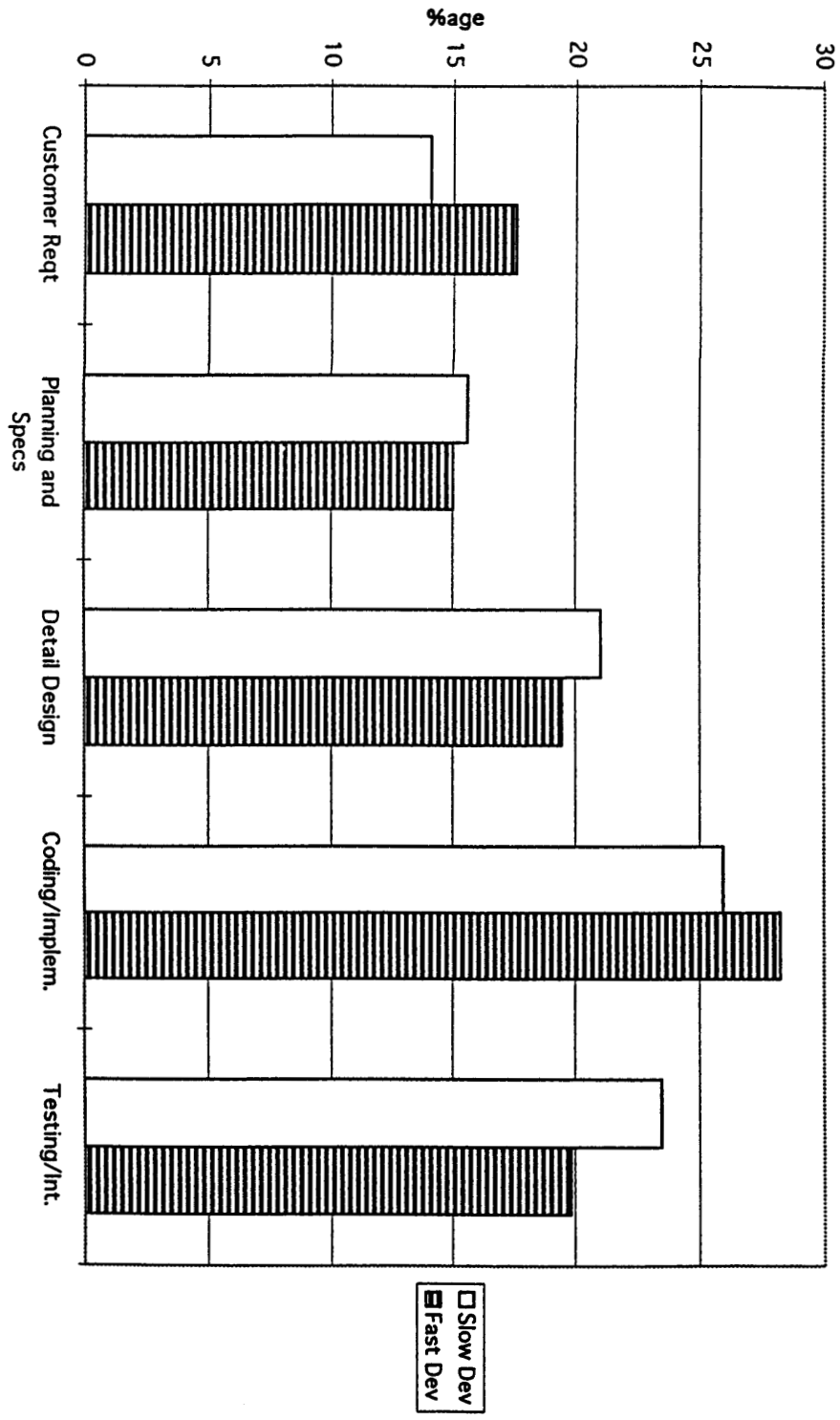

Figure 2. Percentage allocation of time by project stage (fast vs. slow developers: global sample).

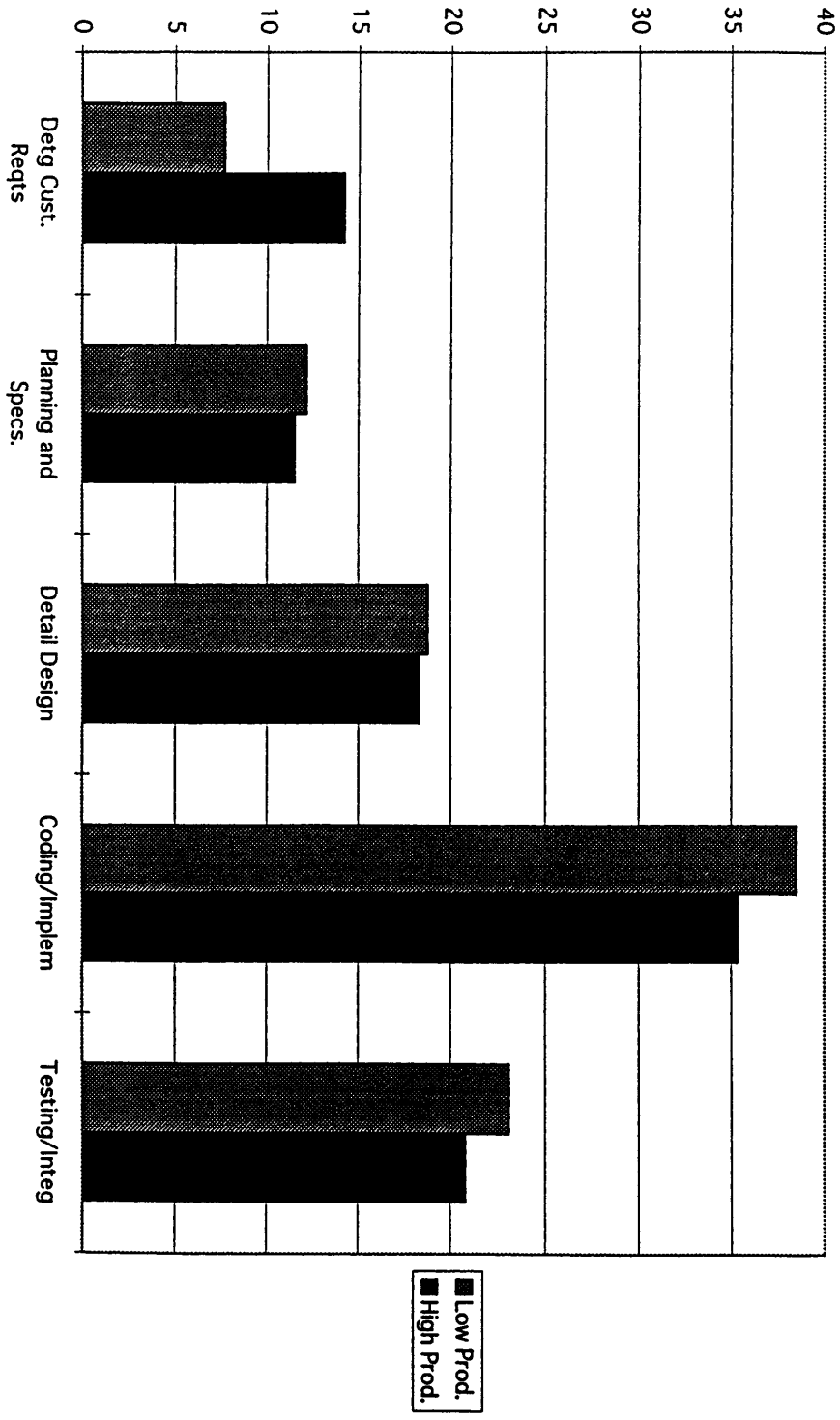

Figure 3. Percentage allocation of effort by project stage (high vs. low productivity developers: global sample).

data in Figures 2 and 3 and the supporting statistical analyses strongly indicate that additional time and effort in the early stages of a project result in reduced time-to-market and greater coding productivity.

Why is more time needed up front? Follow-up interviews with software managers provide one answer: "It's the changes that kill us" is a common complaint. The leading source of time delays software development is rework: the redesign/recoding/retesting cycles made necessary by changes in requirements, in interfaces, and so forth. In the early stages of a development project, the old maxim "haste makes waste" holds true.

The message to managers is clear: invest more time up front to shrink total cycle time and increase productivity later on. To gain speed and productivity, managers must spend more time learning precisely what customers need in a software product and converting those needs into unambiguous specifications.

The Productivity Paradox

Team size in software development lies at the heart of a productivity paradox: responding to deadline pressure, managers add human resources to the project and find to their dismay that not only does productivity degrade but the time delays expand (aggregate production may actually diminish). Brooks [4] has made similar observations, and observers of high-performance work teams have also noted this "more is less" effect.

Our data confirms this effect in software. Figure 4 displays the average team sizes by stage and maximum team size for the fast and slow development groups in our European sample (firms are grouped based on their rate of change in development speed as noted in the preceding section). The figure shows that faster firms tend to have smaller teams at all development stages except for the customer requirements stage. In terms of coding productivity, the same trend is evident: the largest positive correlation between productivity and team size occurs in the customer requirements stage [2]. The subject of the time paradox described in the previous section, the customer requirements stage is the exception to the productivity paradox. Perhaps the critical importance of gathering as much information as possible about customers needs makes larger teams beneficial in this stage. Larger teams may diminish productivity in other stages because of the communication inefficiencies of larger groups. Brooks [4] has argued that communication demands increase in proportion to the square of the size of the team.

To summarize, small teams appear to benefit both cycle time and productivity. The significant exception is in the initial stage where a larger team may be better for determining customer requirements (which, if done right, may make coding more productive and less testing and correction necessary).

Resolving the Paradoxes

To gain more insight into the management practices that support faster development, we performed an analysis of variance on the European sample [2] to learn what management factors and project characteristics explain differences in development speed. We observed that more of the variability is explained by people and management process than by project characteristics. Specifically, people, prototyping, and less rework explain 35% of the difference in development speed, while project characteristics such as size, language, duration, tools and technology, and project newness collectively explain no more than 18% of this difference. Of the 11 project management

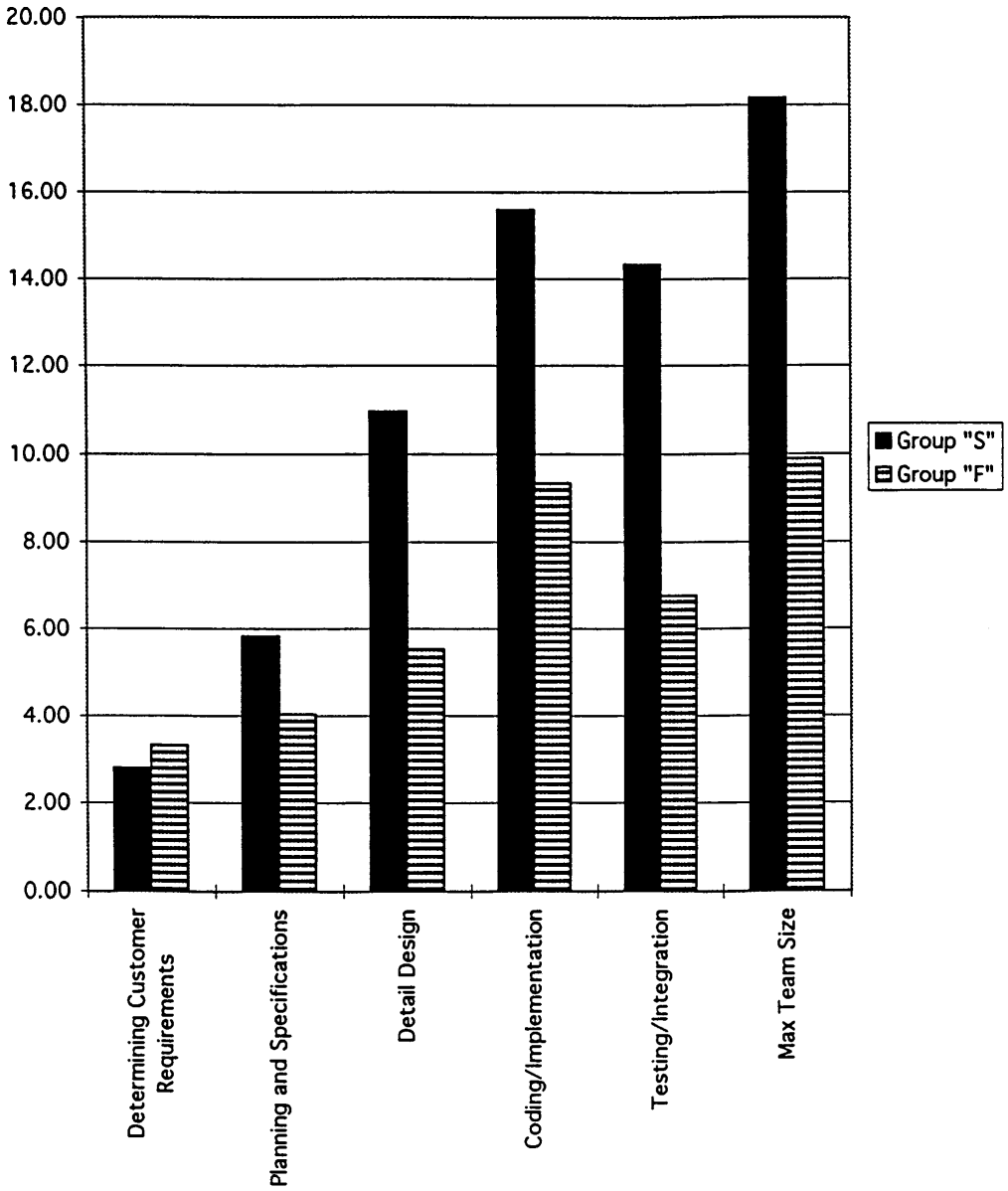

Figure 4. Average team size by project stage in European sample (fast "F" vs. slow "S" developers).

factors, only three—better people and programmers, prototyping, and less rework—had significant positive correlations with increased development speed. Figure 5, which displays a radar plot of the relative importance of these three factors for fast and slow developers, clearly shows that faster firms place relatively more emphasis on these three factors in their management of the software development process.

Many software engineering gurus continue to advocate technology solutions to the time and productivity paradoxes: CASE tools and new programming technologies. But CASE tools received low importance values in terms of reducing development time in

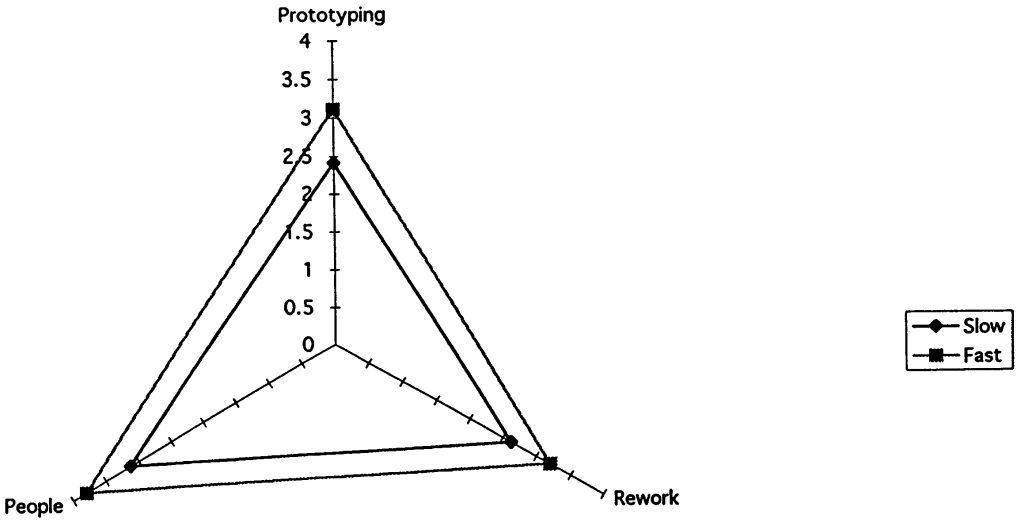

Figure 5. Importance of significant project management factors (global data set).

our interviews and the surveys. Figure 1 shows the relatively low importance given to CASE tools and technology by the respondents in our global sample. Statistical analyses of the European sample [2] found no significant correlation between the use of CASE tools and either development speed or productivity. Yeh [12] noted that "heavy investment in CASE technology has delivered disappointing results, primarily due to the fact that CASE tools tend to support the old way of developing software, i.e., sequential engineering." Research carried out by the Software Engineering Institute on software process maturity [2] suggests that the heavy use of CASE tools may be risky for immature development processes. Although data was not collected on process maturity, it is likely that most of the firms in our surveys are only at level 1 or 2 of the Capability Maturity Model. These observations suggest that process management principles may be more important than (CASE) tools and should precede the introduction of tools.

Our results on software are consistent with theoretical and empirical research on new product development in general. The literature suggests that what matters for hardware design is talented people, small teams, frequent communication, and good, up-front specification of customer requirements. Hardware and software developers have struggled with similar issues: budget and time overruns, time-consuming rework cycles, and the futility of throwing additional resources at projects managed in a sequential, over-the-wall fashion. Fixing the problem in hardware required a fundamental new process design: an overlapping, concurrent process instead of a sequential one. The concurrent process incorporates small teams working in parallel, and good communication across the functions.

The dominant process model for managing software development, the Waterfall model, is sequential. Despite a growing amount of research showing the value of concurrent engineering (CE) principles in hardware design, our interviews suggest these principles have not been embraced in software. In software, the Waterfall model is still "the best known and most widely used overview framework for the software development process."[9] Simultaneous design activity in software tends to occur within iso-

lated project stages, particularly coding and testing.

Our research and experiences in hardware development suggest that with software, it is the process that needs fixing. If a sequential, functional process is the root of the problem, then this may explain why tools and technology are proving so ineffective in speeding up software development. Attempts to automate an obsolete process have failed in other industries, and the evidence suggests that the same thing is occurring in software.

The model we propose for fixing the problem, Concurrent Software Engineering (CSE), is outlined in the remainder of the article. In this model the traditional Waterfall model is supplanted by a concurrent design model that has been found to be effective in hardware development. The CSE model we describe incorporates the management practices that have been adopted by the faster, more productive software developers.

Concurrent Software Engineering

We must first explain what we mean by concurrent engineering since no standard definition exists in the literature. CSE is often described as an unstructured set of concepts and tools. Some writers portray CSE as a simultaneous design activity and others stress the integrated team approach to design in which concerns of different functions in the organization are addressed. Neither approach alone captures the interplay between information and activities that is central to the process.

We define CSE as a management technique to reduce the time-to-market and improve productivity in product development through simultaneous performance of activities and processing of information. Activity concurrency refers to the tasks that are performed simultaneously by different groups; information concurrency refers to the flow of shared information that supports a team approach to development. In a companion article we outline an elaborate structure for Concurrent Software Engineering [1]; here we abstract the details.

Making management's task more daunting is the existence of a hierarchy of activity concurrency—within stage, across project stages, across the hardware/software interface, and across projects and platforms. Ratcheting up the hierarchy of concurrent activity, the key management responsibility is the coordination of information flows to support simultaneity.

To support concurrent activities in development, there are a number of important information flows.[1] In our framework, these flows take three forms: (1) Front Loading, (2) Flying Start, and (3) Two-Way High Bandwidth Information Exchange. Front Loading is defined as the early involvement in upstream software design activities of downstream functions—detailed design, coding, testing, and even customer service concerns. Flying Start is preliminary information transfer flowing from upstream design activities to team members primarily concerned with downstream activities. Two-Way High Bandwidth Information Exchange is intensive and rich communication among teams while performing concurrent activities. The information flow includes communication about potential design solutions and about design changes to avoid infeasibilities and interface problems.

How does CSE help resolve the time paradox? Our findings indicate that the solicitation of customer requirements and conversion to product specifications is a critical step that should not be rushed. Many voices need to be heard at this stage: users, coders

[1]Our model of information concurrency builds upon and extends Clark and Fujimoto's information processing framework for supporting overlapping problem solving activities in design [5].

and testers, hardware designers, and marketers. A key management responsibility is to ensure that all of this information (some of which comes from downstream players) is front-loaded into the requirements process. This activity should be managed concurrently, not sequentially, and project management must maintain a free flow of information, or two-way high bandwidth flow, among the participants.

What specific actions are the leading firms taking? Some leading firms we interviewed have begun a formal process of inspecting specifications, as they do for code. That is, they are spending more time and effort making sure that product specifications, architecture and interfaces are unambiguous, and that downstream concerns are front-loaded into the specification process. (They are following the inspection procedures recommended by Fagan.) Many of these same firms are using tools such as Quality Function Deployment (QFD) to improve the quality of and to quantify customer input.

The faster firms are also using prototyping to gain a flying start. As Figure 5 indicates, two of the most important project management factors are prototyping and less rework. From a CSE standpoint, we view prototyping as a way to promote information flow from users to designers, which increases the likelihood that product design will match customer needs. Eliminating rework is one of the secrets to faster, more productive software development.

How does CSE resolve the productivity paradox? The faster firms realize that smaller teams tend to be more productive (except in the early stages where input from many different sources is needed). As you break problems into smaller pieces, interface complexities grow exponentially and overall productivity suffers because of integration problems, testing difficulties and rework. The objective of CSE is to provide a way out of this productivity trap by maintaining small team sizes and, at the same time, minimizing the interface problems. To accomplish this, project managers must establish two-way high bandwidth flows of information among the teams working on separate pieces of the problem and must ensure that the interfaces are simple and elegant.

These actions to resolve time and productivity problems tend to be confined to the lowest level of concurrent activity: within-stage. Most software firms do this to a degree, especially in the coding stage. But within-stage concurrency only achieves a fraction of the potential gain from CSE. The real potential for gain comes as a firm moves up the CE hierarchy and manages concurrent activity across stages—across the hardware/software interface and across platforms. Our surveys and interviews suggest that the faster firms are beginning to do this.

Across-Stage Concurrency

Concurrent activity across stages is rarely practiced because of developers' risk aversion. In our interviews, managers stated that it is counterproductive to overlap the early stages of software development. Beginning high-level design activities before requirements definition has stabilized, they argue, increases the risk that changing specifications will require redesign, and the cost of reworking a stage can be exorbitant. A manager at a large German electronics firm stated emphatically that stage overlap was incompatible with software development due to the instability of early stages.

Examples from hardware development reported by Clark and Fujimoto [5] show a strong correlation between the degree of stage overlap and shorter development cycles. Some software developers take the risk, reasoning that the time gained from letting

modules progress to the next stage is worth it. One defense contractor, obligated to follow a sequential model, admitted they circumvent the constraint by justifying early coding as mere "prototyping," which is allowed. This may explain the strong correlation in our European surveys between use of prototyping and development speed.

Microsoft is an example of a software firm that employs across-stage overlap. Cusumano and Selby [6] report that, to reduce the time-to-market for large software applications, Microsoft actively manages overlap across stages. Having abandoned the sequential Waterfall model, they have adopted as management practice a procedure to "synchronize and stabilize." Specifications, development and testing are all carried out in parallel but are synchronized with daily builds.

Hardware/Software Overlap

Our research with software managers [2, 3] suggests that the major obstacle to concurrent development of firmware is changing requirements. Because of the complex software/hardware interfaces, changes in one usually translate into changes in the other. Long, repeated rework cycles are the chief cause of cost and time overruns. Although these problems certainly occur in software or hardware designed alone, the interdependencies in firmware increase the frequency and severity of problems. As with stage overlap, two-way high bandwidth information transfer can provide early detection of interface problems. This may not, in itself, decrease the frequency of quality problems, but it can reduce their severity by locating them closer to the source and preventing the "spread of infection" throughout design.

Hardware/software interfaces create another version of the time paradox. Since instability of requirements specifications is a major problem, CSE can address this with simultaneous development of hardware and software requirements. This means that management must make a premeditated decision to spend more time and effort in the initial project stages to gain speed and productivity later.

Across Project Overlap

Software rarely exists in isolation. Like hardware, software is updated and redesigned, and its elements are reused in new products. Firms also carry out parallel development projects for features that will be introduced in different years. Projects thus have genealogical links to predecessors.

How can you achieve concurrent design activity across projects when development cycles do not overlap? The answer is by design reuse. When a component is reused in a subsequent product, the original design work is a form of virtual concurrency; that is, the initial effort is also being carried out for all future products in which that component is used. Concurrency across projects is the most difficult to visualize and accomplish, but also has the greatest downstream rewards.

Development managers recognize that reuse can increase productivity and reduce cycle time, but they are not sure how to encourage it. As one European manager of telecommunications software described the problem: "Our engineers are trained from the time they are at the university to design new things. Reusing old designs goes against their natural inclination to change, to improve." When asked how he encouraged reuse in such a culture, he said: "I don't give them enough time to develop new solutions and new code. They have to reuse whenever possible."

Unfortunately, most software reuse strategies that have been described to us are reac-

tive—reuse occurs more by accident and imposed time constraints than as a planned process. For example, when our Japanese survey responses are partitioned into two groups, new software projects and revisions of earlier projects, the time reduction attributed to reuse is 21.5% for revisions and only 9.5% for new projects. This suggests that firms lack a proactive, design-for-reuse strategy; they are not successfully designing new programs around reusable objects.

Object-oriented programming is expected to change all this, and many firms are studying it (particularly in Japan and the U.S.). In our sample, half the U.S. firms were using languages that support OO development. However, better tools may provide only a partial solution. Front-loading provides the information flow that contributes to stable, well-defined objects with precise interfaces, which are needed for planned, proactive reuse. Front-loaded information benefits designers by providing good forecasts of the requirements for future modules. This information flow helps them develop more robust and reusable programs.

Managers report frustration with their inability to design stable modules. Either the interfaces are unstable due to product changes, or different functionality is required. The problem, they admit, lies not in the tools, but in the management of those tools and a failure to provide the incentives and vision that support architectural modularity.

People and Process Matter, Not Tools

"To build a world-class software development group, you've got to have talented people. My first priority is to hire the best developers." This is how one software manager answered our question about the steps he took to improve performance in his organization. There is no escaping the fact that people are the most important ingredient. Figure 1 provides further supporting evidence on the importance of human capital. Better people and programmers received the highest relative importance ranking from our sample. Statistical analyses confirm that this factor has the highest positive correlation with increasing development speed. This is not a revolutionary finding. Studies of innovation in R&D have shown that people factors explain much of the variance in research productivity. One of the so-called "Microsoft Secrets" reported in [6] is to "find smart people who know the technology and the business."

More surprising is the relatively low importance (in our global samples) given to CASE tools and technology and to changing module size and/or linkages. Although the literature and popular press extol the importance of new tools and technology for enhancing software development, users perceive that these tools have a relatively minor effect on cycle time. In interviews, managers explained that problem complexity and the need to deal with people issues overwhelmed their ability to use the tools effectively. "CASE tools have improved, and we have adopted them. But the improvements in productivity with CASE tools can't keep pace with the increasing complexity of our software projects," said a software manager for a large telecommunications firm.

Respondents may report a low impact from CASE tools simply because they realize that their software development processes are not controlled. Levels of software process maturity are proposed in [9], and further refined into a Capability Maturity Model in [11]. The Capability Maturity Model research has caused companies to take a hard look at their software development processes in order to attain higher maturity levels. Our respondents are likely to be Level 1, 2, or at best, 3 on the CMM and therefore, should not be focusing on the use of CASE tools until their processes are better defined. Johnson and Broadman [10] point out that "less mature organizations do not even

focus on the basic areas of cost, quality and productivity." In an ongoing research project by the authors on time and quality in software development, process maturity is a key requirement for organizations to be willing (and able) to collect defect data that will allow process improvement activities to be undertaken. For less mature organizations, people are critical to successful software development. Because their software development processes are not controlled, the only way to survive is through talented people.

The importance of process and people in improving software development has been noted in other studies. At Raytheon [8], a process improvement initiative was undertaken after its software organization was assessed at CMM Level 1. Raytheon's initiatives were focused in two areas: people, in the form of software working groups, and metrics, to drive process improvement activities. Over an eight-year period, productivity increased 190% and defect density was reduced 65%. A recent survey of European software developers by Dutta and Van Wassenhove [7] found that firms scoring higher in end-user satisfaction and business results also tended to score highly on management practices focused on people and critical process metrics.

Conclusion

Our research indicates that people and process make the difference—you cannot automate your way out of a software crisis. The time and productivity paradoxes indicate that how time and effort are managed in software development can powerfully impact speed, productivity and cost. The CSE framework shows how the information flows should be managed to increase the level of concurrent activity from within stage, which everyone does, to across projects and platforms, which separates best in class from the rest of the pack. Within that framework, it is the people who make the difference because they must execute the plan and, in a creative activity such as design, there is no substitute for talent.

Although software firms have adopted many of the practices we associate with CSE, the impact of these practices is diminished by the piecemeal, reactive way in which they have been invoked, as is the case with reuse. By only attempting concurrent activity in the detailed design stage of a project, firms are robbed of the full benefits that concurrency can bring.

In our assessment, management is the missing piece in the puzzle. To achieve its potential in software development, CSE should be practiced as a coordinated management effort across all stages of the project (and across projects) instead of as a selective application of tools. To do this, management needs a framework, such as the one proposed here, to coordinate the application of concurrency principles throughout the software development process.

References

1. Blackburn, J.D., Hoedemaker, G. and Van Wassenhove, L.N. Concurrent software engineering: Prospects and pitfalls. *IEEE Trans. on Eng. Manage. 43*, 2 (May 1996), 179–188.

2. Blackburn, J.D., Scudder, G.D. and Van Wassenhove, L.N. Improving speed and productivity in software development: A global survey of software developers. In *IEEE Trans. on Soft. Eng. 22,* 12, (Dec. 1996) 875–885.

3. Blackburn J.D., Scudder, G.D. Van Wassenhove, L.N. and Hill, C. *Time-Based Software Development. Integrated Manufacturing Systems 7, 2* (1996), 60–66.

4. Brooks, F.P., Jr. *The Mythical Man-Month, Essays on Software Engineering.* Addison-Wesley,

Reading, MA, 1975.

5. Clark, K.B. and Fujimoto, T. Overlapping problem solving in product development. In K. Ferdows Ed., *Managing International Manufacturing*, North-Holland, Amsterdam, 1989.

6. Cusumano, M.A. and Selby, R.W. *Microsoft Secrets*. Free Press, New York, 1995.

7. Dutta, S. and Van Wassenhove, L.N. Report on the 1995/1996 software excellence survey. *INSEAD Working Paper 96/52/TM*, 1996.

8. Haley, T.J. Software process improvement at Raytheon. *IEEE Software*, Nov. 1996.

9. Humphrey, W.S. *Managing the Software Process.* Addison-Wesley, Reading, MA, 1990.

10. Johnson, D.L. and Broadman, J.G. Realities and rewards of software process improvement. *IEEE Software,* Nov. 1996.

11. Paulk, M.C., Weber, C.V., Curtis, B. and Chrissis, M. B. *The Capability Maturity Model: Guidelines for Improving the Software Process.* Addison-Wesley, Reading, MA, 1995.

12. Yeh, R.T. Notes on concurrent engineering. *IEEE Transactions on Knowledge and Data Engineering 4,* 5 (Oct. 1992), 407–414.