MITSloan

The International Center for Research on the Management of Technology

Alternative Designs for Product Component Integration

Nancy Staudenmayer [1]
Michael A. Cusumano [2]

April 1998 **WP # 177-98**

Sloan WP # ~~4021~~ 4016

[1] The Fuqua School of Business
Duke University
[2] MIT Sloan School of Management

Sloan School of Management
Massachusetts Institute of Technology
38 Memorial Drive, E56-390
Cambridge, MA 02139-4307

ALTERNATIVE DESIGNS FOR PRODUCT COMPONENT INTEGRATION

NANCY STAUDENMAYER
The Fuqua School of Business
Duke University
Box 90120
Durham, NC 27708-0120
Phone: (919) 660 – 7994
Fax: (919) 681 – 6245
E-mail: nstauden@mail.duke.edu

MICHAEL A. CUSUMANO
The MIT Sloan School of Management
Massachusetts Institute of Technology
Cambridge, MA 02142
Phone: (617) 253 – 2574
Fax: (617) 253 – 2660
E-mail: cusumano@mit.edu

ALTERNATIVE DESIGNS FOR PRODUCT COMPONENT INTEGRATION

Managers and researchers have increasingly recognized product development as a core capability in organizations. Recent trends, however, are challenging firms' abilities to perform this central activity. Technologies are simultaneously becoming more complex and advancing at an ever increasing rate. Projects are becoming more distributed across geographic and functional boundaries—all of which is taking place in the context of fierce global competition. This study builds upon a long tradition of research in new product development, which stresses the importance of coordination as a way to meet such challenges. It compares approaches to product component integration in six large-scale software development projects at two firms. The analysis identified three general approaches to product component integration: "big-bang," "roll-up," and "continuous" integration. These approaches differed along multiple dimensions: task allocation, resource investment, incentive structure, and timing. Teams experienced increased cooperation and fewer problems when a dedicated, highly experienced group internal to the project performed the process. This team drew upon a variety of structural and social incentive mechanisms to ensure cooperation. The paper offers an extension to theory by identifying some key design decisions that facilitate component integration and provide other benefits such as enhanced cooperation and motivation among project team members.

INTRODUCTION

Managers and researchers have increasingly recognized product development as a core capability in organizations. While research in product development enjoys a long and illustrious history (Brown and Eisenhardt April 1995), recent trends are challenging firms' abilities to perform this central activity. Technologies are simultaneously becoming more complex and advancing at an ever increasing rate, while projects are becoming more distributed across geographic and functional boundaries. And all of this is taking place in the context of more rapid and fierce global competition.

Improved coordination appears as a common thread in the literature on how firms should respond to such challenges (Lawrence and Lorsch 1967; Allen 1977; Iansiti and Clark 1994). Thus, for example, we see numerous studies on integration processes across functions (Dougherty May 1992), projects (Cusumano and Nobeoka 1992; Meyer and Lehnerd 1997) and time (Gersick 1988). Other researchers stress the influence of team and/or product size and structure on coordination costs. The focus on the role of modularization in system construction is a recent example (Von Hippel 1990). Optimal strategies for system testing and integration (Koushik and Mookerjee 1995) and development environments and tools are also topics of current study (Boehm January 1984). Particularly popular in this line of research are concepts related to concurrent or simultaneous engineering. These concepts emphasize a multifunctional team structure with close working relationships among representatives of product design, marketing, manufacturing, etc., fostered by, for example, overlapping development cycles (Nonaka 1990).

An over-riding message coming out of this body of work is that (1) integration is key to project success and (2) the earlier and more frequently integration is performed, the more likely a project is to succeed (Wheelwright and Clark 1992; Cusumano and Selby 1995). The strength of this result has tended to obscure certain other important questions. For instance, what is the ideal integration schedule—daily, weekly, monthly? And how does this vary by type of project? More importantly, what are the underlying mechanisms that render frequent integration so effective?

Nor do we have an adequate understanding of how projects can achieve such a result. Particularly on large scale efforts extending over several months or years, achieving integration on a regular basis may involve a complex set of organizational and managerial decisions. Finally, it is not clear that integration speed and frequency are the only benefits of a well-designed integration process. There may be other advantages such as enhanced morale and cooperation that are equally important in terms of team functioning and project success.

The goal of this research is to assist large-scale product development projects to achieve integration more easily and consistently. Several aspects of this work distinguish it from traditional research on the topic. First, it focuses on the development of complex software products that involve several hundred people and where the organizational solution is not as simple as forming a multifunctional team. Second, it strives to improve the efficiency of the entire development effort by focusing on the coordination of multiple interdependencies. Most researchers limit their analyses to isolated engineering tasks or coordination of a few functions. Finally, and most significantly, this research adopts an organizational design perspective to the problem of achieving integration.

COORDINATION ON LARGE SCALE PROJECTS

Building and maintaining an automobile, aircraft, computer, or software system represents an extremely complex activity (Brooks 1975; Wheelwright and Clark 1992; Cusumano and Selby 1995; Sabbagh 1995). This complexity arises not only from the inherent complexity of the technology but also due to the difficulties associated with managing the development process.

For example, software systems typically consist of millions of lines of code grouped into hundreds or thousands of files; those lines may execute hundreds of different functions. The first version of Microsoft Windows NT consisted of 5.6 million lines of code organized into 40,000 files. At the peak of the development cycle, approximately 200 developers were working on the NT effort (Zachary 1994). Data further suggest that this complexity may increase over time. One successful real-time telecommunications switching system in its eighth version of release currently stands at 10 million lines of non-commented code divided into 41 different subsystems. Three thousand engineers contribute to its production and maintenance. Nor is this scale and complexity limited to software products, as evidenced by recent data on the Boeing 777 project (Sabbagh 1995).

Because the products themselves are so large and complex, and schedules are usually tight, companies often try to carry out such development efforts in highly distributed, though interdependent, teams. Literally hundreds or even thousands of individuals whose activities are highly related contribute aspects of the product in parallel. For example, people working on parts in the same subsystem need to coordinate with respect to their design and testing. They also need to interact with people working on other subsystems that interface or interact with some of these components, with representatives from other functional areas such as marketing or customer support, and potentially with people on other projects within the firm. Managers face the problem of how to best design such a coordination effort while still meeting the project goals, schedule, and budget.

References from case studies of large-scale development projects highlight both the importance and the difficulty of achieving such balance. One telling indicator of coordination cost is data on design changes. For example, during the construction of the Boeing 777 airplane, the team working on a 20 piece wing flap found 251 interferences where parts occupied the same coordinates in space. All design activity on the project had to be suspended every few weeks during the main design phase. During these periods, team members looked for problems arising because of the interference between one subsystem or set of parts and another (Sabbagh 1995).

Design changes in software development tend to be even more frequent and costly. On the first version of Microsoft Windows NT, there were 150-200 component changes per day in the weeks leading up to release (Zachary 1994). A major telecommunications subsystem experienced approximately 132,000 changes over its 12 year history (averaging approximately 30 per day). As many as 35-40 different team members might "touch" parts of that subsystem on any given day.

Theories and empirical data also support the hypothesis that coordination costs can easily overwhelm or interfere with a team's productive capacity. For instance, the amount of coordination on a team depends on the number of communication links that team members need to establish and increases non-linearly with team size (Brooks 1975; Allen 1977). McCabe's data indicate that typical programmers spend 50% of their time interacting with other team members (McCabe 1976). Observational studies at one large firm revealed that people spent one-half of their time in meetings, and developers attended one meeting, on average, for every line of code they wrote. Interviews with programmers and system engineers working on software development projects also suggest that, as the size of the team increases, communication overhead on the project can quickly get out of hand (Perry, Staudenmayer et al. July 1994).

One popular way of conceptualizing the development process is as a series of implementation steps with iterative relationships between successive phases: system requirements, software requirements, analysis, program design, coding, testing, operations, and maintenance. This waterfall-like approach, which tries to simplify the process by "freezing" a product specification early and then integrating and testing the system at the end, was common in the 1970's and 1980's and remains popular in many industries (Cusumano and Selby 1995).

Recently, however, some researchers have begun to explore alternative development models such as "iterative enhancement" or "spiral" (Boehm January 1984), "concurrent development" (Pimmler and Eppinger 1994), "synch and stabilize" (Cusumano and Selby 1995), and "interpretive" (Piore, Lester et al. 1997). These authors argue that in many industries user needs and desires are so difficult to understand and evolve so rapidly that tasks are much more overlapping and inter-related than commonly supposed. As a result, it is impossible and unwise to design the system completely in advance. Instead, projects should "iterate" as well as concurrently manage as many activities as possible while they move forward to completing the project (Cusumano and Selby 1995). An alternative conceptualization of product development is, therefore, repeated occurrences of a sequence consisting of a development phase and a coordination phase.

The coordination of tasks is clearly a key requirement for project success and yet difficult to achieve. Integration may also be getting more complicated as technologies grow more complex, malleable, and interconnected and industries and markets become more volatile. We therefore need a more sophisticated understanding of how managers can achieve integration. Three research questions, in particular, would seem to be important: (1) What are some alternative approaches to integration, and how are they distinguished in terms of organizational design? (2) What is their impact on project performance? and (3) What underlying mechanisms account for those outcomes? In order to address these questions, this study focused on one integration process in six teams at two companies, as described below.

RESEARCH METHODOLOGY

We collected data regarding six large-scale software development efforts in two firms using multiple methods, including observation and interviews. Cook and Campbell suggest that multi-method analyses are particularly appropriate when conducting comparative investigations (Cook and Campbell 1979). An embedded, multiple case design was used (Yin 1984).

Domain and Sample. The two companies, Lucent Technologies and Microsoft, are in the telecommunications and personal computer (PC) software industries, respectively. Lucent's Network Systems division, the focus of this study, produces switching systems that connect telephone calls and employs approximately 30,000 people. Microsoft, with an employee base of about 20,000, produces software applications and operating systems for the PC market and, increasingly, multimedia, consumer and Internet applications. Both companies generate over $1 billion annually. Industry analysts consider both leaders in their respective industries.

Both Lucent and Microsoft also currently operate in a very competitive environment undergoing significant technical and market transition. In terms of their product development efforts, this translates into a need to develop very large, complex software products that are reliable, competitive in terms of their feature set, and attractive to customers. Competitive pressures further dictate that this be done quickly and efficiently.

Although the two firms face a very similar technical and market situation, they do so from very different historical legacies of success. For example, Microsoft has traditionally operated by instituting per product loyalty and focus. This approach is being increasingly challenged, however, as its products increase in size and become more integrated and system-like (Cusumano and Selby 1995). Lucent, in contrast, is making a transition from being a producer of bundled system products to largely unbundled features. Thus, although the issues concerning product development at the two sites are quite similar, each firm's response strategy in terms of how they manage their integration process should be quite different. The firms also differ markedly in terms of their age and culture. This heterogeneity ensures a wide range of management outcomes consistent with the goals of the study.

The criteria used to select products for the study were: (1) large size and complexity in terms of the amount of code and the number of components; (2) large team size; and (3) products which were almost or already introduced to the market, to eliminate variance due to development stage. The selected projects varied in terms of their (1) degree of innovation (approximately half incorporated new or unfamiliar technology for the firm and/or were marketed to new users); (2) position in the system architecture (some were at the user interface level while others interfaced with hardware); and (3) success (as defined initially by upper management in each firm). We based the choice of selection factors on past research (Kemerer 1997; Boehm January 1984) and discussions with experts in the field.

Methods and Data Sources. Data for the study consisted primarily of three types: interviews, observations, and internal company proprietary documents. We interviewed 71 people who worked on the projects. The subjects were operational or middle level managers, although their functional responsibilities and experience level varied widely. All interviews followed the same general protocol, which combined focused questions about approaches to integration with more unstructured discussion. The interviews lasted from about one hour to over two hours and were recorded on audiotape and transcribed shortly thereafter.

In addition, one researcher spent 2-3 months on-site at each firm where she had the opportunity to observe a variety of formal and informal meetings, discussions and events. Proprietary documents and internal tracking databases yielded information on project outcomes. Other internal materials used in the study included organization charts, product planning documents, engineering diagrams, project accounting reports, materials for internal training and education programs, and internal reports and memorandum, although the sources were not uniformly available across projects. In all cases, we attempted to verify and reconcile findings across sources (described below).

Analysis and Outcome Assessment. We analyzed the data using methods for building theory from case studies (Yin 1984; Eisenhardt 1989). We began by selecting pairs of projects and listing similarities and differences between each pair and categorizing them according to variables of interest, such as the presence or absence of a cross functional structure. Of particular usefulness during this process were various forms of analytical matrices (Miles and Huberman 1984). These matrices not only facilitated cross firm and cross project comparisons but also served to help reconcile the data across sources and standardize the largely qualitative data.

For example, interviews and internal documents yielded information on project outcomes, although the dimensions of performance varied across data sources and projects. We therefore constructed an unordered effects matrix, which summarized evidence of positive and negative performance outcomes along the dimensions of schedule, product component integration, product quality, and team functioning. We also noted explanations for particular outcomes in the display. A second form of outcome display was a case-ordered summed display of team reported problems, which ordered projects according to the type and severity of problems.

RESULTS

This analysis centers on the product component integration process (referred to as the "build" function at Microsoft and the "load" at Lucent). In this process, a project team integrates or "knits together" software component pieces to form one working product. The following quotation from a senior manager at Microsoft captures the centrality of this process in terms of team functioning and activity coordination:

> The build process by its very nature is a bottleneck. Everything developed has to pass through the build. Testers can't test without it. Developers, program managers, and product managers depend on it. These groups need clear and consistent expectations about the build so they can plan their time and work accordingly.

The analysis identified three general approaches to product component integration. These approaches differed along four dimensions: task allocation (the assignment of development, testing, and integration tasks and their organizational and geographic relationship to the project team), resource investment (the level of human and capital resources devoted to the process), incentive structure (the mechanisms used to coordinate and control activities), and aspects of timing. Below we compare and contrast these organizational solutions, illustrated by examples from the cases, before linking them with performance outcomes. Table 1 and Figure 1a-c summarize the organizational decisions that characterize the three approaches.

{Show Table 1 about here}

{Show Figure 1a-c about here}

The first approach to product component integration that we discuss in this paper we call the "big bang." Here, a project team periodically gathers all of the component pieces together and integrates them at one time (Figure 1a). One developer described this approach as "throwing all of the software into a pot and stirring," and noted that on large projects:

> It never makes soup, and we have no idea why or where it is broken. Then people have to run around through the hallways quite a bit before we can get it to build.

A second approach to component integration, the "roll up," sequentially brings together bundles of related components. As opposed to “stirring soup,” the image here is one of baton passing. Different teams sequentially feed their components into an existing base and take responsibility for making sure their components are consistent with what is in the base. Typically, teams of developers and testers work on their components for a couple of weeks followed by an integration roll-up period (Figure 1b). For example, on the Data project, a one week roll-up followed a development phase of 3 weeks.

The third type of component integration is the "continuous" approach. Whereas the big bang and roll-up approaches share certain similarities, continuous integration is radically different. Teams that followed this process established a core set of working functionality early on and proceeded to build the product incrementally as different team members completed various coding units or modules. The project performed the integration very frequently—usually every night—but also quite dynamically depending on the needs of the project. Moreover, these teams explicitly recognized the centrality of the process and made it a high priority in terms of resource investment. In each case, a technically experienced group internal to the project controlled integration, and they drew upon a variety of both structural and social incentive mechanisms to ensure cooperation (Table 1, Figure 1c).

Three Approaches to Product Component Integration

The “Big Bang” Approach

The Tollphone and Autophone projects illustrate how projects typically organize the big-bang approach as well as some of the consequences. Integration on these projects began relatively late in the development cycle. It then occurred every 2-3 weeks on a fixed, preset schedule that upper management determined. Inexperienced, non-technical contractors performed the integration. They were organizationally affiliated with a support department and geographically separated from the development and testing tasks. Developers were responsible for coding and regression testing individual components, while feature testers handled cross component testing. Developers had to submit their code by a deadline, after which the build team performed integration, ran system tests, and released the working product to the team. Change review boards assessed the priority, necessity, and impact of each change request before engineers submitted the notifications. There was usually no penalty if a developer missed a deadline other than having to wait for the next integration cycle to begin approximately 2 weeks later. Figure 1a illustrates this process visually.

Because developers work individually, they are essentially blind to the changes in other pieces of code until the point of integration. One characteristic of component integration is therefore that the number of bugs is proportional to the amount of time between integration points. As the interval increases, the probability of design conflicts also rises:

> If a lot of developers are making changes in the same code, bad things inevitably occur because you can never test interactions that way. Say you have two people modifying code at the same time. Each developer makes changes to the code, and they take those changes and compile them. But each only sees his own changes. Each compiles against the approved base and not against the other's changes. So each tests out OK individually, but when you put them together it won't work.

This ignorance of parallel actions is equally problematic during testing and bug fixing and caused significant problems in both Tollphone and Autophone. As one developer observed, "[Bug fixing is] very error prone. Changes and solutions become meaningless if you iterate too long because so much else has changed in the system." Long intervals between integration not only make problem fixing more difficult, they also complicate problem identification because it can be difficult to trace down the source of the problem after numerous, interdependent changes have occurred. As a result, and somewhat ironically, a big bang approach often appears as more frequent (corrective) builds:

> We end up doing a series of corrective builds to fix some little problem to get over the hurdle. They just keep popping them off like popcorn.

A second set of disadvantages associated with this approach arises from the nature of the schedule. Because the project fixes integration points in advance, their frequency and timing typically reflect some sort of average optimization across many competing needs. Furthermore, the sheer size and number of changes usually results in a longer integration and system test period. On the projects studied here, there was often a delay of up to 3 weeks between when a developer actually submitted his code and when he got feedback. Such blocking negatively impacted productivity and morale because there is a limit to what individuals can do without integrating their work with others. People also tended to forget what they did after a 2-3 week interruption, which further compromised problem-solving abilities.

Finally, the big bang approach results in a high level of duplicate coordination effort on the project because individual developers must collect component pieces and compile in parallel between official integration points:

> One person may have a piece of code. I need to access their code to make my part work. But I don't know who has the code, and it takes time to coordinate and pick them all up. And the coordination is duplicated because everyone is doing that. ...Another problem is that the system starts to slow down. Fifteen or twenty people may be making changes and compiling in parallel.

As alluded to in the second quotation, resource investment decisions made on the team sometimes exacerbated this coordination inefficiency. In the two projects studied here, a series of cost cutting measures that reduced the number of computers and reorganized integration support had a negative impact on the process:

> Computer expenses are quite tangible and therefore the focus of cost saving efforts. If I sit here for a day and one-half waiting for a build or a compile to finish, well, they pay us whether we sit here or not...This Spring the two loaders were moved out of our organization and put in a support department. They eventually took other jobs. The new people are novices who make a lot of errors and poor decisions. It's gone to strangers, and the performance and commitment have dropped.

In summary, projects adopting a big bang approach to product component integration tend to perform integration relatively late and infrequently on a fixed, preset schedule. In reality, however, they rarely achieve this goal without significant problems. Developers take responsibility for individual components, but the project delegates the integration process to another department. The teams tend to rely upon formal rules and procedures to coordinate activities (e.g., deadlines for making changes, change review boards), which people often ignore and bypass.

The "Roll Up" Approach

The roll-up approach differs from the big bang in that integration is somewhat more continuous, typically occurring several times during the development cycle and extending over a longer period. Furthermore, integration is now a shared responsibility on the team, passed from one group to another. The code ownership structure also differs. Whereas previously developers owned components and had to get permission from other owners to make changes, ownership rules are usually weaker in a roll-up model. Teams are relatively free to make whatever changes are necessary to get integration to work while they hold the baton.

One complication associated with the roll-up approach is that the teams need to identify sequential and circular interdependencies in advance in order to determine the correct component roll-up order. This can be difficult to do accurately, particularly when the technology is very complex or new. In fact, the Data team struggled repeatedly to accurately identify the correct sequence in their product:

> Early on we tried to guess at what component depends on others for the rolling up, but we couldn't even determine the order at that point. Later we felt a bunch of pain because it was very difficult to do a weekly roll-up. By Wednesday, we usually found all sorts of problems because the Forms [component] drop, which occurred the same day as the shell [component] drop, didn't work with the new shell. Forms tested against last week's version of the shell, which was perfectly reasonable to do because that's all they had access to.

A predetermined roll-up sequence also introduces constraints such that a team has less flexibility to alter the technology mid development cycle. This caused problems on the Autophone project when the customer demanded some "late-in-the-game" adjustments to delivery:

> So we're trying to bring it together on a different schedule than originally planned. We've all got our pieces—they're done and tested and ready to bring together—but now we want to bring them together piecemeal. They want some of the functional areas, not others, but the software is all wrapped up together. The problem is you don't have the flexibility at this point to break functional dependencies because you didn't design the code to deliver in that manner. Eventually, it becomes a crisis because everything and everyone is blocked. In our case, an official integration load effectively blew up.

A second problem encountered on projects utilizing this approach is how to minimize interference with production during the rolling up period. One option is to temporarily suspend all new development during that phase, but this sacrifices valuable development time. Moreover, developers typically continue to work on their components, resulting in an outpouring of new code and code changes immediately after the freeze. More often, teams end up running dual projects during the weekly roll-up; one code base for integration and a second for current development. This in turn necessitates tracking and mapping code changes between the two bases, which proved complicated and problematic on both of the projects studied here.

In summary, a roll-up approach to component integration is characterized by alternating phases of development and sequential component integration. As in the big bang model, developers own components but now they also share responsibility for integrating those components into the base. The project managers determine the integration schedule in advance, but the number of formal coordination and control mechanisms is somewhat lower than that found in the big bang model. Instead, peer level teams resolve differences and conflicts through informal working relationships (Table 1).

The Continuous Approach

Two of the teams studied, Network and Handphone, followed the continuous approach to component integration. In the Network project, a small, dedicated group internal to the Network department performed the build; 3-4 people were directly responsible for integration and about 9 others tested the results and performed some miscellaneous activities. The build manager, who had twenty years of experience in the software industry, believed very strongly that the build team should play a "policeman or mother hen" role on the project in the sense of controlling both the number and timing of code changes and their quality:

> The quality of the build is inversely related to the number of check-ins. So sometimes we 'open the flood gates' and let everyone check anything in. Other times we use a more controlled phased in approach where we will only allow changes in certain pieces but keep everything else stable. Eventually, when the system gets full and buggy, we go to a more restricted mode where people can only make changes that fix high priority bugs.

Handphone likewise organized product component integration as an internal team function. Although the build manager had less experience than the Network build manager, she worked closely and interactively with the feature engineer on the project.

In both cases, having direct control over the process enabled the teams to integrate components selectively and dynamically, depending on the particular issues and problems that arose. The teams maximized productive capacity by performing integration at night, thus ensuring that developers had a fresh base of code to work with each day. This in turn led to fewer integration problems since developers always had information on other peoples' changes and potentially resulted in faster turnaround time during testing.

Managers on both of these teams indicated that they had thought carefully about how to induce cooperation within the team without resorting to an overwhelming number of formal rules and procedures. The mechanisms they used to create incentives for developers to test and regularly integrate their code changes provide a good example of the type of solution they preferred to impose. Both build teams emphasized the professional responsibility associated with submitting well-tested pieces of code. In Microsoft, incentives that combined aspects of public humiliation and humor further supported this. For example, when a developer "broke the build" he wore goat horns (symbolizing a competitor), had a sign on his door ("Buildmeister"), or paid a small "fine." (The team eventually purchased a stereo with the money collected.) The Handphone team at Lucent used a similar but more subtle prodding factor:

> The first couple of times people were a little sloppy. They didn't have the discipline to always compile and test their check in. So we sent them a notice via email—'You made a mistake. It cost the team because every time we have to rebuild it costs a lot of time.' It embarrassed people but then they got serious and made sure they didn't break it.

Note that, whereas Network used a very public humiliation scheme—potentially the entire team (more than 200 people) knew when someone caused the build to fail-- Handphone privately contacted individuals to point out the impact they were having. These modifications were appropriate, given the very different demographic and cultural profiles in the two firms. Lucent tends to hire a population of older workers who operate in a more respectful and considerate work environment; widespread public humiliation and "goat horn" wearing would have flown in the face of this tradition. However, the data also suggest that managers on both projects targeted their use of incentives to particular personalities. For example, the Network build manager noted that he sometimes communicated privately with developers who were more sensitive to criticism, The Handphone manager would likewise occasionally resort to a public reprobation if a developer was repeatedly careless.

In summary, integration was a high priority on these teams, as indicated by their level of resource investment in it, and yet remarkably free of bureaucratic rules and procedures. Rather, a small central integration team relied on structure and incentive mechanisms to pull work in on a regular basis. Interdependency management under this approach was less about predicting or fixing interdependencies in advance and more about responding to them as they occurred.

Project Performance

Although the data on project performance are incomplete and subject to error, it is possible to draw some tentative conclusions by looking across the data sources and analysis methods.

Qualitative and Quantitative Outcomes. Network and Handphone both shipped on time and experienced few significant software integration problems (Table 2). Data on product quality (as represented by the number of post release bugs) was extremely limited, but neither project appeared to have significant field problems. On Handphone, the number of pre-release bugs (all severity levels) exceeded estimates, but the testing on customer site was very rapid and problem free. The evidence on both projects is also very strong that team members found it to be a very positive working environment. This was particularly true of Handphone, which more than one person interviewed described as "the best working experience of my life."

{Show Table 2 about here.}

The data likewise suggest a clustering of results for the Data and Autophone projects, which exhibited the big bang strategy. There was little if any evidence of positive outcomes, but strong and consistent evidence of poor performance. Both projects experienced significant shipping problems; the Data schedule slipped one year while Autophone was still slipping at the last data collection point. Neither team was able to consistently achieve integration of product components and used similar adjectives to describe the process (i.e., "very unreliable," "integration hell," "a nightmare").

For the Desk and Tollphone projects, the outcome and performance data are less readily interpretable. The Desk release date slipped approximately fifteen days (less than 5% of the total schedule), but the interviews largely attributed this to the fact that another major product went through manufacturing at the same time. Desk had a large number of very severe bugs before and after release, most of which were concentrated in the setup component. Data on the Tollphone schedule and bugs was not available.

Both teams had problems performing software component integration. On Desk, the application teams were working through issues of how to combine their processes. Tollphone experienced some integration problems early on, but the team largely resolved these once it converted to a new process. There appeared to be a moderate amount of conflict and chaos on each project, particularly Desk, but this was largely tempered by high experience levels, respect, and a problem-solving attitude among the team leads.

We can tentatively conclude that, although Desk and Tollphone did experience some negative performance outcomes, it is not entirely appropriate to classify them as poor performers along with Data and Autophone, for two reasons. First, the interviews suggested that team members and managers were quite aware of the sources of the problems and often had plans in place to rectify them in the future. For example, the Desk group was already working on how to reorganize the development of the setup component in the next release. Second, as noted earlier, both projects were undergoing significant transition in their product architecture and product market, which required them to change their processes. The Desk project was trying to integrate five previously independent applications and converting from an unintegrated, asynchronous product release to annual integrated shipment. The Tollphone project was almost moving in the opposite direction-- from annual or biannual system release to continuous streams of smaller functionality.

Reported Product and Process Problems. As described in the methods section, we also classified the projects according to the type and severity of their self-reported problems. The project clusters derived from this analysis largely support the conclusions drawn above. For example, Autophone and Data each had major, on-going problems with product component integration, defining the development process, and creating a standard working environment. Handphone and Network, in contrast, each had only one major problem area (cross project design and code sharing) and a scattering of minor issues.

DISCUSSION

This study examined the continuous day-to-day integration of individual work with the activities of other team members. It focused on the product component integration process, a nexus point for coordinating many different types of functional activities. Product component integration problems are significant because they occur within an integrated system of technologies, human actors, and tasks. Small delays and problems act as distortions in such an environment, which reverberate throughout the system and produce disruptions, waste, and inefficiency. A key proposition emerging from this study is that projects can derive performance leverage off of designing this central process correctly. Just as the Japanese manufacturers demonstrated the power of leverage and system-level thinking in the production process, small well-focused integration design decisions can produce significant, enduring improvements.

Other authors have previously written about the importance of component integration in product development (Cusumano and Selby 1995; Ulrich and Eppinger 1995). The emphasis in this previous work has primarily been on integration frequency; the earlier and more frequently a team integrates components, the better the project performance. The present study supports that conclusion but also offers an extension to theory by identifying some key process design decisions as well as suggesting other performance benefits.

In particular, this study revealed that key organizational design enablors promote cooperative behavior in the project, which affects the frequency with which integration can occur and ultimately project performance. As team members experience the benefits of increased cooperation and frequency, they in turn change their behavior and become more cooperative. The result is a positively reinforcing cycle where integration frequency is a mediator of the relationship between key up front design decisions and project outcomes.

What were some of the processes underlying this cycle? The analysis suggested that we could group them under four general headings: facilitated management of internal and external interdependencies, improved team productivity, enhanced motivation and morale, and self-regulating cooperative behavior. Table 3 links the organizational design enablors with these processes, illustrated by data from the cases.

{Show Table 3 about here}

Facilitated Management of Internal and External Interdependencies. Continuous (daily) integration facilitates interdependency management by reducing the likelihood of coordination conflicts. By integrating their work daily, developers keep abreast of changes and adapt their software code accordingly. Maintaining constant progress on many fronts in this way also reduces the likelihood of surprises later on by ensuring that "hidden" interdependencies surface early and regularly:

> A lot of time people don't realize that they are dependent on something. It's just not obvious. For example, you don't realize that you have a dependency because you are not familiar with that part of the code. Or a dependency that just sort of materializes out of thin air because of a need and is often tracked informally. Or instances where the solution to one dependency creates problems for a third party. The real problems occur

with the hidden interdependencies-- the ones that no one thought about that pop up at the last minute.

> [Interdependency management] is like searching for a nugget of gold under one of one million stones. You need to be very organized and invent a plan because it is too much for a single person. A plan allows more than one human to cast his eyes over the problem. If you don't do this regularly, two months or even one year goes by and stones remain unturned. Then you turn one over and discover a monster has grown and will eat you up.

Other researchers have also observed that one benefit of frequent iteration is that it reduces the likelihood of unpleasant surprises as the product is assembled. A common problem, for instance, occurs when different physical parts overlap or fail to work together (Sabbagh 1995).

Centralizing the coordination (in time and functional assignment) further eliminates so-called "chain errors" and blocking where pieces of code are sequentially interdependent. In order to compile and test a single piece of software, a developer usually needs to identify and gather several pieces (some of which constitute non-obvious linkages):

> For example, my software may depend on someone else's software. But just to get their code together with my code to build these two together can be a big nightmare because they may have code that is dependent on yet another person's code. And that stream can go on and on. You can do it individually and have to deal with all of the dependencies with everyone else, or you can put it into one spot and build it all at once.

Internal centralization also results in coordination flexibility and risk taking, which benefited project performance. Because a team controls the process internally, the team can apply the process dynamically and adapt it to correspond to the immediate needs of the project. It also serves as a quality filter, enabling projects to take more risks while simultaneously being a "good citizen" in the sense of not breaking functionality in other projects:

> When product integration is managed by people internal to your team, it means you can build selectively—only the pieces and times you want—in a more secure environment. If you have problems, you don't have to worry about hurting other projects. It's also more flexible, probably the biggest benefit, so you can take more liberties, beneficial liberties early on.

A key enablor, besides integration frequency and centralization, appears to be a high experience level on the build team. High levels of expertise ensure that complicated technical problems are solved efficiently. As one team member noted, "It's complex, there are so many things that could go wrong, but if you deal with it on a daily basis, it gets easier." Projects where the process was outsourced or assigned to non-technical consultants failed to capture the benefits of this specialization. As a result, integration became a bottleneck—blocking progress and creating lags between tasks and disruptions, which reverberated across the team as well as to other projects within the company.

Improved Team Productivity. Continuous centralized integration saves human and computer resources by eliminating duplicated coordination effort and blocking. By freeing up resources, production activities become more efficient. Clear and consistent expectations about integration also enable people to plan their time efficiently and effectively as well as modify their work accordingly, as opposed to simply reacting to daily interruptions and crises.

Centralizing integration eliminates task duplication, thus saving human and computer resources. In effect, the build team assumes the coordination role on the team, thereby enabling developers to concentrate on production:

> Say I make one line of code change. Then I need to compile and test that change. Fifteen or twenty people may be making changes and compiling in parallel. One problem is that the system starts to slow down. Now we do one compile for everyone. One that is bigger, but that big one does not equal the sum of everyone's little ones. It's much, much less.

The savings are more than mere CPU time, however, because now individual developers do not have to spend time tracking down pieces of software that are interdependent with theirs.

Frequent integration ensures access to functional improvements in the product as they occur, which directly enhances productivity and also reduces developer-to-developer blocking. Perhaps more importantly and subtly, frequent integration improves the diagnostic and bug correction processes. Diagnosis is easier because frequent integration "boxes" the search for problems:

> With a random process, when it breaks no one knows what is responsible. If you incrementally throw things in and test as you go along, it boxes the problem of trying to find the bug and it's easier to pinpoint the source of the problem; the bug is either in the new thing or in the interaction between the new thing and X.

Furthermore, as the interval between integration points increases, bugs and solutions in some sense become less meaningful due to the addition of new code and changes. People therefore waste time and effort either finding bugs that no longer exist or implementing solutions that will not work in a new code context. Shortening the integration interval thus improves the efficiency of the testing processes and potentially reduces testing intervals and turnaround testing time.

Decisions about the distribution of responsibility in a team can result in further productivity enhancements. For example, a dedicated build team develops experience at finding (or forecasting) potential problems as a result of learning effects. As we saw in the cases, some teams also make developers who create bugs responsible for fixing them. This ensures that relatively weak or careless developers will spend more time fixing and therefore less time creating new problems. Making developers responsible for the problems they create may also reduce churn:

> Now they are more likely to realize that 'Oh, if I change that interface it may break a lot of things. What a pain.' It encourages them to search for other creative solutions and helps reduce churn in the code because the pain is on the right person.

Enhanced Motivation and Morale. One of the challenges in a highly interdependent work setting is how to minimize frustration levels and maintain motivation over the course of the project. Frustration levels tend to increase, especially around deadlines or under schedule pressure, when people are blocked because they need access to other pieces of code in order to test their code or perform a fix. On very large projects, it can also be difficult to track progress; hundreds of people may be actively working, but what exactly is the status of their combined effort?

Continuous daily integration serves as a morale booster by keeping people on the team motivated and convinced that things are moving along. Even when integration problems occur, the team benefits from at least knowing the status. Regular integration also manages developer's expectation levels; knowing a new build will be available again in 24 hours tends to reduce the number of interruptions.

People also feel a sense of satisfaction and accomplishment because they see the immediate benefit of what they have done. As one member of the Handphone team described it, "When the first phone call rings successfully, it's really exciting!" Research by Karl Weick has similarly documented the positive motivational value of breaking a large task into a series of small wins or incremental gains (Weick 1984).

Self-Regulating Cooperative Behavior. But perhaps the most interesting and important function of centralized continuous integration is how it helps people to see the value of cooperating and therefore promotes higher quality work and more cooperative behavior. Centralizing the process increases the visibility of interdependence beyond a "near neighbor" level of immediate interdependencies. Individuals are aware of the implications of their actions on others because all work effectively stops when coordination problems occur and integration fails.

Cooperative incentive mechanisms such as public humiliation and professional responsibility further raise people's awareness of other's dependence on them. Other elements of process design support and reinforce these mechanisms. In particular, performing integration frequently and regularly makes it easier to pinpoint the cause of a given error. An internal structure promotes a sense of ownership and responsibility:

> When the process is more random, if it breaks no one knows who is responsible. This way, it is easier to pinpoint and that changes developer's behavior.
>
> Blocking is functional if it incents people to act. If people keep making changes but not integrating, bugs never get fixed because they are not blocking anything. With daily integration, people fix it because they don't want to be on the hot seat holding everyone up.
>
> Doing the integration internally can lead towards people wanting to do better code because it's not Joe Schmo they're hurting—it's their teammate who now won't be able to make the phone call work. It brings it very close to home if it's broken. With an outsourced load, there is a perception that you never want to break that. But it's funny, because even when you break it, what exactly are you breaking? You may be breaking something that is completely divorced from you. So you don't feel the same ownership for what you broke.

Note how cooperative behavior works in several directions—people have an incentive to do things (remove bugs, test code) and not do things (churn code).

When a project has decentralized and random integration, in contrast, individuals feel little if any "pain" when they change their work and negatively impact others. Under such circumstances, people tend to develop a very fatalistic attitude about their work. As one developer on the Autophone project observed, "I can't possibly guarantee that my change works." Such beliefs get translated into feelings of apathy ("It's way too big for me to care about") and sloppy, careless behavior. In particular, developers tend to submit code without evaluating the priority of the change or its potential impact and routinely fail to compile check their code.

Creating a positively reinforcing cooperative cycle through process design has certain other advantages as well. For one thing, it reduces the number of formal rules and ad hoc organizational units (such as change review boards) needed. As one Network manager observed, "A major advantage is the informality of it." Another benefit is the possibility of second order or spill-over effects. Once people see the value and benefits of cooperation in component integration, they may be more inclined to act cooperatively in general. Cooperation also becomes more likely because centralization frees up resources:

> When you asked somebody for something, they were always willing to make time for you no matter how busy.

CONCLUSION

This article described an in-depth investigation of product component integration on six large-scale new-product development teams. It identified some key organizational design decisions as well as the underlying mechanisms that they activate and linked those factors with project performance outcomes. A key argument in the study is that up front design decisions associated with this process can trigger broad patterns of cooperative or non cooperative behavior on a project.

In particular, product component integration in the two highest performing projects served as both a central work coordinating process (by enabling team members to easily respond and adapt to the latest work change made by others) and a motivational pacing mechanism (a visible signal to both managers and team members of the current status and progress on the project). As team members experienced the benefits of cooperation, they in turn changed their behavior to act more cooperatively, effectively setting off a positively reinforcing and largely self-managed cycle of cooperation. In essence, integration acted like a "heartbeat" on the team serving as both a metaphor for keeping the entire team working smoothly and important regulatory and pacing functions.

One limitation of the study was its cross-sectional design. We observed each team applying only one type of process and a large number of factors might account for the performance outcomes. Yet the Desk and Tollphone projects had previously used an integration process similar to that advocated here. Similarly, many individuals had experience with more than one approach. Analysis of their interview data therefore permits direct comparison of alternative approaches, controlling for project and individual differences. The results support the conclusions reported here.

Finally, this work suggests several possible follow-up studies. First is the issue of how applicable are the results to other settings and other types of technology. Centralization of the component integration process as well as frequent integration may be more feasible in software relative to other types of products given its inherent malleability. The basic idea and benefits of frequent component integration, however, clearly apply much more broadly. Many companies in different industries have product development processes that allow design or engineering changes after an initial specification, or they attempt concurrent engineering and overlapping of coupled tasks (Ulrich and Eppinger 1995). All of these kinds of projects must deal with similar issues of integration and coordination as large-scale software projects.

In the automobile industry, for example, we have cases where companies accelerated product development by frequent releases of information on design changes to teams concurrently handling manufacturing preparations, such as building stamping dies. With each change in the body design, the stamping die design team had to modify the die designs (Clark and Fujimoto 1991). In the aircraft industry, Boeing recently built an entire aircraft, the 777, using a computer-aided three-dimensional interactive application (CATIA) for the design process. Using the new CAD technology made it possible to create an aircraft without building physical prototypes. The project, however, required the continual integration of components designed by 238 design and build teams. These teams totaled approximately 5000 engineers and designed or integrated approximately 4 million components over a multi-year period (Sabbagh 1995).

A second issue is whether or not projects can apply the organizational design elements identified here piecemeal or does the result depend on more of a "system" solution. The Network and Handphone cases indicate that firms do need a system to integrate components continuously. They need to coordinate the change-control and component integration processes as well as create a culture where engineers are willing to cooperate and communicate. Nonetheless, the fact that these two projects managed continuous integration somewhat differently indicates that companies or projects have some flexibility with regard to how they design *their* system for component integration.

A final related question for future research is how important other factors such as integration frequency are in coordinating multiple interdependencies versus the design elements identified here. Because this study confounded frequency with design, it was impossible to disentangle the two. Similarly, component modularization and stable interfaces defining how components should interact are important elements in architectural design as well as component integration processes, for any type of product (Ulrich and Eppinger 1995).

Perhaps most importantly, this study illustrates the importance of approaching product component integration as both a technical and very human process. The inherent scale and complexity of the technology demand the application of sophisticated technical and analytical methods. Yet what most distinguished the performance of teams in this study was their attention to and understanding of the human dynamics behind this process. As researchers, we need to open up the dialogue on this relatively narrowly studied topic. As one manager observed:

> Product component integration is very much a social phenomenon, not just technical. We need people to understand the impact they are having on the rest of the team and the value of cooperating.

REFERENCES

Allen, T. J. (1977). Managing the Flow of Technology. Cambridge, MA, MIT Press.

Boehm, B. W. (January 1984). "Software Engineering Economics." IEEE Transactions on Software Engineering **SE-10**(No. 1): pp. 4-21.

Brooks, F. P. (1975). The Mythical Man Month. Menlo Park, Addison-Wesley.

Brown, S. L. and K. M. Eisenhardt (April 1995). "Product Development: Past Research, Present Findings and Future Directions." Academy of Management Review **20**(No. 2): pp. 343-378.

Clark, K. and T. Fujimoto (1991). Product Development Performance: Strategy, Organization, and Management in the World Auto Industry. Boston, Harvard Business School Press.

Cook, T. D. and D. T. Campbell (1979). Quasi-Experimentation: Design and Analysis Issues for Field Settings. Boston, Houghton-Mifflin.

Cusumano, M. A. and K. Nobeoka (1992). "Strategy, Structure, and Performance in Product Development: Observations from the Auto Industry." Research Policy **21**(pp. 265-293).

Cusumano, M. A. and R. W. Selby (1995). Microsoft Secrets: How the World's Most Powerful Software Company Creates Technology, Shapes Markets and Manages People. New York, Free Press.

Dougherty, D. (May 1992). "Interpretive Barriers to Successful Product Innovation in Large Firms." Organization Science **3**(No. 2): pp. 179-202.

Eisenhardt, K. M. (1989). "Building Theories from Case Study Research." Academy of Management Review **14**(No. 4): pp. 532-550.

Gersick, C. J. (1988). "Time and Transition in Work Teams: Toward a New Model of Group Development." Academy of Management Journal **31**(No. 1): pp. 9-41.

Iansiti, M. and K. B. Clark (1994). "Integration and Dynamic Capabilities: Evidence from Product Development in Automobiles and Mainframe Computers." Industrial and Corporate Change **3**(No. 3): pp. 557-605.

Kemerer, C. F. (1997). Software Project Management: Readings and Cases. Chicago, Irwin Publishing.

Koushik, M. V. and V. S. Mookerjee (1995). "Modeling Coordination in Software Construction: An Analytical Approach." Information Systems Research **6**(No. 3): pp. 220-254.

Lawrence, P. R. and J. W. Lorsch (1967). Organization and Environments: Managing Differentiation and Integration. Homewood, Irwin.

McCabe, T. J. (1976). "A Complexity Measure." IEEE Transactions on Software Engineering **SE-2**: pp. 308-320.

Meyer, M. and A. Lehnerd (1997). The Power of Product Platforms. New York, Free Press.

Miles, M. B. and A. M. Huberman (1984). Analyzing Qualitative Data: A Source Book for New Methods. Beverly Hills, Sage Publications.

Nonaka, I. (1990). "Redundant, Overlapping Organizing: A Japanese Approach to Managing the Innovation Process." California Management Review **32**(No. 3): pp. 27-38.

Perry, D. E., N. A. Staudenmayer, et al. (July 1994). "People, Organizations and Process Improvements." IEEE Software: pp. 38-45.

Pimmler, T. U. and S. D. Eppinger (1994). "Integration Analysis of Product Decompositions." Design Theory and Methodology **68**: pp. 343-351.

Piore, M. J., R. K. Lester, et al. (1997). The Division of Labor, Coordination and Integration: Case Studies in the Organization of Product Design in the Blue Jeans Industry. Cambridge, MA, Massachusetts Institute of Technology, Sloan School of Management Working Paper.

Sabbagh, K. (1995). Twenty-first Century Jet: The Making of the Boeing 777. London, Macmillan.

Ulrich, K. T. and S. D. Eppinger (1995). Product Design and Development. New York, McGraw Hill.

Von Hippel, E. (1990). "Task Partitioning: An Innovation Process Variable." Research Policy **19**: pp. 407-418.

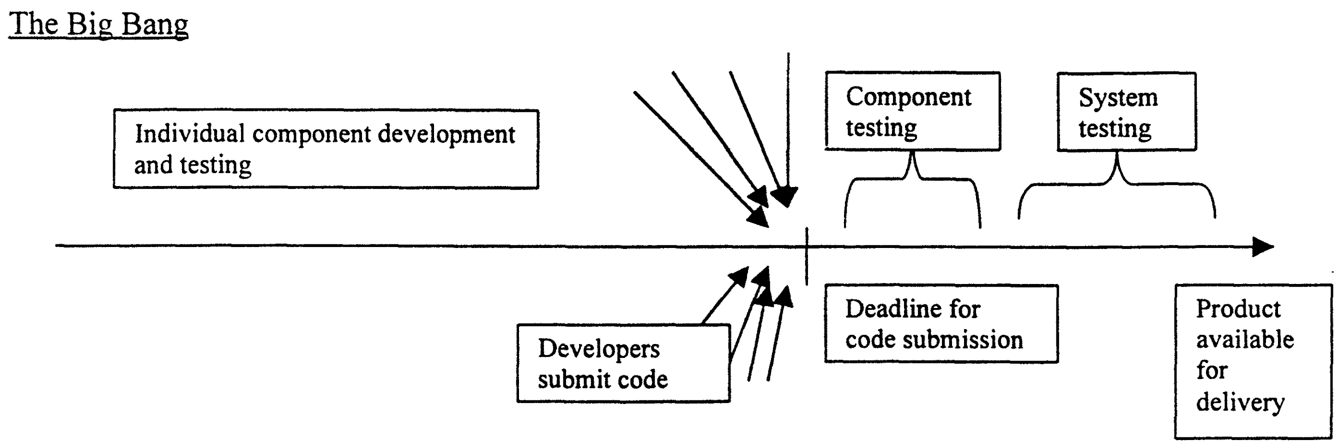
Weick, K. E. (1984). "Small Wins: Redefining the Scale of Social Problems." American Psychologist **39**(No. 1): pp. 40-49.

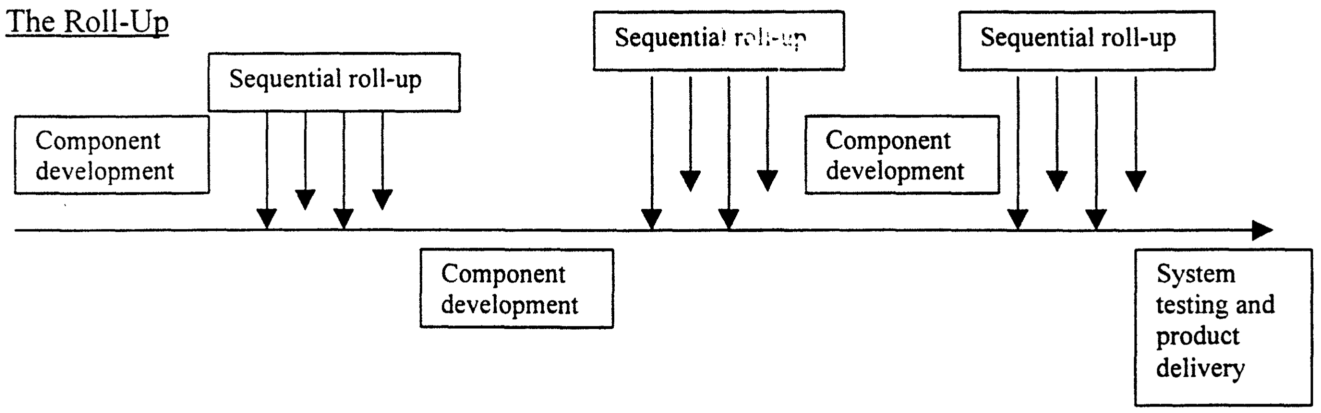
Wheelwright, S. C. and K. B. Clark (1992). Revolutionizing Product Development. New York, Free Press.

Yin, R. K. (1984). Case Study Research. Beverly Hills, Sage Publications.

Zachary, G. P. (1994). Show-Stopper: The Breakneck Race to Create Windows NT and the Next Generation at Microsoft. New York, The Free Press.

FIGURE 1a-1c
THREE APPROACHES TO PRODUCT COMPONENT INTEGRATION


The Big Bang
Individual component development and testing
Component testing
System testing
Developers submit code
Deadline for code submission
Product available for delivery



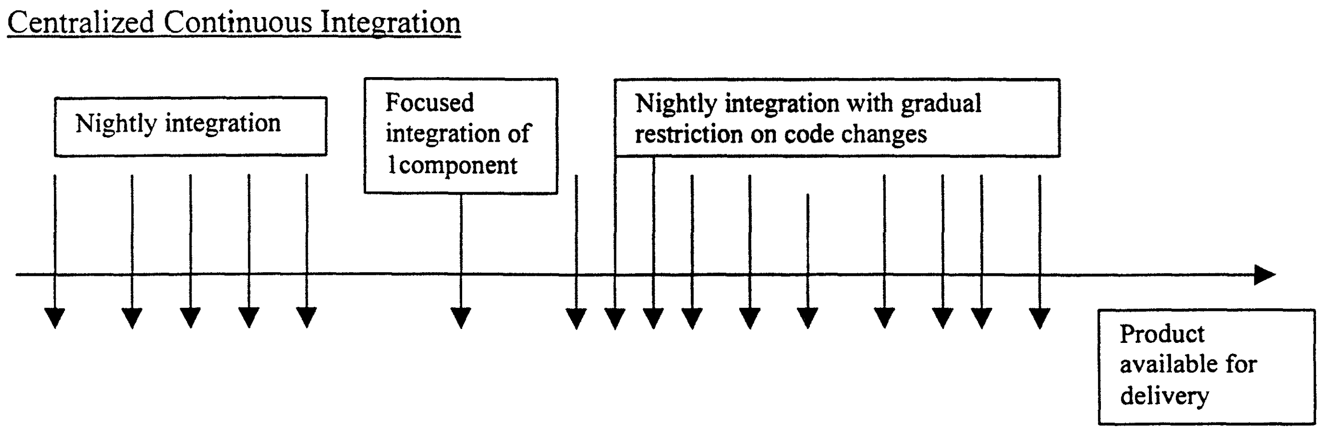

Centralized Continuous Integration
Nightly integration
Focused integration of 1component
Nightly integration with gradual restriction on code changes
Product available for delivery

TABLE 1
COMPARISON OF PRODUCT COMPONENT INTEGRATION DESIGN

	"The Big Bang" Approach	"The Roll-Up" Approach	The Continuous Approach
Sample Cases	**Tollphone, Autophone***	**Desk*, Data**	**Network, Handphone***
TASK ALLOCATION	• developers responsible for component development and regression testing • testers responsible for component-to-component testing • integration performed by an external support department • collocation of development and testing functions; integration geographically distant (separate buildings or states)	• component teams responsible for developing and testing bundles of components • component teams share responsibility for integration (sequentially) • component teams collocated or in separate buildings	• component teams responsible for developing and testing bundles of components (subystems) • integration and system testing performed by a dedicated internal team • collocation of all functions
RESOURCE INVESTMENT	• very low: minimal, slow computers and servers; centralization and outsourcing of integration function (triggered by company wide cost cutting measures)	• medium: intend to upgrade computers and servers in the future	• very high: partnered with hardware suppliers to receive prototypes of fast computers and servers; invested in additional computer queues and servers; placed highly experienced technical people on the integration team
COORDINATION & CONTROL MECHANISMS	• change deadlines • change review boards • formal rules and process manuals	• humor (e.g., hat, "Buildmeister" door poster) • informal rules and procedures (which sometimes varied across component teams)	• appeals to professional responsibility • public and semi-private humiliation • humor (e.g., pay fine) • heightened awareness of implications of actions via frequent, centralized integration
TIMING DECISIONS	• integration schedule determined in advance by product management • (intended) integration once every 2-3 weeks, on average	• roll-up sequence determined in advance by component team leads • 2-3 weeks of development alternating with 1 week roll-up phase	• early and frequent integration (once a day, 5 days a week, on average) • integration performed at night

* The three approaches represent conceptual categories. In reality, a given project often combines aspects of more than one approach. For example, Autophone primarily employed a big bang integration but also sequentially integrated some components. Handphone briefly experimented with a big bang approach before switching to continuous integration when that didn't work out. The projects are classified here according to their modal integration approach, but some of the examples cited in the text do not match this categorization.

TABLE 2

QUALITATIVE & QUANTITATIVE OUTCOME META-MATRIX: **Product & Project Performance Outcomes**

	EVIDENCE OF POSITIVE OUTCOMES & PERFORMANCE	EVIDENCE OF NEGATIVE OUTCOMES & PERFORMANCE	RESEARCHER'S OVERALL ASSESSMENT
DESK	Schedule: RTM date slipped from 6/30 to 7/15 Software Integration: Quality: Team Functioning: some tension betw apps and Desk (loss of autonomy; conflicting reqs) but balanced by high respect; pro-active, problem solving attitude	Schedule: Data slipped for several months, thus delaying Pro version Software Integration: "It sometimes took us days to figure out what's wrong with the build"; "Data is the outlier in terms of wanting to do stuff" Quality: "Majority of bugs were due to setup, which was very unstable" Team Functioning:	Some major problems but can largely be attributed to new product and org'l models: "Many interdependencies were hidden before when [products were produced] sequentially." Primarily localized problems (i.e., setup) Strong evidence that team members are aware of problems and have plans to address them in next release
DATA	Schedule: Software Integration: Quality: Team Functioning:	Schedule: product originally scheduled for shipment in 1996 but delayed to 1997 Software Integration: Yet to achieve a full product integration of all major components on an on-going basis Quality: sacrificed functionality from original product concept Team Functioning: little evidence of cohesiveness or cooperation; no clear leadership; "promises but no action"	Team members think in terms of components, not one product Major, on-going software integration problems On-going design and delivery issues with 4 external component teams
NETWORK	Schedule: product shipped on time Software Integration: daily and weekly builds very early in cycle Quality: 5 months after 3.51 release, 6 high priority repairs on cust sites Team Functioning: very strong team spirit and identity; proud andconfident attitude	Schedule: Software Integration: Quality: Team Functioning:	Few major problems for a product of this size and complexity Functioned effectively as a large team Some blocked component delivery due to higher priority project

	EVIDENCE OF POSITIVE OUTCOMES & PERFORMANCE	EVIDENCE OF NEGATIVE OUTCOMES & PERFORMANCE	RESEARCHER'S OVERALL ASSESSMENT
HAND-PHONE	Schedule: available on schedule when customer wanted it; 6 mo coding interval vs. avg 13 mo on comparable projects Software Integration: some problems at beginning, but quickly corrected after change to new process Quality: handover interval in 1/100 sec vs. requirement of <=1 sec; 17 common site scenarios passed first time; predicted total faults = 500 vs. actual 296 Team Functioning: "best working experience of my life"	Schedule: Software Integration: Quality: Team Functioning:	A highly successful project along a number of dimensions both objectively and in light of some major obstacles at the beginning (i.e., unassigned personnel, very aggressive schedule set by sales force)
TOLLPHONE	Schedule: Software Integration: Quality: Team Functioning:	Schedule: Software Integration: Quality: Team Functioning:	A legacy product (in U.S.) that has had to adapt to changes in technology and market
AUTO-PHONE	Schedule: Software Integration: Quality: Team Functioning:	Schedule: "trouble delivering features to customer on time"; three releases in progress while V1.0 delivered to customer 'on trial basis'; avg delivery interval 30+ mo and counting Software Integration: never able to consistently achieve integration of major components Quality: continually 'losing' functionality (intentionally and not) Team Functioning: "frenzied and chaotic"; "I feel I am part of 6-8 different teams"	No clear leader, ownership or accountability Major software integration and customer delivery problems Clear evidence of a project out of control

TABLE 3
KEY ENABLORS AND PROCESSES

	Management of Interdependencies	Team Productivity	Motivation and Morale	Cooperative Behavior
Organization Design Enablors	• early and frequent integration • centralized integration function	• early and frequent integration • integration centralized in time(at night) • investment in human and capital integration resources • incentive mechanisms to frequently submit code	• early and frequent integration yield "small wins" • centralized integration function yields visible (non) collective progress	• regular and frequent integration • incentive mechanisms for high quality work
Specific Elements of the Process	• reduces likelihood of surprises by surfacing hidden interdependencies • greater flexibility • quality filter for other projects • eliminates blocking and errors due to sequential chains of interdependencies • secure environment promotes beneficial risk taking	• eliminates duplicate coordination; more resources available for production • access to functional improvements as they occur • improved problem diagnostic and fix processes • reduced developer-to-developer blocking • better planning	• frequent, small wins yield a sense of satisfaction and accomplishment • reduced frustration due to blocking • manages expectations (fewer interruptions and crises) • a visible metaphor for large team	• reduced code churn • higher quality code submissions • cooperation spill-over
Illustrative Quotations	*"Since there are so many people working in this project, it's best if we can submit changes everyday and there is someone managing those changes and putting them together."* *"Having control of the build means you can define your own schedule and build and test for your own purposes."* *"This gave us a chance to debug and test our stuff before it went public. It saved other projects a lot of problems and time."*	*"At certain times, especially late in the development cycle, there are so many problems that have to get solved, so many tests to write and run, that the last thing a developer needs to worry about is 'how am I going to get a load to build and test with?'"* *"You can submit your code today and have it back tomorrow instead of waiting 1-2 weeks for feedback."*	*"This was the hub. People were always eager to see how the load was going. In the morning, people were waiting for it. Once they submitted all their stuff, they couldn't wait until the load was ready so they could go test."*	*"Fear of infamy is an incentive mechanism."*