"CONCURRENT SOFTWARE ENGINEERING: PROSPECTS AND PITFALLS"

by
J. D. BLACKBURN*
G. HOEDEMAKER**
and
L. N. VAN WASSENHOVE***

94/65/TM

* Owen Graduate School of Management, Vanderbilt University.

** Research Assistant at INSEAD, Boulevard de Constance, 77305 Fontainebleau Cedex, France.

*** Professor of Operations Management and Operations Research at INSEAD, Boulevard de Constance, 77305 Fontainebleau Cedex, France.

Concurrent Software Engineering: Prospects and Pitfalls

by

Joseph D.Blackburn
Owen Graduate School of Management
Vanderbilt University

Geert Hoedemaker
INSEAD
Fontainebleau, France

Luk N. Van Wassenhove
Technology Management Group
INSEAD
Fontainebleau, France

October, 1994

Concurrent Software Engineering: Prospects and Pitfalls

Abstract-- Software development remains largely a sequential process. Concurrent engineering principles have been more widely adopted and with greater success in hardware. A methodology for marrying concurrent engineering principles to software engineering, or concurrent software engineering, is proposed. A hierarchy of software project complexity from single stage problems to firmware and multi-product, multi-platform projects is defined. Principles of concurrent software engineering are developed for each level in the hierarchy. Research findings that establish limitations to implementing concurrent engineering are also discussed.

Keywords-- Concurrent engineering; software engineering; new product design and development; firmware; overlapping problem-solving; information processing; architectural modularity; software reuse.

I. Introduction

As an industry, software has become one of the largest and it is growing exponentially. Conservative estimates of global expenses on software development exceed $375 billion [1]. However, the software industry is still considered immature; the process of software development closely resembles that of craft production. As a result, software routinely is over budget, exceeds its time target, and is riddled with defects [2]. Attempts to transform the process of software development from a cottage industry into something more like modern manufacturing have had limited success.

The case for concurrency in development is strong and persuasive. The literature teems with examples explaining how Concurrent Engineering (CE) principles have been widely-adopted to replace sequential, "over-the-wall" processes and to shrink lead-times in the design and development of hardware ([3], [4] and [5]). Despite the documented successes with CE in hardware development, the imprint of CE on software development is barely visible. Software development processes remain sequential; simultaneous development activity occurs only in isolated parts of the project, such as coding.

CE principles are used more widely in hardware perhaps because, unlike software, hardware development is closely linked with manufacturing. Leadtimes in manufacturing have been reduced significantly through implementation of just-in-time (JIT) and total quality management (TQM) techniques. Since CE encompasses many of the basic concepts of JIT and TQM, it is more natural for these principles to be transferred to the design process for hardware and for software to lag in learning.

Increasingly, the development of software has become closely coupled with hardware design. Tight linkages between the two processes are critical in firmware, or software designed to be embedded in hardware, where changes in either the product or the software usually require changes in the other. As hardware development times have fallen significantly while software design complexity has increased and development times lag, software often defines the critical path for new product introduction. With increasing competitive pressure to shrink the time-to-market process, management is beginning to recognize that dilatory software development poses a serious problem. Managing large software projects in isolation poses formidable problems, but these difficulties are magnified in firmware by the need for coordination and communication among the hardware and software design teams.

The theme of this paper is that the software development process can be greatly improved in terms of speed, quality and cost, and that there is much to be learned from the experiences in manufacturing and in hardware development with CE. Since this requires a marriage of software engineering principles with those of concurrent engineering, we call the resulting methodology concurrent software engineering (CSE).

This paper makes the case for CSE and describes how it can be applied. The next section presents a summary of the current status of software engineering and explains in more depth why software lags hardware in the adoption of CE principles, drawing upon our field interviews and surveys of software engineers. In Section III, we outline a structured approach to CE, define a set of principles as they apply to software, and construct a framework for application to software and joint software/hardware development projects. In Section IV we present a hierarchy of CSE implementation beginning with the simplest form, within development stage concurrency. This is followed by considering overlapping stages within a project, then the more complex task of managing concurrency across hardware/software projects is analyzed and we conclude with a discussion of the most daunting task-- concurrency across multiple products and platforms. In addition to indicating where CSE can best be applied in this hierarchy, we also show that there exist limits to concurrency. Under certain conditions, additional simultaneous design activity may be counter-productive and certain phases of projects should not be overlapped with others. Section V contains our summary and conclusions.

II. The State of Software Development

Software, instructions required to operate programmable computers, has been developed for commercial purposes since the late 1950s. Initially, writing software was viewed as a creative act in which programs were crafted by highly-skilled artisans. The early software programs were largely individual efforts and this helped foster a cult of "lone wolves" or "cowboy programmers." Some in the industry still consider software design as an art and dismiss software engineering as an oxymoron.

As the young industry grew, growing pains developed. Projects became too large and complex to be carried out by an individual. Since projects were outgrowing the ability to manage them, software began to exceed budget and time constraints with alarming regularity. "Software Crisis" was the term often used to characterize the situation.

Efforts to manage software more effectively led to the creation of a discipline of software engineering [6]. The basic objective of software engineering was to bring to software a more rigorous approach to project management in which division of labor and structured problem solving techniques support creativity. This movement had two branches: One was formalist in nature and consisted of a highly structured approach to programming which sought to rid software of ambiguity and error-prone bad practices. In addition, it encouraged the use of formal methods to impose rigor on specification, design and construction of software systems. Many of these methods were regarded by critics as being academic and overly theoretical. A second branch was pragmatic: it sought to enhance productivity and quality by providing tools and technology to developers at all stages of the process, including process management and quality control. This approach became known as Computer Aided Software Engineering, or CASE, tools.

Out of this effort to structure the process came a number of models for managing software development. The most widely-used model is the Waterfall Model [7]. Most of the firms we have surveyed and interviewed in the U.S., Europe and Japan indicate that they use the Waterfall Model as their process template. As shown in Figure 1 below, the Waterfall Model is a phased approach to development, proceeding through a structured set of stages: Requirements Analysis and Specification, High Level Design, Detailed Design, Coding, and Integration Testing. In practice, the end of each stage represents a milestone in the process at which many firms schedule a phase review before proceeding on to the next stage.

------ Figure 1 goes about here-----

The Waterfall Model has been credited with adding badly-needed structure to what has typically been an undisciplined process. However, it is also important to note that this model is essentially sequential and is accompanied by lots of feedback and redesign. Typically, the only form of concurrency invoked with this model is overlapping activities within a stage, such as coding.

The Waterfall approach is popular with management in the software community because it meshes nicely with the departmental, functional structure of most organizations and does not challenge conventional project management approaches. With this model the different stages of the process can each fall within the purview of a specific function: requirements analysis is the responsibility of marketing, design is the responsibility of software engineers, coding is the responsibility of programmers and other specialists can be employed for testing. The need for cross-functional communication is minimized.

In the past decade there has been increased emphasis within firms on developing a structured, defined software process. A stimulus for this has been Humphrey's Software Process Maturity Model [8] and its successor the Capability Maturity Model [9] . Humphrey defines five evolutionary levels to improve the engineering capacities of a software organization: Level 1 is chaotic, 2- repeatable, 3-defined, 4-managed, and 5-optimizing. An organization that evolves according to this model transforms itself from a software job shop into a software factory, which means adopting the philosophy "that at least some software could be produced in a manner more akin to engineering and manufacturing than to craft or cottage industry practices" [10].

Presently, few software organizations have reached the highest level of the CMM [6]. Cusumano reports some impressive achievements in his discussion of four Japanese software factory initiatives [10]. For instance, Hitachi doubled productivity in two years, cut late projects from 72% to 12%, while bugs per installed application were reduced eight times. Toshiba more than doubled productivity and reduced flaws up to seven times. NEC improved productivity by 26% to 91% and reduced defects by one third. Fujitsu improved productivity by two thirds and cut bugs by one third. Other achievements of software factory initiatives outside Japan are reported by Swanson [11] and Norrgren [12].

More recent research by two of the authors [13] into current global software practice has found that there has been substantial progress in terms of adding structure and discipline to the software process. Stimulated by CMM and ISO 9000 initiatives, software developers are using models such as the Waterfall or Rapid Prototyping to prescribe a documented, repeatable process. Firms in our U.S. and Japanese survey sample reported an average 28% development time reduction over the past five years. Respondents indicated that most of the gains in speed are due to the management of people rather than technology. The largest percentage reductions in development time for new projects were due to better programmers and engineers, improved communication, use of cross-functional teams and concurrency efforts; CASE tools and module size were not judged to be as effective.

The tools and technology have not fulfilled their potential. As the software manager of a large European telecommunications equipment manufacturer reported to us, "CASE tools have improved, and we have adopted them. But the improvements in productivity with CASE tools can not keep pace with the increasing complexity of our software projects."

Managers also report, however, that software development in their organizations remains largely sequential with much of the time and effort loaded into the latter stages of a project. Figures 2 and 3, which show results from surveys by the authors of U.S. and Japanese software managers (as reported in [13]) , indicate that firms still expend a small fraction of their effort in the early stages of software development. Coding and testing comprise almost 50% of the average project cycle time and more than 50% of the effort. Our evidence suggests that concurrent engineering principles have made few inroads into software development practice.

---- Figure 2 and 3 go about here----

III. Concurrent Engineering Defined

CE, like TQM or JIT, means different things to different people. There is no common definition, and most descriptions of CE in the literature present it as an unstructured set of concepts. Some researchers describe CE as a parallel design activity for which the term simultaneous engineering is appropriate. Others stress the integrated team approach to design in which the concerns of the different functions in the organization are addressed. In many papers CE is used as a catch-all term that encompasses a variety of tools that facilitate faster product development:

Quality Function Deployment (QFD), CAD/CAM, Design for Manufacturability and Assembly, Failure Modes and Effects Analysis (FMEA), and others. For managers seeking to improve their development processes, the lack of a clearly-defined set of principles or a template for CE is an obstacle to implementation.

A. Two Forms of Concurrency: Time and Information

Some of the confusion about CE arises because, in the process of design and development, concurrency actually takes two different *forms*: concurrency in *time* and concurrency in *information*. Concurrency in time refers to activities that are performed simultaneously by different people or groups, such as parallel design activities. Information concurrency refers to the integrated, or team, approach to development in which all the concerns of the different functions-- the customer, R&D, design, engineering, manufacturing, sales and service-- are addressed through shared information. Information concurrency includes simultaneously addressing the concerns of other functions in the development process-- sharing information among the team.

The twin concepts of time and information concurrency are synergistic because each tends to support and strengthen the other. Better information sharing in development tends to reinforce efforts to perform problem-solving activities in parallel, or time concurrency. However, these concepts are at too high a level of abstraction to be applied directly to software development. In the next section we subdivide them into a more specific set of principles.

B. Basic Principles of Concurrency

From our studies of new product development efforts at a number of firms practicing CE [14] and the research on new product development in the automobile industry by Clark and Fujimoto [15], we have extracted several important principles that are significant factors in producing time and information concurrency. Clark and Fujimoto introduce a useful framework for understanding concurrency in design based upon overlapping problem-solving cycles, which require integration in *time* and *information* flows. Figure 4 presents a schematic of the main techniques flowing from their work.

-- Figure 4 goes about here--

1) <u>Front loading</u>-- This refers to the early involvement in upstream design activities of downstream functions-- process engineering, testing, manufacturing, and even customer service concerns. Front loading is a form of concurrent information sharing forward in the process. It can yield cycle time benefits by providing an early warning about issues that, not considered, could lead to costly redesign and rework later. In hardware development, front loading of detailed design constraints or manufacturability problems into the early stages of the process can keep them from becoming major show stoppers.

2) <u>Flying Start</u>: This is preliminary information transfer about design activities in the opposite direction, or sharing information about upstream activities with members of the team primarily concerned with downstream activities.. For example, partial design information about new materials to

be used in a product can provide a jump start to design work and process engineering and thus compress the time.

3) Two-way high band with information exchange: This refers to two-way communication among team members involved in parallel activities to support concurrency in time. The information flow includes top-down communication about problem-solving activities to avoid mistakes and bottom-up feedback to avoid infeasibilities.

In addition to these principles of information sharing to support concurrency, the following have emerged from our research as potent enablers of concurrency, particularly in software development.

4) Small Batch Information Processing: This refers to the practice of transferring partial information about design activities downstream frequently and in small quantities to help provide a flying start and support overlapping problem solving. As described in [16] and depicted in Figure 5 below, the ability to process materials and transfer them in small batches is a distinctive competence of time-based manufacturers because it makes JIT or "lean" manufacturing possible. Small batch information processing plays an analogous role in the product design process. As in manufacturing, this activity makes concurrency possible and can dramatically reduce cycle times.

---- Figure 5 goes about here---

5) Architectural Modularity: This refers to the division of design problems into modules with well-defined functions and interfaces. To achieve time concurrency in the design of a complex product, the product must be broken into parts, or modules, to be designed simultaneously by teams. How a product is subdivided into modules is a crucial requirements and high-level design decision because this, in large part, determines the level of information exchange needed to support concurrent activity. In software, for example, the challenge is to define modules that are as independent of each other in terms of interfaces and interactions, yet small enough to be to be tackled by an individual or autonomous work group.

6) Synchronicity: Design work done in parallel needs to remain synchronized because the "plane doesn't fly until all the parts are there". This is primarily the project leader's responsibility to insure that groups working in parallel communicate with each other to synchronize completion times and integrate effectively.

IV. Applying CE to Software: Concurrent Software Engineering

Concurrent software engineering is the application of concurrent engineering principles to the software development process. For a firm engaged in software design but using a conventional process, such as the Waterfall model, there is a hierarchy of opportunities for applying the principles outlined in the preceding section. The concurrency opportunities to be exploited range from the lowest level, within a single stage, to the highest level of complexity, concurrency across multiple projects and platforms (see Figure 6). In the structured approach that follows we examine the implementation of CSE principles in this hierarchy of design complexity :

A. Single-stage concurrency

B. Stage overlap

C. Hardware/software overlap

D. Multi-project or Multi-Platform overlap

---- Figure 6 goes about here----

In each instance we examine the benefits of CSE and the obstacles to its implementation.

A. Single-Stage Concurrency

Parallel design activity is a central principle of concurrent engineering which is introduced into a software life cycle phase through modularization (as illustrated in Figure 7). Modularization involves the decomposition of a software system into elementary building blocks. This technique is applied as a problem solving and structuring technique to enhance the quality of the software system.

---- Figure 7 goes about here----

Most software organizations practice modular development in the coding stage and many report significant leadtime reduction resulting from these practices. However, there are no well-documented methods to achieve architectural modularity. The objective of architectural modularity is to define modules that are as independent as possible. This prevents regression errors -- new flaws propagated to other modules resulting from fixes in a particular module -- and enhances reusability. In addition, it reduces the managerial effort required to coordinate parallel development activities.

The conventional approach to module definition is top-down functional decomposition. The first division of the software system involves breaking it down into subsystems each of which perform a specific function. Every subsystem is further decomposed until a sufficient level of detail has been obtained. It is argued that the resulting tree-like system, although often a good match to the requirements, is rigid and difficult to change or enhance. A more recent approach to architectural modularity is that of bottom-up, object-oriented composition. Starting with generic objects, e.g. a customer, the system is built. The resulting system tends to be have greater flexibility but with the disadvantage of complex interactions between its modules. The best approach presently known appears to be a combination of the top-down and bottom-up methods. Functional decomposition is still the dominating practice, although many of our interviewees reported that their firms are beginning to use or consider object-oriented design.

As with general new product development, the implementation of modular design introduces several organizational problems. If the modules are not completely independent, interfaces must be well defined and additional communication is necessary to keep the different modules compatible. Martin [17] discusses the use of a central coordination platform into which the modules are plugged and which organizes the communication among the modules through interfaces which are defined up front. Changing interfaces is done at the platform level, and every developer affected by this change must be informed, so that the most efficient way to change this interface can be negotiated. This requires the application of two CSE concepts: front loading and high bandwidth information exchange. Front loading includes the round of negotiations to fix the interfaces prior to parallel module development. The coordination platform supports high bandwidth information exchange.

Concurrency achieved through subdividing a design activity into modules ultimately involves tradeoffs. Our research on the effectiveness of within-stage concurrency has shown that limits exist beyond which increased simultaneity can actually decrease expected stage (and project) completion time [18]. The concept that increasing the number and decreasing the size of modules designed in parallel can have adverse side effects on communication requirements was suggested in earlier work by Brooks [19] and Allen [20]. For example, Brooks notes that the amount of communication between teams working in parallel increases as a function of the square of the number of teams. Further evidence of the effect of increased complexity was provided in our field interviews with software developers in telecommunications who stated that as parallel activity in coding increases, the probability of interface errors and integration problems also increases, expanding the overall project completion time.

A subsequent analytical study by the authors confirms, and quantifies, limits to concurrency. In this study we construct a simple model of stage decomposition into n modules and show that, when effects of communication complexity and the probability of rework due to interface errors are factored in, there is an optimal module decomposition, n*, beyond which additional subdivision actually increases expected completion time. Moreover, as additional complexity effects are included in the model, the optimal number of modules decreases. These results also show that a key to managing within-stage concurrency lies in architectural modularity to minimize the negative effects of communication linkages and integration problems. Improperly managed, these negative effects can undo the time benefits of concurrent development. However, other companies denied this adverse effect. Two firms we interviewed in the defense and aerospace industry reported that good coordination of the communication flows could preserve the time reduction resulting from introducing parallel operations.

To summarize, concurrent software engineering can be introduced into a life cycle phase in several ways. Architectural modularity facilitates parallel activities. However, since modules are generally not independent, they must communicate through interfaces. Front loading activities consist of the negotiation of interfaces and the way changes in interfaces are handled during development. When deviations from the original interfaces occur, high bandwidth information exchange insures that all parties affected by the change are informed and that interfaces be fixed. Modules should be developed such that they contribute to the optimal performance of the system. That is, their interaction is at least as important as their individual performance.

What are the challenges for management? More time spent on the up front stages seems to be the key to achieving successful architectural modularity by breaking the problem up in the right way. In our interviews this principle seems to apply across the board-- from cars and airplanes to software. Many products may break up into logical parts-- automobiles, for example, into engine, drive train, heat exchange-- yet the difficulty arises in the design interfaces and in minimizing the need for constant two-way communication. In software, practitioners have stated that one key is to minimize the number of interfaces; this minimizes the links and minimizes the number of ways that things can go wrong. Software still has much to learn from hardware design in this area.

B. Stage overlap

Implementing concurrent engineering across different life cycle stages is advantageous because overlapping reduces the total leadtime if, in so doing, the individual stage completion times can be kept from expanding. However, it is even more advantageous to apply concurrent engineering techniques to the whole development cycle. This may not only result in further leadtime reductions, since stages are allowed to overlap, but quality also may be significantly improved since downstream concerns are recognized.

We can implement overlapping software life cycle phases in three ways (as shown in Figure 8):

(1) proactive overlapping (early downstream involvement);
(2) preparatory overlapping (concurrent preparation of downstream life cycles);
(3) passive overlapping (problem-solving activity simultaneously in different phases).

---- Figure 8 goes about here----

Synchronicity and small batch information processing are key skills in successful stage overlapping. Once overlapping has commenced, the subsystems or modules may not remain in lockstep, even if they are of similar complexity. Therefore they may get "out-of-sync" or out of phase. It may happen that some modules are still in the design phases, while others are already being coded or even being tested. This would not pose any problem if all modules were independent. However, if they are not, upstream detection of flaws may have dire consequences for the development leadtime and costs. This is the case if it triggers the rework of many downstream modules. A continual flow of information in small batches helps compress the time between occurrence and detection of problems.

As with within-stage concurrency, there is substantial evidence that some stage overlap is counter-productive. Overlapping stages and a flying start of design activity increase the risk of design iterations required to fix errors. Eppinger [21] notes that stage overlap " succeeds in reducing overall task time only when earlier iteration (overlapping) eliminates later iteration (feedback) which would have taken longer." Software developers are divided in their support of stage overlap. Many argue that it is counter-productive to attempt to overlap the early stages of software development. By beginning high-level design activities before requirements definition has stabilized, they argue, you merely increase the likelihood of having to rework the design in the face of changed specifications. Previously, organizations such as the U.S. Department of Defense (DOD) have required contractors to apply a sequential life cycle model (such as the Waterfall

Model) in their software development. In our interviews with a large German electronics firm, they stated that phase overlap was incompatible with software development due to the instability of early stages and the fact the process is harder to manage.

However, letting modules progress to a next phase is often considered too advantageous in terms of time gained to forego. Software organizations contractually bound to a sequential model told us they usually circumvented the problem by justifying early coding as mere "prototyping", which is allowed. This illustrates that very different views are held on how software should be developed.

Analytical models provide further insight into this problem. Kowal and Shtub [22] investigate a time/cost tradeoff and show that, under certain conditions, overlapping stages can reduce project completion time but increase the total effort expended and, thus, the cost. They also suggest that stage concurrency is best achieved in the later stages of a development process. In [18] we use analytical models to investigate the desired degree of project overlap and show that there is an optimal degree of overlap, which is diminished by communication and interface complexity, thus underscoring the importance of architectural modularity. We further show that the desired degree of overlap is a function of the number of modules into which a stage has been subdivided.

Phase overlapping can be carried out in a more proactive way. During high level design, information on a module can be passed to detailed designers who can begin preliminary detailed design. This facilitates a flying start. Similarly, detailed design information can be passed to coders who can begin analysis of data structures. Care should be taken, however, not to "jump to the solution". A major reason of the poor quality of early software development was the neglect of design.

Another principle of concurrent engineering is taking downstream interests into account while performing upstream activities. An example is design for maintenance or serviceability. This implies that a product is designed such that fixes are easily performed. In a CSE context it means also creating provisions for enhancements. This puts heavy flexibility requirements on the design. It also requires exhaustive technical documentation so that maintainers who were not involved in the system development have no problem maintaining it. Another example is design for testing. Since large systems are impossible to test exhaustively, it is necessary to limit system testing to some specific cases. This requires user involvement from the very start.

The needs of the customer should be translated into testable characteristics and tolerances should be negotiated. However, user requirements tend to be extremely volatile, since often the "customer has no clue what he exactly wants". A popular technique to test out and help firm up requirements is rapid prototyping. This front loading technique involves the quick development of a "prototype" of the eventual system. This prototype consists of a representation of the system to be developed which allows the user to use all of the user interface while the underlying functionality is represented by dummies. Quality function deployment (QFD) is also a useful, but under used, technique to translate customer needs into testable specifications [23]. During system development, test cases should be prepared, so that testing can start as soon as all modules are completed and integrated.

Summarizing, concurrent software engineering can be introduced in the following ways. Parallel activities can occur by allowing modules to progress to the next life cycle phases without waiting for other modules to complete their current life cycle phase. More proactive approaches involve early downstream involvement in which preliminary information is passed to downstream stages. This facilitates a flying start of those stages. Lastly, parallel activities can be established by starting concurrent preparatory activities such as test case preparation and market introduction facilitation. To prevent waste of downstream effort due to upstream stages, high bandwidth information exchange is imperative. Two-way communication shortens the feedback cycles so that errors can be detected early and their adverse effects can be limited. The eventual goal of the approach is to improve quality and reduce development time. Those objectives induce a focus on the system as a whole.

C. Hardware/Software Overlap

Increasingly, software is embedded into a larger system which integrates hardware and software subsystems. Examples are numerous and range from a small integrated circuit that controls an amplifier to an automated air defense system. In this case we consider the development of neither a pure hardware product nor a pure software system, but a joint design project for firmware in which both hardware and software must be integrated into a workable system.

When we have an integrated system including both software and hardware subsystems, the complex interdependencies tend to increase the leadtime beyond what would be expected for the independent design of the subsystems. The interdependencies create coordination and quality problems. Consider testing, for instance: software cannot be tested when the hardware to interact with it is lacking, and the hardware often cannot be tested without its controlling software. Simultaneous development applying concepts of concurrent engineering can help coordinate the hardware/software activities and reduce overall leadtime.

To avoid wasted time due to waiting on the completion of all the subsystems and the lengthy recycling upon flaw detection, simulators are constructed for preliminary testing purposes. Moreover, delaying the integration of hardware and software systems until the first testable hardware prototype tends to reduce the quality of the system. As Morris and Fornell [24] note:

> "Bringing hardware and software together at this late state is troublesome for several reasons. Engineers have little time to correct design problems, and fixes are more costly than they are earlier in the design process. Options for revisions are much more limited; because of the rigidity of the hardware design changes are usually made in the software, at the expense of system performance."

When embedded software is only a minor subsystem, such as in copiers, interaction between hardware and software development is often limited to the passing of requirements from the hardware designers to software engineers. Those requirements tend to change frequently, frustrating the software engineers who grow weary of "those hardware guys that always come up with more changes." One-way information flows of this type are common in a traditional, sequential development process. However, a better solution to the problem requires more involvement of software people in hardware design (or front loading) and two-way communication which can provide them insight in the origins of changing requirements so that they learn to predict changes (flying start); that is, the problem requires the application of CSE principles.

A CSE model for the development of firmware (as depicted in Figure 9) is an interactive, parallel process instead of the conventional iterative one in which much of the hardware development precedes software. Our interviews and surveys of software managers ([13], [14]) suggest that the major obstacle to concurrent development of firmware is changing requirements. Because of the complex software/hardware interfaces, changes in either 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 both the frequency and severity of problems.

--- Figure 9 goes about here---

As is the case with stage overlap, JIT principles such as small batch 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" to other parts of the design.

Since requirements specifications are a major problem, CSE addresses this with simultaneous development of hardware and software requirements. Quality Function Deployment (QFD), is a tool that is finding increasing use to reveal the interdependencies between customer needs and specifications of the hardware and software systems [23]. QFD is a form of front loading and two-way information transfer because it connects the "voice of the customer" with the team members responsible for determining hardware and software requirements. In a CSE environment the hardware and software requirements are determined simultaneously and so many of the design concerns are front loaded into this early stage. QFD provides a jumpstart on high-level design and provides a convenient platform for dealing with issues such as modularity and interface simplification.

Our research into software practice suggests that, although product development managers recognize the importance of up-front planning and "getting requirements right the first time," they still do not allocate much time to these critical early stages of the process. When questioned about the use of QFD and other tools to sharpen product specifications, many managers admit to an unfamiliarity with the techniques, some are "studying it," and a smaller number (predominantly in Japan) are using it in their processes. Adoption of QFD is much more widespread in hardware development than in software.

The architectural design of the larger system is used to front load the simultaneous development of the hardware and software subsystems. Two-way high bandwidth communication between the hardware and software departments keeps the parts compatible and helps both parties predict possible changes in the systems (i.e. due to changing market requirements or technical infeasibilities) so that expensive surprises are limited to a minimum. If the systems are better coordinated with one another, integration should be much easier.

Our preliminary hypothesis, based on field surveys and interviews, is that the keys to faster firmware development lie in the consistent application of the basic principles of CE: front loading, architectural modularity and two-way high bandwidth communication in small batches between the hardware and software teams. Unfortunately, not much is known about the root causes of time delays in firmware development. There is anecdotal evidence that the hardware/software interface causes the most serious time delays and cost increases. However, insufficient data have been reported to determine whether these problems are due to changing requirements or simple design errors in one creating the need for redesign in the other.

D. Multi-project Concurrency

Software rarely exists in isolation. Like hardware products, software is updated, redesigned and parts of it are reused in new releases and new products. There are genealogical links to predecessor projects. For example, in consumer electronics firms such as Philips and Sony introduces new color TVs on an annual basis, and many elements of the hardware and software are essentially unchanged from one year to the next. Sony may introduce a number of different versions of the Walkman during a year, each with a high degree of commonality. At the same time, these firms are carrying out parallel development projects for features that will be introduced in different years.

What is the potential for cross-product concurrent engineering? Concurrency across projects and platforms is the highest level within the firm and poses the most difficult management challenges. However, the payoffs in terms of time and cost increase as you move up the hierarchy from simple within-stage activities to concurrency across products. The project manager's role also increases with the scope of concurrent activity.

From our interviews with development managers, the clear consensus is that there is no secret to achieving concurrency across products and platforms: the answer lies in *design reuse*. The trick is in getting it. Reuse is difficult to achieve because it requires stability, but product development is an environment of constant change and this tends to scuttle most reuse programs. One European manager of telecommunications software described his problem in the following way: "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." Nevertheless he went on to explain that reuse is important. 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."

Most of the reuse strategies that have been described to us are reactive. In software, reuse occurs more by accident and by imposed constraints than as a planned process. A German electronics manufacturer outlined their strategy for concurrent projects as follows [14]: "We plan on product releases every nine months. Our metaphor is a train. Every nine months the train leaves the station. Development of features proceeds concurrently and if the software is ready, it goes on the train. Otherwise it waits for the next one to leave the station." So, in this case, reuse often occurs because of delays in completing the replacement design.

Planned reuse, on the other hand, is a form of virtual concurrency. Designing a reusable component means that when one is designing the component, one is simultaneously doing the design work for all future products in which that component is used. The leverage in terms of productivity, quality and in time compression is substantial. Reuse produces economies of effort as well as time compression.

Planned, proactive reuse is a management responsibility. They must provide the clear vision of requirements for future products so that stable objects, and platforms, can be defined and kept in place. To foster reuse, the two most important CSE principles are front loading and architectural modularity. Frontloading is important because designers need good estimates of what form future modules should take. Architectural modularity is the ability to define stable, well-defined modules that are robust enough to transfer across platforms. Management must also put incentives in place so that designers are rewarded for following these CSE principles in the design of reusable modules. Research by Sanderson and Uzmeri [25] on the Sony Walkman suggests that the ability of Sony to spin off so many products from one platform was due in large part to decisions about product architecture designed around reusable modules.

To achieve planned reuse in software most practitioners expect the answer to lie in object-oriented programming. The core concept of object-oriented languages-- portable modules with precisely-defined interfaces and functionality-- is congruent with reuse. Reusable objects are much easier to use than fragments of code, and the structure of object-oriented languages strongly supports architectural modularity.

For most managers, the promise of object-oriented tools remains largely unfulfilled. Most report frustration with their failure control change across products and an inability to retain stable modules: either the interfaces change because of product changes or different functionality is required. The problem, they admit, is not in the tools, but the management of those tools and the projects to which the tools are applied. The tools and technology will continue to improve, and management must learn how to apply CSE principles in order to achieve greater leverage from these resources.

V. Conclusion

Software development has come a long way since the days when entire programs were largely individual efforts. The problems have become more complex and have outgrown any individual's ability to tackle them alone. As a result, the process of software development has become more of a structured team effort. What has remained constant, however, is the inability of software projects to meet time, cost and quality targets.

For other forms of product development, concurrent engineering has become a natural activity. The evidence continues to accumulate that CE is giving firms the tools to reduce product development cycle times, lower their costs and improve the quality of their designs. For software to emulate the successes of hardware development, our research suggests that software engineering needs to evolve toward concurrent software engineering. Experiences with hardware development provide a good learning model, however the techniques of CE are yet to be widely adopted in software.

We have proposed a hierarchy of CSE application, ranging from low level, within-stage concurrency to cross-product and multiple release overlapping. Our research shows that concurrency is a common practice within stages, particularly coding and testing, but that its use diminishes as we move up the hierarchy and the managerial problems become more complex. To a large extent, the real potential of CE is not being realized in software.

As we show, the applications of the principles of CE are somewhat different at each level, and there appear to be limits to concurrency at each of these levels. Within stage overlapping has limits imposed by increasing communication, testing and integration requirements. Managers should be very cautious about overlapping across stages of software development, except in the early stages of requirements definition and high-level design. On the other hand, for hardware/software development the early stages may be precisely where more concurrency is needed to nail down the requirements and define the interfaces so that software development can proceed in parallel. The highest level (across products and multiple releases) of concurrency is the most difficult to achieve but it also has the highest potential payoff. This is where object-oriented reuse promises to bring enormous productivity gains.

In software, the payoff from CSE comes with a price: it places a heavier responsibility on project managers. Concurrency complicates rather than simplifies management's job . As the information flow rate increases, complexity increases and the project becomes more difficult to manage. Consequently, there is much research that remains to be done to develop a CSE methodology. Empirical studies are needed to help managers understand the critical drivers of concurrency, such as the links between time and quality in software development. There is also much to be learned about managing the high-bandwidth information flows in a team environment.

The most formidable barrier to implementing concurrency in software appears to be inertia based on the belief that "software is different." Our purpose in outlining a CSE methodology is to show that software can benefit greatly from applying the tools and techniques for fast product development that have been established in hardware. The firms most likely to achieve sustainable advantages in hardware/software development are the ones that stress the similarities and apply concurrent engineering principles to their software development.

REFERENCES

[1] S. R. Schach, *Practical Software Engineering*. Homewood, IL: Irwin, 1992.

[2] M. Van Genuchten, "Why is Software Late? An Empirical Study of Reasons for Delay in Software Development," *IEEE Trans. Software Eng.*, vol. 17, pp. 582-590, 1991.

[3] H. Takeuchi, and I. Nonaka, "The New New Product Development Game," *Harvard Business Review*, 64, 1, 137-146, 1986.

[4] A. Rosenblatt and G. F. Watson (ed.), "Concurrent Engineering," *IEEE SPECTRUM,* July 1991, pp. 22-37.

[5] J.R. Hartley. *Concurrent Engineering*. Cambridge: Productivity Press, Inc., 1992.

[6] W.S. Humphrey, D H. Kitson, and J. Gale, "A Comparison of U.S. and Japanese Software Process Maturity," 13th International Conference on Software Engineering, Austin, Texas, May, 1991.

[7] B.W. Boehm, "A Spiral Model of Software Development and Enhancement," *IEEE Computer*, 1988.

[8] W.S. Humphrey, W. Sweet, et al., "A Method for Assessing the Software Engineering Capability of Contractors," Software Engineering Institute, February 1989.

[9] M.C. Paulk, B. Curtis, M.B. Chrissis and C.V. Weber, "Capability Maturity Model for Software, Version 1.1," Software Engineering Institute, Pittsburgh, PA, 1993.

[10] M. Cusumano. *Japan's Software Factories, A Challenge to U.S. Management*. Oxford: Oxford University Press, 1991.

[11] K. Swanson, D. McComb, J. Smith and D. McCubbrey, "The Application Software Factory: Applying Total Quality Techniques to Systems Development", *MIS Quarterly*, December 1991.

[12] F. Norrgren, "Designing and Implementing the Software Factory--a Case Study from Telecommunications," *R&D Management*, 20, 3, 1991.

[13] J. Blackburn, G. Scudder, L. N. van Wassenhove, and C. Hill, "Time-Based Software Development", *Proceeding of the First European OMA Manufacturing Strategy Conference,* Cambridge, England, 1994.

[14] J.D. Blackburn, G. Hoedemaker, and L.N. van Wassenhove, "Interviews with Product Development Managers," Vanderbilt University, Owen Graduate School of Management Working Paper, 1992.

[15] K.B. Clark and T. Fujimoto. *Product Development Performance: Strategy, Organization and Management in the World Auto Industry.* Boston: Harvard Business School Press, 1991.

[16] J.D. Blackburn (ed.). *Time-Based Competition: The Next Battleground in American Manufacturing*. Homewood: BusinessOne Irwin, 1991.

[17] J. Martin. *Rapid Application Development.* New York: Macmillan Publishing Company, 1991.

[18] G.M. Hoedemaker, L N. van Wassenhove and J. D. Blackburn, "Mathematics of Concurrency: An Investigation of the Limits to Simultaneous Operations," INSEAD Working Paper, May 1993.

[19] F.P. Brooks, Jr. *The Mythical Man-Month, Essays on Software Engineering*. Reading, MA: *Addison*-Wesley, 1975.

[20] T. Allen. *Managing the Flow of Technology*. Cambridge, MA: M.I.T. Press, 1977.

[21] S.D. Eppinger, "Model-Based Approaches to Managing Concurrent Engineering," International Conference on Engineering Design 91, Zurich, 1991.

[22] M.T. Kowal and A. Shtub, "Concurrent Engineering and its Effect on New Product Development, " Vanderbilt University, Owen Graduate School of Management WP 91-29, 1991.

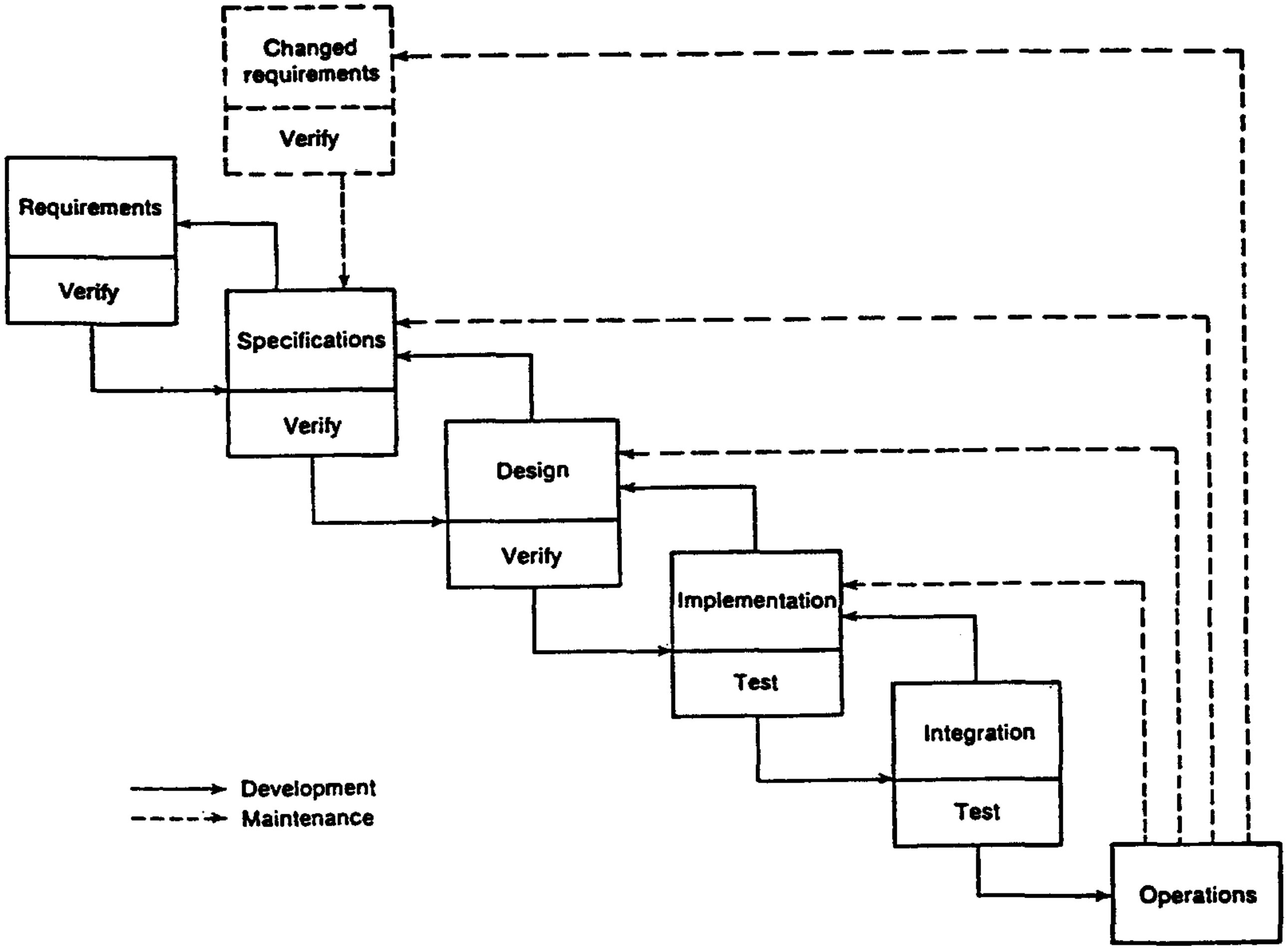
[23] R. Thackeray and G. van Treeck, "Applying Quality Function Deployment for Software Product Development," *Journal of Engineering Design*, Vol 1, No. 4, 1990.

[24] R. Morris and P. Fornell, "Linking Software and Hardware Design," *High Performance Systems*, June 1990.

[25] S. W. Sanderson and V. Uzmeri, "Strategies for New product Development and Renewal: Design Based Incrementalism," Center for Science and Technology Policy Working Paper, Rensselaer Polytechnic Institute, 1991.

Figure 1

Waterfall Development Model



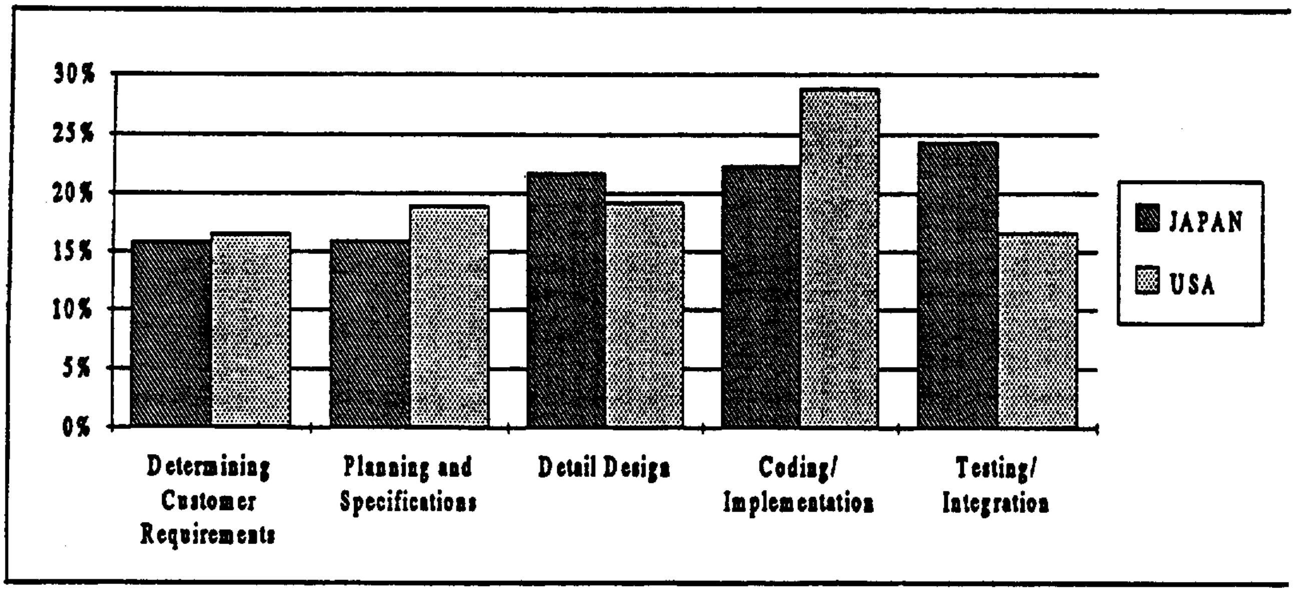


Figure 2
Development Stage as Percent of Total Development Time

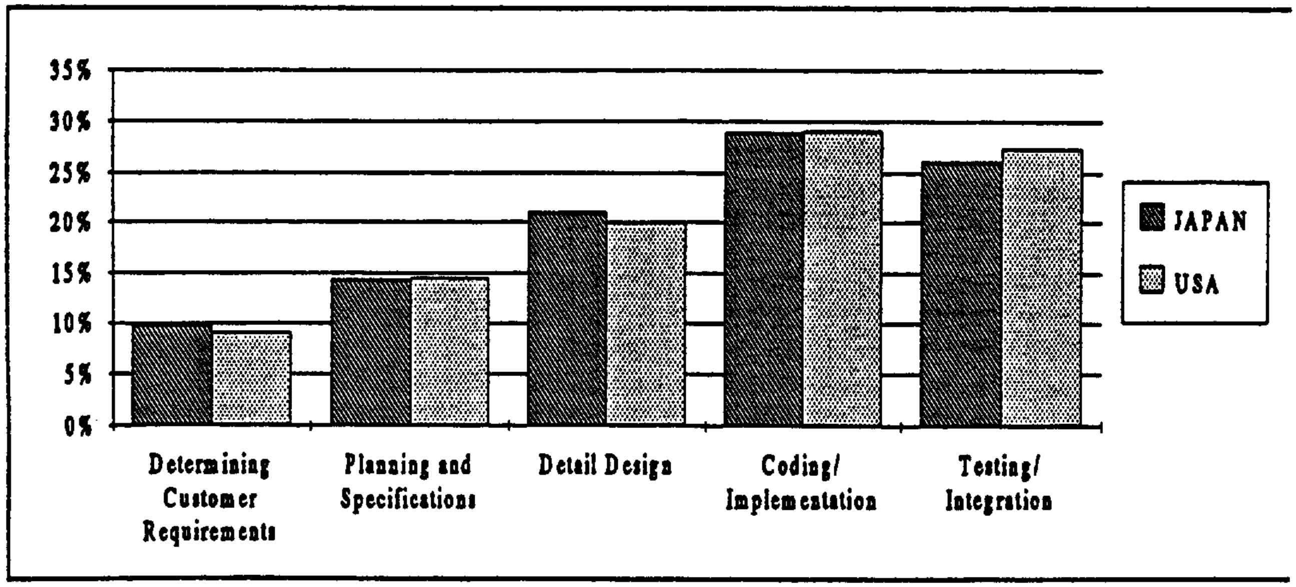


Figure 3
Percent Allocation of Effort in Man Months by Stage

Figure 4:

Concurrent Engineering Concepts

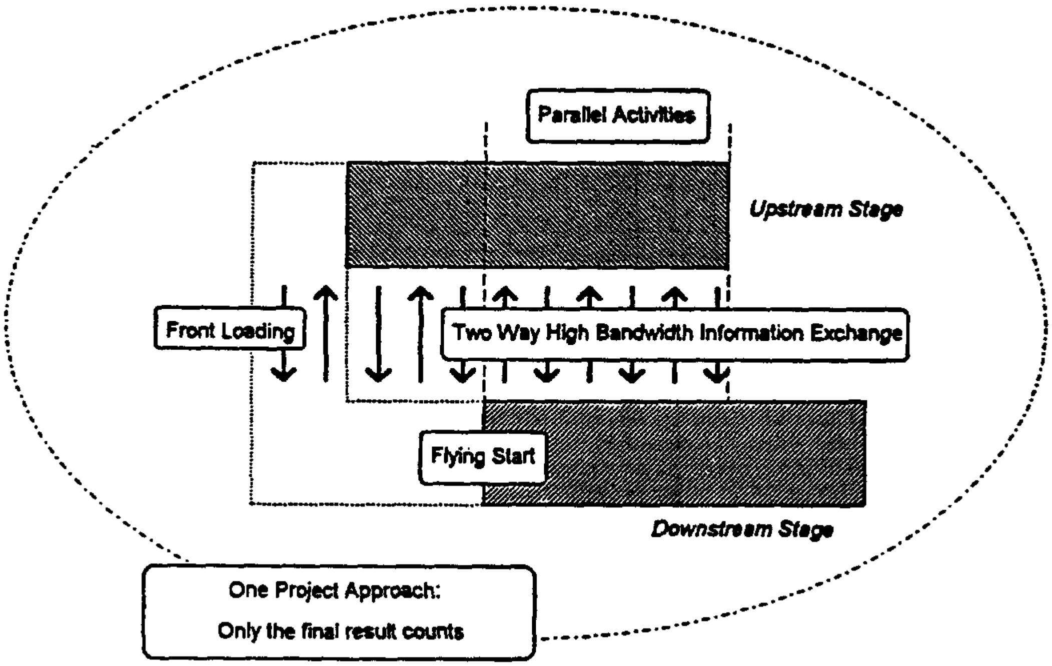
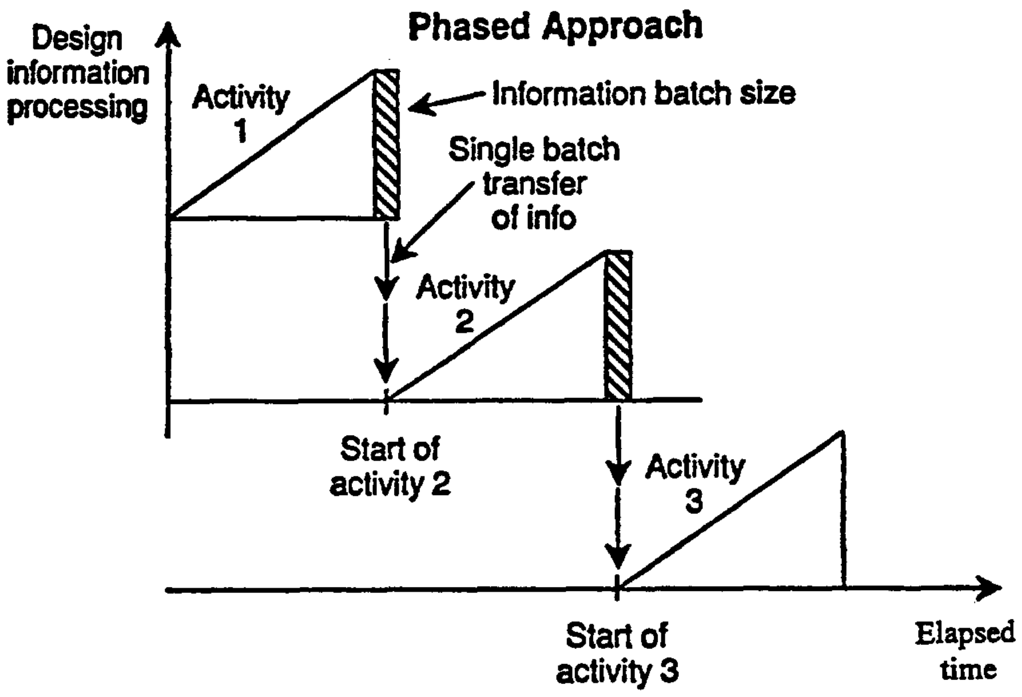


Figure 5

Phased versus Overlapping Approach to New Product Development



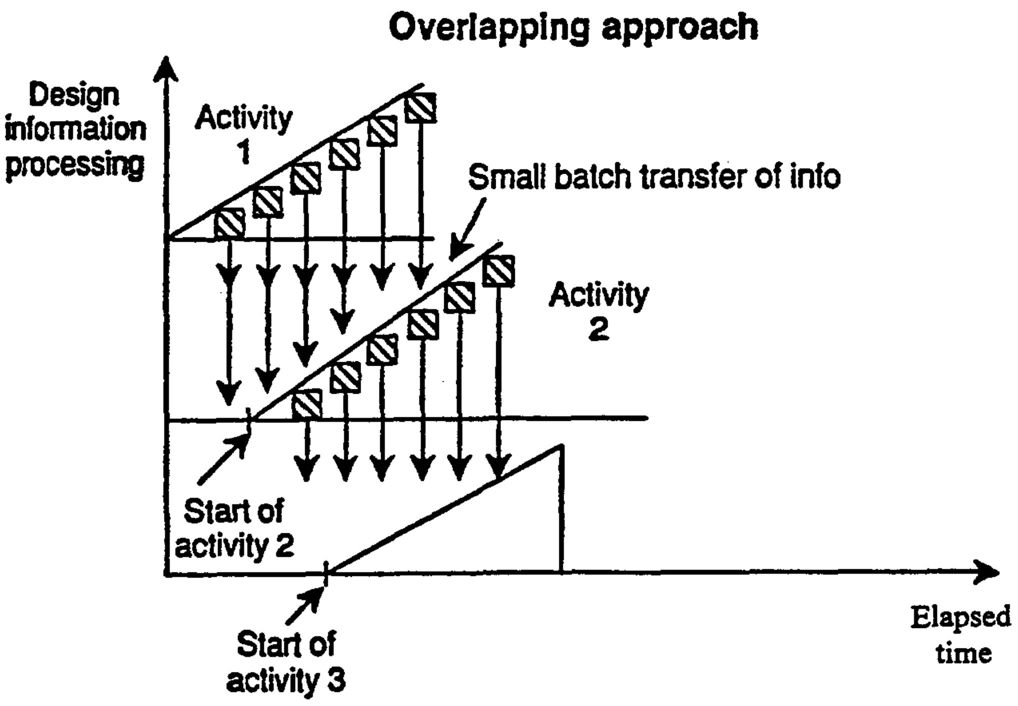
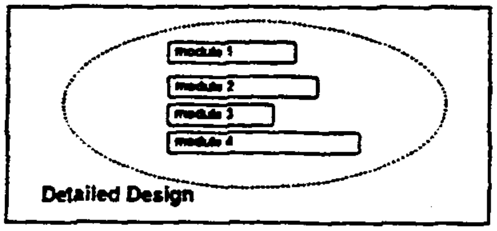
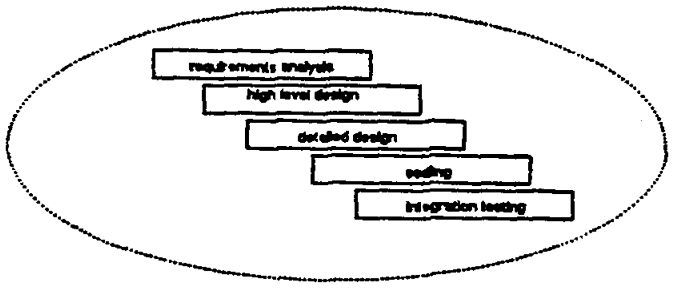


Figure 6:

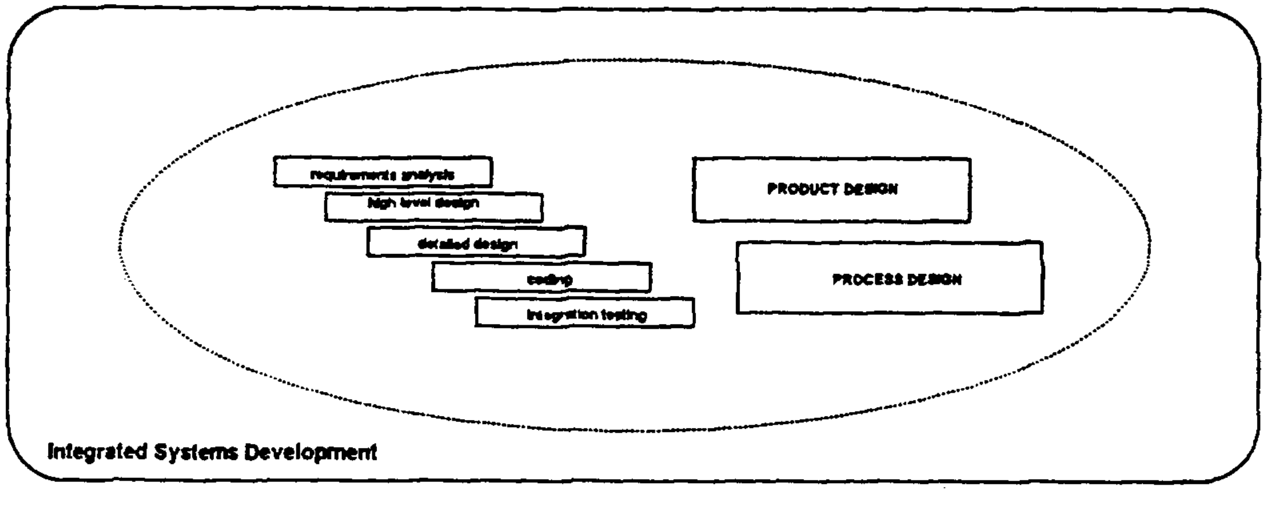
Hierarchy of Design Complexity



1. Overlapping Within a Software Life Cycle Phase



2. Overlapping Software Life Cycle Phases



3. Overlapping Across Hardware and Software Development

Figure 7:

Overlapping Within a Software Stage

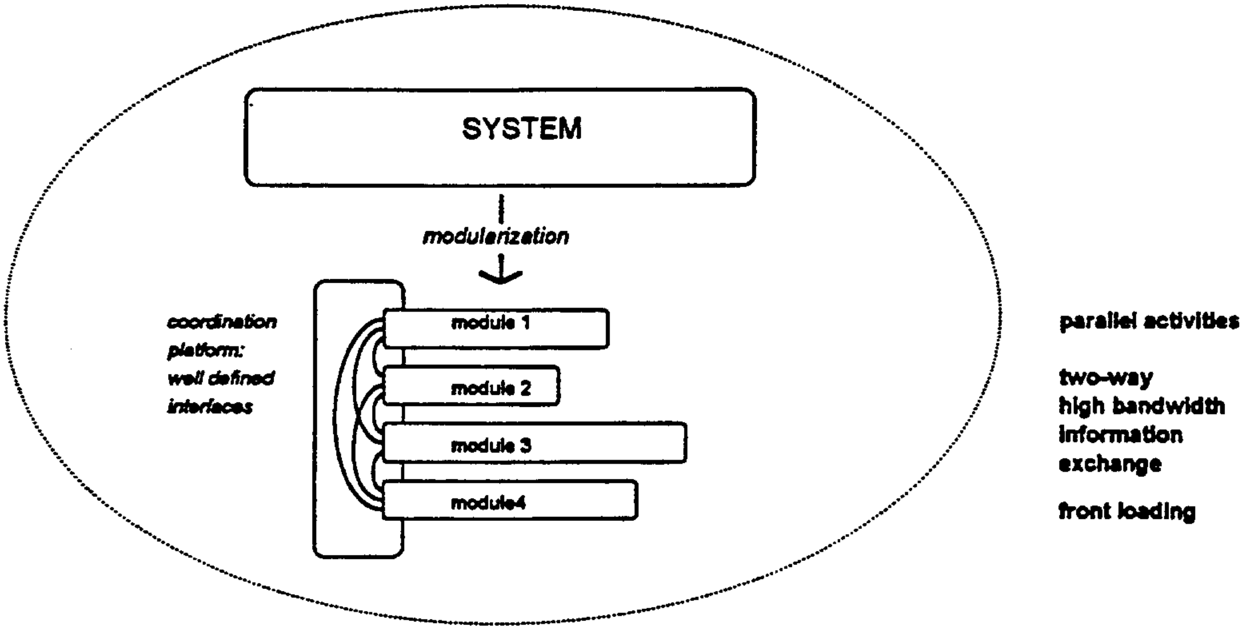

SYSTEM
modularization
coordination platform: well defined interfaces
module 1
module 2
module 3
module4
parallel activities
two-way high bandwidth information exchange
front loading

Figure 8:

Overlapping Across Software Phases

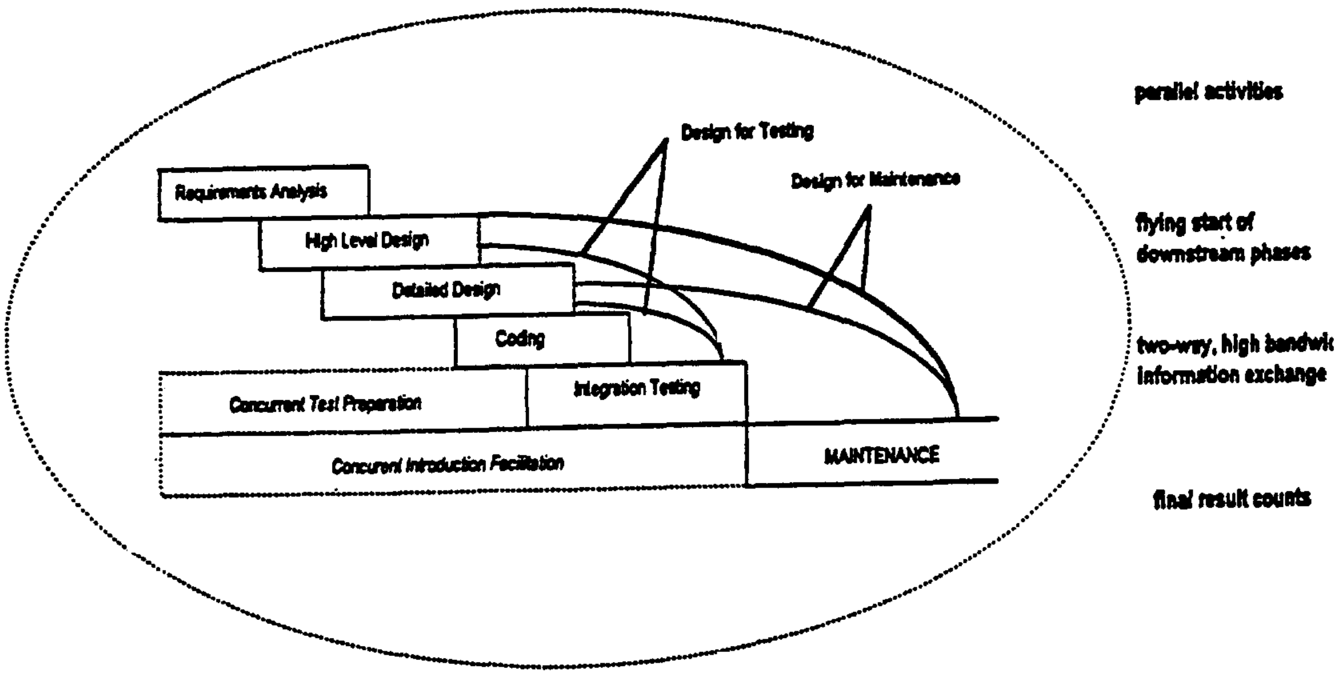
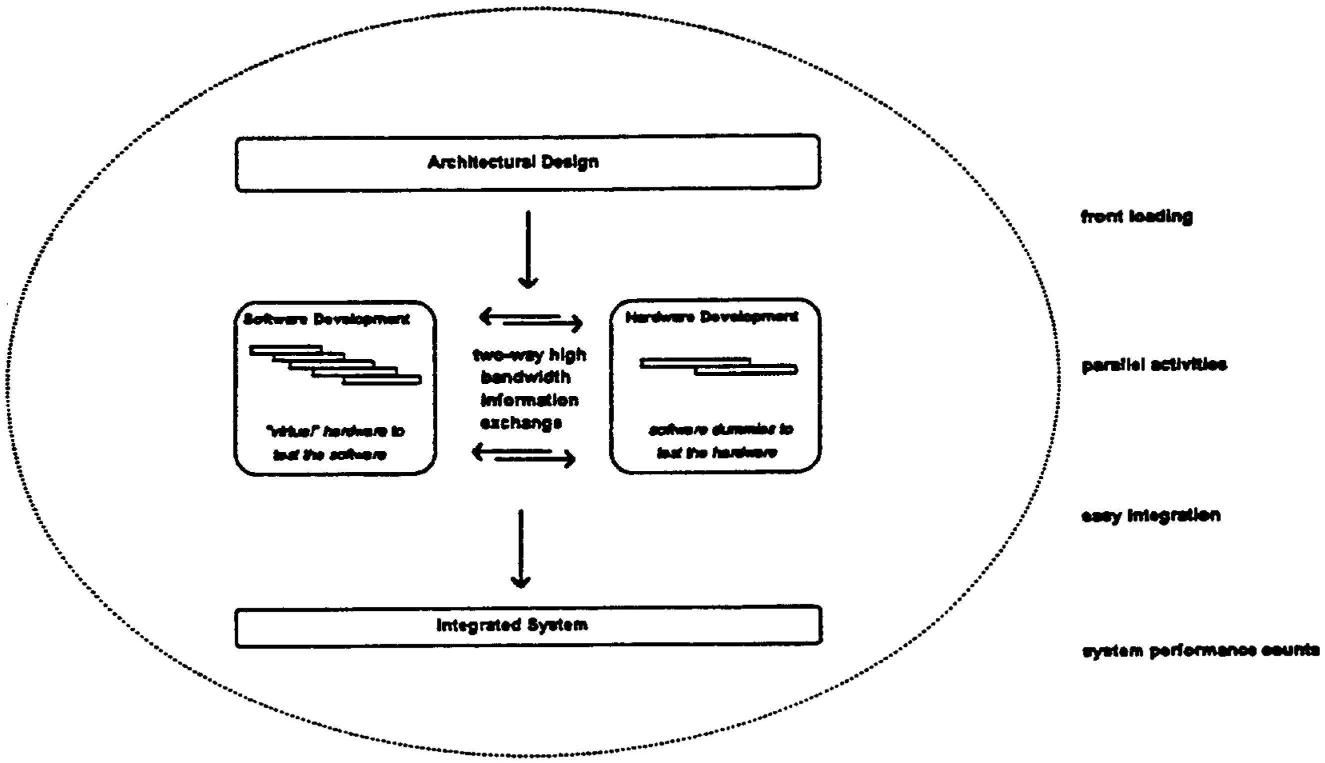


Figure 9:

Hardware/Software Overlap


Architectural Design
Software Development
'virtual' hardware to test the software
two-way high bandwidth information exchange
Hardware Development
software dummies to test the hardware
Integrated System
front loading
parallel activities
easy integration
system performance counts