INTRODUCTION

SOFTWARE IS CRITICAL IN TODAY’S MARKETS. The importance of information and communication technologies, and thus the software that makes them function, is growing rapidly in both industrial and consumer markets. E-commerce, the Internet, enterprise-integration systems, and wireless networking are just some of the high-profile systems and applications dependent on effective software development. The U.S. Department of Commerce estimates worldwide sales of packaged software alone will reach $258 billion in the year 2003, with average annual sales increases of 14 percent during this period (Industry and Trade Outlook 2000). Software is clearly a key driver of the growing information economy (Evans and Wurster 1997; Harter, Krishnan, and Slaughter 2000).
Software development projects are notorious for being late and over budget. A "software crisis" was declared in 1969 as a result of poor development performance (Cusumano 1991). The situation does not appear to have improved to date, judging from the continuing volume of articles discussing the software development problem (DeMarco 1995; Blackburn, Scudder, and Van Wassenhove 1996; Maxwell, Van Wassenhove, and Dutta 1999). Although software is an important contributor to the functioning of society, the software development process has not been mastered despite years of attempts to improve its efficiency. There are similar problems with the effectiveness of software development, another aspect of the "software crisis." Although strides have been made towards systematizing software development and therefore controlling the quality of the final product, software "bugs" of varying degrees of severity are found frequently and in many contexts (Business Week 1999; Krishnan et al. 2000). It appears that software developers need to improve the quality and reliability of their products as well as the cost and speed of the development process. The importance of this in economic terms is heightened when it is recognized that among the world's 500 foremost software providers, the average firm allocates 19 percent of its revenue back to R and D (Software Magazine 2000).

The importance of software and the need for improvement in many facets of its development has led to practical and research interest in improving the management of the software development process. The establishment of an "appropriate" software development environment is critical to success (Humphrey 1995). A number of software process models have been discussed in the (mostly technical) literature. However, relatively little management research has examined the software development process, either as a unique task or in the context of new product development. Further, which process model is most appropriate under any given set of conditions remains unresolved.

This chapter integrates managerial and technical literatures with the software development process to develop a conceptual framework of software development effectiveness. The classic model of software development and several variations of it are described in the next two sections. This allows consideration of the attributes, benefits, and shortcomings of various software development process models described in the literature. No one development process is appropriate for all software development situations. The following section presents the conceptual framework. This framework characterizes different types of software development tasks, and identifies effective matches of task type and development process approach. Theoretical and practical implications of the conceptual framework, as well as directions for future research, are addressed in the next section, followed by the chapter conclusion.

**THE WATERFALL MODEL**

The classic model for the software development process, known as the waterfall or life cycle model, was first suggested by Royce (1970). Boehm (1976) refined the basic model in a seminal article on software engineering. His version of the waterfall model, with slight variations in wording or minor embellishments, is still in use today as the standard for software development. One reason the waterfall model gained this widespread acceptance was its use by the U.S. Department of Defense as a standard for software contractors (Gabig 1991). A fairly basic version of the waterfall model is depicted in Figure 9.1 (Davis, Bersoff, and Comer 1988).

There are several software development life cycle stages which can be found in all versions of the waterfall model: requirements, design, coding, test, and maintenance. The development process is iterative, with cycles sometimes occurring between the various phases of development. At the end of each phase, a document is generated and reviewed, and its approval triggers the initiation of the next stage of development. Such documents are often called "milestone" documents since they denote conclusion of a
phase. The inclusion of a maintenance phase in the waterfall model makes it a true life cycle model, since maintenance changes the functionality of the product, it acts as a catalyst for incremental improvements and releases, and thus supports the extension of the product’s life until the next generation can be developed.

**Requirements**

The requirements or specification phase of software development involves the identification of what the software product will do (Boehm 1976). Customer requirements for the product must be identified and translated into general guidelines or objectives for the software development effort (often called a software specification or "spec"). The requirements phase is critical to a successful project because a thorough identification of, and understanding of, requirements up front can minimize expensive and time consuming corrections in later stages (Murphy 1990; Brooks 1987; Ramamoorthy et al. 1984). Requirements definition, however, is the most difficult software development task to perform well (Brooks 1987). Customers are often unsure of their exact needs and have difficulty articulating their requirements. In addition, the requirements definition process is filled with conflicts, both on the development side between marketing and programming personnel and on the customer side between users and buyers (Rosenthal and Saltzman 1990). Many ad hoc software organizations skip requirements definition completely, since it is easy to delay or avoid (Schach 1997; Boehm 1976). While the requirements task is obviously important, most software development research has focused on the more technical aspects of software engineering.

**Design**

The design phase takes abstract software requirements and translates them into more specific and detailed plans for implementation (Davis, Bersoff, and Comer 1988; Schach 1997). These plans can be in the form of flowcharts, pseudocode, diagrams, or other forms of representation (Shooman 1983). A good design clearly defines the relationships between elements of the development task. Requirements are partitioned into design elements which are then grouped into modules that form the basis for coding (Ramamoorthy et al. 1984). The design phase is particularly important for object-oriented or modular approaches, which require that individual modules or objects be identified and their interactions defined from the beginning of the design effort (Fichman and Kemerer 1993).

**Implementation**

The implementation or "code and debug" phase is often considered to be the easiest of the software development life cycle phases, particularly since the advent of high level languages (Gabig 1991). On the other hand, programmers are sometimes viewed as independent and eccentric people who work long hours on the relatively mysterious task of writing code (Schwartz 1991). Regardless, this phase often ends in development disasters, particularly when the requirements and design are not done thoroughly or are not created with coding issues in mind. If a design were perfect, an automated code generator could be used to implement the final product without requiring human effort. However, requirements are almost always incomplete or even partially uncertain when coding begins, and so the programmer’s expertise becomes a factor in effective project completion (Rowen 1990; Brooks 1995).

Perhaps the most important managerial aspects of this phase are the organization of the programming team (Schach 1997; Brooks 1995) and the productivity of the individual team members (Cusumano 1991; Rasch and Tosi 1992). No one person can code the entire system for large projects, so the task must be shared among programmers. At the same time, code elements are highly interdependent, and agreement among and communication between programmers is crucial (Hauptman 1990). Adding more staff to a late project often makes it later, as personnel interactions consume more of the available work hours (Brooks 1995). Individual programmer productivity also varies widely, and is primarily a function of individual ability (Rasch and Tosi 1992). Overall, implementation can be uneventful if the requirements and design are complete and clear, but in reality this is rarely the case. Therefore, coding can be a difficult task as a group of programmers struggles to develop a complex system, guided by incomplete or changing specifications.

**Testing**

The testing phase is crucial for ensuring the quality of the product, but is also the most costly of the "traditional" development phases of requirements, design, implementation, and testing (Cusumano 1991). On average, over 30 percent of the errors found in the testing stage result from incorrect or misinterpreted specifications (Ramamoorthy et al. 1984). Therefore, testing should start early in the project to reduce the potential cost of downstream changes (Boehm 1976). Some argue that testing should be done at every phase of the project, preferably by a separate software quality assurance (SQA) organization (Pfleeger 1992; Humphrey 1995). The main testing phase
(after coding is complete) should approach the problem in a bottom-up manner, first checking for individual component correctness and then attacking increasingly higher levels until the system is checked as a whole. Schach (1997) states that five facets of software quality should be tested: utility, reliability, robustness, performance, and overall correctness. One of the most difficult aspects of testing is deciding how to measure these dimensions, since correctness alone is hard to measure (Pfleeger 1992) and the others nearly impossible to assess objectively. Testing can be systematized, however, to conquer ambiguities in measurement. In an "ideal" software development environment, the software development process can be brought under "statistical control" in the same way as a manufacturing process (Humphrey 1995).

**Maintenance**

The final phase of software development is ongoing maintenance. There are three types of maintenance (Schach 1997) which are performed throughout the life of software: corrective (fixing errors), perfective (adding functional enhancements), and adaptive (coping with environmental changes). Although maintenance is often not considered to be part of the software development process, it is still an important facet of the software life cycle. Maintenance is the most costly (and time consuming) of all life cycle phases, comprising an estimated 60 to 80 percent of total life cycle costs (Murphy 1990). Maintenance can be laden with confusion and added difficulties because it is often performed by a separate group of programmers who are not as skilled as developers and are not familiar with the code (Schach 1997). To mitigate these effects, earlier process phases should consider the impact of decisions on software maintainability (Boehm 1976; Murphy 1990). Further, earlier stages should include substantive documentation efforts since good documents are essential for good software (Brooks 1995). However, the importance of good documentation is often not realized until the maintenance phase, where it is often too late to document products. It has been argued that documentation efforts should begin no later than the design phase (Brooks 1995).

**Summary**

Overall, the waterfall model has been useful in helping academics and practitioners conceptualize the software development process. The widespread use of the waterfall model in the software engineering community today, over 20 years after the original version was presented, attests to its descriptive powers. For a basic understanding of the process steps and their ideal order, the waterfall model is indispensable. However, there are shortcomings to the waterfall model in practice (Humphrey 1995). First, it assumes that software development follows a relatively linear progression through the phases with some iteration—an assumption many practitioners would argue is rarely met. Second, the waterfall model does not accommodate recent advances in software engineering, such as automated processes (for example, computer-aided software engineering), which can alter the structure of the development process. Third, the waterfall model relies on milestone documents written in technical language, which are difficult for customers to understand and often result in customer inability to follow development progress (Schach 1997). Fourth, the waterfall model does not handle the common case where specifications or designs are incomplete when coding begins (Rowen 1990). Finally, the waterfall model does not work well on projects which tackle unfamiliar problems (Gabig 1991). Thus, the waterfall model is useful in conceptualizing the development process, but may not be as helpful in the actual management of all types of software projects.

**ALTERNATIVE PROCESS MODELS**

Several alternatives to the standard waterfall model have been suggested in response to its weaknesses and changes in software engineering over time. These revised models take into account such advances in software engineering as prototyping, systematic reuse, and automated software development (Davis, Bersoff, and Comer 1988; Bersoff and Davis 1991; Rowen 1990). More specifically, technological advances including computer-aided software engineering (CASE) and its individual components (automated code generation, flowcharting, project management, and automated testing), object-oriented programming (OOP), fourth generation languages, and estimation models for software costs all may change either the structure of the waterfall model or the execution of its various phases. This section describes some of the major competing descriptions of the software development process.

**The V-Shape Model**

The V-shape model is in substance identical to the waterfall model, but is unique because it helps individuals conceptualize changes in the degree of detail involved in activities throughout the software development process. The V-shape model (Figure 9.2) was originally presented by Jensen and Tonies (1979) and is further discussed by Rowen (1990). As the early development phases proceed towards coding, the level of abstraction in the project decreases as specifications become increasingly detailed. The coding
phase entails the highest degree of detail in the project. During the testing phase, the level of abstraction again increases as development moves from the component level to the subsystem level and finally to the system level. The final step is the acceptance test, which signifies approval of the product by the customer or a customer representative.

In the V-shape model, correspondence between the requirements and design phases and which signifies approval of the product by the customer or a customer representative. The V-shape model depicts a linear and controlled process, and implicitly assumes that the customer will understand the content and activities of all software development tasks and sub-tasks (even the most detailed). The V-shape model is therefore unlikely to provide an accurate description of software development practice (Rowen 1990).

**Prototyping Models**

There is a great need to handle ambiguous customer requirements and incomplete specifications while still making progress in the software development process. Many software researchers have suggested the use of prototypes to cope with these difficulties (Brooks 1995; Schach 1997; Ramamoorthy et al. 1984; Rowen 1990; Bersoff and Davis 1991; Davis, Bersoff, and Comer 1988). The prototyping approach has been characterized as a W-shape model, a version of the V-shape model in which a system prototype brings the development process to a less detailed, higher, or overview level at some point during the process rather than only at the end of the process (Goldberg 1986). If multiple prototypes are used during development, the W-shape model becomes a sawtooth model (Rowen 1990). There are three distinct generic prototyping approaches: rapid or throwaway, evolutionary or incremental, and operational (Rowen 1990; Bersoff and Davis 1991; Davis, Bersoff, and Comer 1988). All of these approaches have a common goal: to clarify customer needs and gain greater understanding of customer requirements before full-blown coding begins. The primary characteristics of these various prototyping approaches are now described.

**Rapid or throwaway** prototyping methods involve a "quick and dirty" construction of a piece of the overall system. Rapid prototypes are built quickly, used to validate the developer's understanding of customer requirements for difficult or unclear aspects of the project, and then discarded. They are not meant to be fully functional software, but simply to bring the customer and developer closer to an understanding (Rowen 1990). The rapid prototype is discarded after it has served its purpose because it is not designed to be reliable and quality cannot be retrofitted (Bersoff and Davis 1991).

**Evolutionary** or incremental prototypes are high-quality builds which continuously evolve towards the final product (Bersoff and Davis 1991). These prototypes are made to be of final product quality and user-ready at each release, rather than to be discarded after use. The goal of gathering user feedback remains, and each successive iteration incorporates knowledge gained from experience with the previous one. The prototype in this case eventually becomes the final product (Schach 1997). Evolutionary prototypes add functionality in small, almost continuous steps while incremental prototypes are built in larger stages with a full design in mind at the time of construction (Rowen 1990; Davis, Bersoff, and Comer 1988). Each successive prototype in both cases comes closer to meeting actual customer requirements.

**Operational** prototypes, an extension of the evolutionary method, consist of relatively continuous development towards a final product. The operational prototype approach is similar to the evolutionary method, but operational prototypes are designed to be implemented while the product is actually in use (Bersoff and Davis 1991). A prototyper works at the customer site and collects customer feedback on the operating prototype. The prototyper frequently updates the system in the field with "quick and dirty" or throwaway prototypes incorporating customer feedback on the software. Periodically, the prototyper returns to the development site, where the "quick and dirty" changes are removed and replaced by final-quality updates. The new version of the software then becomes the new operational prototype.

Overall, prototyping methods are useful for coping with uncertainty in the specification and design phases of development (Rowen 1990). The waterfall model with prototyping is much the same as the standard model, except that it incorporates concrete...
customer feedback and involves some code writing in the early development phases. Thus, it should reduce the costs of making changes late in the process. The incremental, evolutionary, and operational methods make the waterfall process more iterative, as successive product builds are written and changed.

**Reuse Models**

Systematic code reuse has been discussed as perhaps the best way to improve the productivity, quality, cost, and speed of software development projects (Ramamoorthy et al. 1984; Cusumano 1991). Modular and object-oriented coding are two ways in which a systematic reuse approach can be implemented (Cusumano 1989; Fichman and Kemerer 1993; Schach 1997; Stark 1993; Weinberg, Guimaraes, and Heath 1990; Garceau, Jancura, and Kneiss 1993; Scholtz et al. 1993). Under these approaches, code elements are isolated from one another and are relatively generic in nature. This way, program modules or “objects” can be written to have flexible functionality which is achieved through standardized or “black box” interfaces. Further, these code elements can be thoroughly tested and then kept in electronic libraries for future use in similar software.

**Systematic reuse approaches** are relatively new. Problems exist in generating code for reuse, storing reusable code in an accessible way, and building systems from previously used code (Bersoff and Davis 1991). Programming libraries are sometimes difficult and time consuming to use, and so programmers may resist their adoption. In addition, since reuse libraries contain modules and objects designed for general purposes, code built from these elements may not run as efficiently for any particular situation as code written from scratch. On the other hand, cases have been recorded where a 90 percent reuse level has been achieved with simultaneously high productivity and quality levels (Swanson et al. 1991). Perhaps most importantly, code reuse can decrease dependence upon skilled programmers in the coding phase (Cusumano 1991).

The reuse concept can be extended beyond the code itself to include previously proven designs, requirements specifications, and test plans (Ramamoorthy et al. 1984; Bersoff and Davis 1991). If programming language choices or coding styles vary from project to project, requirements and designs may be easier to reuse than the code itself (Ramamoorthy et al. 1984). The benefits from reuse in other phases of the development process are much the same as in the coding phase. Reliability is increased due to the use of proven materials, and development time and costs are decreased due to reduced duplication of previous efforts (Bersoff and Davis 1991).

To establish systematic reuse in the software development process, a number of organizational obstacles must be overcome. First, a library of reusable code modules must be developed, requiring an investment of resources which may not provide an immediate payoff. Second, programmers may resist the change (Swanson et al. 1991) because code reuse may threaten their job security, the proper modules may be hard to find, or there may be no incentive to reuse code rather than build potentially better suited modules from scratch. Even without planned reuse, modular or object-oriented approaches alone may be a good first step towards reuse since reusable code elements will then be available.

**Software Factory Models**

The next step in the evolution of the systematic reuse model of software development is the software factory model. The underlying principle of the software factory is systematic reuse combined with standardized software development tools and controls (Cusumano 1989). Software factories frequently use automated tools (such as computer-aided software engineering) to enable narrowly skilled workers to develop software (Cusumano 1989, 1991). In addition, the software factory approach lends itself best to the development of a group of software products which are related in type and/or function (Cusumano 1989, 1991). The factory approach has been compared to total quality management because it also has process improvement, training, and consistency as central goals (Swanson et al. 1991).

The factory method requires separation of the coding phase of development from the front end specification and high-level design phases (Cusumano 1989). Factory proponents argue that these are two fundamentally different types of tasks, and different organizations should be put in place to specialize in each. This way, the factory can focus on the task of efficient and effective code production, while another organization focuses on determining customer requirements and setting software specifications. The factory concentrates upon implementing software specifications rather than on the full design and development process.

The overall purpose of the software factory is to move software development away from the craft practices of the past and towards a true engineering discipline. This thrust involves making software development a more standardized and controllable process. Due to the reuse strategy and automated support along with process standardization and separation, less skilled workers can be used in a more production-like coding process. More highly skilled “craft” workers are then free to concentrate on the demanding tasks of developing front end specifications and designs.

Factory approaches to software development have been used most widely in Japan. This approach seems to stem not only from the Japanese management style, but also from a shortage of skilled programmers (Cusumano 1991). Japanese factories tend to
emphasize reuse and automatic code generation, resulting in decreased time spent in the coding phase (Cusumano and Kemerer 1990). They have also concentrated primarily on semi-custom applications in familiar areas, leaving high product innovation to developers in the United States (Cusumano 1989, 1991).

Hybrid Models

The waterfall and related models do not accommodate tailored approaches to each phase of the software development life cycle and resist iteration and go-backs between individual phases (Plyler and Kim 1993). A few hybrid models in the literature suggest ways to do so, thus building on advantages and counteracting limitations of other models. The spiral model of software development (Boehm 1988), in which risk analysis is used at the beginning of each phase of the development process to decide which approach is most appropriate at that point, is the best known example of a hybrid model. The risk analysis is based on the objectives at the stage in question, the alternatives for implementation, and the constraints on the decision such as cost or schedule. The spiral model can include other model types (including the waterfall and prototyping methods as well as reuse strategies) as special cases of a more generalized framework. The packaged software model, on the other hand, calls for two iterative loops in development—a “requirements loop” using a prototype approach and then a more traditional “quality loop” encompassing coding and testing (Carmel and Becker 1995). Another hybrid approach, the concurrent process model, uses formal specifications to guide multiple, simultaneous coding efforts which are integrated and tested as a unit after completion (Aoyama 1993). All of these hybrid models combine different aspects of the other models into an approach tailored for the project in question.

CONCEPTUAL FRAMEWORK OF SOFTWARE DEVELOPMENT EFFECTIVENESS

There is widespread agreement that following a process for software development improves development outcomes such as product development cycle time, development cost, and product technical performance (Goldberg 1986; Garrette 1990; Davis, Bersoff, and Comer 1988; Carmel 1995). Process models (such as those reviewed in the previous sections) provide needed structure and rigor for software development. Although the benefit of having a process is clear, which process should be used has not been answered. Following is a conceptual framework of software development effectiveness that addresses which process model to employ in a given software development situation.

This is a “contingent” framework because it asserts that no one software development approach is appropriate for all software development projects. The framework is developed in several stages. First, we review literature motivating the contingency perspective and identifying contingent factors. Second, we characterize different types of software development projects in terms of their underlying task characteristics. Third, we address what “effectiveness” means in the software development context. Then, drawing upon literature and our field experience, we present the contingent framework identifying specific software development process models for effective development of particular software development project types.

Software Development Contingencies

The variety of software development process models that are available presents a challenge in determining which model is most effective for a particular project. The traditional waterfall model is most appropriate for large scale, customized systems (Carmel and Becker 1995), situations where there is little technological innovation and a tight schedule (Mahmood 1987), and “precedented” systems in familiar domains and with stable requirements (Gabig 1991). However, the waterfall model works poorly in situations including interactive end-user applications (Boehm 1988), unfamiliar systems (Gabig 1991), and complex systems for which the requirements are not well understood (Bersoff and Davis 1991; Mahmood 1987). Hough (1993) argues that any project which uses the waterfall approach risks inadequate functionality, schedule overruns, and even abandonment due to its “monolithic” nature (as opposed to incremental or iterative approaches). In all, the waterfall model appears appropriate for some but not all software development situations.

Other evidence supports the belief that certain development models are most effective only for a specific class of development situations. For example, exploratory data analysis has found that software projects facing very challenging conditions require greater “management power” (risk control, collaboration, requirements definition, conflict resolution, resource availability, and project planning) to be successful (Deutsch 1991). The “user involvement” literature contains considerable empirical evidence that, under conditions of high complexity and uncertainty about user requirements, development approaches involving user participation (such as prototyping methods) are most strongly related to project success (Mahmood 1987; Tait and Vessey 1988; McKeen, Guimaraes, and Wetherbe 1994). The literature on reuse and factory models, while primarily conceptual or descriptive, likewise suggests that the success of these approaches is contingent on the software development task (Banker, Kauffman, and Kumar 1992; Cusumano and Kemerer 1990; Griss 1993; Hofman and Rockhart 1994).
The literature clearly supports the view that relationships between software development process choice and development effectiveness are contingent, and that the contingent or deciding factor has to do with characteristics of the specific software project. Still, the theoretical and practical question of which process model to employ in a given situation remains unanswered due to the variety of results in the literature.

**Software Development Task**

Specifying a contingent framework for software development effectiveness requires a characterization of the different types of software development projects firms undertake. This characterization is the "software development task type," and is based on underlying characteristics of the software development task.

We first look to the user involvement literature to help characterize the software development task because it is the best developed area of research highlighting contingent relationships. Naumann, Davis, and McKeen (1980) argued that the appropriate degree of user involvement in the requirements phase of software development should be contingent upon the level of uncertainty in the project (measured by the project size, degree of structuredness, user task comprehension, and developer task proficiency). Other research has found that the most successful user involvement approach to software development is contingent upon the system complexity as measured by the number of interacting parts and lack of structure to represent them (Tait and Vessey 1988; McKeen, Guimaraes, and Wetherbe 1994; Tatikonda and Rosenthal 2000a). Related research has found similar results for the contingent variable of task complexity as measured by ambiguity in the tasks to be computerized (McKeen, Guimaraes, and Wetherbe 1994) or by the extent to which the tasks are highly structured or well understood (Edstrom 1988). All in all, the extant literature on user involvement indicates several bases for characterizing the contingent factor of software development project task.

More general research on software development poses a parsimonious conceptual classification of software development task attributes. Deutsch (1991) discusses project adversity as a contingent factor influencing the relationship between project management and project performance in software development. Project adversity consists of five elements: project size and character, business constraints, novelty of the technology to the organization, external interface adversity (interactions between the software and the surrounding environment), and number of users involved. The first three elements combine to form a single dimension of the software development project task type, which we describe below as "complexity." The "external interface adversity" and the "number of users involved" elements deal with the external environment's relationship to the software, and comprise a second dimension that we describe below as "conformity."

---

**TABLE 9.1 Intrinsic Characteristics of Software.**

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Discussion (Brooks 1987)</th>
<th>Representative Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complexity</td>
<td>&quot;no two parts are alike&quot;</td>
<td>Application domain</td>
</tr>
<tr>
<td></td>
<td>&quot;large numbers of states&quot;</td>
<td>Interdependence</td>
</tr>
<tr>
<td></td>
<td>&quot;elements interact . . . in some nonlinear fashion&quot;</td>
<td>Number of unique elements</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Project size</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Novelty</td>
</tr>
<tr>
<td>Conformity</td>
<td>&quot;many human institutions and systems to which . . . interfaces must conform&quot;</td>
<td>Customization</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Purpose</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Physical product interfaces</td>
</tr>
<tr>
<td></td>
<td></td>
<td>User process interfaces</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Hardware interfaces</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Operating system interfaces</td>
</tr>
<tr>
<td>Changeability</td>
<td>&quot;constantly subject to pressures for change&quot;</td>
<td>Functional enhancement</td>
</tr>
<tr>
<td></td>
<td>&quot;pure thought-stuff, infinitely malleable&quot;</td>
<td>Error correction</td>
</tr>
<tr>
<td>Invisibility</td>
<td>&quot;unvisualizable&quot;</td>
<td>Control flow</td>
</tr>
<tr>
<td></td>
<td>&quot;not inherently embedded in space&quot;</td>
<td>Data flow</td>
</tr>
<tr>
<td></td>
<td>&quot;no ready geometric representation&quot;</td>
<td>Dependency</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Time sequencing</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Name-space relationships</td>
</tr>
</tbody>
</table>

Brooks (1987) has also identified essential characteristics of a software development task (see Table 9.1). His "complexity" and "conformity" characteristics are consistent with our categorization of Deutsch's elements into two dimensions; however, his "changeability" and "invisibility" characteristics describe differences between software and physical products rather than discriminate between different software development tasks.

**Complexity**

Software is inherently complex in nature (Brooks 1987). The complexity of a software development effort, however, may vary with a number of factors. First, some software application types may be more complex than others due to size of the program, number of function points, number of interdependencies among elements of the program, and the number of input and output sources. For example, a user-interactive application is typically more complex than an offline batch processing system, or a word processor more complex than a programming language compiler. Second, software
with a structure involving more elements and greater interdependence between elements will be more complex to develop. Third, software development efforts involving larger project teams will be more complex. Finally, software development efforts involving a type of application or technology that is new to the organization (as opposed to a familiar problem domain and technology) will be more complex. For example, virtual reality products and packages designed to help users explore the World Wide Web are relatively new application types (to most organizations). Similarly, the first software written using object-oriented programming techniques in an organization represents use of a new technology to that organization. So, in general, more complex software projects are larger, more difficult to develop, and require more new software technology development than less complex software projects.

**Conformity**

Conformity refers to the fact that software is often designed to conform to an existing interface or process, or to fit the needs of a particular extant system (Brooks 1987; Salzman and Rosenthal 1994). The conformity of a software product thus depends in part on whether it is customized for an individual user or group of users (high conformity) or designed as an “off-the-shelf” package (low conformity). In addition, the degree to which the software is designed to be part of a physical product, to run on a certain platform, or to use a particular operating system determines the extent to which it must conform to a particular set of requirements. Brooks (1987) states that the conformity of software to external interfaces is a critical descriptor of the software task. Carmel and Becker (1995) have argued that packaged software development requires a new process model because previous literature focused primarily on large-scale custom applications versus smaller scale, standardized products. Additionally, Whang (1992) reports that the U.S. government uses the degree of software customization to distinguish between software “products” and software “services.” Case evidence indicates that software conformity to a physical product interface can also pose unique demands on the software development organization (Rosenthal and Tatikonda 1993). In general, projects of high conformity are more custom designed and more constrained by user processes and requirements, hardware and operating system protocols, and physical product interfaces than projects of low conformity.

**Complexity and Conformity**

Complexity and conformity together describe the “project adversity” or challenge posed by the software project task to the development organization. Complexity generally represents internally posed challenges by capturing project size, newness of the technology and application area to the organization, and quantity of internal task elements and their interdependencies. Conformity represents externally posed challenges since it captures the degree to which specific external interface requirements (such as the user process, hardware and operating system constraints, or whether the software is embedded in a physical product) must be met. These two dimensions define the software development task type (see Figure 9.3), and identify four specific task types. Knowing the task type helps determine the most effective software development management approach. Before presenting the full framework of software development effectiveness, we first consider what “effectiveness” means.

**“Effective” Software Development**

There are many perspectives on what constitutes “effectiveness” or “success” in product development. Research in the physical product development arena has explored the determinants of new product and development project success. Market acceptance and commercial significance are criteria often used to assess new product success (Marquis 1969; Cooper 1979). Financial outcomes such as market share and profitability have also been used (Rothwell et al. 1974; Cooper 1982; Zirger and Maidique 1990). For physical product development projects, success is generally a combination of technical, cost and schedule performance—development success is clearly a multidimensional phenomenon (Rubenstein et al. 1976; Rosenthal and Tatikonda 1993; Griffin and Page 1996; Smith and Reinertsen 1998).

The management information systems (MIS) literature views software development success differently, in part because this literature focuses almost exclusively on
software development that is internal to the firm. DeLone and McLean (1992) define information systems success in six dimensions: system quality, information quality, use, user satisfaction, individual impact, and organizational impact. This view of software development success is quite internally focused, however, it does reflect to some degree the market acceptance and technical success dimensions observed in physical product development research. Investigations of the effectiveness of software development project management approaches under situations of differing project task types characterize performance in terms of user satisfaction, requirements achievement, comparative product quality, cost performance, or schedule performance (Deutsch 1991; Hauptman 1986). We use "effectiveness" and "success" to encompass overall development performance including technical, budget, schedule, and market acceptance outcomes.

Contingency Framework

We now present the contingent framework for software development project effectiveness. It is based on the literature, our software development work experience, field observations of software development practice, and discussions with experienced software development managers and programmers. Figure 9.4 depicts the overall model, and Figure 9.5 shows specific contingent relationships. Each quadrant of Figure 9.5 shows the software development process model we hypothesize is associated with the greatest execution success for the corresponding task type. Each cell (that is, each software development task type) and its corresponding process model is described in the following paragraphs.

High Conformity/High Complexity

Software development projects in this cell have the highest overall project adversity. Software of this type is a classic "one-of-a-kind" system, a complex and innovative piece of code which is customized for a particular use. The waterfall model is not effective for such customized and innovative systems because specifications are typically uncertain (Boehm 1988; Gabig 1991; Bersoff and Davis 1991; Mahmood 1987). Literature indicates that a prototyping approach compensates well for these difficulties. Prototypes clarify customer requirements when specifications are incomplete or unclear (Rowen 1990). User involvement in general and prototype methods in particular are effective in complex system situations (Naumann and Jenkins 1982; McKeen, Guimaraes, and Wetherbe 1994; Tait and Vessey 1988; Mahmood 1987); when the underlying problem is unstructured (Edstrom 1988; Anderson 1983; Ives and Olson 1984); when user acceptance is important (Gibson 1977; Ives and Olson 1984); and when the overall project environment is uncertain (Naumann, Davis, and McKeen 1980; Naumann and Jenkins 1982; Leonard-Barton and Sinha 1993). Prototypes are more effective in cases where the system is customized or "tailored" to user requirements (Whang 1992; Sioukas 1995; Naumann and Jenkins 1982) and when the system is a radical innovation (Leonard-Barton and Sinha 1993; Holtzblatt and Beyer 1993; Nickols 1993). Thus, prototyping in cases of high conformity and complexity helps clarify ambiguities in the customer requirements, bring the customer into the design process to enhance satisfaction, and maintain flexibility to changing needs and technologies (Hough 1993).

High Conformity/Low Complexity

Systematic reuse systems work best for software tasks of this type. Reuse (and particularly code reuse) is effective for relatively narrow, well understood, and static software tasks (Biggerstaff and Richter 1987). Systematic reuse is appropriate when producing "families" of related applications where there is some commonality between systems and thus complexity is lower (Griss 1993; Swanson et al. 1991). Reuse provides
process flexibility and the ability to customize systems within a particular software application domain (Griss 1993; Swanson et al. 1991). When a library of modules from similar projects exists, a wide variety of software programs can be assembled with little additional effort (as in the case of modular product designs for physical products, see Ulrich 1995). The Japanese approach to software development places great emphasis on systematic reuse and has been quite effective for custom and semi-custom applications that are not highly innovative (Cusumano and Kemerer 1990).

**Low Conformity/Low Complexity**

Software tasks in this cell are relatively familiar and routine to the development organization. Such tasks include software that is similar to previously developed products (and hence is of low complexity to the development organization). Software factories are effective for such familiar, less innovative, and less customized systems (Cusumano 1989, 1991; Swanson et al. 1991). Accordingly, a basic software factory, which combines systematic reuse, standardization, and automation, is most effective for projects of this type. Software development tasks of this type are the least "uniquely software" of the four quadrants, suggesting that such projects are more amenable to physical—or production—management techniques (hence the term "factory"). For systems of moderate conformity, a so-called "flexible" software factory uses less skilled labor for computer-aided implementation and more skilled staff for the specification and design tasks (Cusumano 1989; Griss 1993). In a flexible software factory, computer-aided software engineering tools are combined with extensive reuse for quick development of software (Hofman and Rockhart 1994). As conformity decreases, less front-end effort and comprehensive reuse is necessary. Instead, a basic automated software factory approach is appropriate. Therefore, as one moves from high to low conformity within the category of low complexity systems, the emphasis on automated tools should increase and the emphasis on reuse may decrease.

**Low Conformity/High Complexity**

An organization is unlikely to be economically viable in the long run if its products are only of the low conformity/high complexity type. The constant development of unfamiliar but off-the-shelf products displays unfocused competitive strategy and insufficient development of long-term organizational competencies (we believe this software task type is not common in practice). However, an organization might have an individual project of this type at a given time. Such a project might represent a market entry point for a new product family. Still, the firm’s long-term position should be in other quadrants to ensure company survival. This suggests a "portfolio" strategy for the firm. In product development organizations, a portfolio of projects with different characteristics will exist at any point in time (Wheelwright and Clark 1992). Projects over time are often grouped in product families that rely on a common platform (Meyer and Utterback 1993; Cusumano and Nobeoka 1994; Tatikonda, 1999).

When a project of low conformity and high complexity is the first in a product family, future projects should be of lower complexity since the application area would be more familiar to the firm. Hence, less innovation is necessary in such future endeavors. An organization in this situation should employ either a systematic reuse strategy (in which future software products would be customized versions of the first product) or a factory strategy (in which future products would be both off-the-shelf and of a similar application type) in the long run. In either case, a library of code and design information for future reuse must be prepared. Modular or object-oriented programming enables such a strategy (Stark 1993; Weinberg, Guimaraes, and Heath 1990; Garceau, Jancura, and Kneiss 1993; Scholtz et al. 1993). The development organization must anticipate moving from one quadrant to another, and should plan its development processes accordingly.

**IMPLICATIONS**

The literature review and conceptual framework pose theoretical implications, directions for future research, and prescriptions for software development management practice.

The perspective of different task types and execution approaches has simple theoretical links to organizational information processing theory. This theory argues that organizations undertake tasks that vary in the uncertainty they pose to the organization conducting the task. Uncertainty is essentially "lack of knowledge," an incomplete understanding of what the task at hand is and how to execute it properly. Task uncertainty is reduced by choosing better understood tasks or employing appropriate organizational approaches to conduct the task (Galbraith 1973, 1977; Tushman and Nadler 1978). In our conceptual framework, task uncertainty is represented by project adversity and its two dimensions of conformity and complexity. Further, in this framework, task uncertainty reduction is represented by different software development process models or approaches. Information processing theory posits, just as we have in the conceptual framework, that appropriate matching of tasks (uncertainty sources) and organizational approaches to their execution (uncertainty reducers) is necessary if high task effectiveness is to occur. This highlights an organization theory basis for the conceptual framework, and may aid in future construct measurement, empirical tests, interpretation of findings, and integration of broader literatures with technological innovation.

The conceptual framework treats a software development project as one "large" task which has many phases or sub-tasks (as illustrated by the waterfall model). We
believe the multiple phases of work can be treated as a whole, and that the whole project is characterized by its complexity and conformity. However, some who study development of physical products argue that the "fuzzy" front end of a product development effort and its more execution-oriented back end are qualitatively different in style, objectives, and appropriate use of personnel (Smith and Reinertsen 1998). Others describe the two ends in terms of task definition and execution, calling them "product definition setting" and "product definition management" (Bacon et al. 1994). In the software development context this would mean that the front end (requirements and design) should be managed as one task while the back end (coding, test, and maintenance) should be treated separately and managed differently. Such a dichotomy is implied in the software factory model, and is consistent with the hybrid models reviewed earlier. Further, the common problem of incomplete and incorrect specifications might be ameliorated through special emphasis on the front end.

If the two ends are fundamentally different tasks, then they presumably need to be managed in fundamentally different ways rather than as two sub-tasks in one large task. Alternatively, "bundling" of separate, possibly related approaches for the two ends under a single hybrid approach may be appropriate. Most software development research focuses on the more technically oriented coding, testing and maintenance phases of the software life cycle. The front end phases of specification and design merit additional research attention. Further, whether software development should be managed as one or two large tasks remains unresolved and merits further research.

The means by which organizations develop software continues to increase in variety and technical sophistication (Buxton 1990), and new software development technologies continue to emerge. Two new technologies—object oriented programming and design, and computer-aided techniques—were discussed in the context of the conceptual framework. Other new technologies include "virtual reality" techniques which can be employed to capture user requirements for software functionality (Schacht 1994; Kumada 1992), and "artificial life" which develops software code that uses genetic algorithms to rewrite itself over time (Brandt 1994). The constant stream of new technology offers new options for managers determining which software development approach to employ, and requires periodic reconsideration of the conceptual framework. The "best" execution approaches for given task types may well be different in future points in time due to new development technologies.

The conceptual framework applies to large-scale systems produced for sale as software products or as a component of physical products. The concepts may also apply to smaller systems and in-house MIS projects. Future research should address appropriate execution approaches for development of small software projects (such as those completed by a single person in a few days or weeks), and determine which execution techniques are effective for both small- and large-scale systems.

The conceptual framework essentially presents propositions about the contingent effectiveness of various software development execution approaches for different software task types. This sets the stage for empirical validation of the framework. A major challenge before validating the framework is operationalizing the constructs of interest: software development task types, process models, and project outcomes. There are few survey scales, measures, and instrumentation procedures for these constructs in the extant software development literature. Accordingly, development of measures may require descriptive case-based field research, which is in turn followed by a larger-sample survey study to empirically confirm propositions inherent in the model.

This chapter (especially the first two sections) identifies lessons about software development practice. We find that software development is rarely a simple, linear process (although it is easy to describe it that way); rather, in the real world, it is multi-stage, iterative, and overlapping. Determining user requirements and product specifications is quite challenging for many firms. Further, software development requires a perspective well beyond requirements and coding—maintenance must also be considered because it significantly impacts the usable life and long-term profitability of a given product. This highlights the need for thorough product documentation, an effort which must start early in the development process.

A significant practical concern arising from the literature review is that while various software development process approaches have strengths and weaknesses, no one approach is appropriate for all software projects. This is because there are fundamentally different types of software projects which in turn pose different execution challenges. Accordingly, firms should evaluate the type of project they plan to undertake (by analyzing the underlying task characteristics), and then match the project to an appropriate execution approach in order to achieve high project effectiveness (as shown in the conceptual framework, Figures 9.4 and 9.5). Alternatively, firms can choose software tasks consistent with the development execution approach in which the they already have significant capability.

The framework poses implications for firms that have several software development projects underway simultaneously or in sequence. A firm may wish to have a "focused" or "differentiated" software development product portfolio. The focused portfolio has only software development project tasks of a similar nature at a given time. These tasks are in the same conceptual framework quadrant and require broadly similar execution approaches. A differentiated portfolio has a mix of projects underway that are different tasks. In this case the firm must be capable of multiple execution approaches because the tasks are in different conceptual framework quadrants.

A related issue is the "evolution" of software task types and execution approaches. As a firm gains familiarity with a market, new software development projects undertaken by that firm may be of lower complexity or conformity than before. This suggests...
the firm should plan movement from quadrant to quadrant, and generate new skills in development execution (Wheelwright and Clark 1992; Cusumano and Nobeoka 1994; Tatskonda and Rosenthal 2006b). In some cases, the development approach already employed can serve as a basis for the approach necessary for a different software development task type. For example, "systematic-reuse" is an element of the broader, more sophisticated "software factory." Firms may choose their markets (or may choose to develop their markets) so that the sequence of organizational capability development is feasible and efficient. Compounding the need for new organizational competencies is the reality that new software development technologies continually emerge. Firms must monitor these potential capabilities and gain skills in the appropriate ones.

Different software task types, differentiated product portfolios, and software product evolution suggest a fascinating and complex environment for software developers. These also suggest a rich research opportunity in devising management approaches and development techniques which are effective for more than one software task type. Such approaches would likely be quite encompassing but flexible in application, have strong emphasis on database support, and have planned generation and multiple application of code modules.

**CONCLUSION**

The importance of new software to industry, government, and our personal lives continues to increase. However, software development projects are notorious for being late, over budget, and resulting in products with incomplete technical performance. To address software development effectiveness, this chapter reviewed literature on management of the software development process, alternative development models and approaches, and stages in software development. There is limited research on the management of software development. Further, empirical studies to date present partial and conflicting findings on how to effectively manage software development efforts. As Frederick Brooks (1987) said, there is "no silver bullet" for software development: no one technology or management approach will cure all the problems of software development.

This chapter integrated technical and managerial literatures to develop a conceptual framework of effective management of different types of software development projects. The framework reflects the belief that there is no one method for managing software projects because each project type has unique requirements and management implications. In developing the framework, we first considered the characteristics, benefits, and shortcomings of various software development execution approaches. Second, we characterized software development projects along two dimensions, complexity and conformity, which are based on underlying task characteristics. These dimensions represent different challenges to the development organization, and result in four archetypal software development project tasks. Third, based on the literature and our field experience, we presented a contingent framework identifying effective software development execution approaches for different software development task types. This framework is a basis for further management research on effective software development. In addition, the framework has implications for software development firms regarding: choice of software development tasks to undertake, product portfolio content and strategy, implementation of software development execution capabilities, and strategies for product type evolution and corresponding renewal of software development organizational capabilities.

**REFERENCES**


Contents

Preface ix 

Section 1: Integrating New Technologies into Supply-Chain Operations

1. The Relationship-Technology Interface: A Path to Competitive Advantage
   Sandy D. Jap 3

2. Digital Marketplaces and Efficient Supply Chains: Can They Co-Exist?
   Ram Ganeshan and Katherine Gehman 24

3. Internet Retailers' Dilemma of Operational and Market Efficiencies
   Joseph P. Bailey and Elliot Rabinovich 39
4. A Case Study of an ERP Implementation: Promises, Phantoms, and Purgatory
   Gigi G. Kelly  
   58

5. Web-Enabling the Supply Chain: An Exploratory Case Study
   Ram Ganeshan  
   73

Section 2: Technology-Based Product and Service Development

6. Identifying and Managing Programmatic Efficiency Differences across Technology-Based Product Development Capabilities and Uncertainty Environments
   K. Scott Swan  
   93

7. Management of Information Technology Driven Product Development Processes
   Nitin R. Joglekar and Ali A. Yassine  
   125

8. Mass Customization: Strategic and Operational Considerations
   Rebecca Duray  
   153

   Mohan V. Tatikonda and Main Lorence  
   171

    Mohan V. Tatikonda and Valerie A. Zeithaml  
    200

Section 3: Knowledge Management and Supply Chain Integration Issues

11. Inward Technology Transfer: A Key Link in the Technology Supply Chain
    Mohan V. Tatikonda and Gregory N. Stock  
    237

12. Experience-Based Productivity Improvements in Project Environments
    Tonya Boone  
    272

    K. Scott Swan, Brent Alfred, and Aldor Lanctot  
    282

14. Pricing Supply Flexibility: The Impact of Variability and Information Asymmetry
    Alexander O. Brown  
    306

15. Vendor-Managed Inventory in Production Constrained Settings
    Michael Fry, Roman Kapuscinski, and Tava Lennon Olsen  
    325

16. Supply-Chain Integration: Putting Humpty Dumpty Back Together Again
    352

Index
   377

About the Contributors
   381