Information Technology and Management 2, 53–70, 2001

Traditional, Iterative, and Component-Based Development: A Social Analysis of Software Development Paradigms

DANIEL ROBEY * and RICHARD WELKE

Department of Computer Information Systems, P.O. Box 4015, Georgia State University, Atlanta, GA 30302-4015, USA

DANIEL TURK

Colorado State University, USA

Abstract. Information systems have always been developed through social processes, wherein actors playing a variety of specialized roles interact to produce new business applications of information technology. As systems development practices continue to evolve, an ongoing assessment of their social implications is required. This paper develops a framework for understanding the potential social implications of an emerging, *component-based development* paradigm. Like two alternative paradigms for systems development, the *traditional life-cycle* and the *iterative-incremental* paradigms, the new component-based paradigm requires that certain generic roles be performed to build a desired application. For each paradigm, we identify the actors who play different roles, specify the nature of their interdependence, and indicate the requirements for managing conflicts constructively. The framework may guide research into the social dynamics of system development and serve as a tentative guide to the management of information systems development.

Key Words: software development paradigms, component-based development, software development roles, social implications

1. Introduction

Information systems (IS) development is a social process that involves actors in various interacting social roles [23,24]. While certain roles have become generic across the range of development approaches that have evolved in practice, the particular mix of actors varies with the development method used. The distinction between the user and developer roles has been the primary focus of prior research (e.g., [3,21,36,37]). Users are typically responsible for specifying information requirements, whereas developers retain responsibility for formal analysis, design, coding, testing, and maintenance. Additional roles include managers, who are responsible for facilitating IS development and allocating resources and attention to the overall process [47], and guarantors of the applications developed [14,34]. Usually, different people perform each of these roles,

* Corresponding author. E-mail: drobey@gsu.edu.

but sometimes individuals play multiple roles. Actors engaged in IS development may be employed by the organization designing the system, or they may work for external parties such as consulting firms, equipment and software vendors, or corporate partners [47]. Thus, because of the diversity of actors and arrangements, IS development remains essentially a social process.

IS development is assisted by methodologies and methods, which may affect the interactions among actors. Dozens of methodologies are used today, and each specifies a sequence of steps or procedures for completing the process of system development. Although methodologies are more commonly differentiated by some underlying technical feature, such as the degree of automation in code generation and the extent of object-oriented programming, they also differ in their social implications. By assigning responsibility for different activities and by indicating how roles are to interact, development methodologies may bestow more power to certain roles while diminishing others in importance [7,44]. For example, activities like requirements analysis may call for systems analysts to conduct one-on-one interviews with users. Other activities, such as structured walkthroughs, may occur in group meetings led by analysts with users in attendance. Thus, in addition to imposing a technical discipline, formal development methodologies also shape the character of social interactions.

All social processes are affected by the amount of role differentiation among actors and the degree of interdependence among them [2,4,15,39]. Together, these variables influence the relationships among parties engaged in work, potentially affecting communication, power and conflict among participants. The effects of IS development methodologies on these behaviors during systems development can, in turn, potentially account for some of the success or failure of development projects. Prior research has shown the importance of communication to project success [18,53]. The frequency and types of communication may vary, ranging from an honest and trusting partnership in one situation to a defensive and error-prone relationship in another [13,49]. Prior research has also focused on the management of conflict during systems development [3,41–43]. When used constructively, conflict can enhance system development outcomes by encouraging meaningful disagreement among participants. Where mismanaged, conflict can become a destructive and demoralizing force [5].

In this paper we present a framework for studying and managing the social consequences of alternative IS development methodologies. We begin by grouping development methodologies into three paradigms for system development: the traditional life-cycle paradigm, the iterative-incremental paradigm, and the emerging component-based development paradigm. We distinguish these paradigms from each other on the basis of four dimensions that describe their social implications rather than their technical requirements. We then relate these dimensions of development paradigms to the social consequences of their use. Finally, we discuss the implications of our analysis for managing the system development process.

2. Systems development: methodologies and paradigms

Paradigms, methodologies, methods, and tools are nested concepts, as shown in figure 1. *Paradigm* is used in this paper as a general concept that captures the general characteristics shared by a group of methodologies. We consider *methodology* to mean rules and specifications for conducting the overall process of IS development, including rules for prioritizing and sequencing activities in the process. Some methodologies, such as Martin's Information Engineering [31–33] are described in great detail, while others are described in less detail (e.g., RAD, Prototyping, SDLC), but each methodology encompasses the overall process of software development. By contrast, we use the term *method* to refer to analytic techniques used within a given methodology. Furthermore, methods may be executed with a variety of *tools* that consist of packaged software applications.

The vast number of methodologies may be classified within two established paradigms. The traditional life-cycle paradigm is based on the assumption that proper execution of each step will eliminate corrections later in the process and will thus result in reduced overall costs. By contrast, the iterative-incremental paradigm assumes that it is improbable to produce a correct system without the repetition of cycles and the close interaction of users and developers contributing to the creation of an evolving system.

Recently, a third IS development paradigm has assumed prominence among IS professionals and user communities [38,54]. The component-based development (CBD) paradigm regards systems development as a user-centered activity in which component brokers supply modules and utilities to the user who assumes responsibility for assembly, testing, operation and maintenance. The CBD paradigm assumes that a knowledgeable user community comprehends its own information needs, is able to obtain the required components for building systems, and is competent enough to assemble these components into useful and meaningful applications. Given the commercial availability of high-level, "plug-and-play" components, users can develop and maintain their systems with greater autonomy than permitted by the other two paradigms. Although the CBD paradigm appears to bypass the historically difficult relationship between users and in-house system professionals, it introduces new social relationships that may prove equally

Concept	Definition
Paradigm	Conceptually captures principles common to a group of methodologies.
Methodology	Defines the overall process for developing applications and identifies the role of methods within the process.
Method	Prescribes techniques that comprise a specific methodology.
Tool	Delivers software packages that implement specific methods.

Figure 1. Nested relationship among IS development concepts.

problematic. In place of interdependence among roles housed within the same organization, the CBD paradigm brings users into greater contact with commercial suppliers external to an organization. Navigating electronic markets creates new communication needs and requires alternative methods for resolving conflicts among parties contributing to IS development. In this paper, we seek an understanding of these social relationships and their managerial implications.

3. Dimensions of IS development paradigms

The three paradigms for system development can be distinguished on the basis of four analytical dimensions: linear–iterative, developer-centered–user-centered, new development–reassembly, and structured–unstructured. The basic definitions of these dimensions and their hypothesized effects are described below.

3.1. Linear–iterative

A linear development process is where each phase of development is completed and approved before moving to the next phase. This sequential process does not provide for returns to previous phases unless the whole process is commenced anew. The waterfall model [40] is an extreme example of the linear approach. Just as water does not flow uphill, neither are prior steps of IS development to be repeated. By contrast, an iterative process is designed to include repetition so that work completed in an earlier cycle can be refined and corrected. Interactive prototyping is an example of the iterative approach.

The primary effect of using linear methodologies is to enforce a sequential interdependence among parties contributing to the development process. Sequential interdependence effectively limits parties' participation to specific activities that occur after other activities have been completed. By contrast, an iterative methodology brings parties into reciprocal, team-like interaction wherein participants work together simultaneously. In general, the more intense interpersonal interactions created in interactive methodologies are likely to increase the opportunities for direct communication and also increase the opportunity for conflict [41].

3.2. Developer-centered–user-centered

A developer-centered process is driven by developers, who structure and control users' activities. Although users may play essential roles in proposing and justifying an IS project, and in specifying their information requirements, the process of development is conducted under the assumption that the developer knows best. A user-centered process, by contrast, elevates users to a more prominent role by giving them responsibility for initiating, organizing, and structuring the work of developers and other contributors [25]. In the user-centered approach, users and developers collaborate, each contributing their own expertise to build software solutions.

The primary implication of this dimension is for the distribution of power among participants. Those who control the process are in a better position to rule on controversial issues and to allocate resources in a manner that fulfills their objectives [7]. Developer-centered methodologies place developers in control and give them greater power than their clients. By contrast, user-centered methodologies reverse this balance of power and place users in control of development.

3.3. New development–assembly from reusable components

New development requires that applications be developed from "scratch", building software functionality from low-level source code. Assembly from reusable components, by contrast, relies upon previously built components that are available either in existing applications or commercially from external suppliers. For example, when an application requires data to be sorted, the new development approach would direct a developer to write new code for the sort routine. Assembling the same application from reusable components, however, would direct the developer to reuse a sort routine from a library of existing components. Components for other functions, such as user interfaces, database storage and retrieval, or network connectivity, could also be acquired from existing software sources.

The social effect of this dimension is to alter the dependence of users on developers. When relying upon system developers to produce original code, users become more dependent upon them. Less dependence upon developers occurs when assembling systems from available components.

3.4. Structured–unstructured

This dimension refers to the formality of the development process. Structured development follows a specific plan, executed according to formal rules and methods, to produce pre-defined deliverables. Unstructured methodologies follow a more spontaneous, *ad hoc* approach in which the development plan is adjusted based on knowledge that is generated as the project proceeds. With an unstructured approach, products may emerge at any time, not just as regularly scheduled deliverables. For example, whereas a structured process may require user trials to follow formal testing, an unstructured development process may generate working prototypes at any time.

The degree of structure in a development methodology potentially affects the power relationship between users and developers. More structured methodologies favor developers by prescribing the rules for a complex social process. By contrast, unstructured methodologies offer greater potential for balanced power relationships [7].

A development paradigm's social profile may be portrayed by rating subjectively its relative emphasis on each of these four dimensions. Figure 2 illustrates the social profiles of all three development paradigms. The profiles shown illustrate the ideal, or pure portraits of each paradigm. In reality, individual methodologies will vary from the ideal and not exhibit the four dimensions to the extremes shown here. The ideal profiles are used here to illustrate the key differences between the three paradigms.

4. Social analysis of three is development paradigms

In any social process, interactions occur among actors playing particular roles. In IS development, many roles have become generic insofar as they appear in methodologies within all three development paradigms. Clearly, the roles of user and developer are the focus of most inquiries into the system development process [36]. However, other important roles can be identified. A manager role is responsible for identifying needs, allocating resources, monitoring progress, and making decisions affecting the scope of an IS project and the needs it addresses [47]. Also, a guarantor role stands behind the system's performance, assuring its quality and suitability for the application [14,34]. While in specific instances there may be additional roles, such as the project champion [6], or more specialized versions of these roles (such as the project sponsor which is normally a subset of the manager role), these four occur in most approaches to information system development. The actors performing these generic roles vary from one paradigm to another. For each paradigm, therefore, a table is provided summarizing the actors associated with the four generic roles.

4.1. Traditional life cycle paradigm

The social profile for the traditional life-cycle paradigm is located at the far-left side of the scale in figure 2. The traditional life-cycle paradigm follows a linear approach to systems development, proceeding through analysis, design, coding, testing,

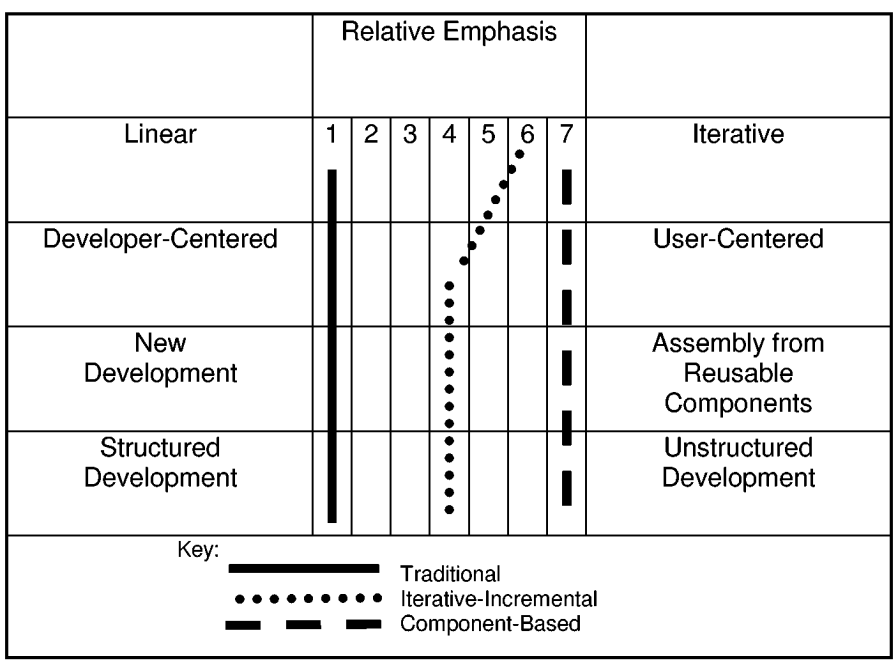

Figure 2. Social profiles for three IS development paradigms.

and maintenance. The general rule in methodologies that follow the traditional life-cycle paradigm is that the developer does not proceed to the next phase until the current one is completed. The assumption underlying the traditional life-cycle approach is that proper execution of earlier activities reduces the number of later corrections in the process and thus reduces overall costs of development.

Traditional approaches tend to be developer-centered. Although users may initiate requests for applications to be developed, developers control the development process. Developers hold meetings to elicit requirements, they review specifications, conduct testing, orchestrate walkthroughs, and so on. Users participate by providing information and by certifying that the end product meets their functional requirements, but users are peripheral to the design process itself. Development methodologies within the traditional life-cycle paradigm are typically employed in the development of new systems. While developers following this approach may have experience with many similar systems, specific components and objects tend not to be reused. Designs, source code, libraries, and other products developed for other systems tend to be reused only when they perform identical functions in both applications. New features are developed by drawing from developer experience, not the tangible residue of that experience.

Methodologies within the traditional life-cycle paradigm also tend to be structured and formal. Because each phase must be approved before being handed off to the next phase, it is especially important to devise formal procedures and standards for judging acceptability at each phase. Structured methodologies accomplish this through formal checklists and procedures.

Typical methodologies. One methodology that exemplifies the traditional approach is structured analysis and design [19,55]. The defining characteristics of structured analysis and design are thorough documentation, separation of logical design from physical design, and graphic aids to support analysis and design decisions [40]. Data flow diagrams, data dictionaries, and functional decomposition are specific methods that embody these characteristics and that are central to methodologies in the traditional life-cycle paradigm. Such methods allow system professionals to assess business operations, document existing activities, model information flow and source materials, and create diagrams and data dictionaries that show the essential operations of the organization. These processes are performed at multiple levels of abstraction in order to obtain both high- and low-level views of a business process.

Roles and interdependencies. Table 1 provides the role/actor matrix for the traditional life-cycle paradigm. There are two predominant roles in the traditional approach: the developer and the user. There are several IS professional actors who fill the developer role: analyst, designer, and programmer. The analyst performs an assessment of the current system, finding out what is currently being done and what is needed. The designer creates a proposed solution based on current activities, current technology, future plans, and organizational constraints. The programmer implements the design in a specific programming language. End-users use the system interactively, or obtain information from it in the form of reports, and provide information that becomes input for the system.

Table 1
Actors and roles in the traditional life-cycle paradigm.

Actors	Roles			
	Developer	User	Manager	Guarantor
Analysts	X			
Designers	X			
Programmers	X			
End user		X		
User manager		X	X	
IS manager			X	X

In addition, end users typically provide requested input as developers assess needs and design new systems. User managers are responsible for the business area for which the application is being developed.

The manager role is performed by both user managers and IS managers. User managers allocate resources through charge-back payment systems and monitor the performance of the system once it is in production. IS managers allocate human resources to various projects and assume responsibility for project completion and budget adherence.

The developer holds the prominent role in the traditional life-cycle paradigm. Although, the developer is dependent on the user to obtain information, the developer provides the structure for gathering information and avoids dependence by imposing a sequential order to the entire systems development process. As the person in charge of the process and as possessor of the specialized knowledge of methods, the developer's dependence on the user is minimized. In traditional life-cycle development, users are also dependent on developers for maintaining applications that the users need.

The four roles are distributed among different actors with very little overlap, thus creating a high degree of specialization and need for coordination. The managerial role affords the greatest overlapping, being shared by both IS managers and user managers, and this creates ambiguity within the traditional life-cycle paradigm.

Conflict and its management. Conflicts between any two roles in an organization are likely to occur when the roles are highly interdependent, where their responsibilities and methods differ, and where they must accomplish work jointly under resource constraints and time pressure [2,39]. These conditions clearly describe the traditional relationship between users and IS professionals, and the traditional life-cycle paradigm has persevered because it offers hope for controlling potential conflicts. With its highly structured and sequential discipline, the life cycle attempts to regulate conflict by imposing rules on the process, which theoretically reduces the potential for conflict because actors work independently of each other.

Unfortunately, this kind of regulation only works where there are few deviations from planned activities. In contrast to the implied assumption of few deviations, field research on system development provides evidence that deviations are frequent, and that project performance is often frustrated by dependence on a sequential plan [44]. When deviations do occur, the traditional life-cycle paradigm proves to be remarkably inflexi-

ble. Where conformity to traditional development methods is strictly enforced, conflicts may simply be deferred until after implementation and then be manifested as resistance to and/or unintended uses of the system [28,45].

4.2. Iterative-incremental paradigm

The second paradigm for system development is the iterative-incremental paradigm. Methodologies within this paradigm are designed on the assumption that it is possible to identify and meet system needs more accurately by continuously revisiting requirements with users. Thus, system development is designed as an iterative process, repeating various activities until design specifications are better understood and more fully developed. Methodologies within the iterative-incremental paradigm are centered neither the developer's nor user's camp. While developers may direct some activities, developers and users assume joint responsibility for producing results.

Typical methodologies. Many iterative-incremental methodologies fit within an object-oriented (OO) paradigm that focuses on software reuse, even though some new development efforts are undertaken using the iterative-incremental approach [20,35]. Perhaps the most general methodology is prototyping, which involves the construction of a model of a system or object that contains only essential features [8,40]. Prototypes do not include all features of a system or an object, and they do not provide the level of accuracy or performance that is expected with traditional life-cycle methodologies. Prototypes are generally created quickly, and they provide timely feedback on the feasibility and usefulness of an application's design and specifications.

The spiral model [9] defines four major activities that are performed repeatedly throughout the IS development process: planning, risk assessment, engineering, and evaluation. Because not all of these activities are directly related to system development, the spiral model is frequently considered to be a more general iterative process for managing IS projects. However, because iterations recur within each main activity of the spiral model, technical system development may be regarded as a recurring engineering activity within the overall systems development process.

Object-oriented methodologies have come into vogue because they provide a valuable perspective on how the data and functions of a program fit together and how hierarchies of concepts may be used to model the real world more efficiently. OO methodologies generally fit within the iterative-incremental paradigm because they take advantage of prototyping in the development process [11,12,46]. The OO approach focuses on defining and categorizing real-world abstractions and modifying their computer representations until they are detailed enough for implementation.

Roles and interdependencies. As shown in table 2, the developer role is expanded in the iterative-incremental paradigm to include both traditional developers and users. Developer and user are the two central roles in the iterative-incremental paradigm, but users perform both of them. As in the traditional approach, users provide information about needs, and developers perform actual design and development work. However,

Table 2
Actors and roles in the iterative-incremental paradigm.

Actors	Roles			
	Developer	User	Manager	Guarantor
Analysts	X			
Designers	X			
Programmers	X			
Consultant; integrator	X			X
End user	X	X		
User manager	X	X	X	
IS manager			X	X

in the iterative-incremental methodolgies, actors work more closely to identify domain abstractions and to create analysis documents that describe the functional characteristics of the domain. This process is accomplished iteratively, with higher-level abstractions evolving over time to become more detailed descriptions. While either developers or users may be nominally in charge of the process, power is more balanced than in the traditional life-cycle paradigm.

The interdependence between users and developers in the iterative approach is reciprocal rather than sequential. Work and communication move back and forth as participants rely upon each other to provide information and knowledge contributing to the shared goal of understanding the domain and the system. Reciprocal interdependence allows each party's contribution to be on display as it is created. By approaching the system development process with an assumption of mutual cooperation among equally important roles, the iterative-incremental development process hopes to build systems more effectively [10].

Conflict and its management. Unlike the traditional approach of reducing conflict through carefully sequenced activities, the iterative paradigm actually increases the potential for conflict by bringing actors closer together. With greater requirements to work together, and with more equal power, actors may indeed create more conflict. However, the hope is that such conflicts can be constructively managed by providing greater opportunity for open communication and mutual influence [41,42]. Project management and leadership are key to the success of such development efforts, so many iterative processes are led by facilitators skilled in both conflict management and in the methods of system development [30,53].

4.3. Component-based development (CBD) paradigm

The CBD paradigm depends upon the availability of a wide variety of reliable utilities and business-application components, which must be easily connected to create and configure business applications, then recreated and reconfigured as business requirements change. Unlike objects, components are platform dependent, and thus concrete enough to avoid the risks and problems of instantiating general objects on a particular

machine and within a specific application at the language level. Components supporting user interface design, data storage and retrieval, and data communications are examples of general utilities. Business-application components also include support for inventory tracking, point-of-sale processing, and data analysis. Most component-based development relies upon the use and reuse of objects available from independent component suppliers.

Component-based development has become a very important topic in recent years [29,38,52,54]. Evidence of this includes special issues of established journals (e.g., [26]), the first component-base software conference in 1996 [27], many organizations instituting component technology development [1], and research groups focusing on component development (e.g., the Department of Management Science and Statistics at the University of Alabama, Tuscaloosa). The ready availability of commercial component-based infrastructures (e.g., DCOM and CORBA) and plug-ins for software such as Adobe Acrobat, Visual BASIC, and Netscape, have made component based development become a reality.

Unlike the first two paradigms, which separate the developer and user roles, the CBD paradigm typically has both roles played by users. Given the high-level functioning of commercially available components, users can develop and maintain systems more quickly and more effectively than could full-time system developers working through a backlog of projects. The primary advantage of the CBD paradigm is that users may combine their deep knowledge of an application domain with fairly limited technical knowledge, yet still produce useful systems. The user can configure new applications or modify existing ones when needed rather than waiting for a development team to design an enhancement to an existing application. However, the CBD paradigm requires that users interact directly with external providers of components and services, thereby entering into market-based transactions that are quite different from their relationships with in-house systems professionals.

Typical methodologies. Because the CBD paradigm has emerged as a user-led phenomenon, little attention has been given to the use of formal methodologies. Rather, development typically proceeds using visual programming and employs components with high functionality. The responsibility for creating systems thus shifts to the user, who must depend upon component suppliers and brokers. In turn, the suppliers and brokers must deliver compatibility and functionality to the user-developer so that components can be assembled into working applications. Suppliers must market components that provide inventory control, text editing, Internet connectivity, and many other functions, which can be plugged and played with other components acquired from disparate sources. Prior to the emergence of the CBD paradigm, one might have obtained, at best, so-called "structural support" components from external vendors (or an internal reuse library) for B-tree indexing, screen layout, graphic display, sorting, or other lower-level functions. Today, components have a very high degree of flexibility, generality, quality, and reliability.

Table 3
Actors and roles in the component-based development paradigm.

Actors	Roles			
	Developer	User	Manager	Guarantor
Component supplier	X			X
Component broker	X			X
Architect	X			X
End user	X	X		X
User manager	X	X	X	
Auditor				X

In the CBD paradigm, component suppliers are responsible only for pieces of the final application while user-developers retain responsibility for assembly. This arrangement poses potential difficulties in guaranteeing system performance and reliability. While in the other paradigms, the guarantor of the system was a single organization, in the CBD case the role of guarantor is diffused among the component suppliers and brokers. Ultimately, end-users are responsible for guaranteeing the fitness, capability, and appropriateness of their assembled applications.

Roles and interdependencies. As table 3 shows, the actors in the CBD paradigm differ dramatically from those described in the two preceding paradigms. In essence, the developer role is assumed by external agents: component suppliers, component brokers, and architects. The end-user and user-manager become central actors in development, essentially removing their traditional reliance upon IS professionals. Moreover, a new actor, the auditor, appears in table 3, sharing the role of guarantor with the other suppliers of components [54].

The radical shifts in roles and responsibilities within the CBD paradigm place tremendous pressure on the user. Users are not in a favorable position to compensate for components with marginal quality or questionable reliability. To obtain high-performing applications, it is essential that users acquire high-level resources from component suppliers. The role of guarantor becomes more important in the CBD paradigm than in the other two paradigms. Because the user is unlikely to be schooled in traditional analysis and design techniques, the guarantor must assure that developed systems are ready to use and can be maintained.

The interdependence among roles in the CBD paradigm is less well defined than in the traditional life-cycle and iterative-incremental paradigms. Instead of being guided by a formal methodology prescribed by professionally trained analysts, users engage in commercial transactions with independent suppliers of utilities and components. This dependency on the marketplace introduces an element of risk that the outside organization may discontinue providing or supporting compatible components that the end user needs. This risk is generally not present in the other two paradigms, where trained developers are available to modify software modules. The CBD paradigm, unfortunately, leaves the user in the position of "plug and pray" that everything will fit together and that future enhancements will be available on the market.

Although there is reciprocal interdependence in the CBD paradigm between users and suppliers, conventional opportunities for communication and conflict resolution are not present. Rather, this crucial relationship is governed by a contractual relationship in which the self-interests of each party assume primacy. Users do control the development of their own applications, and they may shop freely among suppliers to obtain the functionality needed at a competitive price. However, beyond the economic incentives regulated by the market and pricing mechanisms, no relationship is established between the parties. User-developers are motivated by organizational needs and demands, whereas suppliers are motivated by the desire for profits made from selling their components. Suppliers clearly benefit if they keep customers happy, so suppliers are likely to respond if a sufficient number of customers request the same features or updates. However, when customers require components that are not widely requested, suppliers are unlikely to provide them at a competitive price.

Conflict and its management. Although the historical conflicts between users and developers may be absent in the CBD paradigm, conflict and its management remain primary concerns. Although interdependence among parties internal to the organization may be reduced, external interdependence has increased. Moreover, the types of resources available to manage conflicts with internal parties in the traditional and iterative paradigms are unavailable in the CBD approach. The structured methodologies that are most characteristic of life-cycle methodologies are unavailable in component-based development, and the trained facilitators used in the iterative-incremental methodologies are not likely to be assigned to resolve conflicts between user-developers and suppliers. The mechanism of market price is unlikely to regulate the potential conflicts between user-developers and suppliers in the CBD paradigm. Thus, the relationship between users and suppliers poses considerable potential for conflict without much recourse to means for managing it.

Two solutions to these difficulties are likely to appear as the CBD paradigm becomes more firmly established. First, markets can be supplemented by closer, strategic alliances such as those found among outsourcing partners. IS managers have already learned that outsourcing contracts are no substitute for more secure alliances with external providers [22,48]. User-developers should seek closer relationships with component vendors so that the complex issues surrounding product performance, delivery, service, maintenance, and updates may be resolved constructively.

The second development likely to emerge in the CBD paradigm is the creation of intermediary roles. Two roles in particular should become prominent. First, given the lack of technical training for user-developers, it may be valuable to have system *architects* design the "big picture" within which users may develop their own applications [50,51]. An architect does not design system applications, but rather designs the overall information architecture for the organization and provides the framework and standards within which user-developers may create their own applications. This architectural framework helps to alleviate some of the risks of having untrained users perform systems development. Second, system *auditors* would be helpful to verify system qual-

ity and to certify the functionality and maintainability of user-developed systems [54]. System auditors may be deployed internally or hired on a contractual basis, much like the practice of hiring financial auditors.

5. Summary and conclusion

The social implications of three IS development paradigms have been discussed using a multidimensional framework. We distinguished among the traditional life-cycle, the iterative-incremental, and the component-based paradigms for system development. Each paradigm implies different actors for certain generic roles, and the interactions among these actors may also differ. It is important to identify the actors and their interdependencies because IS development is a social process, historically rife with conflicts among actors. Because it is vital for an organization to develop useful applications of information technology, it is also vital for conflicts to be managed and resolved, regardless of the methodology chosen for development work.

The primary implications of our analysis are summarized in table 4. Each paradigm is associated with the same generic roles, yet these roles are played by different actors who interact in different ways. The potential for conflict and the most likely means for managing conflict are also identified in table 4 for each paradigm.

The traditional life-cycle paradigm implies a carefully designed social process, but its reliance on a linear process may be ill suited for complex problem solving. The dominant position of professional developers and their regulated sequential interaction with users may suppress potential conflicts and leave end-users frustrated in their attempts to obtain systems that meet their needs. More effective mechanisms for conflict resolution attempt to surface latent conflicts and deal with them constructively [16,17].

The iterative-incremental paradigm emerged in response to the problems experienced in the traditional life-cycle methodologies. Iterative-incremental development brings the expertise of users and developers together, attempting to draw the best information and capabilities from each. It provides a process that is structured enough so that development may proceed in a managed way, while providing flexibility needed for revising documents, plans, and systems. And it attempts to make use of reusable components as much as possible, while not being so reliant on them that systems are determined solely by available pre-built components.

The component-based development paradigm gives the user the power to build complete systems from components available from external sources, thereby removing dependence upon in-house IS professionals. However, because end-users often lack formal IS training, they may become significantly dependent on external component suppliers and component brokers. While the CBD approach is attractive because of end-users' ability to develop and refine applications that directly benefit them, potential conflicts remain and need to be managed. Significantly, the CBD paradigm provides little guidance beyond the operation of market mechanisms for managing potential conflicts. The emergence of system architects and auditors as intermediaries may fill an important need in CBD.

Table 4
Summary of IS development paradigms.

Paradigm	Major roles and actors	Potential for conflict	Management of conflict
Traditional life-cycle	Developer Analyst Programmer User End user User manager Manager User manager IS manager Guarantor IS manager	Low due to concentration of power in developer role and sequential process.	Regulated by formal methods that keep roles separate and sequential. Conflicts not addressed directly and may remain unresolved.
Iterative-incremental	Developer Analyst Programmer Consultant Integrator End user User manager User End user User manager Manager User manager IS manager Guarantor Consultant Integrator IS manager	High due to equal power distribution and frequency of interaction between developers and users.	Regulated through direct confrontation in a team setting, facilitated by project leader. Conflicts addressed as part of the development process and are likely to be resolved.
Component-based	Developer Component supplier Component broker Architect End user User manager User End user User manager Manager User manager Guarantor Component supplier Component broker Architect Auditor	High due to uncertainty about component performance and the multiple performers of the developer and guarantor roles.	Regulated by market. Conflicts addressed through buyer-seller negotiation without recourse to cooperative problem-solving.

The emergence of the CBD paradigm presents a ripe opportunity for researchers to investigate its social consequences. Far more research has been compiled about the management of conflict during IS development under the traditional life-cycle and incremental-iterative paradigms, but these findings cannot be generalized to the CBD environment. Specifically, researchers could investigate the ways in which disagreements among component suppliers, brokers, architects, and users arise and examine the ways in which they may be successfully resolved. Recourse to the normal assurances of fair market practices may be insufficient to protect users who seek to develop high-performing applications on their own. We expect that intermediaries will fill important roles in the component-based development environment, but empirical studies need to illuminate the social dynamics and consequences of these roles.

For the practitioner, although the suggestions of this analysis surely need to be supported with empirical evidence, our analysis does suggest caution as one moves toward the CBD paradigm for developing new systems. Greatest ambiguity exists in the guarantor role; it is not obvious to anyone who will vouch for the performance of systems constructed from components purchased from different market vendors. While we await the emergence of intermediaries who will assume responsibility (and liability) for performance requirements, the user will have to bear the risk inherent in component-based development. Whether such risk is worth taking remains an empirical question that hopefully will be stimulated by this analysis.

References

[1] R. Applebaum and M. Guttman, Transitioning to component technology, Component Strategies 1(6) (1998) 36–44.
[2] W.G. Astley and P. Sachdeva, Structural sources of intraorganizational power: A theoretical synthesis, Academy of Management Review 9 (1984) 104–113.
[3] H. Barki and J. Hartwick, User participation, conflict, and conflict resolution: The mediating roles of influence, Information Systems Research 5(4) (1994) 422–438.
[4] S.R. Barley, Contextualizing conflict: Notes on the anthropology of disputes and negotiations, in: *Research on Negotiation in Organizations*, Vol. 3, eds. M.H. Bazerman, R.J. Lewicki and B.H. Sheppard (JAI Press, 1991) pp. 165–199.
[5] R.A. Baron, Negative effects of destructive criticism: Impact on conflict, self-efficacy, and task performance, in: *Managing Conflict: An Interdisciplinary Approach*, ed. M.A. Rahim (Praeger, 1989).
[6] C.M. Beath, Supporting the information technology champion, MIS Quarterly 15(3) (1991) 355–372.
[7] C.M. Beath and W.J. Orlikowski, The contradictory structure of systems development methodologies: Deconstructing the IS-user relationship in information engineering, Information Systems Research 5(4) (1994) 350–377.
[8] B. Boar, *Application Prototyping: A Requirements Definition Strategy for the 80s* (Wiley, 1984).
[9] B. Boehm, A spiral model of software development and enhancement, IEEE Computer 21(5) (1988) 61–72.
[10] R.J. Boland, Jr., The process and product of system design, Management Science 24(9) (1978) 887–898.
[11] G. Booch, *Object-Oriented Analysis and Design with Applications* (Benjamin/Cummings, 1994).
[12] G. Booch, J. Rumbaugh and I. Jacobson, *The Unified Modeling Language User Guide* (Addison-Wesley, 1999).

[13] R.P. Bostrom, Successful application of communication techniques to improve the systems development process, Information & Management 16(5) (1989) 279–295.
[14] C.W. Churchman, *The Design of Inquiring Systems: Basic Concepts of Systems and Organizations* (Basic Books, 1971).
[15] A.T. Cobb, An episodic model of power: Toward an integration of theory and research, Academy of Management Review 9 (1984) 482–493.
[16] R.A. Cosier and D.R. Dalton, Positive effects of conflict: A field assessment, International Journal of Conflict Management 1 (1990) 81–92.
[17] R.A. Cosier and C.R. Schwenk, Agreement and thinking alike: Ingredients for poor decisions, Academy of Management Executive 4(1) (1990) 69–74.
[18] B. DeBrabander and G. Thiers, Successful information system development in relation to situational factors which affect effective communication between MIS-users and EDP-specialists, Management Science 30(2) (1984) 137–155.
[19] T. DeMarco, *Structured Systems Analysis and System Specification* (Prentice-Hall, 1979).
[20] M.E. Fayad, W.-T. Tsai and M.L. Fulghum, Transition to object-oriented software development, Communications of the ACM 39(2) (1996) 108–121.
[21] C.R. Franz and D. Robey, An investigation of user-led system design: Rational and political perspectives, Communications of the ACM 27(12) (1984) 1202–1209.
[22] J.C. Henderson, Plugging into strategic partnerships: The critical IS connection, Sloan Management Review 31(3) (1991) 7–18.
[23] R. Hirschheim and H. Klein, Four paradigms of information systems development, Communications of the ACM 32(10) (1989) 1199–1216.
[24] R. Hirschheim, H. Klein and M. Newman, Information systems development as social action, Omega 19(6) (1991) 587–608.
[25] J.E. Hunton and J.D. Beeler, Effects of user participation in systems development: A longitudinal field experiment, MIS Quarterly 21(4) (1997) 359–388.
[26] IEEE Software 15(5) (1998), Special focus on component-based software engineering.
[27] T. Jell, *CUC96: Component Based Software Engineering* (SIGS Books, 1998).
[28] R. Kraut, S. Dumais and S. Koch, Computerization, productivity, and quality of work-life, Communications of the ACM 32(2) (1989) 220–238.
[29] M. Laitkorpi and A. Jaaksi, Extending the object-oriented software process with component oriented design, JOOP 12(1) (1999) 41–50, 67.
[30] R.J. Lewicki, S.E. Weiss and D. Lewin, Models of conflict, negotiation and third party intervention: A review and synthesis, Journal of Organizational Behavior 13(3) (1992) 209–252.
[31] J. Martin, *Information Engineering, Book I: Introduction* (Prentice-Hall, 1989).
[32] J. Martin, *Information Engineering, Book II: Planning and Analysis* (Prentice-Hall, 1990).
[33] J. Martin, *Information Engineering, Book III: Design and Construction*, (Prentice-Hall, 1990).
[34] R.O. Mason and I.I. Mitroff, A program for research on management information systems, Management Science 19(5) (1973) 475–487.
[35] J.D. McGregor and T.D. Korson, Integrated object-oriented testing and development processes, Communications of the ACM 37(9) (1994) 59–77.
[36] M. Newman and F. Noble, User involvement as an interaction process: A case study, Information Systems Research 1(1) (1990) 89–113.
[37] M. Newman and D. Robey, A social process model of user-analyst relationships, MIS Quarterly 16(2) (1992) 249–266.
[38] O. Nierstrasz, S. Gibbs and D. Tsichritzis, Component-oriented software development, Communications of the ACM 35(9) (1992) 160–165.
[39] L.R. Pondy, Organizational conflict: Concepts and models, Administrative Science Quarterly 12(2) (1967) 296–320.
[40] R. Pressman, *Software Engineering: A Practitioner's Approach* (McGraw-Hill, 1997).

[41] D. Robey and D.L. Farrow, User involvement in information systems development: A conflict model and empirical test, Management Science 28(1) (1982) 73–85.
[42] D. Robey, D.L. Farrow and C.R. Franz, Group process and conflict during system development, Management Science 35(10) (1989) 1172–1191.
[43] D. Robey, L.A. Smith and L.R. Vijayasarathy, Perceptions of conflict and success in information systems development projects, Journal of Management Information Systems 10(1) (1993) 123–139.
[44] D. Robey and M. Newman, Sequential patterns in information systems development: An application of a social process model, ACM Transactions on Information Systems 14(1) (1996) 30–63.
[45] D. Robey and M.-C. Boudreau, Accounting for the contradictory organizational consequences of information technology: Theoretical directions and methodological implications, Information Systems Research 10(2) (1999) 167–185.
[46] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy and W. Lorensen, *Object-Oriented Modeling and Design* (Prentice-Hall, 1991).
[47] R. Sabherwal and D. Robey, An empirical taxonomy of implementation processes based on sequences of events in information system development, Organizational Science 4(4) (1993) 548–576.
[48] R. Sabherwal and J.J. Elam, Overcoming the problems in information systems development by building and sustaining commitment, Accounting, Management & Information Technologies 5(3/4) (1996) 283–309.
[49] G. Salaway, An organizational learning approach to information systems development, MIS Quarterly 11(2) (1987) 244–264.
[50] N.F. Simenson, The architect: Roles and responsibilities, American Programmer 10(7) (1997) 14–17.
[51] D.F. Sittig, S. Sengupta, H. Al-Daig, T.H. Payne and P. Pincetl, The role of the information architect at king faisal specialist hospital and research center, in: *Proceedings of the 19th Annual Symposium on Computer Applications in Health Care* (1995) pp. 756–760.
[52] J. Voas, Maintaining component-based systems, IEEE Software 15(4) (1998) 22–27.
[53] D.B. Walz, J.J. Elam and B. Curtis, Inside a software design team: Knowledge acquisition, sharing, and integration, Communications of the ACM 36(10) (1993) 63–77.
[54] R. Welke, The shifting software development paradigm, Data Base 25(4) (1994) 9–16.
[55] E. Yourdon, *Modern Structured Analysis* (Prentice-Hall, 1989).