Silver Pellets for Improving Software Quality

Evan W. Duggan, University of Alabama, USA

ABSTRACT

In his timeless article, Fred Brooks asserted that the essential difficulties of developing software would continue to ensure the futility of any search for a "silver bullet" to reproduce (for software engineering) the catalytic effects that electronics, transistors, and large-scale integration had on computer hardware development. Since his article, software development has become even more difficult and organizations have magnified the struggle to overcome what has been called "the software crisis." There is unlikely to be a silver bullet, *but this article discusses a variety of user-centered and process-oriented systems delivery methods, philosophies, and techniques available to the software engineering community, that may be used in innovative permutations to tranquilize the dragon of poor software quality. The context for the applicability of these approaches and some advantages and weaknesses where indicated in the research literature or gleaned from practitioner accounts are also discussed.*

Keywords: *software engineering; software development; software quality*

INTRODUCTION

Despite several years of information technology (IT) innovations, the persistent theme is that "software development is in trouble."[1] Information systems providers have failed to exploit IT capability to establish systems that provide appreciable impact on their sponsoring organizations' value chain (Brynjolfssen, 1993; Gibbs, 1994). Consequently, organizations (and eventually the national economy) seldom derive commensurate benefits from IT investments. This, presumably, has contributed to the perception of a productivity paradox, which was first insinuated by economist Robert Solow.

Brooks (1987) has metaphorically linked the search for solutions to the software crisis to the search for a silver bullet to slay the legendary werewolf, and suggested two major categories of software development problems—essential and accidental difficulties. The former, which are inherent in the process of conceptualizing the several constructs that comprise this largely intangible product, have no known solution. The latter are controllable; they are merely incidental properties of the programmatic representation of these constructs. He concluded that no silver bullet existed to slay this werewolf of essential software difficulties, and none was foreseeable on the distant horizon. In an equally

conclusive article, Cox (1995) offered the apparent minority view that a silver bullet indeed existed for those who will summon the tenacity to shift from the software building paradigms that we have embraced toward engineering-based construction models.

In this article I describe several techniques that have been employed to address the quality concerns of information systems delivery. I recommend an attack on the problem from several sources simultaneously, with relevant combinations of philosophies, methods, and techniques—**silver pellets**—that may assist developers and users chip away at the vital areas of the monster's anatomy. No single silver bullet is presumed; only the disciplined, faithful application of relevant combinations of these approaches will eventually tranquilize the werewolf beyond its capacity to generate terror.

ANALYZING THE PROBLEM FURTHER

None of the four reasons—conformity, changeability, invisibility, and complexity—to which Brooks (1987) attributed the essential difficulties of software engineering, has disappeared. Software continues to "conform" to organizational politics and policy, human institutions, and other systems. Computer systems are still subject to many corrections, adaptations to business process changes, and extensions beyond their original scope. We are yet unable to create visual models to make intangible design products appreciable during construction; software quality is still only experienced through utilization.

In fact, the complexity of software systems development has increased (Al-Mushayt et al., 2001), primarily because of the increased reliance of businesses on IT to support corporate missions and priorities, either for strategic enablement or competitive necessity. This accentuates the challenges that face IS developers and user domain experts to interact more effectively (Vessey & Conger, 1994). Another reason for the increased complexity stems from organizational drives to apply more sophisticated technologies and establish flexible communications networks to accommodate a variety of data and processing distribution strategies, multimedia operations, and integrated systems. Yet another reason is the increased security concern that now attends the greater movement of data. The need for high quality software is glaring.

There is general agreement that the crucial parameters that set the tenor of expectations to determine what value software systems can eventually deliver are scope, quality, timing, and cost (Friedlander, 1992). These interdependent factors, which must be monitored and controlled during systems development, may be traded off against each other, but one cannot be adjusted independently without repercussive effects on the others. The tendency has been to use quality as a surrogate, aggregating construct to capture the value-adding attributes of these parameters.

From the combined definitions of several scholars (Erikkson & McFadden, 1993; Grady, 1993; Hanna, 1995; Hough, 1993), a high-quality information system (IS) is one that reliably produces required features (with a high probability of correct response) that are relatively easy to access and use. It should provide consistently good response times, and is delivered on time so that it retains business relevance beyond deployment. Problems that occur during field use are discoverable with normal effort, and can be easily isolated and corrected. It is scalable both with regard to functionality and usage, and the overall

benefits during its useful life outstrip life-cycle cost.

This set of quality characteristics concentrates on "development" attributes, however. Lyytinen (1988) extended IS quality requisites by reminding us that a system must also be utilized to qualify as successful; several technically sound systems have been abandoned post development for lack of use. Systems success is not always determined by technical validity, but also by affective and behavioral responses of the intended beneficiaries (Markus & Keil, 1994; Newman & Robey, 1992). A socio-technical focus is required to address both development and usage factors of systems success.

Several terms, including software engineering, software development, systems development, and systems delivery (the preferred term in this article) are used in the literature to describe the process of establishing and deploying information systems. Unfortunately none does justice to the intricacy of the socio-technical dynamics of the process. According to Robey et al. (2001), information systems are developed through social processes that require the interaction of several stakeholders and information technologies. The nature of the interaction influences outcomes. The actors are supported by, but also react to adopted paradigms, methodologies, methods, and tools (Table 1). Organizations have only flirted with the socio-technical implications of systems delivery. They have focused more on development-related success factors (e.g., accurate requirements, excellent design, programming standards, and delivery speed).

Instead of pragmatic responses to contemporary needs, more focused pre-project analyses of combinations of available socio-technical development practices and systems delivery methods are required. IS developers should emulate the business tactics adopted by the organizations they serve. Most are forced to pursue a multi-level attack on the werewolf of profitability by simultaneously addressing issues of technology adoption, product design, process management, and customer relationship management.

The overarching thesis of this article is that the confluence of a similar blend of IS delivery strategies can successfully assail the essential difficulties of systems development. These include socio-technical

Table 1: Classification of Systems Delivery Approaches

Concept	Description	Examples
Systems delivery paradigm	Commonly distinguishing features of a family of life cycle approaches	Waterfall SDLC model
Systems development methodology	Comprehensive set of principles for structuring the delivery process that describes what, when, and how activities are performed, and the supported methods and tools	Information Engineering; Method 1; Navigator
Systems delivery method	Set of principles and objectives for driving specific development technique	Rapid application development; extreme programming; component-based development
Systems delivery tool	Specific implementation support for a variety of methods	CASE; data flow diagrams; use case

best practices, process assessment and management to establish the highest capability and inject process management rigor, and process methods, which are classified and described within three systems development paradigms. The descriptions of these techniques are accompanied by considerations of the contexts for which particular approaches may have relevance, known caveats associated with their use, and claimed benefits. With the exception of those described under socio-technical systems, the plethora of tools (e.g., CASE, data flow diagrams, use cases, etc.) that support these approaches are beyond the scope of this paper.

SOCIO-TECHNICAL PHILOSOPHIES AND METHODS

Socio-technical systems (STS) thinking recognizes information systems as combinations of social and technical subsystems in which people organized into work units (teams or departments) interact with technology to satisfy some instrumental (work-related) objective. It acknowledges the social consequences of systems delivery and considers human, ethical, and organizational concerns jointly with technology decisions. The environment in which the socio-technical interactions occur, the people and the functions they perform, and the roles they play individually and in groups, are within its scope. *De facto* and *de jure* operating practices, rules and regulations that prescribe expected behavior, and sanctions for violations are also components of STS. In STS terms, the process of thinking about these issues is equated with the systems design process itself (Al-Mushayt et al., 2001).

Many information systems problems originate in the complexities of the social structures in which they are executed (Fox, 1995), and equivalently, the technology embedded in the social (sub) systems may produce human issues that contrive obstacles to system use (Shani & Sena, 1994). However, systems delivery efforts are focused on identifying functionality and the technical components that enable them and rarely, if at all, on social impact analysis. STS analysis resolves this sub-optimality by including both affective and instrumental issues while co-generating social and technical requirements. This practice allows the impact of different technical solutions to be assessed against potential social obstacles to system use and helps to pre-identify ethical issues in a particular technology implementation before critical incidents arise (Doherty & King, 1998).

While STS approaches identify and juxtapose social and technical requirements, other related user-centered themes (covered in the following sections) help to satisfy STS objectives, and enable effective user-developer interactions that can ameliorate essential difficulties of systems delivery. The philosophy of user-ownership (the ultimate form of user association with an IS project) and methods such as prototyping, joint application development, participatory design, and quality function deployment, which help to implement user-ownership, but also address other significant socio-technical goals, are elaborated.

User Ownership

The value of involving users in systems delivery has been well recognized (Barki & Hartwick, 1994). Potential homeowners do not disengage from the construction of their new building until occupation time; they remain with the process throughout. Similarly, potential systems owners should be closely associated with decisions affecting their business pro-

cesses. Users involved in systems building often endorse system goals, and this increases their perception of the usefulness of the system (Carmel et al., 1995). This positive identification in turn produces a high level of commitment to system outcomes (Borovits et al., 1990). It also creates a sense of ownership, the quintessence of user association with a systems project. User owners often become champions—systems "missionaries" who promote the system and encourage others to use it (Dodd & Carr, 1994; Hough, 1993).

Empirical research results have not consistently confirmed this theory of positive correlation between user involvement and system success (Carmel et al., 1995). Barki and Hartwick (1994) attributed these inconclusive findings to inadequate definition of user-association constructs. For example, they distinguished between user participation—the intellectual occupation with the accomplishment of assigned tasks and user involvement—the psychological state that attaches significance to the affiliation with a systems delivery project. These terms are often used interchangeably in the literature. User ownership goes beyond involvement and motivates pride of possession and a sense of responsibility for the success of the system.

Prototyping

The erection of physical structures (which is now supported by computer-based visualization techniques during conceptualization) takes increasing visible shape and form as the implementation progresses. Software development, however, remains a conceptual exercise. Its "goodness" or otherwise is typically experienced in use. "Look and feel," the term borrowed from software copyright battles, provides a qualitative measure of how closely software products resemble each other. Prototyping is software's approximation of visualization. It provides this look and feel dimension to simulate usage experience.

During prototyping, developers and users interact to build progressively refined models of the intended system that demonstrate its appearance and behavior. The initial model is seeded by partially defined requirements, and subsequent models are refined and expanded based on users' feedback from the trial of the latest model. The process progresses iteratively toward the discovery of the requirements (Boar, 1984; Davis, 1994; Dekleva, 1992; Weinberg, 1991). Prototyping may also be used to isolate and discover design errors, to obtain a more realistic sense of the proposed solution, and to uncover potential deployment problems. It has been used successfully for automobile and aircraft design, and in other areas of manufacturing, where it has been dubbed "rapid prototyping" (Adamopoulus et al., 1998).

The maturity of non-procedural languages and the increasing popularity of CASE tools have made this approach viable by providing the required programming productivity to produce succeeding versions expeditiously. Once the requirements or design alternatives have been decided, the prototype may either be discarded (a "throwaway" model) and development continued with other technologies, or continued toward full functional form ("evolutionary" prototyping) on the same design platform.

This technique is recommended to reduce the risk of failure of complex systems, where requirements cannot be rigorously pre-specified (Weinberg, 1991), either because of intricate business relationships, divergent stakeholder interests, or highly politicized environments. The close

user-designer working relationship forged during prototyping and its support for more effective communications about system features, moderate both quality-related and usage failure factors (Baskerville & Stage, 1996). The former occurs because of more accurate, user-centered requirements; the latter as a result of enhanced user involvement. Another benefit derives from the early assessment of the impact of the system on the user environment, which may also facilitate more effective change management (Adamopoulos et al., 1998).

A disadvantage of prototyping is that users sometimes overemphasize aesthetic requirements, such as format of screens and reports, while ignoring the reliability issues that bear more on system performance (Korac-Boisvert & Kouzmin, 1995). It can also lead to false expectations about the level of remaining effort to convert the prototype into the final product (Weinberg, 1991), and in some cases users have been known to press for the deployment of the unfinished product (the prototype). Alternatively, the opportunity for repeated revisions may induce scope creep (Wetherbe, 1991), and may become counterproductive because of over analysis.

Joint Application Development

JAD emphasizes the communication and relationship aspects of establishing systems requirements (Liou & Chen, 1993-94). IBM first introduced it in the 1970s as an alternative to interviewing individual users, which was the dominant technique for eliciting requirements. Instead, JAD is conducted in a series of structured, face-to-face meetings (workshops) where, under the guidance of a trained facilitator, system developers, users, and managers pool their knowledge and perspectives to decide appropriate information requirements. The formal JAD protocol consists of the five phases (Wood & Silver, 1995) shown in Table 2.

JAD has provided several advantages over the conventional techniques. The requirements generated within such a group are likely to be more accurate and consistent than those aggregated from several interviews. A typical JAD session runs for three to five days depending on the scope and complexity of the project. This contributes to a significant reduction (from months and years to weeks) in the time required for specifying requirements using conventional methods. The workshop also provides a vehicle for strengthening user involvement, which typically leads to greater commitment to the decisions (Andrews, 1991; Wood & Silver, 1995).

JAD seeks synergistic process gains but the freely interacting group technique it uses often causes the social and emotional dynamics of group relationships to

Table 2: JAD Phases

Phase	Activity
1. Project Definition	The facilitator works with system owners and sponsors to determine management perspectives and expectations regarding purpose, scope, and objectives.
2. Research	The facilitator explores the current business processes and related computer systems in place.
3. Preparation	Planning for the workshop, the logistics, and training of the scribe.
4. The Session	The main event in this process, where the facilitator guides the deliberations toward the accomplishment of its objectives.
5. Documentation	Preparation of the final documents containing the decisions and agreements.

negatively impact task accomplishment. Facilitation excellence is therefore a pivotal requirement for overcoming the problems that Kettelhut (1993) identified: destructive dominance; freeloading; groupthink (the fixation on preserving group harmony); risky-shift behavior—where groups shift from the risk propensities of individual members; and the Abilene paradox—conflict avoidance which produces group decisions that do not reflect the desires of its members. Several recommendations have been advanced for addressing these problems (Kettelhut, 1993; Wood & Silver, 1995).

Participatory Design

Participatory design (PD), the facilitated, group participation scheme that occupies the extreme eastern edge of the developer-led/user-led systems development continuum, originated in Scandinavia. It requires much stronger user/developer interactions than JAD. PD is rooted in the Scandinavian sociopolitical and cultural background, and provides a basis for operationalizing user ownership. It is based on the philosophy of workplace democratization by empowering workers to engage in organizational decision-making (including those concerned with technology adoption) to improve their working conditions (Carmel et al., 1993).

While JAD was instituted to exploit opportunities for user-developer teams (social entities) to experience synergy during technical specification and design, PD stresses technical knowledge sharing to enhance social interactions (Carmel et al., 1993). It promotes the cross-fertilization of ideas, knowledge, and skills necessary for mutual learning and empathy, so that users and developers can help each other understand their respective domains. PD forces systems developers to consider the sociology of the workplace to ensure mutual fit between technical and social systems (Kjaer & Madsen, 1995).

The popularly cited benefits of PD include more informed decision-making, broader commitment to the final system, and better systems fit to users' needs, which usually transcend the immediate impact of a newly developed system to more long-term morale benefits (Mankin & Cohen, 1997). The reduction in politically motivated abandonment after implementation is also an identified advantage. There are claims that the Scandinavian cultural predisposition is a necessary (though not sufficient) condition for PD success, and that it is not particularly useful outside of the specific climate fostered by their co-determination laws (Muller et al., 1993). Even if this is true, however, the increased focus on global systems development demands attention to the impact of cultural particularism on global teams.

Quality Function Deployment

Quality Function Deployment (QFD), which features extensively in the TQM literature, is primarily a customer-oriented approach. It has been applied successfully in both manufacturing and service organizations to facilitate enterprise-wide quality control. It was first introduced in manufacturing processes to capture, quantify and prioritize customer preferences and quality requirements, and monitor their delivery throughout product design and manufacturing. The deliberate focus on the customer typically leads to better products and services, and often provides process improvement opportunities.

QFD is supported by the house of quality matrix, through which product performance requirements are mapped to de-

sign decisions to prioritize technical characteristics. Several other QFD methods are also available for enmeshing and managing customer requirements throughout the production process. These include affinity diagrams that establish a hierarchy of association among expressed requirements, and hierarchy trees for assimilating the interrelationships. Relations diagrams help with problem analysis, discover priorities, and even unearth associated but unexpressed requirements. Process decision program diagrams are sometimes used to study potential failure factors of proposed design solutions, and the analytic hierarchy process (AHP) to incorporate qualitative factors in ranking the importance of both requirements and proposed product features.

Similar quality consciousness throughout the systems development process with deliberate focus on users' desires is crucial for providing useful and usable information systems. Most systems development methodologies do not require explicit articulation of user quality requirements, nor do they provide for systematic consideration of quality concerns progressively through development. QFD may be incorporated in systems development by deliberately capturing users' quality requirements, prioritizing and mapping them to measurable system features, and reviewing the results with customers (Erikkson & McFadden, 1993). Elboushi (1997) empirically demonstrated its effectiveness in producing efficient, robust, and consistent requirements for customers with diverse needs in an object-oriented application.

QFD, however, is time consuming and the process has to be accompanied by excellent group decision-making techniques to resolve conflicting requirements (Ho et al., 1999). Martins and Aspinwall (2001) identified several QFD implementation problems with team dynamics, cultural indisposition, and behavior management. These considerations may prejudice its wider appropriation in systems environments already encumbered with communication problems and pressured for rapid systems delivery. The recommendations for improving JAD may be equally applicable to QFD.

PROCESS ASSESSMENT AND MANAGEMENT

The extensive focus on IS process capability assessment provided by the capability maturity model (CMM) and other formal process management approaches are relatively recent systems development preoccupations to help address mounting quality and productivity concerns (*CIO Magazine*, April 2001). The introspection enabled by CMM-based assessment provides a baseline for improving process management capability. IT organizations have also begun to identify more with the importance of formal systems development methodologies in process management and rely less on individual and project group heroics.

Capability Maturity Model

CMM was introduced by the Software Engineering Institute (SEI) of Carnegie Mellon University and sponsored by the Department of Defense. It is an instrument for assessing the maturity and the degree of discipline of an organization's information systems development processes against a normative ideal. This assessment, which may be conducted by the organization or assisted by a third party, indicates five maturity levels.[2]

In April 2001 *CIO Magazine* conducted research on software process de-

velopment methods. Although the concentration on software process improvement was novel in many of the participating organizations, some were already claiming CMM assessment benefits such as shorter development cycle, reduced rework, higher development productivity, and higher quality systems. Other indirect benefits mentioned were more proactive IS groups; valuable, unsolicited end-user feedback about improvement opportunities; improvement in their capability to assess project risks; and cost-reduction opportunities enabled by more standardized software.

The indications of a CMM assessment, however, cannot be used to predict process outcomes—the quality of the system. It is process-oriented and makes no statement about the product. Good or poor systems will not automatically follow from the particular indication of an assessment. Similarly, it is difficult to use CMM results comparatively across organizations; it is an introspective tool to help them effect measures for moving their process capability to the next level, and eventually as far as they can go.

Systems Development Methodology

A systems development methodology (SDM) is a prerequisite for Level 3, CMM designation. It is the primary vehicle for structuring and managing the systems delivery process. It prescribes consistent, stable, repeatable protocols and procedures for each systems delivery project. The methodology details the activities, roles, and deliverables with their quality standards, and appropriate tools and techniques for each of its stages. Several commercial and proprietary systems development methodologies exist. Conventional wisdom is that a single organization adopts a single methodology on which all developers are trained. This allows for consistent process management across different projects, which eliminates project start-up inertia to decide the "how," and helps the project team focus on the situational characteristics of the current project.

Organizations that make effective use of methodologies may also be able to dynamically deploy project participants across multiple projects and often minimize the risk of project failures due to attrition. But methodologies may also be burdensome and some (particularly small) development efforts may well be frustrated by onerous process structures (Fitzgerald, 2000; Flatten, 1992). It is not unusual for a methodology to be altered either temporarily for a specific project, but fully sanctioned by the organization, or permanently to effect some desired improvement.

SYSTEMS DELIVERY METHODS

At least three systems delivery paradigms exist (adapted from Robey et al., 2001). There is the traditional systems development life cycle (SDLC) paradigm that is embodied in the waterfall model, which is characterized by the sequential execution of life cycle phases with the possibility of recycling to an earlier stage to resolve design issues. This method is well known, has longevity, and has been described ad infinitum; it will not be elaborated in this paper. Consequently, structured techniques that have been supporting pillars of SDLC will not be elaborated.

The iterative/incremental development paradigm describes a set of systems delivery methods that produce manageable pieces of the system iteratively and progressively toward the completion of the final product and/or delivers usable systems functionality incrementally. These methods, such as rapid application development,

cleanroom software engineering, and extreme programming, conform to Brooks' (1987) suggested approach of "growing" rather than building information systems. In some cases they substantiate a "less sooner" delivery strategy to secure early buy-in, and to varying degrees indulge user-centered philosophies.

The third paradigm includes development methods that focus extensively on software reuse and abstracting and managing units of software functionality (object-orientation and components-based development). Much imprecision exists in the notions of modules, classes, components, and objects; they are usually described in overlapping terms. Such fuzziness may indeed obscure the ability to distinguish nuances of value. But the clear implications of reuse are well understood. Third-generation programming languages reuse blocks of procedures repeatedly in a program, and subroutines provide commonly required functionality across several programs. Programming productivity was the primary target of these approaches, but object and component technologies have expanded this ambition to also attack systems building quality directly by abstracting away scale-related complexities (Szyperski, 1998).

Rapid Application Development

An information system, like money, has a time value. Sometimes the elapsed time between conception and deployment, during which the business processes could change, relegates even brilliantly conceived and technically sound systems to failure. Rapid Application Development (RAD) is based on the philosophy that the risk of system failure increases with elongated development. It enables faster delivery of functional parts of an application (Hough, 1993) by segmenting a large development into smaller, more manageable modules that are developed in parallel in an evolutionary process.

RAD embraces several user-centered design techniques: JAD (optional), structured analysis and design techniques for modeling and representing procedural logic and data relationships, and prototyping. These help to blend analysis, design, and construction during systems development life cycle and accelerate systems delivery. CASE tools and fourth-generation languages are typical, but optional.

In RAD, requirements are not completely pre-specified. Preliminary, high-level requirements provide the basis for project segmentation (Hough, 1993) for parallel development. RAD then applies structured techniques to construct preliminary system models, from which a prototype is built. The software module under construction progresses toward finalization through a series of user-developer interactions involving feedback on the prototype, which generates improvements to the model and then refinement of the prototype. The end result of this iterative process is a documented, deployment-ready module or system. If necessary, other system segments are created similarly in parallel development, until the entire system is built and implemented (Hanna, 1995; Hough, 1993).

RAD seems better suited to applications (with clearly defined user groups) that can be partitioned easily, and are not computationally complex, than for large infrastructure projects and distributed information systems that use corporate-wide databases (Beynon-Davies, 1998). Large projects are typically considered only when they can be segmented into smaller components that can be delivered independently (Hough, 1993). RAD focuses more on quickly producing systems with high busi-

ness value (to reduce the likelihood of project abandonment) than on engineering excellence. Its use requires a fairly sophisticated development environment and organizational familiarity with a variety of techniques.

RAD contributes to shorter development cycle times, which helps to reduce development costs and increases the odds of securing good system-business fit. By limiting the project's exposure to change, it helps to combat scope and feature creep, and produces user-involvement benefits similar to prototyping. RAD, however, forces several trade-offs for the compressed development cycles, which some view as sub-optimization on speed of development; quality may be sacrificed when users ignore rigor and robustness in favor of quick business value (Bourne, 1994). Project delays are avoided by reducing requirements rather than postponing and rescheduling deadlines (Redding, 1995), and rework is sometimes necessary in order to integrate independently developed modules (Hanna, 1995).

Cleanroom Software Engineering

The name, cleanroom software engineering (CSE) was borrowed from the semi-conductor industry where contaminant-free environments were established to prevent defects during the production of silicon chips. Its objective is to produce high-quality software with statistically certified reliability for failure-free performance in field use. A major focus is on preventing, as opposed to detecting, software defects. This incremental development practice is based on mathematical foundations of referential transparency and function theory. CSE deviates from the traditional repetitive sequence of code-test-correct-retest and uses more rigorous, controlled, engineering-based practices.

In this approach, small development teams generate software modules under statistical quality control with independent sub-teams for specification, development, and certification. For larger projects, several small teams may develop portions of the system concurrently. Peer reviews of incremental deliverables are conducted to increase the probability of generating error-free code. After the construction of a module, a subset of its possible uses is sampled and inferential statistical analysis used to determine whether that module has defects. The test statistic may be errors per thousand lines of code (E/KLOC) or mean time to failure (MTTF). If the pre-established quality standards are not attained, the module is discarded and reconstructed.

Several authors have cited successful applications where cleanroom techniques have improved software quality (Agrawal & Whittaker, 1993; Hausler et al., 1994; Head, 1994; Trammell et al., 1996) and significantly increased development productivity (Gibbs, 1994). Practitioners have also reported greatly reduced errors per line of cleanroom generated code (Jerva, 2001) of eight to ten times less than the average for other processes (Spangler, 1996). Indirect benefits that may accrue from the reduction of software failure in field-use include longer product life and the reduction of systems development resources devoted to maintenance activities. However, opponents of the cleanroom technique have focused on the method's philosophical aversion to traditional testing techniques (Beizer, 1997), its mathematical complexity, and the high demand for human resources (Jerva, 2001), which may account for the relative sparseness of cleanroom applications.

Extreme Programming

Extreme programming (XP) is a systems delivery technique that uses evolutionary deployment strategies to quickly produce small-scale systems. It is typically used by small, co-located development teams. Key features of this approach are its "process-light" methods, its use of two-person programming teams (dyads), and customer involvement throughout development. Systems planning, analysis, design, and construction are blended into a single phase of activity to speedily deliver small releases of system functionality.

The system is produced by managed interactions among system builders (Development) and business domain experts (called Business). Business and Development engage in a three-step "planning game" to produce requirements, and design and construct systems in a series of small releases. The first step is known as exploration, in which Business produces "story cards" of desired features, and Development estimates the implementation requirements. In the second step, called commitment, Business prioritizes the story cards into essential, not essential but value adding, and nice-to-have features and Development classifies them according to implementation risk. Based on the resulting matrix, Business selects the next feature(s) to be developed and deployed. In the third step (steering), Business and Development review progress and make adjustments as necessary.

The "planning game" is followed by an "iteration planning game" that is played by Development alone, with the same three cycles of exploration, commitment, and steering. In exploration, story cards are converted to task cards that contain implementation details. During commitment, responsibilities are distributed and the workload balanced among dyads. In steering the various dyads construct, test, integrate, and deploy new functionality. Typically, the iteration planning game occurs in the interval between steering cycles of the planning game.

There is skepticism about this approach among some systems development pundits who claim that some XP terminology are mere euphemisms for questionable systems practices—for example "refactoring," the XP concept of continuous design for "rework" (Glass, 2001). But proponents of the approach attest to its efficacy particularly to stabilize systems quality and develop applications rapidly in environments where business processes are volatile (Paulk, 2001). They also refute the claim that XP does not subscribe to modeling and documentation. While admitting that both are deliberately "light" (moderate) in XP, they say this is a deliberate strategy to disencumber the process (Paulk, 2001). XP supporters also concede its unsuitability for developing large multi-site (or global) applications with large development teams (Glass, 2001).

Object-Oriented Development

Object-orientated development (OOD) technology represents a paradigm shift from traditional systems-building approaches. Before object-oriented (OO) techniques gained acceptance, the separation (independence) of data and programs was axiomatic and almost sacrosanct. Data embedded in programs contribute to frequent program changes because of the volatility of data attributes. Object orientation is based on the opposite concept. It depicts a computer system as a collection of interacting objects—entities that an organization wishes to track and keep data about. Objects package both attributes (the

data elements) and the program procedures (also called behavior, methods, or operations) that are allowed to create, read, update, delete, and otherwise manipulate the attributes.

The key features of the OOD include:

- Encapsulation, the idea of making an object self-sufficient by abstracting and isolating its attributes and procedures, as in a "capsule," and hiding them from other objects. The encapsulated procedures are the only instructions that can act on data within an object; both are protected from outside interference. Objects need not keep track of the complications of (and the changes to) another object's internal structure; they interact by exchanging messages that request information or the activation of particular procedures.
- Inheritance is accommodated by the fact that objects are modeled as classes (a generic grouping) or instances (specific concrete invocation of an example in application, i.e., with attribute values). Inheritance is enabled by defining special cases, subclasses or subassemblies, which share the common attributes, procedures, and relationships of a more general class (superclass), but may have their own peculiar attributes and procedures. Existing objects may also be augmented with additional procedures to create new classes with enhanced functionality.
- Polymorphism exploits this hierarchy of object classification to apply the same named procedure in different ways depending on the characteristics of the object class in which it is invoked.

The close affinity of notional objects with physical objects helps to reduce conceptual complexity during analysis. The major benefit of objects, however, is their reusability (with or without modification) in other applications (Schmidt & Fayad, 1997; Vessey & Conger, 1994). The focus on reuse may also enhance reliability resulting from more disciplined object design (Johnson et al., 1999). Organizations that adopt the OO philosophy pursue this reuse benefit to establish more flexible systems that are easier to maintain (Agarwal et al., 2000) and thereby decrease information systems costs.

OO techniques have been available since the 1960s (with a simulation language called Simula67) but were not widely used until the 1990s. Several reasons account for the slow pace of acceptance: the steep learning curve for developers trained under other paradigms; the initial inability to integrate object-based modules with legacy systems (Bordoloi & Lee, 1994; Johnson et al., 1999), now partially addressed by object wrappers; and the unavailability of analytical models and process methods (Fichman & Kemerer, 1992). The latter has been remedied with the adoption of the unified modeling language (UML) as the de facto modeling and notations standard. UML and common object request broker architecture (CORBA), a protocol that facilitates object communication on a network, have been adopted as standards by the Object Management Group. Several OO process methodologies, including rational objectory process (ROP) and rational unified process (RUP), have also emerged.

Component-Based Development

Component-based development (CBD) refers to the erection of information systems by assembling prefabricated, platform-neutral "packages" of software functionality called components. These components, whose granularity may vary

considerably, include self-contained objects, subroutines, or entire applications that deliver cohesive software services that are reusable across multiple applications. Independently provisioned software components are acquired and interfaced to construct a functioning system (Szyperski, 1998).

CBD involves the three basic steps of provisioning, inventory, and assembly. Technologists use software-engineering disciplines to architect reusable components (provisioning) in compliance with standards. Components provide presentation and interface functionality, server-side data manipulation, and application logic services. There are infrastructural components for providing commonly used and generic software services, domain components with specific and proprietary business application logic, and wrapped components to facilitate the interfacing of pre-existing functionality provided in legacy systems. Components are provisioned independently of immediate need and the services they provide registered with component inventories that support application assembly.

Reuse is a large benefit of CBD. It contributes to delivery speed and eliminates repetitive development steps to produce infrastructure that all applications need. Reuse also reduces the delivered cost of software, which is amortizable over multiple applications (Szyperski, 1998). CBD assemblers need not be expert systems builders; systems may be assembled by solution providers who are knowledgeable professionals in the problem domain (Robey et al., 2001). Components are also replaceable, which enables evolutionary system upgrade by changing specific, outdated, and/or inadequate pieces in the link instead of the entire assembly (Szyperski, 1998). Some claim that the reuse benefit is often overplayed, and that interface problems continue to plague "plug and play" capabilities especially in the absence of agreed interoperability standards (Sparling, 2000). Perhaps, the most immediate restriction, though, on the wider applicability of CBD is the lack of a critical mass of inventoried components.

SUMMARY AND CONCLUSION

Table 3 summarizes the salient features of the systems delivery approaches discussed. It provides synoptic statements about the primary objective, the condition most favoring the applicability of, and any major caveat associated with each approach.

Systems developers, it seems, possess both the desire and the aptitude, but fail persistently, to establish successful systems. The systems delivery process is complex and there are no silver bullets. But perhaps the search for one has been the largest reason the problem endures. In this pursuit, each new technique, like the Pied Piper, attracts a following of dancers who sometimes abandon time-tested "waltzes." Instead of casting alternative methods as competitors, we might exploit synergistic combinations in flexible ways. For example, some XP practices may be gainfully incorporated into RAD, or cleanroom techniques in OOD. No method is inherently better than another.

Figure 1 illustrates that some development methods are associated with particular paradigms. However, process support techniques, socio-technical concepts, and a variety of systems development tools may be used by any method and under any paradigm. The dominant message of this paper is that a well-considered amalgam of socio-technical philosophies, systems delivery methods, and process techniques (selected appropriately to match the con-

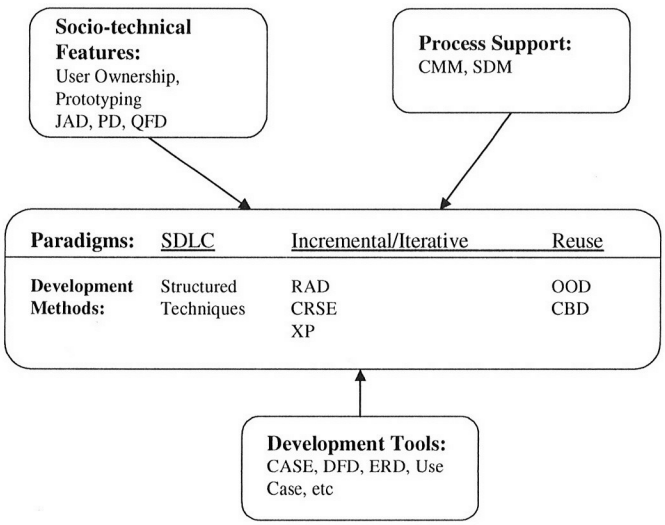
Table 3: Summary of Approaches

Approach	Objective	Applicability	Major Caveat
Socio-Technical Features			
User Ownership	User responsibility for systems success	Where operational feasibility is threatened	Depends on the sophistication of users and developers
Prototyping	Simulate usage experience to discover problems	Difficult to discover requirements	Requires powerful development environment
JAD	Improve requirements & design through enhanced communication	Pooling of knowledge, heterogeneous stakeholders	Needs excellent facilitator to combat relational problems
PD	Technical knowledge sharing to enhance social interactions	Worker participation in decision making	Needs cultural predisposition
QFD	Capture user quality requirements	Intense customer focus	Time consuming; requires group structuring mechanism
Process Management			
CMM	Organizational assessment of systems delivery process	Improved process quality	No immediate impact; cost
SDM	Consistent, stable, repeatable process management	Multi-project environment	May be cumbersome if followed religiously
Methods			
RAD	Fast delivery of functional application (modules)	Expeditious solution implementation	Focus on development speed instead of quality
CSE	Failure-free software in field use	High-risk projects with low fault tolerance	Complexity; human resource requirement
XP	"Process-light" user-developer interaction with co-located teams	Small teams, small systems, quick value-added	Development rigor
OOD	Systems building with flexible, reusable, interacting "objects"	Complex projects; reductionism	Understanding the OO philosophy
CBD	Assembling systems from fully functional platform-neutral pieces	User-developer environment	Critical mass of available components; lack of interface standards

text in which it will apply) is required to overwhelm the creative complexity of the software delivery process. This recommendation embraces an extension of the existing relationships illustrated in Figure 1 to include the possible combination of methods to effect a more useful project-context fit and establish comprehensive systems delivery process support within a variety of development environments.

Such a call for a "method mix" has fairly wide support among researchers (Broady et al., 1994; Fitzgerald, 1998; Hocking, 1998; Jaakkola & Drake, 1991; Korac-Bosvert & Kouzmin, 1995; Lee & Xia, 2002; Meso & Medo, 2000; Saberwal & Robey, 1995; Vessey & Glass, 1998; Vlasblom et al., 1995). Practitioners may not be as convinced, however. Many systems delivery methods have been overhyped by zealous originators and enthusiastic proponents who hyperbolize claims of efficacy in subjective allegiance to particular methods. To a large extent, practitioners are ambivalent about the legitimacy of these claims and in some cases even confused by inconsistent use of terms in the literature.

Figure 1: Relationship among the process techniques



I have, therefore, attempted a balanced and objective review of the literature in this area based on both empirical observations and seemingly dispassionate practitioner accounts. Three objectives guided this effort. The first was to explicate the nature, intentions, and the contextual applicability of these fairly popular, alternative practices and methods and to provide supported statements about their validity or otherwise. The second was to assist with the assimilation of literature in this area by clarifying some of the terminology. The third objective was to guide decision-making about the selection of context-relevant approaches (or combination of methods) to increase the probability of overcoming the pervasive problem of deploying low-quality information systems.

The techniques described in the preceding discussion are by no means exhaustive. Typically, they are the precursor (and probably the best known) representatives of a family of similar approaches. They are still widely used and have extensive "name recognition." Several derivatives of these "root" approaches like dynamic systems development method (DSDM), a RAD-based technique used extensively in the UK (Barrow, 2000; Beynon-Davies et al., 2000), have become very popular, with claimed improvements over the original methods. However, their similarity to the original approaches, and space and time preclude their elaboration in this paper. The techniques included are also quite interesting from the perspective that there is no convergence of opinions about their efficacy; claims of usefulness and disputed declarations of ineffectiveness coexist in the literature.

Some of these techniques are also fairly "old." They are explicated here (to accommodate the recommended method mix) as members of a possible candidate set of approaches with known strengths in particular contexts. If they are already well known by decision-makers, older methods may serve as a point of reference (a baseline) for the evaluation of other development options. However, the operational details of even long-standing approaches may not be sufficiently familiar to potential

adopters and practitioners to allow them to decide whether they fit a particular development setting.

Research findings have supported the general proposition that high-quality, mature systems delivery processes and methods produce high-quality systems (Harter et al., 1998, Kalifa & Verner, 2000; Ravichandram & Rai, 1996). Further research is needed to more effectively describe the scope of the applicability of these approaches, to legitimize the recommendations for a combination of systems delivery methods, and to empirically validate appropriate mappings of individual and combined methods and techniques to particular development settings. This could eventually provide valuable insights into contingency considerations for the adoption of effective combinations of methods and approaches, and suitable criteria for grouping methods.

ENDNOTES

1. Examples of reports of the software crisis:

- Mousinho (1990) reported on runaway projects (two, three, or even four times over budget, at least 50% behind schedule, or both)
- Niederman et al. (1991) identified the improvement of software development quality among the top ten issues of information systems management.
- KPMG-Peat Marwick (*Computerworld*, April 25, 1994) reported the startling finding that 95% of all major computer projects slide into cost and time overruns and that 65% of those become runaway projects.
- Charles B. Kreitzberg, President of Cognetics Corp., a software consulting firm in Princeton Junction, New Jersey, claims that CIOs estimate the failure rate for software projects at 60%; 25% are abandoned.
- In 1998, the Standish Group International Inc. indicated that, although the picture is improving, only 24% of IS projects in Fortune 500 companies could be considered successful (up from 9% in 1994).
- According to a June 2001 survey of MIS executives by *CIO Magazine* approximately 50% expressed dissatisfaction with the quality of their organization's business software.

2. CMM Levels:

- Level 1, the "initial" stage, is characterized by the absence of and formal procedures—a state of near chaos, where proper project planning and control are non-existent. Organizations may experience successful implementations, but these depend on heroic individual effort.
- Level 2 is called the "repeatable" stage, where basic project management capability exists. The organization can establish and track schedules and cost, and monitor and report on progress. But process management discipline does not exist and individual project leaders supply their own process management techniques. Project success depends greatly on the skill of the project group.
- An organization at Level 3, the "defined" stage, uses a common, institutionalized process management method (systems development methodology) for all its IS projects. The process management discipline produces consistent, stable, and predictable outcomes. This allows dynamic redeployment of human resources and reduces attrition-related failure risks.
- At Level 4, the "managed" stage, organizational learning is a key objective. Establishing a common methodology is not viewed as the end-all of process man-

agement. The organization collects detailed measures of process and product quality, which is used to refine the development process.

- An organization at Level 5 has "optimized" its process management capability. It uses the metrics from Level 4, acknowledged best practices, and benchmarks, and the results of controlled experiments and pilot studies to adjust the process to achieve continuous process improvement.

REFERENCES

Adamopoulos, D.X., Haramis, G., & Papandreou, C.A. (1998). Rapid prototyping of new telecommunications services: A procedural approach. *Computer Communications*, 21, 211-219.

Al-Mushayt, O., Doherty, N., & King, M. (2001). An investigation into the relative success of alternative approaches to the treatment of organizational issues in systems development projects. *Organization Development Journal*, 19(1), 31-48.

Andrews, D. C. (1991). JAD: A crucial dimension for rapid applications development. *Journal of Systems Management*, 42(3), 23-27,31.

Agrawal, K. & Whittaker, J. A. (1993). Experiences in applying statistical testing to a real-time, embedded software system. *Proceedings of the Pacific Northwest Software Quality Conference.*

Agarwal, R., De, P., Sinha, A. P., & Tanniru, M. (2000). On the usability of OO representations. *Communications of the ACM*, 43(10), 83-89.

Barki, H., & Hartwick, J. (1994). Measuring user participation, user involvement, and user attitude. *MIS Quarterly*, 18(1), 59-82.

Barrow, P. D. & Mayhew P. J. (2000). Investigating principles of stakeholder evaluation in a modern IS development approach. *Journal of Systems and Software*, 52(2-3), 95-103.

Baskerville, R. L. & Stage, J. (1996). Controlling prototype development through risk analysis. *MIS Quarterly*, 20(4), 481-504.

Beizer, B. (1997). Cleanroom process model: A critical examination. *IEEE Software*, 14(2), 14-16.

Beynon-Davies, P., Mackay, H., & Tudhope, D. (2000). Its lots of bits of paper and ticks and post-it notes and things: A case study of a rapid application development project. *Information Systems Journal*, 10(3), 195-216.

Beynon-Davies, P., Tudhope, D., & Mackay, H. (1999). Information systems prototyping in practice. *Journal of Information Technology*, 14(1), 107-120.

Boar, B. H. (1984). *Application Prototyping*. New York: John Wiley & Sons.

Bordoloi, B. & Lee, M. (1994). An object-oriented view: Productivity comparison with structured development. *Information Systems Management*, 11(1), 22-30.

Borovits, I., Ellis, S., & Yeheskel, O. (1990). Group processes and the development of information systems. *Information & Management*, 19, 65-72.

Bourne, K. C. (1994). Putting the rigor back in RAD. *Database Programming & Design*, 7(8), 25-30.

Broady, J. E., Walters, S. A., & Hartley, R. J. (1994). A review of information systems development methodologies (ISDMS). *Library Management*, 15(6), 5-19.

Brooks, F.P. Jr. (1987). No silver bullet: Essence and accidents of software engineering. *IEEE Computer*, 20(4), 10-19.

Brynjolfssen, E. (1993). The productivity paradox of information technology. *Communications of the ACM*, 36(12), 67-

77.

Carmel, E., Whitaker, R. D., & George, J. F. (1993). PD and joint application design: A transatlantic comparison. *Communications of the ACM*, 36(6), 40-48.

Carmel, E., George, J.F., & Nunamaker, J.F. (1995). Examining the process of electronic-JAD, *Journal of End User Computing*, 7(1), 13-22.

Cox, B.J. (1995). No silver bullet revisited. *American Programmer Journal,* (November).

Dodd, J. L. & Carr, H. H. (1994). Systems development led by end-users. *Journal of Systems Management*, 45(8), 34-40.

Doherty, N. F. & King, M. (1998). The consideration of organizational issues during the systems development process: An empirical analysis. *Behavior & Information Technology*, 17(1), 41-51.

Dekleva, S.M. (1992). The influence of the information systems development approach on maintenance. *MIS Quarterly*, 16(3), 355-372.

Elboushi, M.I. (1997). Object-oriented software design utilizing quality function deployment. *The Journal of Systems and Software*, 38(2), 133-143.

Erikkson, I. & McFadden, F. (1993). Quality function deployment: A tool to improve software quality. *Information & Software Technology*, 35(9), 491-498.

Fichman, R. G. & Kemerer, C. F. (1992). Object-oriented and conventional analysis and design methodologies. *IEEE Computer*, 25(10), 22-39.

Fitzgerald, B. (1998). An empirical investigation into the adoption of systems development methodologies. *Information and Management*, 34(6), 317-328.

Fitzgerald, B. (2000). Systems development methodologies: The problem of tenses. *Information Technology & People*, 13(3), 174-182.

Flatten, P. O. (1992). Requirements for a life-cycle methodology for the 1990s. W. W. Cotterman & J. A. Senn (Eds.), *Challenges and Strategies for Research in Systems Development* (pp.221-234). New York: John Wiley & Sons.

Friedlander P. (1992). Ensuring software project success with project buyers. *Software Engineering Tools, Techniques, and Practices*, 2(6), 26-29.

Fox, W. M. (1995). Sociotechnical system principles and guidelines: Past and present. *The Journal of Applied Behavioral Science,* 31(1), 91-105.

Gibbs, W. W. (1994). Software's chronic crisis. *Scientific American*, 271(3), 86-95.

Glass, R. L. (2001). Extreme programming: The good, the bad, and the bottom line. *IEEE Software*, 8(6), 111-112.

Grady, R. B. (1993). Practical results from measuring software quality. *Communications of the ACM*, 36(11), 62-68.

Hanna, M. (1995). Farewell to waterfalls? *Software Magazine*, 15(5), 38-46.

Hausler, P.A., Linger, R.C., & Trammell. C. J. (1994). Adopting cleanroom software engineering with a phased approach. *IBM Systems Journal*, 33(1), 89-109.

Harter, D. E., Slaughter, S. A., & Krishnan, M. S. (1998). The life cycle effects of software quality: A longitudinal analysis. *International Conference on Information Systems*, 346-351.

Head, G. E. (1994). Six-Sigma software using cleanroom software engineering techniques. *Hewlett-Packard Journal*, 45(3), 40-50.

Ho, E. S., Lai, Y. J., & Chang, S. I. (1999). An integrated group decision-making approach to quality function deployment. *IIE Transactions*, 31(6), 553-567.

Hocking, L. J. (1998). Developing a framework for examining systems development and its environmental context: The relationship between Gidden's structuration theory and Pettigrew's contextualist analysis. *Proceedings of the 4th Association for Information Systems Americas Conference*, 835-837.

Hough, D. (1993). Rapid delivery: An evolutionary approach for application development. *IBM Systems Journal*, 32(3), 397-419.

Jaakkola, J.E., & Drake, K.B. (1991). ASDM: The universal systems development methodology. *Journal of Systems Management*, 42(2), 6-11.

Jerva, M. (2001). Systems analysis and design methodologies: Practicalities and use in today's information systems development efforts. *Topics in Health Information Management*, 21(4), 13-20.

Johnson, R. A., Hardgrove, B. C., & Doke, E. R. (1999). An industry analysis of developer beliefs about object-oriented systems development. *The DATA BASE for Advances in Information Systems*, 30(1), 47-64.

Kettelhut, M. C. (1993). JAD methodology and group dynamics. *Information Systems Management*, 14(3), 46-53.

Khaliffa, M. & Verner, J. M. (2000). Drivers for software development method usage. *IEEE Transactions on Engineering Management*, 47(3), 360-369.

Kjaer, A. & Madsen, K. H. (1995). Participatory analysis of flexibility. *Communications of the ACM*, 38(5), 53-60.

Korac-Boisvert, N. & Kouzmin, A. K.N. (1995). It development: Methodology overload or crisis. *Science Communication*, 17, 81-88.

Lee, G. & Xia, W. (2002). Flexibility of information systems development projects: A conceptual framework. *Proceedings of the Eighth Americas Conference on Information Systems* (pp. 1390-1396).

Liou, Y. I. & Chen, M. (1993/4). Using group support systems in joint application development for requirements specifications. *Journal of Management Information Systems*, 8(10), 805-815.

Lyytinen, K. (1988). Expectation failure concept and systems analysts' view of information system failures: Results of an exploratory study. *Information & Management*, 14(1), 45-56.

Mankin, D. & Cohen S.G. (1997). Teams and technology: Tensions in participatory design. *Organizational Dynamics*, 26(1), 63-76.

Markus, M.L. & Keil, M. (1994). If we build it they will come: Designing information systems that users want to use. *Sloan Management Review*, 35(4), 11-25.

Martins, A. & Aspinwall, E.M. (2001). Quality function deployment: An empirical study in the UK. *Total Quality Management*, 12(5), 575-588.

Meso, P. & Madey, G.R. (2000). A complexity-based taxonomy of systems development methodologies. *Association for Information Systems 2000 Americas Conference* (pp. 238-244).

Mousinho, G. (1990). Project management: Runaway! *Systems International*, 6, 35-40.

Muller, M.J., Wildman, D.M., & White, E.A. (1993). Taxonomy of PD practices: A brief practitioner's guide. *Communications of the ACM*, 36(4), 26-27.

Newman, M. & Robey, D. (1992). A social process model of user-analyst relationships. *MIS Quarterly*, 16(2), 249-266.

Niederman, F., Brancheau, J. C., & Wetherbe, J.C. (1991). Information systems management issues for the 1990s. *MIS Quarterly*, 15(4), 474-500.

Paulk, M. (2001). Extreme programming from a CMM perspective. *IEEE Soft-*

ware, 8(6), 19-26.

Ravichandran, T. & Rai, A. (1996). Impact of process management on systems development quality: An empirical study. *Proceedings of the Americas Conference on Information Systems*.

Robey, D., Welke, R., & Turk, D. (2001). Traditional, iterative, and component-based development: A social analysis of software development paradigms. *Information Technology and Management*, 2(1), 53-70.

Sabherwal, R. & Robey D. (1995). An empirical taxonomy of implementation processes based on sequences of events in information systems development. In G. P. Huber & A. H. Van de Ven (Eds.), *Longitudinal Field Research Methods: Studying Processes of Organizational Change* (pp. 228-266). London: Sage.

Schmidt, D.C. & Fayad, M.E. (1997) Lessons learned building reusable OO frameworks for distributed software, *Communications of the ACM*, 40(10), 85-87.

Shani, A.B. & Sena, J.A. (1994) Information technology and the integration of change: Sociotechnical system approach. *The Journal of Applied Behavioral Science*, 30(2), 247-270.

Spangler, A. (1996). Cleanroom software engineering: Plan your work and work your plan in small increments. *IEEE Potentials*, (October/November), 29-32.

Sparling, M. (2000). Lessons learned through six years of component-based development. *Communications of the ACM*, 43(10), 47-53.

Szyperski, C. (1998). *Component Software, Beyond Object-Oriented Programming*. New York: Addison-Wesley.

Trammell, C. J., Pleszkoch, M.G., Linger, R.C. & Hevner A. R. (1996). The incremental development process in cleanroom software engineering. *Decision Support Systems*, 17(1), 55-71.

Vessey, I. & Conger S.A. (1998). Strong vs. weak approaches to systems development. *Communications of the ACM*, 37(5), 102-112.

Vessey, I. & Glass R. (1994). Requirements specification: Learning object, process, and data methodologies. *Communications of the ACM*, 41(4), 99-102.

Vlasblom, G., Rijsenbrij, D., & Glastra, M. (1995). Flexibilization of the methodology of system development. *Information and Software Technology, 37*(11), *595-607.*

Weinberg, R. S. (1991). Prototyping and the systems development life cycle. *Information Systems Management*, 8(2), 47-53.

Welke, R. J. (1994). The shifting software development paradigm. *Database*, 25(4), 9-16.

Wetherbe, J. C. (1991). Executive information requirements: Getting it right. *MIS Quarterly*, 15(1), 51-65.

Wood, J. & Silver, D. (1995). *Joint Application Development*. New York: John Wiley & Sons.

Evan W. Duggan is Assistant Professor of MIS in the Department of Information Systems, Statistics, and Management Science at the University of Alabama. He holds MBA and PhD degrees from Georgia State University and a BSc from the University of the West Indies. He has over 25 years of industry experience in information systems delivery up to the position of Information Services Director. Dr. Duggan has a special interest in the implementation and management of information systems in corporations, and his research involves human and methodological contributions to information systems quality, and systems development and deployment success factors. He has taught a variety of information systems and decision sciences courses in graduate and undergraduate programs in major U.S. and international institutions.