

AGILE SOFTWARE DEVELOPMENT: ADAPTIVE SYSTEMS PRINCIPLES AND BEST PRACTICES

PETER MESO, assistant professor of information systems at Georgia State University, earned his Ph.D. degree in information systems from Kent State University. His research dealing with requirements and software engineering, the processes of system development, and information systems in underdeveloped nations appears in several leading journals.

RADHIKA JAIN is finishing up her doctoral work in information systems at Georgia State University. In August 2006, she begins her appointment as an assistant professor at the University of Memphis. Her research interests include business process management, healthcare systems, software development, and ubiquitous computing. Her work appears in several leading journals and conferences.

Peter Meso and Radhika Jain

Today's environments of increasing business change require software development methodologies that are more adaptable. This article examines how complex adaptive systems (CAS) theory can be used to increase our understanding of how agile software development practices can be used to develop this capability. A mapping of agile practices to CAS principles and three dimensions (product, process, and people) results in several recommendations for "best practices" in systems development.

TODAY'S SOFTWARE DEVELOPMENT units face old and new risks, including shortfalls in real-time performance, externally performed tasks, externally furnished components, unrealistic schedules and budgets, development of wrong or unwanted information systems (IS) solutions, obsolete features, and changes in requirements from multiple sources [including customers, technology, social factors, overhead, and competition (Conboy, Fitzgerald, & Golden, 2005)] that need to be implemented at lightning speed (Kwak & Stoddard, 2004). Thus, organizations put a premium on software development approaches that increase their responsiveness to business change.

Agile software development methodologies provide organizations with an ability to rapidly evolve IS solutions (Highsmith, 1999; Sutherland & van den Heuvel, 2002). To date, these methodologies have been employed primarily in Internet and Web software development contexts (Glass, 2003; Paulk, 2001). However, the growing emphasis on these methodologies' capabilities to respond to rapidly changing

requirements from various sources has led to a growing interest in their application in developing large mission-critical software solutions (Baskerville, Ramesh, Levine, Pries-Heje, & Slaughter, 2003).

In this article, we argue that complex adaptive systems (CAS) theory provides us with a theoretical lens for understanding how agile or "Internet-speed" development of IS solutions can be used to advantage in volatile business environments. Specifically, we apply several principles of CAS theory (Anderson, 1999; Highsmith, 1999; Kauffman, 1991) to develop a better understanding of agile software development methods. We then use this as a lens through which to develop a set of best practices for software development to improve software process and product quality. The Agile Manifesto (<http://agilemanifesto.org/>) identifies a set of values to improve overall software development. Although the principles driving these values overlap with the best practices identified in this article, this article goes a step further to identify a more comprehensive set of best practices that are grounded in theory and

As in the past with heavyweight methodologies, organizations often develop their own customized versions of these named agile development approaches to suit their development needs and organizational environment.

span across the three important aspects of software development; namely, people, product, and process.

HEAVYWEIGHT AND LIGHTWEIGHT SOFTWARE DEVELOPMENT METHODOLOGIES

To bring about some discipline to what can otherwise be chaotic processes of software development, many development methodologies have been proposed and adopted in practice. These methodologies impose a disciplined process on software development with the aim of making it more predictable and efficient. These software development methodologies can be classified as being either heavyweight or lightweight, based on the extent to which they emphasize factors such as (1) planning and documentation, (2) well-defined phases in the development process, (3) composition of the development team, and (4) use of well-elaborated modeling and coding techniques to generate software development artifacts, among others (Blum, 1996; Highsmith, 1999; Krutchen, 2001).

- *Heavyweight methodologies* promote up-front overall planning of the roles to be played, activities to be performed, and artifacts to be produced (Germain & Robillard, 2005). Proponents of these methodologies argue that planning leads to lower overall cost, timely product delivery, and better software quality. Traditional methodologies such as waterfall methodology and RUP (Rational Unified Process) to some extent are classified in this category (Germain & Robillard, 2005).
- *Lightweight or agile methodologies* put extreme emphasis on delivering working code or product while downplaying the importance of formal processes and comprehensive documentation. Proponents of these methodologies argue that by putting more emphasis on "actual working code," software development processes can adapt and react promptly to changes and demands imposed by their volatile development environment (Krutchen, 2001).

In addition to various named agile methodologies (such as eXtreme Programming [XP] and Scrum), numerous homegrown agile methodologies belong to this category. As in the past with heavyweight methodologies, organizations often develop their own customized versions of these named agile development approaches to suit their development needs

and organizational environment. Although agile approaches are not a panacea or the silver bullet, practices promoted by them shed some light on how adaptivity can be achieved in software development (Paulk, 2001). And although lightweight and heavyweight methodologies represent two extremes in software development methodology, most development methodologies used in organizations can be positioned between these two extremes (Bansler & Havn, 2003; Boehm, 2002).

Despite the diversity in their development practices, agile methodologies in general can be characterized as follows (Abrahamsson, Warsta, Siponen, & Ronkainen, 2003):

- Incremental (small software releases with rapid development cycles)
- Cooperative (a close customer and developer interaction)
- Straightforward (easy to learn and modify and are sufficiently documented)
- Adaptive (an ability to make and react to last-moment changes)

Table 1 presents a brief description of four leading agile development methodologies.

CAS THEORY AND SOFTWARE DEVELOPMENT

A complex adaptive system (CAS) is defined as one that is capable of adapting to its external environment and its internal state so that it survives despite the circumstances that befall it. Studies of CAS are based on the holistic premise that there is more to a holistic system than the sum of its components and linkages (Simon, 1996). CAS shows properties that are truly emergent, irreducible to explanations that take into account only those properties of its lower level components. From a review of CAS literature (Anderson, 1999; Arthur, Defillipp, & Lindsay, 2001; Chiva-Gomez, 2004; Cilliers, 1998, 2000; Eoyang, 1996; McKelvey, 1999, 2001; Rhodes & MacKechnie, 2003; Simon, 1996; Tan, Wen, & Awad, 2005), the major principles of CAS can be summarized as follows:

1. *Principle of open systems.* A CAS is an open system. It interacts with its environment. It exchanges energy or information with its environment and operates at conditions far from equilibrium.
2. *Principle of interactions and relationships.* Constituent elements of a CAS interact dynamically and exchange energy or information with each other. Even if only specific elements interact with a few others,

TABLE 1 Leading Agile Development Methodologies

Methodology	Description
Extreme programming (XP)	<p>A lightweight process targeted at development projects that are ill understood or have rapidly changing requirements (Beck, 1999)</p> <p>Empowers developers to confidently respond to changing customer requirements, even late in the life cycle, emphasizing teamwork</p> <p>Managers, customers, and developers are all part of a team dedicated to delivering quality software</p> <p>Improves a software project development process in four essential ways: communication, simplicity, feedback, and courage</p> <p>Its development process has various unique features such as requirements as stories, pair programming, test-driven design, frequent unit testing, and continuous integration</p> <p>Primary activities in each XP iteration cycle include new design, error fix, and refactoring (Alshayeb & Li, 2005)</p>
Adaptive Software Development (ASD)	<p>Places emphasis on production of high-value results emanating from the rapid adaptation to both external and internal events, rather than on the optimization of process improvement techniques</p> <p>Targets development teams in which competition creates extreme pressure (both high-speed and high-change) on the delivery process</p> <p>Adaptation is significantly more important than optimization (Highsmith, 1997, 1999)</p>
Feature-Driven Development (FDD)	<p>Uniquely sees very small blocks of client-valued functionality, called features, organizing them into business-related groupings</p> <p>Focuses on producing working results every two weeks and facilitating inspections; managers know what to plan and how to establish meaningful milestones (Palmer & Felsing, 2002)</p> <p>Reduces risk by emphasizing the frequent delivery of tangible working results</p> <p>Provides for detailed planning and measurement guidance; promotes concurrent development within each increment</p> <p>Its motto is “design by feature, build by feature” (Coad, LeFebvre, & Luca, 2000)</p>
Scrum	<p>A team-based approach for controlling the chaos of conflicting interests and needs to iteratively, incrementally develop systems and products when requirements are rapidly changing</p> <p>Can improve communications and maximize cooperation</p> <p>Is scalable from small single projects to entire organizations (Rising & Janoff, 2000)</p>

effects of these interactions are propagated throughout the CAS. The behavior of such a system is determined greatly by the nature of these interactions, and not simply by what is contained within these elements.

3. *Principle of transformative feedback loops.* Interactions across a CAS’s boundary result in many direct and indirect transforming feedback loops among the components of the system. Change in one part of the system is transmitted via its boundaries to other parts of the system, causing recipients of feedback information to change or react in some fashion. These secondary changes (reactions) are transmitted back to the originator, causing the originator to change again, and so on. Within an organizational context, feedback loops have the potential to spur continuous improvement and provide a shared sense of community involvement.

4. *Principle of emergent behavior.* Because interactions among different entities of a CAS are rich, dynamic, nonlinear, and fed back, the behavior of the CAS as a whole cannot be predicted simply from an inspection of its components. To understand the implications of these interactions and make predictions about a CAS’s behavior, the notion of “emergence” and “emergent behavior” is needed. Unpredictable and novel ideas may emerge (which may or may not be desirable) as a result of these interactions, but by definition they are not an indication of malfunction. These emerging ideas should not be censored simply because they were unanticipated.
5. *Principle of distributed control.* A CAS cannot thrive when there is too much central control. This certainly does not imply that there should be no control; rather, control should be distributed throughout the system. However, one should not go overboard

This implies that agile methods used to foster organizational adaptation must allow for dynamic and effective interplay among them.

with notions of self-organization and distributed control.

6. *Principle of shallow structure.* A CAS works best with shallow structure. This should be interpreted to suggest that a CAS should have some structure, but only the minimal amount of structure necessary for it to effectively achieve its objectives.
7. *Principle of growth and evolution.* In a CAS, survival is enhanced through continuous growth and evolution occurring in minuscule increments as the system responds to emerging internal and external environmental changes. This principle fosters adaptation by allowing the CAS to pay immediate attention to the needs of its immediate surroundings. In doing so, the CAS reorients itself incrementally to more distant/remote environmental conditions. Adaptation thus occurs in minuscule incremental steps.

CAS AND LIGHTWEIGHT METHODOLOGIES

Truex et al. (1999), Highsmith (1999), and Kauffman (1995) posit that enterprise information systems supporting today's digital economy are highly complex and exhibit a need to be adaptive. They also point out that developing and improving IS solutions is in itself a complex-adaptive task that exhibits the basic properties of a CAS. Although there is little published research on the application of CAS theory in software development, there is an increasing consensus that CAS informs the shift toward agile software development methodologies. For example, Gerber (2002) states:

... by incorporating the principles of self-organization called isomorphies, new methodologies ... including object-orientation, eXtreme Programming (XP), and lightweight methodologies use some concepts and principles of natural complex systems ... that's why they produce better results than conventional methodologies if they are applied in appropriate areas.

Highsmith (1999), in his book on adaptive software development, suggests emergent order as an alternative to a belief in and a dependence on imposed order. By the same token, Eoyang (1996) describes how CAS theory helped unravel complicated issues that threatened to stymie a customer support system's GUI (graphical user interface) development. In

providing an argument for a complex adaptive view of software development, he states:

... our old measures of product quality and process efficiency are always unrealistic and frequently destructive when they are applied to development of complex, adaptive software products ... we must change the way we look at the world of software development and business processes.

MAPPING CAS THEORY TO COMMON AGILE DEVELOPMENT PRACTICES

In this section we describe seven commonly followed agile practices and how various CAS principles provide insights into each of them. The process of developing IS solutions comprises interactions among three dimensions:

1. Stakeholders in a development project (people)
2. Process-related rules/guidelines used to provide a direction to the development effort (process)
3. Software artifacts generated as a result of this development effort (product)

Software development evolves through a myriad of complex interactions among these three dimensions. This implies that agile methods used to foster organizational adaptation must allow for dynamic and effective interplay among them. CAS theory provides an underpinning for dynamic interplay among people, process, and product dimensions and informs how agile methods enable such interplay. The result is a mapping of agile practices and CAS principles, as summarized in [Table 2](#).

Note that the first CAS principle (open systems) essentially applies to most of the agile practices discussed below. As will be apparent from the following discussion, agile methodologies place a heavy emphasis on open interactions among various stakeholders (including development team, management, and customers) to receive feedback and to manage the life cycle of a development project. Agile software development as a whole therefore exhibits this behavior of open systems, so this principle is not associated with any of the specific practices in [Table 2](#). Similarly, the sixth CAS principle (shallow structure) is not included in [Table 2](#) because it does not fit pragmatically in any one row or apply to any one agile practice. Rather, it provides a framework for various agile practices such as continuous frequent feedback, minimalist design teams, etc. For example,

TABLE 2 Mapping Agile Practices along Three Dimensions (People, Process, and Product)

#	Agile Practice	CAS Principle	Best Practices		
			Product Dimension (Artifact)	Process Dimension (Development)	People Dimension (Software Team)
1.	Frequent releases and continuous integration	Principle of growth and evolution	Develop the software artifact iteratively, allowing it to acquire increasing complexity in size and capability in each succeeding iteration.	Reevaluate development methodology frequently. Best practice is to start simple and modify methodology as the need for advanced or different processes and tools emerges.	Reevaluate team configuration frequently, scaling and transforming team composition and size as it progresses through various stages of development.
2.	Need for frequent feedback	Principle of transformative feedback loops	Iteratively test and validate software artifact, making necessary improvements to it.	Time-box development efforts, use measurable process milestones, and evaluate process effectiveness and efficiency continuously.	Pragmatically involve stakeholders and fellow developers by carefully seeking and listening to their comments and concerns.
3.	Proactive handling of changes to the project requirements	Principle of emergent order	Allow the solution being developed to be responsive to the emerging changes in project requirements by taking into account feedback gained from the exercise of frequent releases and integration.	Allow actual process employed to emerge or be determined by immediate local needs. In other words, process tailoring needs to be done to take into account contextual factors.	Allow actual team configuration to emerge or be determined by immediate local needs.
4.	Loosely controlled development environment	Principle of distributed control	Strive for componentization and loosely coupled software artifacts.	Use iterative, time-boxed, and/or modular development process.	Delegate responsibility and decision making to local development units.
5.	Planning kept to a minimum	Principle of growth and evolution	Minimal product planning done where and when necessary; limited emphasis on documentation, but some documentation is required.	Process planning done where and when necessary; always some process assessment undertaken, but in minimalist fashion.	Minimal human resource and stakeholder involvement planning, performed where and when necessary; regular assessment of efficacy of project team composition and size.
		Principle of emergent order	Let actual solution emerge naturally rather than from heavy planning and solution design.	Let actual process emerge naturally rather than from heavy planning and method engineering.	Allow for team interactions and working patterns to emerge naturally.
6.	Enhancing continuous learning and continuous improvement	Principle of growth and evolution	Allow for manageable experimentation in product design and for learning from the mistakes made. Learning results in better quality product.	Allow for manageable experimentation with various processes and for learning from mistakes made. Learning increases process effectiveness and efficiency.	Involvement in agile development effort enhances individual IS development skills.
		Principle of interactions and relationships	Allow for comparison of artifact solutions from current and past efforts. Comparison fosters reuse and enhances continuous improvement of previously developed solutions.	Allow for comparison of agile methods/processes from current and past efforts. This fosters reuse and enhances continuous improvement of existing agile methods.	Allow for pairing or teaming up of developers. Teamwork enhances learning, increases competence, and enhances communication.
7.	Emphasis on working software product	Principle of path of least effort (based on Zipf, 1949)	Foster an environment that simplifies and enhances the delivery of a functional, error-free IS solution.	Let the process that is most effective for rapid production of a working solution prevail. Emphasis on not fixing emergent process if it remains effective.	Let the team configuration that leads to the most rapid production of an effective working solution without forceful centralized control prevail.

Successful completion of each subsequent iteration boosted the development teams' confidence and helped to reduce the project risks.

using minimal structure (i.e., teams without a deep hierarchy), software development teams are able to seek and receive feedback frequently and more quickly.

Frequent Releases and Continuous Integration

Literature on agile software development suggests that for a complex systems development project to be successful it should be implemented in small steps, each with a clear measure of successful achievement and with an option of rolling back to a previous successful step upon failure (Larman & Basili, 2003). In addition, conducting frequent releases is crucial to accommodate constantly changing requirements. Thus, adaptation needs to occur in miniscule, incremental steps.

The CAS principle of growth and evolution provides a theoretical backing for such behavior in agile software development. This is evidenced in the product dimension in the form of incremental development of artifacts. In the process dimension, it is evidenced in the form of time boxes — finite modular periods of development time — comprised of identifiable process steps and tools. These time boxes or iterations become more intricate as the project matures due to the increasing complexity in the artifact being developed. In the people dimension, this principle is evidenced as a finite configuration of development personnel, the composition and structure of which may radically alter as the project shifts from iteration to iteration.

Grenning (2001) reported on a process-intensive company that introduced XP. After this company had launched XP, development teams carried out monthly releases and continuously integrated new features into the existing product. Successful completion of each subsequent iteration boosted the development teams' confidence and helped to reduce the project risks. Team members knew exactly what was happening and had a good grasp of the progress made at all times in the course of development (Grenning, 2001). Cao et al. (2004) also observed that development teams employing agile software development methodologies focused primarily on delivering production code that supports end-to-end functionalities rather than developing heavily integrated software modules. This practice was observed also at two software giants, Netscape and Microsoft (MacCormack, Verganti, & Iansiti, 2001). It is

seen as a way of increasing customer support and satisfaction (Beedle et al., 2001).

Best practices supported by the CAS principle of growth and evolution for this agile practice are thus:

- Develop the IS solution iteratively, with each iteration being focused on adding some well-defined capability to the continuously evolving IS solution.
- Start with a simple set of development processes and tools and modify development strategy in successive iterations as the need for more advanced procedures and tools becomes necessary.
- Start with a minimalist development team and scale up and/or reconfigure team's structure and composition as the project grows in size and complexity.

Need for Frequent Feedback

Software development by its nature involves interactions among stakeholders, development tools, software components in the solution being developed, and interactions of stakeholders with these tools, solution, or both. These interactions are message exchanges occurring between entities related or interconnected in some fashion to each other. Consequently, feedback comes not only from clients and other significant stakeholders but also from messages (such as error messages) received from the components that make up the IS solution, from development tools being employed, and/or from artifacts that document the developmental details within the process (Highsmith, 1999).

When a development team establishes transforming feedback loops across a boundary that divides the team from users, other development agents, and stakeholders, a complex adaptive development environment is created. In a bid to remain responsive, the development team frequently adjusts its focus, structure, and composition to best address these emerging requirements. The feedback loops also generate creative tension that propels the development team toward developing novel IS solutions that are suited to addressing the emerging requirements. To take advantage of feedback and to accommodate changes as a result of feedback, software processes also need to be flexible and easily changeable. These characteristics are a manifestation of the principle of transformative feedback loops in the people, product, and process dimensions of agile software development.

In a bid to remain responsive, the development team frequently adjusts its focus, structure, and composition to best address these emerging requirements.

According to MacCormack et al.'s (2001) study of IS development at Netscape and Microsoft, seeking feedback from various stakeholders was considered crucial to the fate of a project. Both organizations placed heavy emphasis on user feedback and carried out alpha and beta testing to obtain feedback from potential users. These development teams had multiple mechanisms to accommodate feedback from various stakeholders. One such mechanism was to prioritize the elements of the received feedback and incorporate them into subsequent iterations based on their criticality (MacCormack et al., 2001). Cao et al. (2004) report that when it was not possible to seek feedback directly from the customers, the development team had their project managers and business analysts (who had direct contact with customers) act as the surrogate customers. In both instances, the need for feedback was paramount.

Based on this discussion, suggested best practices are:

- Test and validate an IS solution at the end of each development iteration to obtain feedback on its efficacy and on required modifications.
- Time-box the development process and use measurable process milestones within the development iteration.
- Obtain pragmatic involvement from stakeholders and fellow developers by carefully seeking and listening to their comments and concerns.

These approaches have the potential to enhance programmatic and frequent feedback elicitation that allows agile development to remain responsive to the emergent business environment.

Proactive Handling of Changes to the Project Requirements

The CAS principle of emergent order helps explain the phenomenon of unanticipated requirements and capabilities evident in IS solutions. Within the software development context, CAS theory suggests that unanticipated and perhaps novel requirements may emerge in software development projects. The desirability of these emergent requirements is context sensitive. Nonetheless, they are not an indication of malfunction or malformation, and thus need not be censored. These emergent

requirements allow software artifact to remain current and reflective of the changing business environment.

One of the major attributes of agile software development methodologies is that they embrace changes in requirements from various sources, even in later development phases (Conboy, Fitzgerald, & Golden, 2005). This underscores an emphasis of agile methods on willingness to embrace emergent requirements and unanticipated changes regardless of when they occur in a project's lifetime. To ensure delivery of a "nonobsolete" solution, it is important that any changes to project requirements are handled proactively by making necessary adjustments to the development process (process dimension) and the development team (people dimension). This discussion implies that the process and people dimensions conform to the CAS principle of emergent order. Also, adjustments and modifications to the development processes and the team undertaking those processes shape outcomes of the IS solution (product dimension). In this regard, the IS solution also exhibits "emergent order."

At Netscape and Microsoft, systemic changes in a project's definition and basic direction were managed proactively (MacCormack et al., 2001). For example, during the development of a Web browser, end users were invited to play with the browser. These users had not used such software before. As the users played around with the beta versions of the browser software, they gained more understanding regarding its limitations and potential capabilities. These users then provided development teams with numerous suggestions for improvement of the software. Had the development teams ignored users' feedback, neither of these companies would have been able to deliver browser software that was truly reflective of their external users' emerging requirements.

Based on the above discussion, the best practices underpinned by the principle of emergent order are:

- Allow for sufficient flexibility in a development process.
- Allow for sufficient flexibility in a development team to facilitate quick response to immediate local needs.
- Accommodate emerging requirements in the IS solution to develop a solution that is reflective of current business needs.

Although on the surface agile methodologies may appear to follow unplanned and undisciplined development practices, these methodologies emphasize a minimum, but sufficient, amount of planning within each iteration.

Loosely Controlled Development Environment

CAS theory suggests that in conditions of high uncertainty, flexible and adaptive organizational units are more appropriate than rigid and static ones. Organizational units operating in such situations tend to flourish when distributed decision making and control mechanisms are utilized across a network of interconnected entities (Rihania & Geyer, 2001). Agile development teams exhibit flexible and distributed control structures. This type of structure tends to be minimal and highly accommodating. This in itself is reflective of adherence to the CAS principle of distributed control.

Published studies on agile software development provide support for this principle. For example, project managers at FinApp (Cao et al., 2004) believed that deep hierarchical organizational structure could lead to unresponsive environments with high inertia. Eoyang (1996) reports that better quality user interfaces and greater efficiencies in development were achieved when the IS development team was reorganized to allow for localization of decision making. Flexible work setting, focus on results rather than micro-management, and empowering people who are actually doing the work were seen as key factors affecting developer morale and motivation (Cao et al., 2004; Cusumano & Yoffie, 1999).

Cusumano and Yoffie (1999) also observe that managers at Microsoft and Netscape tried not to control the development process too rigidly in order to be responsive. At Microsoft, MacCormack et al. (2001) note that developers could exercise veto powers at the local level based on their technical expertise in a problem domain being addressed. At Netscape, senior developers along with members of the marketing department could assign higher priorities to their favorite features and the product managers, engineers, and developers could negotiate IS solution terms with their clients (the marketing department, for example) when things didn't go quite as per plan (MacCormack et al., 2001).

These observations might lead one to think about the role project managers should play in agile development teams. Fowler (2002) and Blotner (2003) suggest that a manager can and should help the team be more productive by offering some insight into the ways in which things can be done in an agile environment. Grenning (2001) observed that having senior people monitor the team's progress at a monthly design-as-built review meeting accelerated

the development process while lowering the number of bugs. Any bugs found were made part of the next iteration.

The recommendations for best practice based on the above discussion are:

- Leadership and decision making on a software development project ought to be decentralized, with more decisions being made at local units where problems are encountered.
- Successive iterations in a development process need to be fairly independent or loosely coupled.
- The structure of an IS solution being developed needs to foster componentization, loose coupling, and high cohesion within the finished product.

Planning Kept to a Minimum

Given the volatile and rapidly changing business environment, it's practically impossible to specify ahead of time everything that should go into a new product or release. CAS theory lays significant emphasis on the emergent order — the principle of emergent order — accounting for minimal planning practice to accommodate various unforeseen requirement changes. Although on the surface agile methodologies may appear to follow unplanned and undisciplined development practices, these methodologies emphasize a minimum, but sufficient, amount of planning within each iteration.

For example, XP emphasizes iteration-centric planning. Rather than planning, analyzing, and designing for the far-flung future (which is highly prone to change), XP exploits reduction in the cost of changing software to do all of these activities a little at a time, throughout the development process (Beck, 1999). The scope of the planning process is kept primarily to the current iteration. Feature-Driven Development (FDD) also advocates for detailed planning and measurement guidance within each "design by feature, build by feature" increment (Coad et al., 2000). Only after executing and evaluating the outcome of the current plan are plans for subsequent iterations generated (Martin, 2000). These practices were observed at Netscape and Microsoft, where managers clearly believed that some planning, albeit minimal, was necessary to build complex products (Cusumano & Yoffie, 1999).

Within the product, process, and people dimensions, suggested best practices are that (1) the planning for the IS solution, (2) the actual processes used to develop the solution, and

Another practice that is a feature of agile software development is the minimal emphasis on documentation.

(3) the structure and composition of the development team are best addressed within each development iteration. This keeps planning to a minimum and steers the focus to the immediate needs of the development project.

Enhancing Continuous Learning and Continuous Improvement

CAS theory stipulates that adaptation is a learning mechanism. It occurs because a CAS is able to learn about changes in its environment or its internal state and react to those changes. Additionally, each successful adaptation is reinforced for reuse in similar future situations. The CAS principle of growth and evolution provides a theoretical underpinning for such behavior. Agile methodologies put a great emphasis on people and their talents, skills, and knowledge, suggesting that for agile development to be effective team members must be responsive, competent, and collaborative (Boehm, 2002; Cockburn & Highsmith, 2001).

Adaptation occurs by continuous reactions to changes in environmental conditions sensed locally by specific entities within the system. Because these entities tend to be specialized, they need to interact with other entities to enact adequate behavior to bring about adaptation in the CAS. Consequently, relationships within the CAS result in information exchange that produces learning. Different units of a development team must interact with each other. For example, in XP, developers with different levels of experience and skill are typically paired for development purposes (Cao et al., 2004; Grenning, 2001). It is suggested that such pairing fosters learning for all involved parties. To facilitate this kind of learning, however, the communication capabilities of developers become crucial (Cao et al., 2004). Various tasks within a development process are interrelated and dependent on each other. Effective completion of one task impacts completion of others. Thus, agile methodologies' emphasis on continuous learning and improvement finds theoretical support in the CAS principle of interactions and interconnectedness, on the one hand, and that of growth and evolution, on the other hand.

Inferences from this discussion suggest that:

- Componentized IS solution development fosters reuse and enhances learning from past successes and failures in developing effective components.

- Tolerance for manageable experimentation with various approaches and focus on attainment of short-term objectives enhances learning by doing, allows the development team to fine-tune the processes and tools used for the development effort, and steers the development project toward expected outcomes.
- Interactions among members of the development team and various stakeholders will foster an exchange of ideas and second-order learning.

This practice also fosters the emergence of natural, unforced collaborative partnerships, thereby leveraging productivity, creativity, and quality.

Emphasis on Working Software Product

Another practice that is a feature of agile software development is the minimal emphasis on documentation. The motivation for this practice is twofold: (1) to make the development team more effective in responding to change and (2) to reduce the cost of moving information between people (Cockburn & Highsmith, 2001). Martin (2000) suggests that teams should create detailed plans that are meaningful only for that iteration. Grenning (2001) argues that a good way to protect future software maintainers is to provide them with clean and simple source code, rather than with binders full of out-of-date paper, and to keep the documentation at a high level so that usual maintenance changes and bug fixes don't affect it. Cusumano and Yoffie (1999) found that managers at Microsoft and Netscape tolerated incomplete documentation because they put a premium on creating a working product.

This is an area in which CAS theory provides weak support. Hence, we borrow from the theory of "path of least effort" offered by Zipf (1949), who explains this emphasis on a working product with minimalist documentation orientation:

Nothing is gained by calculating a particular path of least effort to a greater degree of precision, when the added work of so calculating it is not more than offset by the work that is saved by using the more precisely calculated path.

Consequently, it is best to let the "path of least effort" emerge in the course of completing a development task. This helps us understand the agile practice of minimal emphasis on documentation, and also on formal planning, while

By identifying the correct set of metrics and mechanisms to manage their interdependencies, project managers can effectively steer the project toward desired outcomes as planned.

putting heavy emphasis on building a working product.

This discussion prompts the identification of the following best practices:

- Foster an environment that simplifies and enhances delivery of a functional, error-free IS solution.
- Let the development process that is most effective at producing a working solution prevail.
- Allow for working relationships of development team members to produce natural configurations that foster rapid production of a working solution.

Summary

In summary, CAS theory provides insights into how agile development methods foster delivery of responsive and highly evolvable software solutions. Such insights are crucial when identifying metrics for agile software development, because the same CAS principles may govern these metrics. The metrics so developed ought to capture each of the three dimensions of software development; namely, people, process, and product. However, as indicated in the previous section, these three dimensions are not completely independent of each other. Thus, these metrics should be identified and developed in such a way that they *incorporate the interactions across all of these dimensions*. For example, the principle of emergent behavior is applicable to all three dimensions via the common practice of proactive handling of changes to the project requirements. This has direct implications not only for researchers but also for practitioners. By identifying the correct set of metrics and mechanisms to manage their interdependencies, project managers can effectively steer the project toward desired outcomes as planned.

A key limitation of this article is that it stops short of deriving and validating metrics for agile software development for the theory-based best practices suggested. However, it does present a basis upon which such metrics can be developed. The argument here is that CAS theory, as presented in this article, is an effective theory upon which to underpin such metrics. The discussion of how these theory-based best practices could be implemented within each of the three core dimensions of an agile software development undertaking — people, process, and product — provides a starting point for the elaboration and refinement of metrics that can capture each best practice

succinctly. We see this as an opportunity for future research in this domain.

CONCLUSION

As Highsmith (1999) succinctly points out, without a clear understanding of underlying theoretical principles behind software development approaches, organizations are at high risk of being nonadaptive. This article contributes to the literature on complexity science and IT development by providing a clear understanding and appreciation of the sources, nature, and types of interdependencies between individuals, organizations, and IS solutions within the context of fast-paced agile software development. Based on discussion presented in this article, we argue that organizations that use heavyweight methodologies need to reconsider these agile practices, rather than discounting them as “hacking,” and integrate them as suitable.

Opportunities for future research include exploring the finer differences in how various agile software development methodologies support the proposed best practices; the development of a customizable set of metrics for agile software development methodologies; and the identification of mechanisms for embedding these metrics into automated development tools and integrated development environments commonly employed in agile software development projects.

Other areas that need to be addressed include the development of design principles that are also rooted in CAS theory, the empirical evaluation of these principles to establish their effectiveness and rigor, and the identification of various metrics to assess product quality and process efficiency based on CAS theory. This understanding can also be used to inform us about why certain agile practices are more amenable than others. Availability of this knowledge can make the tailoring of agile practices more informed, rather than based simply on intuition. ▲

References

- Abrahamsson, P., Warsta, J., Siponen, M. T., & Ronkainen, J. (2003). *New Directions on Agile Methods: A Comparative Analysis*. Paper presented at the 25th International Conference on Software Engineering, Portland, Oregon.
- Alshayeb, M., & Li, W. (2005). An empirical study of system design instability metric and design evolution in an agile software process. *Journal of Systems and Software*, 74(3), 269-274.

- Anderson, P. (1999). Complexity theory and Organization Science. *Organization Science*, 10(3), 216-232.
- Arthur, M., Defillipp, R., & Lindsay, V. (2001). Careers, Communities, and Industry Evolution: Links to Complexity Theory. *International Journal of Innovation Management*, 5(2), 239-256.
- Bansler, J., & Havn, E. (2003). *Improvisation in Action: Making Sense of IS Development in Organizations*. Paper presented at the International Workshop on Action in Language, Organisations and Information Systems (ALOIS 2003), Linköping, Sweden.
- Baskerville, R., Ramesh, B., Levine, L., Pries-Heje, J., & Slaughter, S. (2003). Is "Internet-speed" software development different? *IEEE Software*, 20(6), 70-77.
- Beck, K. (1999). Embracing Change with Extreme Programming. *IEEE Computer*, 32(10), 70-77.
- Beedle, M., Bennekum, A. V., Cockburn, A., Cunningham, W., Fowler, M., Highsmith, J., et al. (2001). *Principles behind the Agile Manifesto*. Retrieved Feb 15th, 2004, from <http://agilemanifesto.org/principles.html>
- Blotner, J. (2003). *It's More than Just Toys and Food: Leading Agile Development in an Enterprise-Class Start-Up*. Paper presented at the Agile Development Conference (ADC 2003).
- Blum, B. (1996). *Beyond Programming: To a New Era of Design*. Oxford University Press.
- Boehm, B. (2002). Get Ready for Agile Methods, with Care. *IEEE Computer*, 35(1), 64-69.
- Cao, L., Mohan, K., Xu, P., & Ramesh, B. (2004). *How Extreme Does Extreme Programming Have to Be? Adapting XP Practices to Large-Scale Projects*. Paper presented at the 37th Annual Hawaii International Conference on System Sciences (HICSS'04), Big Island, Hawaii.
- Chiva-Gomez, R. (2004). Repercussions of complex adaptive systems on product design management. *Technovation*, 24(9), 707-711.
- Cilliers, P. (1998). *Complexity and Postmodernism: Understanding Complex Systems*. London; New York: Routledge.
- Cilliers, P. (2000). What Can We Learn From a Theory of Complexity? *Emergence*, 2(1), 23-33.
- Coad, P., LeFebvre, E., & Luca, J. D. (2000). *Java Modeling in Color with UML: Enterprise Components and Process*. Prentice Hall.
- Cockburn, A., & Highsmith, J. (2001). Agile Software Development: The People Factor. *IEEE Computer*, 34(11), 131-133.
- Conboy, K., Fitzgerald, B., & Golden, W. (2005). Agility in Information Systems Development: A Three-Tiered Framework. In R. Baskerville (Ed.), *Business Agility and Information Technology Diffusion. IFIP TC8 WG 8.6 International Working Conference*. Atlanta, Georgia, USA.: Springer.
- Cusumano, M., & Yoffie, D. (1999). Software Development on Internet Time. *IEEE Computer*, 32(10), 60-69.
- Eoyang, G. (1996). Complex? Yes! Adaptive? Well, maybe ... *Interactions*, 3(1), 31-36.
- Fowler, M. (2002, March). *Agile Development: What, Who, How, and Whether*, from <http://www.fawcette.com/resources/managingdev/interviews/fowler/>
- Gerber, M. (2002). *Keynote Speech: Lightweight Methods and Their Foundations in Chaos Theory*. Paper presented at the 6th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2002), Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland.
- Germain, E., & Robillard, P. N. (2005). Engineering-based processes and agile methodologies for software development: a comparative case study. *Journal of Systems and Software*, 75(1-2), 17-27.
- Glass, R. (2003). Questioning the Software Engineering Unquestionables. *IEEE Software*, 20(3), 119-120.
- Grenning, J. (2001). Launching Extreme Programming at a Process-Intensive Company. *IEEE Software*, 18(6), 27-33.
- Highsmith, J. (1997). Messy, Exciting, and Anxiety-Ridden: Adaptive Software Development. *American Programmer*, X(1).
- Highsmith, J. (1999). *Adaptive Software Development: A Collaborative Approach to Managing Complex Systems*. New York, NY: Dorset House Publishing.
- Kauffman, S. (1991). Antichaos and Adaptation. *Scientific American*, 256(2), 78-84.
- Kauffman, S. (1995). *At home in the Universe*: Oxford University Press.
- Krutchon, P. (2001). Agility with the RUP. *Cutter IT Journal*, 14(12), 27-33.
- Kwak, Y. H., & Stoddard, J. (2004). Project risk management: lessons learned from software development environment. *Technovation*, 24(11), 915-920.
- Larman, C., & Basili, V. R. (2003). Iterative and Incremental Development: A Brief History. *IEEE Computer*, 36(6), 47-56.
- MacCormack, A., Verganti, R., & Iansiti, M. (2001). Developing Products on "Internet Time": The Anatomy of a Flexible Development Process. *Management Science*, 47(1), 133-150.
- Martin, R. (2000). eXtreme Programming Development through Dialog. *IEEE Software*, 17(4), 12-13.
- McKelvey, B. (1999). Avoiding Complexity Catastrophe in Coevolutionary Pockets: Strategies for Rugged Landscapes. *Organization Science*, 10(3), 294-321.
- McKelvey, B. (2001). Energising Order-Creating Networks of Distributed Intelligence: Improving the Corporate Brain. *International Journal of Innovation Management*, 5(2), 181-212.

- Palmer, S., & Felsing, J. (2002). *A Practical Guide to Feature-Driven Development*. Upper Saddle River, NJ: Prentice Hall PTR.
- Paulk, M. (2001). Extreme Programming from a CMM Perspective. *IEEE Software*, 18(6), 19-26.
- Rhodes, M. L., & MacKechnie, G. (2003). Understanding Public Service Systems: Is There a Role for Complex Adaptive Systems Theory? *Emergence*, 5(4), 57-85.
- Rihania, S., & Geyer, R. (2001). Complexity: an appropriate framework for development? *Progress in Development Studies*, 1(3), 237-245.
- Rising, L., & Janoff, N. (2000). The Scrum Software Development Process for Small Teams. *IEEE Software*, 17(4), 26-32.
- Simon, H. (1996). *The Sciences of the Artificial* (Third ed.). Cambridge, MA: MIT Press.
- Sutherland, J., & van den Heuvel, W.-J. (2002). Enterprise Application Integration and Complex Adaptive Systems. *Communications of the ACM*, 45(10), 59-64.
- Tan, J., Wen, H. J., & Awad, N. (2005). Health care and services delivery systems as complex adaptive systems. *Communications of the ACM*, 48(5), 36-44.
- Truex, D., Baskerville, R., & Klein, H. (1999). Growing systems in emergent organizations. *Communications of the ACM*, 42(8), 117-123.
- Zipf, G. (1949). *Human Behavior and the Principle of Least Effort*. New York: Hafner Publishing Company.