Communications of the Association for Information Systems

Volume 24 Article 43

6-1-2009

Controls in Flexible Software Development

Michael L. Harris
Indiana University – Southeast, harris60@ius.edu

Alan R. Hevner
University of South Florida

Rosann Webb Collins
University of South Florida

Recommended Citation
Harris, Michael L.; Hevner, Alan R.; and Collins, Rosann Webb (2009) "Controls in Flexible Software Development," *Communications of the Association for Information Systems*: Vol. 24 , Article 43.
DOI: 10.17705/1CAIS.02443
Available at: https://aisel.aisnet.org/cais/vol24/iss1/43

Communications of the Association for Information Systems

CAIS

Controls in Flexible Software Development

Michael L. Harris

Indiana University – Southeast, New Albany, IN, harris60@ius.edu

Alan R. Hevner

University of South Florida, Tampa, FL

Rosann Webb Collins

University of South Florida, Tampa, FL

Abstract:

Control and flexibility may appear an unlikely pair. However, we propose that effective flexible software development processes must still provide clear control mechanisms to manage the progress and quality of the resulting software products. This paper presents a conceptual study to understand the types of control found in flexible software development processes, termed controlled-flexible approaches. Control theory is used as a lens to study the control mechanisms found in plan-driven and flexible processes. We extend current thinking to include emergent outcome controls and clan controls for team coordination in our taxonomy of control mechanisms. Several popular flexible processes are analyzed for control mechanisms. We conclude with a brief discussion of future research directions.

Keywords: software development, control methods, flexible methods, agile processes, emergent outcomes

Volume 24, Article 43, pp. 757-776, June 2009

I. INTRODUCTION

Flexible[1] software development processes include subtle, yet essential, control mechanisms to manage the progress and quality of the resulting software products. To better understand the controls found in flexible processes, we use control theory as a lens to examine the central tension between control and flexibility in managing software development. It is generally understood that a key managerial responsibility is to exert controls that guide employees' behaviors to ensure compliance with organizational goals. The challenge for software development managers today is how to appropriately employ controls when dynamic environments require more flexibility. In addition, the need for explicit and formal controls is increased when development is distributed and/or developers come from multiple organizations. This control versus flexibility tension is evident in the software industry's long running debate over the relative merits of plan-driven development approaches versus flexible approaches [Boehm and Turner 2004].

So, how should software teams balance flexibility and control of their work processes? In the early days of software development, programmers tended to use an unconstrained, ad hoc approach for the construction of software systems. The outcomes were often unpredictable and unrepeatable. Faced with poor results from ad hoc development, teams turned to planned approaches. The waterfall method, the archetype for planned approaches, establishes several stages for the development process. A team must complete a stage and gain stakeholder agreement to stage completion before it progresses to the next stage.

One of the first stages in the waterfall process is the development of a detailed work plan that becomes an output control [Ouchi 1977] for the balance of the software development process. Like all output controls, the plan represents a detailed specification of the deliverables throughout the entire project. Due to the central role of the plan in managing this class of development methods, they have been termed plan-driven approaches [Boehm and Turner 2004]. Plan-driven approaches have been welcomed for the structure they bring to the development process. In a recent study [Neil and Laplante 2003], organizations reported that the waterfall approach was the most popular development method.

However, critics of plan-driven approaches have pointed out that it is not only difficult, but sometimes impossible to fully specify software before development begins [McConnell 1996]. When knowledge is tacit or when the technology is new, the team may need to test software process alternatives before they can select the best approach. Furthermore, in quickly changing environments the team may find that user or market needs change during the course of development.

This has led to a search for intermediate alternatives – processes that are more flexible than a plan-driven method yet are more controlled than an ad hoc approach. We term these approaches controlled-flexible. The list of these choices seems to grow daily. Boehm and Turner [2004] list ten flexible processes, but this is far from a complete list. Methods in practice may be even more varied as organizations implement their own interpretations of various flexible approaches.

The agile manifesto [Agile Manifesto 2001] states principles that are used by many of the flexible approaches. However, the manifesto principles leave open the theory behind the flexible processes. The manifesto doesn't explain why various principles are important, nor does it explain how the principles are intended to be enacted. A more formal approach for understanding the use of control mechanisms in order to achieve organizational goals is embodied in control theory as articulated by Ouchi [1977, 1979, 1980]. In the study presented in this paper, we use a conceptual research approach to marry the observations of formal control theory with the practical knowledge embedded in existing flexible processes. Our goal is to better understand the role and use of controls in flexible software development processes. In order to reflect actual control practices, we propose an expansion of control theory to encompass emergent dynamic controls.

[1] We use the term flexibility instead of agility to be more inclusive of processes that encourage change during development but that may not meet some definitions of agile processes.

II. FLEXIBILITY IN SOFTWARE DEVELOPMENT

Although plan-driven development processes are in broad use, the demand for more flexible alternatives is evident. In a recent study, 85 percent of CIOs indicated that agility is part of their core business strategy [Ware 2004]. While it is widely recognized that software development methods must be flexible, it is very difficult for managers to determine what forms of flexibility are best suited for a given instance of development.

As a case in point for the need of flexibility, consider the FBI Virtual Case Project that was cancelled after $170 million in expenditures. According to SAIC, the contractor, the problem was an evolving design [Hayes 2005]:

…And what the FBI called software deficiencies were really more changes in requirements. And users kept rejecting SAIC's software designs, taking what one SAIC executive complained was a "trial-and-error, we-will-know-it-when-we-see-it approach to development."

The article's author offered his opinion regarding the issues:

And in the frantic days after Sept. 11, SAIC should have spotted that stable requirements for this project just weren't in the cards. The FBI needed results in the face of a crisis. SAIC should have shifted gears and methodologies to start producing working deliverables right away, no matter how far the project was from a complete set of requirements.

One insight from this case is that the SAIC executive dismissed the FBI's design process as a "trial-and-error" approach. Although this instance may have been an example of trial-and-error (ad hoc) development, the quote highlights a problem with flexible methods; managers may not be able to identify and effect flexible control mechanisms and, thus, they may not be able to differentiate between a flexible approach and an ad hoc development approach.

MacCormack, Verganti, and Iansiti [MacCormack et al. 2001] provide another example that illustrates the confusion over the workings of flexible controls. The researchers asked managers at a company to identify a successful project and an unsuccessful one. Analysis revealed that the 'successful' project was a well structured project with no changes and no surprises once the design was locked down. In contrast, the 'unsuccessful' project underwent continual change in response to market and competitive changes. However, surprising news was revealed when objective measures of success were examined. The 'unsuccessful' flexible project had higher quality levels and used fewer resources relative to its level of complexity.

In this research we develop a theoretical framework for controls in software development based on control theory and an analysis of development methods in use. We held an informal focus group to help stage the research. The focus group confirmed that some developers believe a plan-driven approach is 'correct' and that a flexible approach is less favored. Upon further discussion, it became clear that the problem may be there is not a common understanding of the distinction between a flexible, 'anything goes' approach and a *controlled-flexible* approach.

This confusion might trace its roots back to the creation of the 'waterfall' approach to development. Early software projects were unstructured, ad hoc initiatives. The problems with these initiatives were addressed through the introduction of the waterfall approach or related plan-driven derivatives. Using this as a frame, a development approach might be classified using a continuum between a plan-driven approach and an ad hoc, flexible approach as illustrated in Figure 1.

Plan Driven
Ad Hoc

Figure 1. Continuum between Plan-Driven and Ad Hoc Development Processes

Although there is some insight in the continuum in Figure 1, we argue that this picture is limiting and that it does not fully present the range of controlled-flexible approaches. The Figure 1 continuum creates two end-points: 1) It envisions a fully plan driven method wherein development is completely specified and controlled by the plan; and 2) it envisions a fully ad hoc method wherein developers are given no guidance and complete freedom with regard to both the feature set and the timing of the eventual deliverables. This worldview would then interpret any point along the continuum as a development approach that relaxes the plan-driven controls to allow more flexibility.

However, we argue that a controlled-flexible approach is not a plan-driven approach with fewer controls. Instead, we suggest that controlled-flexible approaches have *different* controls. This viewpoint was supported in our focus group discussions. The participants were introduced to several flexible control mechanisms, such as: daily software builds, interim releases for stakeholder review, daily meetings, and pair programming. This led participants to recognize that controlled-flexible approaches may contain controls, but they are different from those in plan-driven approaches. Instead of Figure 1, we propose a more two-dimensional view of flexible alternatives in software development. This is illustrated in Figure 2, below.

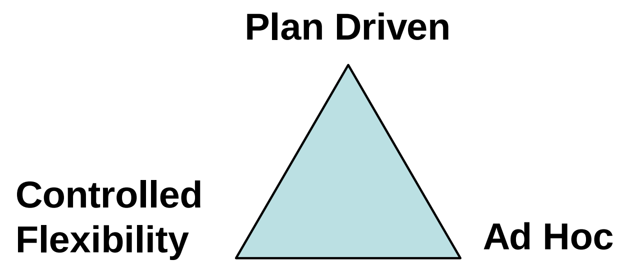


Figure 2. A Two-Dimensional View of Software Development Approaches

This figure implies that multiple dimensions exist. A plan-driven approach requires stakeholder agreement on the plan before development begins. In an ad hoc approach, developers are given a broad vision and are expected to report back with a completed software product. A controlled-flexible approach has its own controls that are different from those in a plan-driven approach. Any given instance of development may exist anywhere on this ‘map’ of development alternatives. Let’s consider a case where the specificity of the plan is reduced, thus giving more freedom to the developer. This flexibility may be offset with increased use of flexible controls, such as daily software builds. This would represent a movement along the left edge of the triangle from Plan-Driven toward Controlled-Flexible. If, however, the plan becomes less specific but there are no additional constraints added, the approach can be classified closer to the right edge of the figure in the Ad-Hoc region.

In practice, an individual instance of a development approach is not limited to points along the edge of the figure. As an organization adopts a development process it also adapts the process to its own idiosyncratic needs [Boehm and Turner 2004]. Preliminary interviews conducted for this study suggested that many organizations end up with processes in practice that consist of tradeoffs between the process choices. Since there is no language for understanding and comparing processes and their control mechanisms, it is difficult for adopting organizations to understand whether an appropriation of a process is faithful [DeSanctis and Poole 1994].

Much of the existing research on software process improvement focuses on a single process and uses assertions and proofs of concept as evaluation techniques [Zelkowitz and Wallace 1998]. Research on agile processes is no exception to this trend. There is no established framework for comparing, analyzing, and evaluating flexible processes. Similarly, there is no theory that can be used to differentiate a controlled-flexible process from a purely ad hoc approach.

Thus, a research objective of this study is to develop a common language for analyzing and comparing flexible processes based on their use of common control mechanisms. The current dialog centers on individual mechanisms recommended by specific agile processes. It is difficult to know whether these separate concepts are complementary or if they are substitutes for one another when there is no established theoretical base to explain the purposes of various mechanisms. By drawing on the control theory literature, we hope to provide a beginning taxonomy for understanding controls in flexible processes. In addition, we expect to aid practitioners who seek to adopt flexible processes. This study can help adopting organizations achieve a better understanding of the role and purposes of the control mechanisms in a flexible development method.

III. CONTROL THEORY

In this research we use control theory to establish a taxonomy for analyzing flexible processes. The initial work in control theory established three types of controls that organizations use to manage towards objectives [Ouchi 1977, 1979, 1980; Ouchi and Johnson 1978]. These control types can be briefly defined as:

- Behavioral control: Appropriate when the behaviors that transform inputs to outputs are known
- Outcome control: Appropriate when a process' output can be measured
- Clan control: Appropriate in ambiguous circumstances where neither the behaviors nor outputs can be predicted a priori. Clan members belong to a common organization and share values, beliefs, and attitudes [Cardinal et al. 2004; Ouchi 1980].

The relationships among the types of control are shown in Table 1. Two factors determine the proper control approach: the availability of outcome measures and the knowledge of the transformation process. Outcome measures are useful if an outcome can be specified a priori and if the individual's contribution can be tied to the outcome. Alternatively, behavioral control can be exercised by prescribing the transformation behaviors that produce the end product and measuring adherence to those behaviors. This behavioral control approach not only requires well known transformations that will result in success, but it also requires that behaviors be observable. Furthermore, the controller must be knowledgeable enough to understand the appropriate behaviors in order to observe them [Kirsch 1997].

Table 1. Organizational Use of Control Types

		Knowledge of Transformation Process	
		Perfect	Imperfect
Availability of Outcome Measures	High	Behavioral or Output	Output
	Low	Behavioral	Clan

Adapted from Ouchi [1977, 1979]

The initial work in control theory was produced in a relatively deterministic world (e.g. an automobile assembly line). Output control assumed that you could clearly and completely specify an output a priori. Furthermore, success or failure in regards to an a priori output was binary with no ambiguity. Likewise, behavioral control was expected to be straightforward. If every employee successfully performs specified behaviors, then every employee will deliver acceptable results.

Ouchi recognized that employees might have multiple duties, and thus a single employee might face multiple controls. Furthermore, he recognized that future organizations might not be so deterministic. He referred to the emerging literature of the time and stated that clan control might be the most appropriate choice as uncertainty increased [Ouchi 1979] .

Since Ouchi's initial work there has been significant research on control theory. Specific to our interests in this study on software development are the extensions related to clan control, portfolios of control, and dynamic controls. Researchers have suggested two different controls related to the management of software development: self control and team control [Kirsch 1996; Choudhury and Sabherwal 2003]. In order to reconcile these extensions with Ouchi's initial work, we have chosen the terms "Clan-Attitude" control and "Clan-Team" control.

When Ouchi talked about clan control [Ouchi 1979, 1980] he talked about control through common values and beliefs. This type of control is used when formal monitoring is not possible. This corresponds to the concept of self-control discussed by some researchers. We support this view of control, but prefer the term Clan-Attitude control over the term 'self-control'. The purpose of any control is to manage individuals to achieve organizational objectives. Self-control might imply a lack of organizational influence. Ouchi points to the organization's role in shaping clan decisions through selection, training, and other means of socialization that shape attitudes.

In addition, researchers have pointed out that peers can also influence decisions to achieve a type of team control [Kirsch 1996; Choudhury and Sabherwal 2003]. Ouchi did state that co-workers can subtly influence one another, but this understates the clear signals that team members can send one another. Consider, for example, the practice of requiring team members to wear dunce caps when they break software builds [Cusumano and Yoffie 1999].

Newer research has also emphasized the need for a portfolio of controls rather than just relying on one type of control [Choudhury and Sabherwal 2003; Henderson and Soonchul 1992; Kirsch 1997, 2004; Nidumolu and Subramani 2004; Orlikowski 1991]. Ouchi did recognize that multiple controls could exist. For example, he described a retail environment in which one control might use commissions to focus employees on sales while using other controls to focus employees on duties such as stock arranging [Ouchi 1977]. However, Ouchi's work focuses on the role of different goals to encourage different duties that an individual might have. The work on control portfolios discusses the need for multiple controls for a single task when the task is sufficiently complex that a single control might not be sufficient.

We find that control theory falls short in the discussion of dynamic environments, such as those involving flexible development methods. There have been studies that investigate changes in controls over time [Cardinal et al. 2004; Kirsch 2004]. However, these studies examined predictable changes in control approaches through the lifecycles of projects and organizations. In the current context, we examine shorter term changes. In the short term, our goal is not to change controls, but to establish controls that can manage change.

The primary recommendation of control theory in "conditions of ambiguity, of loose coupling, and of uncertainty" [Ouchi 1979 p. 845] is to use clan control. While we certainly see the role of clan control as an element in a portfolio of controls, we feel that the flexible development techniques contain more than clan controls. In making this observation, we see a correspondence between clan control and organic organizations. We also feel a kinship with Eisenhardt and Tabrizi's [1995] study of high technology companies, wherein they note that:

"… calling this organic does not capture the sense of structure that we found."

IV. A TAXONOMY OF DYNAMIC CONTROLS

We have identified some shortcomings regarding the application of control theory to flexible development. However, it would be a mistake to abandon control theory and start anew. Control theory has been applied many times to understand how organizations achieve management objectives. Our goal is to marry this existing wisdom with the insights embedded in the field of flexible development in order to extend the theory so that it encompasses both past findings and the new requirements of flexible development.

The work of Ouchi [1979] points to limits of both output and behavioral control in uncertain environments, and suggests the use of clan control. As discussed earlier, in order to shape attitudes of employees, the mechanisms of clan control are achieved through selection of personnel, training, and socialization, not through direct control of the task. However, even a casual observation of flexible development methods will reveal specific control methods that are much more tangible than the dictates of clan control.

In fact, we specifically note that, despite the expectations of control theory, the agile manifesto [Agile Manifesto 2001] elevates the importance of outputs in the management of agile processes. The manifesto deemphasizes the role of a planning document, but it maintains a central focus on the output of the process (working software). In order to understand this seeming contradiction with control theory, we examine the meaning of output control as defined by control theory and compare that to the practices in flexible software development.

Output/Outcome Control and Dynamic Development

First we must understand the nature of output controls. According to control theory, outcomes are deterministic: produce to this specification and you will achieve an acceptable product. The most obvious instance of outcome control in systems development is the use of a detailed plan in plan-driven development. The plan is developed upfront. Once the stakeholders agree to the plan, it becomes an outcome control that governs development and delivery of the software.

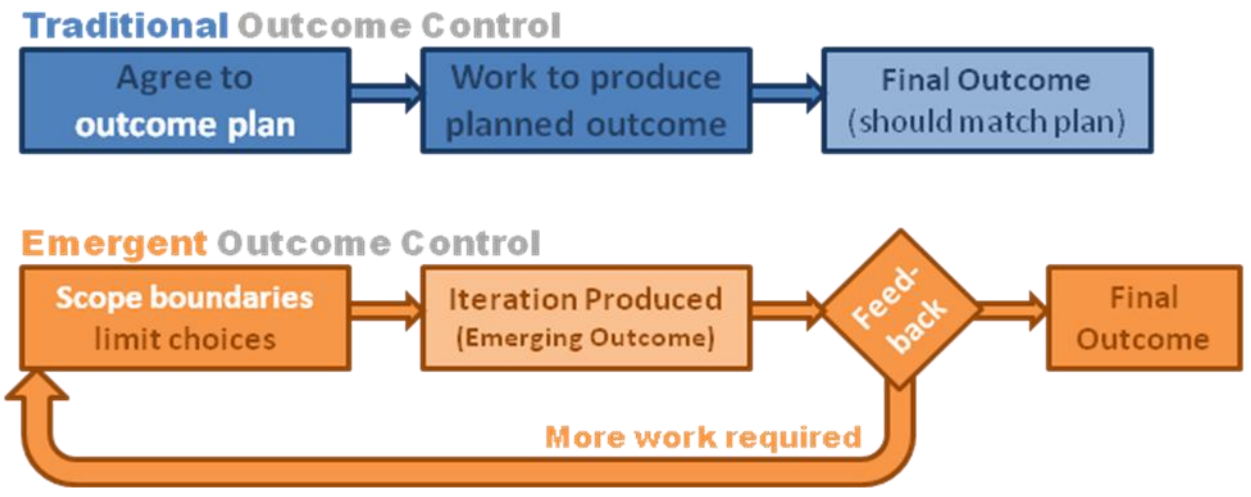
Output controls are akin to the cybernetic concept of control -- of which one simple example is the operation of a thermostat. Let's say that the target temperature for the thermostat is 72 degrees and the actual temperature is 78 degrees. The thermostat recognizes the actual condition is above the target (78>72), and it tells the system to add cool air until the measured temperature reaches the target level (72=72).

The visible target in outcome controls is not just something that is measured at the end of the project. The target is used to actively direct the work as it converges toward the target. Among other criticisms, Hofstede [1978] observes that there are problems with cybernetic control when objectives are missing, unclear, or shifting. Envision programming a thermostat to control the temperature so that it was "comfortable." Not only is this term undefined, but the definition might depend on the person(s) in the room, the activity level, the time of day and many other factors. Without an objective, an agreed-upon standard, the thermostat cannot work.

This is the problem with using output controls in flexible development – the final outcome is only known when the process is complete. Without a fixed, a priori output definition the development team will have no guidance during development, Furthermore, since there is no detailed plan, there is nothing that the final output can be measured against.

Because of these problems, traditional control theory would lead us to expect that managing outcomes would be ineffective in the control of flexible development. However, as we look at the agile manifesto and consider examples of flexible development, we see that the actual software developed is a primary concern of the control mechanisms: "We value: … *Working software* (emphasis added) over comprehensive documentation" [Agile Manifesto 2001].

The departure between actual flexible development and control theory is in the lifecycle state used to manage and control output (Figure 3). Traditional outcome control manages toward the final outcome. In contrast, flexible methods cannot and do not establish a final goal because the end game is unknown during development. Instead, flexible methods manage the output as it evolves or emerges from the production process. This difference is illustrated in Figure 3 which shows the focus of Traditional Outcome control on its last step (Final Outcome), and the focus of Emergent Outcome Control on the second step (Iteration Produced).


Traditional Outcome Control
Agree to outcome plan
Work to produce planned outcome
Final Outcome (should match plan)
Emergent Outcome Control
Scope boundaries limit choices
Iteration Produced (Emerging Outcome)
Feed-back
Final Outcome
More work required

Traditional outcome control focuses on the final outcome, whereas emergent outcome control focuses on the outcome as it continuously emerges or evolves during the development process.

Figure 3. Traditional Outcome versus Emergent Outcome

Emergent outcome controls guide the way that flexible processes evolve towards a final outcome (software deliverable). The developer is given freedom to create the best solution as new learning is uncovered; however, this freedom is constrained to ensure that the developer satisfies organizational objectives. The risk with flexibility is that the developer might wander off in unproductive or idiosyncratic directions. Later in this paper we will demonstrate how these controls work by classifying specific controls used by sample development processes. We propose two key types of emergent outcome controls: scope boundaries and ongoing feedback.

Scope boundaries: These controls limit the amount of flexibility available by defining the feasible space for exploration. The software vision is the crudest version of a scope control.

Although flexible methods do not use detailed plans, they may rely on software architectures and partial specifications to further define the set of feasible development choices. A partial specification can establish an architecture, or it can set APIs for external systems. A partial specification may also lock down some features, and leave others open for exploration. These partial specifications may exist as formal documents, but they may also be enacted through other means. Some examples: A legacy environment may constrain the available choices; the team might begin development with an architectural stub; and each team member may be confined to a specific area for exploration.

An alternative type of boundary limits the *amount,* not the area, of exploration. For example, consider the practice of delivering weekly releases. The team is limited to a week's worth of changes between software releases and reviews. Resource management can also be used to limit the amount of changes. For example, the number of team members assigned to a given area may vary based on the organization's needs.

Ongoing Feedback: Ongoing pervasive feedback between stakeholders is another type of emergent outcome control. Traditional feedback compares progress against a known plan, but the standard for comparison in emergent outcome control equates more to "I will know good work when I see it." Because of the tacit nature of this comparison, the team requires feedback in order to ensure progress is acceptable. In order to reduce the risk of wasted work, this feedback must be pervasive and it must occur continuously. Feedback occurs between team members and all stakeholders including: other team members (daily builds), user representatives, management, and the marketplace through frequent releases.

As a result of this discussion, we consider traditional outcome control to be insufficient for describing control of flexible processes. We propose the amendment of outcome control to include two separate classes of control: Traditional Outcome Control and Emergent Outcome Controls. Our classification expands Emergent Outcome Controls into the controls of scope boundaries and ongoing feedback. Scope boundaries do not specify details. Rather, they describe the set of feasible choices that can be made. This can be accomplished by constraining the environment (resources, API, tools) or by describing the feasible solution set. Feedback directly addresses the emergent outcome and its match against explicit or tacit models held by various stakeholders.

Behavior Control Mechanisms

Behavioral control focuses on behaviors that transform inputs to desired outputs. In the software development context this includes the specification of work methods and procedures [Henderson and Soonchul 1992]. Consistent with other types of control, behavioral control in software development is not as deterministic as Ouchi may have initially envisioned. Even with a plan-driven approach, two different developers who work on the same plan using the same methods would be expected to produce different software development artifacts. The variability would be even greater if a flexible method was used. This supports our contention that a single behavioral control may not be sufficient, and that control of flexible methods may require the use of a portfolio of overlapping controls.

Clan Control Mechanisms

As we discussed previously, there are two types of clan controls. Clan-Attitude controls achieve broad goal congruence through personnel selection, training, and socialization. The "clan" in attitude control refers to the common beliefs and values of the clan. There is no active control over tasks; rather, individuals are socialized to work in congruence with the clan's goals. This approach can be useful [Cardinal et al. 2004], but it is another instance of a control that cannot achieve repeatable results if it is used in isolation.

The other type of clan control involves more direct team control of tasks. Peer pressure from team members can affect the outcome of tasks. An example of this occurs in extreme programming, in which team members' activities are continuously visible. Everyone can see the progress being made by each person and there is an expectation that the team's work is unified, so that there are no mavericks and no surprises. Team clan control can keep the team members synchronized, but it will not necessarily keep the team in concert with users' needs. Again, this is an example of a useful, primarily internal control, but it cannot be used in isolation to match software to stakeholder needs.

Dynamic Control Taxonomy

In summary, our proposed control taxonomy retains the full richness of control theory with the addition of emergent outcome controls. The following taxonomy of dynamic controls is proposed:

Outcome controls: measure performance against a priori specifications

- Emergent outcome controls: manage outcomes in an evolutionary fashion
  - Scope boundaries: constrain creativity to insure focus on key areas
  - Ongoing feedback: provides corrective feedback as development occurs.

Behavior controls: measure adherence to behaviors that transform inputs to outputs

Clan controls

- Team: Team members provide task feedback to support coordination and communication.
- Attitude: Individuals make decisions based on attitudes and values that are congruent with organization's attitudes and values.

Even with the addition of emergent outcome controls, no single control will sufficiently constrain flexible development in order to achieve predictable, repeatable results. It is only through the use of an overlapping portfolio of the above controls that a flexible process can be effective.

V. DYNAMIC CONTROL AND FLEXIBLE PROCESSES

As a test of the concept of dynamic controls, we apply the control mechanism taxonomy to analyze the controls found in three popular flexible software development processes: Extreme Programming, Synchronize and Stabilize, and the Rational Unified Process. Our goal is to better understand where control mechanisms exist in these flexible processes in order to support improved management of flexible software development. For completeness, we also analyze the controls found in the Waterfall process. These analyses are depicted in Tables 2-6 in the Appendix at the end of the paper, with the control mechanism taxonomy developed from the control literature as categories for the key practices of each development process.

This analysis was done in two phases. First, the authors identified key practices for each development process from industry and trade literature sources as a representation of the methods in use. Second, the authors classified each practice mechanism into one or more parts of the control taxonomy to reveal the underlying type of control that was exercised. To further validate the author classification, we created two surveys, each with a description of the types of control and software practices and with half of the practices to classify (to minimize time to completion). In the surveys, Masters-level students with software development experience were asked to classify each practice into one, and only one, type of control, or to answer “Don’t Know”. Fourteen subjects completed these surveys. The results of the classifications by the authors and surveyed developers are shown in Tables 2-5 in the Appendix (the numbers indicate how many individuals made each classification; results are shown for any classification done by at least 2 individuals). The results of this analysis demonstrate how the different types of control are operationalized in development processes, while also identifying differences in control portfolios in the different development processes.

Controls in Extreme Programming (XP)

We used XP to guide development of the dynamic control taxonomy. The XP key practices [Beck and Andres 2005] are classified according to the taxonomy in Table 2. As the table reveals, the XP mechanisms are consistent with the expectations of the dynamic controls taxonomy. The emergent outcome mechanisms, behavior controls, and clan-team controls are the primary forms of control used by the process.

Although XP developers are given freedom to create new solutions, scope boundaries limit technological wandering [McDonough and Leifer 1986]. For example, XP iterations of one to three weeks limit the amount of change developers can introduce. XP also utilizes user stories to define broad requirements and further bound creativity. These mechanisms define the area within which developers will create the solution. Detailed feedback lets XP developers constantly gauge progress. A user co-located with the development team provides one source of continuous feedback. However, XP also realizes that a single user may not be representative of the market as a whole. Another important reason for the short release cycle is that it allows the team to continually check their evolving design against the needs of the broader market of users.

As suggested earlier, XP’s behavior controls all have a focus on immediacy. Daily builds mean that bugs must be fixed as they are found, not added to bug lists. Weekly releases mean that a one day slip can result in a 20 percent schedule overrun. Ten-minute builds make it quick to compile the system. This sense of immediacy works with the short cycle times to keep the team on course. Although the team has freedom to create, they do not have time for idle wanderings. Compare this to a large, plan-orientated approach where the next milestone may be weeks away. XP epitomizes the adage: ‘Don’t put off until tomorrow what you can do today’.

The analysis indicates that control theory with emergent outcomes provides a useful way of thinking about the activities of extreme programming.

Controls in Synchronize and Stabilize

The Synchronize and Stabilize (S&S) [Cusumano and Yoffie 1999] approach provides an interesting contrast to XP. In many ways, the S&S approach is similar to a plan-driven method. S&S has been used to manage large teams consisting of hundreds of developers. Projects begin much like plan-driven development methods. Functional specifications are built, major milestones are established, and feature teams of three to eight developers are established. Standards, such as user-interface design rules, are also set [Iansiti and MacCormack 1999]. However, these startup processes stop short of specifying every detail. About 30 percent of the product design evolves from the development activities of the feature teams. Since the 70 percent of fixed requirements include infrastructure items, such as the system architecture, the amount of freedom given to the feature teams is considerable. Each feature team is expected to discover and implement the best possible solution for their product area. Many structures are used to guide the developers, but many feature orientated decisions are left to the developers.

Two differences between S&S and XP relate to 1) architecture and 2) code ownership. S&S develops architecture before coding begins. The architecture constrains flexibility and ensures that any subsequent creative solutions are true to the overall system needs. XP uses a minimalist architecture. For XP, architecture is important but for each iteration, the architecture only supports features in that iteration. The XP philosophy is that planning an architecture for future iterations is a waste of time since the feature set will change. The second difference relates to the ownership of code. S&S carefully segments the code. Each feature team has ownership of only its own code segment. In XP, the entire team has ownership of the entire code base.

Despite these differences, S&S and XP are both considered agile processes. They both encourage the developers to pursue creative approaches and they both consider documentation an output of the process rather than an input. Thus, we would expect S&S to have similar characteristics to XP even though its mechanisms are quite different. Table 3 shows how the mechanisms in S&S map to control mechanisms.

The specification list provides a scope boundary; an emergent outcome control, not a fixed outcome control. This list provides broad outlines for development but it can change as development proceeds. Scope boundaries are also evident in other areas. For example, a feature team must fit their code into the established architecture. This architecture is not an outcome control since it does not dictate what the team must build. However, it determines how the team's code interacts with other modules and, thus, places boundaries on the extent of the changes that can be made. S&S also relies on role definitions to place boundaries on the teams. A feature team only has responsibility for a specific feature-set and ownership privileges are defined for each piece of code.

As can be seen, S&S's use of scope boundaries is different from XP's. XP relies primarily on short cycle times to reign in ad hoc development. XP developers don't have time to wander too far afield because the next release is no more than a few days away. S&S uses a partial specification, defined roles, a fixed architecture, and limits on code ownership to constrain developers. Both methods use daily builds to ensure that maverick programmers do not get too far out of step with the rest of the organization. In total, all of these mechanisms can be seen as alternative ways to bound the creativity of the team. Although the team is given permission to innovate, these mechanisms ensure that the innovation does not stray too far from the intent of the software.

S&S also includes mechanisms for ongoing feedback [MacCormack et al. 2001]. Multiple releases are offered to the market in order to gather user input. For example, Netscape 3.0 underwent six beta releases to gather user input before it was released to market [Iansiti and MacCormack 1999]. Furthermore, the team takes any chance possible to get continuous end-user reviews of the work in progress. This ongoing user feedback is as important as functional/bug testing.

S&S is not quite as 'extreme' as XP. There is still a sense of immediacy derived from daily builds and testing that occurs in parallel with development. However, this sense is muted. Although software is built daily, an individual piece of code may go several days before being checked into a build. Since there is no user representative on each team, the feedback is less current. Slightly more structure upfront allows for a greater use of traditional outcome controls. However, this is a relative comparison between the two techniques. In general, the profile of controls is the same. They both rely primarily on emergent outcome controls, they both keep a sense of immediacy in their behavior expectations, and they both include team oriented clan controls.

Controls in the Rational Unified Process

The Rational Unified Process (RUP) [Kruchten 2000] provides an interesting contrast in our study because it is a plan-driven approach, but it allows for learning throughout the process. In a plan-driven process, such as RUP, the specification of the software occurs separately from the software development. Creation of the design either occurs at the front of the process (Waterfall) or in spurts between development cycles (RUP). The programmer executes the plan but does not directly modify the plan. In contrast, a flexible environment requires the programmer to make real time design decisions. The RUP developer will consider both the user needs and the technical capabilities of the environment.

When we talk about emergent outcomes we are describing a continuous design process that involves the programmer in design decisions. Although RUP does allow for evolving designs, it does not allow for emergent designs. RUP design decisions are agreed upon by a change committee before a programming iteration begins. The programmer implements the design. Any creative suggestions must wait for the committee's action and for the next iteration.

Table 4 maps RUP development techniques onto the control taxonomy. The table shows fewer controls for RUP as compared to the agile processes. This is because of the central role of the design document in RUP. RUP includes business modeling plans, architecture plans, and a formal change management board. The results of all of these processes are embedded in the design specification. Therefore, the inclusion of this single document factors in the decisions of all of these processes.

The two shaded rows at the bottom of the table indicate that specific aspects of RUP are adaptive. However, this ability to make changes is in the hands of project management, not the developers. Between each set of iterations the change management board can adjust the controls by changing the specification.

From the developer point of view, RUP is not as flexible as the agile processes XP and S&S. As the table reveals, RUP leans heavily on traditional outcome control. Developers are assigned due dates and detailed specifications for each iteration. Testing helps determine if the specified outcome is successfully delivered. Certain behaviors, in terms of tools and standards, are also pre-specified.

Controls in the Waterfall Process

The previous discussion contrasted two types of agile methods (XP and S&S) and demonstrated that despite their differences, they contained similar approaches from the viewpoint of dynamic control theory. It then revealed the dynamic control profile for an iterative, plan-driven approach (RUP). As a next step we analyze the Waterfall development process using the same approach. This will help us understand how dynamic control theory distinguishes between the agile and plan-driven approaches.

Table 5 shows how the mechanisms of the waterfall method influence development. The most significant control again is the design document. This document encapsulates the results of many detailed sub-processes in one formidable control tool.

As can be seen, the waterfall approach has a significantly different control profile from the agile approaches. The waterfall approach relies primarily on traditional outcome controls. The flexible methods (XP and S&S) use a more broad-based set of controls with special emphasis on emergent outcome controls.

VI. CONCLUSIONS AND FUTURE RESEARCH

This initial study of controls in flexible software development processes demonstrates that traditional control theory has shortcomings when applied to dynamic situations, such as new software development. Traditional control theory relies heavily on clan control for these situations. However, even if clan control is possible, it is not clear that it is sufficient. The analyzed agile processes do use clan control, but they supplement it with other types of control.

The analysis suggests that extensions to control theory are needed to understand control mechanisms in dynamic situations. Specifically, it recommends the addition of emergent outcome controls. This new control mode consists of two key mechanisms. Scope boundaries define the limits on the developer’s creativity. Ongoing feedback is used to steer the creative process. In addition, the study recommends a re-categorization of informal controls. Self control becomes clan-attitude control. It involves creation of shared attitudes and values across the clan. Clan-team control is created to capture the concept of intra-team coordination of tasks.

Our analysis demonstrates that the new dynamic control taxonomy can be used to classify flexible control mechanisms. This allows researchers and practitioners to understand the relationships between controls in various development processes. For example, consider the finding that one purpose of the short cycle times in XP is to limit the scope of development to minimize technological wandering. S&S does not use short cycle times, but it places scope boundaries via carefully defined developer roles that restrict developers to specific feature areas. An organization that is developing its own flexible development process may consider one of these scope limitation devices or they may come up with an approach of their own to achieve the same purpose. See Table 6 for a synthesis of control mechanisms by control types and development processes.

Our analysis also supports the portfolio view of controls of software development. All the development processes analyzed employ more than one category of control. Two of the more flexible processes, XP and S&S, use many more types of controls than RUP and the Waterfall process do. This finding underscores the importance of understanding the nature of controls in a theoretical way. Managers adopting a flexible process need to be able to deploy a varied and sophisticated set of control mechanisms with a clear understanding of how and why they are using each key control mechanism.

A common feature of the agile processes is the manner in which they create a sense of immediacy. Developers are required to be ever ready to build and demonstrate the system. This creates continuous pressure on developers to perform. In contrast, some plan-driven approaches establish their pacing through the use of infrequent milestones. Developers may relax when the milestones are first established and gradually increase their pace as the due dates approach.

Clan-team control is also a key practice of flexible methods. Unlike clan-attitude control, this team based approach does not require lengthy socialization. Team control involves team members directly interacting to coordinate and influence each other's tasks. Since this is more directive than attitude control, it can even be used in relatively new teams. However, this type of control mechanism is likely to be more difficult in a distributed setting when interaction is not face to face but via a communication medium. This type of control may also be more challenging when members of the team come from different organizations and cultures (e.g. when consultants are used, or when development is partially outsourced to different countries). More research is clearly needed to understand how best to apply clan-team controls in distributed software development environments.

We note two limitations to the analysis reported in this paper of how software development practices embody various types of control. First, the list of software development practices analyzed for each development approach is not intended to be exhaustive, but rather representative of the most commonly used practices of each approach. The goal here is to demonstrate that different types of control are used in a reasonably large set of development practices. In particular, we study well known controls in four commonly used development approaches. Second, the survey of developers is used here to investigate the proposed model of software development controls. The results show clear support for the authors' conceptualization of control types and the authors' classification of a range of software development practices into one or more types of control. Further empirical work is needed to fully validate the taxonomy of control mechanisms and the proposed extensions to control theory.

This study provides an initial test of the new conceptualization of control mechanisms in the software development context by analyzing XP, S&S, RUP, and the Waterfall process. As future research we plan to validate this taxonomy of control mechanisms in an extended field study of industrial software development projects. This will allow us to verify the role of emergent outcomes in controlling development, establish the distinction between clan-team and clan-attitude controls, and explore any boundary conditions that might suggest when the uses of emergent outcomes are more or less appropriate.

ACKNOWLEDGMENTS

The Associate Editor review led to several significant improvements in the final paper. An earlier version of this paper was presented at the 2006 HICSS Conference [Harris et al. 2006].

REFERENCES

Editor's Note: The following reference list contains hyperlinks to World Wide Web pages. Readers who have the ability to access the Web directly from their word processor or are reading the paper on the Web, can gain direct access to these linked references. Readers are warned, however, that:

1. These links existed as of the date of publication but are not guaranteed to be working thereafter.
2. The contents of Web pages may change over time. Where version information is provided in the References, different versions may not contain the information or the conclusions referenced.
3. The author(s) of the Web pages, not AIS, is (are) responsible for the accuracy of their content.
4. The author(s) of this article, not AIS, is (are) responsible for the accuracy of the URL and version information.

Agile Manifesto (2001). Manifesto for Agile Software Development, http://agilemanifesto.org, published 2001.

Beck, K. and C. Andres (2005). *Extreme Programming Explained: Embrace Change, 2nd ed. XP Series* Boston: Addison-Wesley. p. 189.

Boehm, B. and R. Turner (2004). *Balancing Agility and Discipline: A Guide for the Perplexed* Boston: Addison-Wesley.

Cardinal, L. B., S. B. Sitkin, and C. P. Long (2004). "Balancing and Rebalancing in the Creation and Evolution of Organizational Control," *Organization Science* 15(4), p. 411-431.

Choudhury, V. and R. Sabherwal (2003). "Portfolios of Control in Outsourced Software Development Projects," *Information Systems Research* 14(3), p. 291-314.

Cusumano, M. A. and D. B. Yoffie (1999). "Software Development on Internet Time", *IEEE Computer* 32(10), p. 60-69.

DeSanctis, G. and M. S. Poole (1994). “Capturing the Complexity in Advanced Technology Use: Adaptive Structuration Theory,” *Organization Science* 5(2), p. 121-147.

Eisenhardt, K. M. and B. N. Tabrizi (1995). “Accelerating Adaptive Proceses: Product Innovation in the Global Computer Industry,” *Administrative Science Quarterly* 40, p. 84-110.

Harris, M., A. Hevner, and R. Collins. “Controls in Flexible Software Development”, *Proceedings of the 39th Annual Hawaii International Conference on System Sciences* (HICSS39), Hawaii, January 2006.

Hayes, F. (2005). “$170 Million Lesson,” Computerworld, http://www.computerworld.com/governmenttopics/government/story/0,10801,100335,00.html (current May 6, 2006).

Henderson, J. C. and L. Soonchul (1992). “Managing I/S Design Teams: A Control Theories Perspective”, *Management Science*, 38(6): p. 757-777.

Hofstede, G. (1978). “The Poverty of Management Control Philosophy,” *The Academy of Management Review* 3(3), p. 450-461.

Iansiti, M. and A. MacCormack (1999). “Living on Internet Time: Product Development at Netscape, Yahoo!, NetDynamics, and Microsoft,” *Harvard Business School Case Stud*, 9-697-052, pp. 12.

Kirsch, L. (1996). “The Management of Complex Tasks in Organizations: Controlling the Systems Development Process,” *Organization Science* 7(1), p. 1-21.

Kirsch, L. (1997). “Portfolios of Control Modes and IS Project Management?,” *Information Systems Research* 8(3), p. 215-239.

Kirsch, L. J. (2004). “Deploying Common Systems Globally: The Dynamics of Control,” *Information Systems Research* 15(4), p. 374-395.

Kruchten, P. (2000). *The Rational Unified Process: An Introduction, 2nd edition,* Reading, MA: Addison-Wesley.

MacCormack, A., R. Verganti, and M. Iansiti (2001). “Developing Products on ‘Internet Time’: The Anatomy of a Flexible Development Process,” *Management Science* 47(1), p. 133-150.

McConnell, S. (1996). *Rapid Development*, Redmond, Washington, Microsoft Press.

McDonough III, E. F. and R. P. Leifer (1986). “Effective Control of New Product Projects: The Interaction of Organization Culture and Project Leadership,” *Journal of Product Innovation Managemen*, 3(3), p. 149-157.

Neill, C. J., and P. A. Laplante (2003). “Requirements Engineering: The State of the Practice”, *IEEE Software,* November-December pp. 40-45.

Nidumolu, S. R. and M. R. Subramani (2004). “The Matrix of Control: Combining Process and Structure Approaches to Managing Software Development,” *Journal of Management Information Systems* 20(3), p. 159-196.

Orlikowski, W. J. (1991). “Integrated Information Environment Or Matrix of Control? The Contradictory Implications of Information Technology,” *Accounting, Management & Information Technology* 1(1), p. 9-42.

Ouchi, W. G. (1977). “The Relationship Between Organizational Structure and Organizational Control,” *Administrative Science Quarterly* 22(1), p. 95-113.

Ouchi, W. G. (1979). “A Conceptual Framework for the Design of Organizational Control Mechanisms,” *Management Science* 25(9), p. 833-848.

Ouchi, W. G (1980). “Markets, Bureaucracies & Clans,” *Administrative Science Quarterly* 25(1), p. 129.

Ouchi, W. G. and J. B. Johnson (1978). “Types of Organizational Control and Their Relationship to Emotional Well Being,” *Administrative Science Quarterly* 23(2), p. 293-317.

Ware, L. C. (2004). “The Benefits of Agile IT,” CIO www2.cio.com/research/surveyreport.cfm?id=74, (current September 5, 2005).

Zelkowitz, M. V. and D. R. Wallace (1998). “Experimental Models for Validating Technology,” *IEEE Computer* 31(5) p. 9.

APPENDIX: CONTROL TABLES

The tables begin on the next page. This introduction explains the layout of the tables.

Fourteen experienced subjects completed surveys to classify key development practices according to the type of control that was represented. The results of the survey are shown in the following tables.

- Each table represents a development method (XP, Synchronize and Stabilize, RUP, Waterfall).
- Each row is a key practice for the given development method (e.g. develop a specification).
- The table columns represent the types of controls that could be used. Note the inclusion of Emergent Outcome Controls, represented by Scope Boundaries and Ongoing Feedback.
- Each cell indicates the number of respondents who felt that a given key practice was an example of a given control type. For example, the first row of the XP table shows that three people felt that “Sit Together” was an example of “Ongoing Feedback”, and that eight people thought it was an example of a “Clan-Team” control.
- The responses are only shown for cells that have more than one respondent.
- In order to keep the survey to manageable size, it was split into two parts. Each respondent only saw half of the possible key practices. As a result, the number of respondents is not identical for all practices.

Table 2. Extreme Programming Controls

	Outcome Controls			Behavioral Controls	Clan Controls	
		Emergent Outcomes				
	Traditional Outcome	Scope Boundaries	Ongoing Feedback		Clan – Team	Clan – Attitudes
Sit Together			3 Progress observable by teammates		8 Teammates accessible for advice	
Whole Team			4 Feedback from customer representative		6 Team is unified. No mavericks.	
Informative Workspace	6 Visible results		10 Outcomes observable	3 Daily work observable	3 Teammates see each other's progress	
Energized Work				4 Work at fast pace		6 When we are at work we will do our best
Pair Programming			4 New ideas tested with partner	4 Work together	10 Activities transparent to team members	
Stories	2 Describe actions to be supported	8 Stories are broad targets -- not detailed	2[2]			
1-3 Week Cycle	2 Know that release is very 1-3 weeks	8 Limit change that can occur in an iteration	5 Market feedback every 1-3 weeks	3 Urgency ➔ even small delays change schedule		
Quarterly Cycle		4 Focus on Business constraints	5 Feedback of Management Issues			
Slack		8 Timing constraints restrict options		3 Weekly cycles more important than features		
10-Minute Build			4 Easy to demonstrate	4 Don't break the system	3 Code must work well with others	
Continuous Integration			6 Always be ready to demo latest product	6 Fix bugs as they occur	3 Maintain synch. with others. No surprises	
Build test cases first		3 Develop a detailed goal for each feature			4 Team agrees to goals before programming begins	
Incremental Design		6 Each iteration focuses on a few things	6 Iterations dem-onstrated to stakeholders	3 Don't over plan for future features you may not need		

[2] Examination of the question and consideration of the responses has led us to conclude that these two responses indicate a problem with the phrasing of the question, and not necessarily a valid response.

Table 3. Synchronize and Stabilize Control Mechanisms

	Outcome Controls			Behavioral Controls	Clan Controls	
		Emergent Outcomes				
	Traditional Outcome	Scope Boundaries	Ongoing Feedback		Clan – Team	Clan – Attitudes
Start with Vision	5 Can-provide some control – lacks measurable detail	5 Carves out area of development	4 Provides some basis for comparison			
Up to 70% specified before development	3 Partial specification may change – accountability is difficult	6 Although any feature may change, majority of features set	3 Subsequent 30% based on feedback			2 Workers fill in missing pieces based on their interpretation
Architecture & Feature team assignments first		4 Each team focuses on its own area				
Daily Builds			7 Progress Visible	3 Must keep code in working condition	3 Dunce cap if break the build	3 Each worker chooses when to add to daily build
Continuous End-User Reviews			8 Focused on delivered functionality, not code			
Major Milestones	5 Specify due dates for feature bundles	3 Major milestones only specify broad targets		2 Must plan for broad stakeholder review		
Development & Testing done in parallel		2 Development bounded by test constraints		4 Always keep code in working condition		
Code Reviews			3 Independent view of progress		8 Code must be acceptable to other team members	4 Developers share work without being defensive
Check-in and Change Tracking		4 Limit who can change a section of code at any given time				

Table 4. Rational Unified Process Control Mechanisms

	Outcome Controls			Behavioral Controls	Clan Controls	
		Emergent Outcomes				
	Traditional Outcome	Scope Boundaries	Ongoing Feedback		Clan – Team	Clan – Attitudes
Developer Controls: developers are controlled using a plan-driven approach. They do not have much flexibility on a daily basis. The ability to adjust direction comes between iterations as shown in “Team Controls” below.						
Design Document	8 Deliver to the specification; Proposed changes make later iterations					
Iteration Due Dates	5 Measurable outcomes that can be tracked	2 Change based on outcome of prior phase				
Testing Workflow	7 Makes outcomes visible		2 Assess whether project is satisfying requirements	3 Sets how testing will be done		
Environmental Workflow		2 Limited to given set of tools		6 Set tools and standards		
Configuration Manager		6 Define code sections a developer can access. Sets rules for sharing code.				
Team Controls: The two rows below are related to the team’s management., Developers are subject to non-flexible plan-driven controls on a daily basis. The flexibility of the process comes from adjustments made between iterations.						
Multiple Iterations		4 Limit changes for each iteration to specific areas	2 Feedback given at each iteration			
Stakeholder Reviews			11 Provide feedback throughout the project			

Table 5. Waterfall Process Control Mechanisms

	Outcome Controls			Behavioral Controls	Clan Controls	
		Emergent Outcomes				
	Traditional Outcome	Scope Boundaries	Ongoing Feedback		Clan – Team	Clan – Attitudes
Design Specification	7 Deliver to specification established at the beginning					
User Participation in Design Specification	2 Specify deliverable before development					
Project Due Dates	6 Must deliver at times established at the beginning					
Project Budget	6 Set budget at beginning of project	3 Constraint is budget amount				
Unit Testing	2 Done based on test plans and expected outcomes	2 Boundary is unit tested				
Integration Testing	4 Done based on test plans and expected outcomes	2 Limitation on testing space				
Testing	11 Make outcomes visible. Focus on technical issues, not end-user acceptance					
Environment				9 Determine tools and standards that the implementer must use		
Configuration Management		5 Define part of code developer can access. Set rules for sharing			2 Code sharing and coordination	
Deliverable at end of Phase	3 Known items to deliver for each phase		2 Assess progress against budget, schedule, and resources			
Sign-off at end of Phase	2 Measures delivery against specification		2 Provides feedback on acceptability			

Table 6. Synthesis of Control Methods by Control Types and Development Process

Control Types/ Development Processes	XP	S&S	RUP	Waterfall
Outcome Controls				
Traditional Outcome Controls	**Visibility** of results through an informative workspace	1. **Partial specification** that lacks detail and may change 2. Due dates for **feature bundles**	1. Delivery is to the **design document specification** 2. **Measurable due dates** that are tracked 3. Testing workflow makes **outcomes visible**	1. Delivery is to the **design document specification** 2. **Delivery time** is established at beginning 3. **Testing** for technical issues makes outcomes visible 4. Formal **sign-off** on delivery
Emergent Outcomes – Scope Boundaries	1. **Time**: limited, by short iterations (1-3 weeks, quarterly) 2. **Breadth** of target in overall goal, but **Detailed** goal for each feature that emerges via test cases	1. **Areas of development** established 2. Feature specification may **change** 3. **Team matched to development area**, limits on who can change a section of code	**Code changes limited:** 1. Which area a developer can access and how code is shared 2. By iterations, by area	**Code changes limited** to which area a developer can access and how code is shared
Emergent Outcomes – Ongoing Feedback	Feedback through: 1. **Observability** of outcomes to teams & customers 2. **Continuous testing** with teammates 3. **Continuously demonstrable** to customers	**1. Progress visible** 2. Progress **compared to initial vision** 3. **Independent review** of progress through code reviews	**Stakeholders review** to provide continuous feedback	Assess progress versus **budgets** at major milestones
Behavioral Controls				
Behavioral Controls	1. **Work pace**: fast with a sense of urgency **2. Collaborative** 3. **Weekly schedule** goals 4. **System always working**, bugs fixed immediately	**Code** must always be kept in **working condition** via daily builds and concurrent coding and testing	**Set tools and standards**	**Set tools and standards**
Clan Controls				
Clan - Team	1. **Teammates**: accessible for advice, unified, no mavericks, synchronized, no surprises 2. **Transparency** of activities to others in team 3. **Code** works well with others in 10 minute builds and continuous integration	1. **Code reviewed** by other team members 2. **Visible sanction by teammates** (dunce cap) if team member breaks the build		**Code sharing**
Clan – Attitudes	**Work Value**: When we are at work, we do our best			

ABOUT THE AUTHORS

Michael L. Harris is an assistant professor of Business Administration at Indiana University Southeast. He held managerial positions in the software field for 20 years before earning his Ph.D. in Business Administration from the University of South Florida. Dr. Harris' research interests include management of innovation, organizational change, knowledge management, and entrepreneurship. He has refereed research in *ISR (forthcoming), Communications of the ACM*, *ICIS*, *HICSS*, *AMCIS*, *SMA*, and *ISOneWorld*.

Alan R. Hevner is an eminent scholar and professor in the Information Systems and Decision Sciences Department in the College of Business at the University of South Florida. He holds the Citigroup/Hidden River Chair of Distributed Technology. Dr. Hevner's areas of research interest include information systems development, software engineering, distributed database systems, healthcare information systems, and service oriented computing. He has published more than 150 research papers on these topics and has consulted for a number of Fortune 500 companies. Dr. Hevner received a Ph.D. in Computer Science from Purdue University. He has held faculty positions at the University of Maryland and the University of Minnesota. Dr. Hevner is a member of ACM, IEEE, AIS, and INFORMS.

Rosann Webb Collins is an associate professor of Information Systems and Decision Sciences at the University of South Florida. Her current research focuses on global information systems, systems development, the impact of information technologies on work, and behavioral issues in information markets. Her publications include a book, *Crossing Boundaries: The Deployment of Global IT Solutions*, and research articles in *MIS Quarterly*, *Information Systems Research*, *IEEE Transactions on Software Engineering*, the *Journal of the American Society for Information Science*, and other MIS and information science publications. Dr. Collins has consulted with numerous businesses, community organizations, libraries, and educational organizations on information technology use and issues. She is a member of Beta Gamma Sigma, AIS, ACM, and IEEE.

ISSN: 1529-3181