# Best Practices in Agile Software Development

**Article** · April 2006

CITATIONS
2

READS
3,339

**2 authors:**

Steven R. Haynes
Pennsylvania State University
**47** PUBLICATIONS **376** CITATIONS
SEE PROFILE

Marc Friedenberg
Pennsylvania State University
**6** PUBLICATIONS **53** CITATIONS
SEE PROFILE

# Best Practices in Agile Software Development

Steven R. Haynes, Ph.D.
College of Information Sciences & Technology
The Pennsylvania State University
shaynes@ist.psu.edu

Marc Friedenberg
College of Information Sciences & Technology
The Pennsylvania State University
marcf@psu.edu

**Technical Report No. 0018**
**College of Information Sciences and Technology**
**The Pennsylvania State University**
**May 19, 2006**

## Abstract

This report describes an investigation into best practices in *agile* software development. Agile software development methods represent the latest incarnation of software engineering methods designed to be user-centric, low cost, and focused on rapid delivery of high-quality software systems. This survey of empirical and theoretical studies of agile methods provides a summary of the practices found to be most effective in promoting the creation of high quality software systems using relatively flexible prescriptions in the software development process. Our review of agile development literature, case studies, and empirical findings leads us to a number of conclusions, some of which run counter to the tenets of agile development. First, organizations should take an adaptive, pragmatic approach to an agreed-upon methodology, one that carefully and continually considers how and why elements of agile development may, or may not work well in different project and team contexts. Second, sustained, constructive customer involvement in development projects is a key factor in project success. Third, system development organizations and teams should focus on closely monitoring and managing the *tempo* of development projects. Core writings in agile methods promote rapid development and small iterations as critical to project success, but published experiences suggest that the appropriate tempo for a given project is always dependent on a range of project, team, and context-specific factors. Finally, the surveyed work suggests that agile development's lack of an architectural focus often leads to poorly designed software systems, and a level of disorientation on the part of development team members.

# Introduction

As in any design discipline, deciding on the best approach to software development methods involves resolving a number of trade-offs. A project management adage says that you can have any product designed and developed to be good, fast, and cheap, as long as you can live with only ever achieving two of these criteria. As a discipline, software systems development is immature and its brief history may be viewed as the struggle to control software development costs while at the same time delivering high quality systems that meet the needs of system users and other project stakeholders. A range of methods, or collections of methods bundled as methodologies, have been proposed to meet these objectives in some reasonably predictable manner. These methodologies range from highly prescriptive, phase/deliverable focused processes (Humphrey, 1990) to simple development heuristics and guidelines designed to create a professional atmosphere conducive to high quality intellectual work (DeMarco & Lister, 1987).

The current status of software development methodology pits "traditionalists" against those committed to more flexible or agile approaches to building software systems (Boehm, 2002). Traditionalists argue that a carefully defined and followed set of planned, discrete tasks is best suited to ensuring predictable, successful software development projects. Adherents of the recently emerged agile software development (ASD) approaches argue *against* this rigid specification of knowledge work and in favor of methods that explicitly allow for the intellectual and creative nature of software design, while at the same time prescribing a set of practices to help achieve working software within constraints of time and cost. Among the most compelling arguments for agile methods is that they explicitly acknowledge how much can change between initial requirements analysis and final delivery of a working system. Business and market operating environments, project stakeholder goals and priorities, and technical standards and constraints are constantly in flux. The objective of the modern systems development team is to design and employ a software development process sufficiently adaptive to this dynamic environment while at the same time capable of predictable and repeatable system delivery success.

Software system development has been characterized as a continual cycle of analysis and experimentation (Mathiassen & Stage, 1992). Developers employ analytic techniques (e.g. modeling and specification) to reduce the complexity of a given problem space through functional decomposition, which results in the identification of abstract entities, attributes, and behaviors at successively more fine-grained levels of detail. However, this process of analysis and decomposition results in increased uncertainty as the development team makes decisions about what will be included in the system and how these entities will behave, and, importantly, what will be excluded from the scope of the design. Each such analytic decision raises the question: *have we got it right*? One way to answer this question is to employ experimentation, often referred to as prototyping in the software domain, to determine whether the decisions made result in a design that more or less accurately reflects users' functional requirements in the application domain. Agile methods represent a skewing of this analysis-experimentation dichotomy towards that of experimentation. Implicit in the approach is that analysis of complex domains is of marginal utility when the critical features of the domain change and evolve with respect to the organization's requirements and its operating environment. Agile methods therefore focus on delivering 'good-enough' functionality quickly, allowing development teams to avoid implementing complete, but obsolete specifications and to evolve systems that better map to dynamic organizational priorities.

### *Study Method*

We survey the state-of-the-art in agile development methods to derive a set of best practices to guide creation of a corporate methodology. These best practices are gleaned from two sources: empirical studies of agile methods in action and theoretical writings from industry experts. While the former is our preferred source of data as it is grounded in the practical application of ASD techniques, we carefully selected from the writings of some highly expert researchers and developers, such as Barry Boehm, Robert Glass, and Kent Beck, based on their previous successes as analysts and predictors of how the software development industry would evolve. We reviewed about 30 papers and derived from these those aspects of ASD found to be most effective in practice and those mitigating factors that diminish or inhibit success with ASD methods.

### *Report Structure*

Following this introduction, the report first provides a brief overview of agile software development with a particular emphasis on *eXtreme Programming*, arguably to most pervasive of the agile methods. The next part reports on findings from our review of the literature in agile software development and presents a set of best practices derived from our reading. This is followed by a critical discussion of what appear to be the most important aspects of ASD and the steps that can be taken by development organizations to help ensure project success. The final section presents our conclusions and some recommendations for how to best employ ASD *adaptively* to projects with different levels of risk.

## Agile Software Development

Agile software development is an approach to building systems that emphasizes evolutionary development, customer-centricity, and low-documentation/specification overhead. Agile methods are often further defined with respect to what they are not, traditional, rigid, plan-based approaches typified by the so-called waterfall model. Though there are numerous methodologies that fit loosely under the agile umbrella (see for example Marchesi, 2003), this report uses the term *agile software development*, or *ASD*, to refer to both the shared *ethos* of these different methods, with specific techniques drawn from many and with particular focus on the most popular ASD, extreme programming (Beck, 2000). The central tenets of extreme programming are provided below as an exemplar for the ASD approach.

**Table 1 - Tenets of Extreme Programming, an Exemplar Agile Method (adapted from Beck, 2000)**

| ***Tenet*** | ***Description*** |
|---|---|
| *The Planning Game* | Continually plan the next (small) development iteration or release. The plan should closely integrate organizational objective and priorities with technical specifications and constraints. Planning is driven by user 'stories'. Customers and developers work together to write, estimate, and then prioritize the stories that will be realized in the system. Implementation schedules derived from stories are constantly renegotiated by the development team and customer representatives as new requirements are identified and/or technical or other issues cause schedule delays. The number and granularity of the stories is used as the major input to system release planning (see below). |
| *Small releases* | Keeping software release sizes small contributes to project success in a number of ways. First, it focuses development team members on delivering a small piece of working software in a short amount of time. Second, it limits the downstream damage of design or coding errors by applying unit testing and customer verification to the each mini-deliverable (Kivi, et al., 2000). Small releases are also significantly easier to estimate though issues arise when developers attempt to bid on large system projects using ASD as the basis for the estimates (Pelrine, 2000). |

| ***Tenet*** | ***Description*** |
|---|---|
| *Metaphor* | Architecture and design is driven by a simple story of how the system will work. The differences between metaphor in the XP sense and architecture as is usually defined for software development is the simplistic and narrative form that the metaphor. The metaphor describes what the system is all about, what it is 'like'. |
| *Simple design* | Design for XP development should be as minimalist as possible while still meeting the functional requirements of the user story that drives its purpose. 'Gold plating' and elaborate abstractions built to support *envisioned* but not expressed future requirements are to be avoided. |
| *Testing* | In contrast to its reputation as an undisciplined development methodology, XP strictly prescribes especially two kinds of tests: "white box" unit tests written by developers before coding is started and "black box" acceptance or functional tests written by customer champions, again before any actual software development has begun. |
| *Refactoring* | Programmers continually review designs and code to assess whether either can be made simpler while still meeting the functional and performance requirements expressed by the customer. |
| *Pair programming* | Developers work in teams of two on a single workstation. The idea behind pair programming is to embed often-skipped code reviews into the development process and to ensure that while one developer is 'driving', the other is thinking about possible alternative designs, how the code being written might be improved, and how it might fail. |
| *Collective ownership* | The entire code base for a given project is open to all development team members to review and to change. The idea of individual ownership in the sense of the original developer having sole rights to changing the code he/she has written, is eschewed in favor of a collective approach to code ownership, change rights, and accountability for code quality. |
| *Continuous integration* | Code units are continually integrated into a compile-able version of the system under construction. Integration of a |

| ***Tenet*** | ***Description*** |
|---|---|
| | programming pair's code occurs at least every day. Unit and acceptance tests are applied at each integration to ensure that the current build of the system is capable of passing all pre-specified tests in its current form. |
| *40-hour week* | Members of the development team are encouraged to work within their cognitive and physical limits. 'Hero' programming and 'death march' projects are avoided by ensuring that each developer can face each day's challenges with professionalism, deliberation, and care. |
| *On-site customer* | Every development project has at least one full-time representative drawn from the customer/user population. This customer is responsible for assisting with development of the system metaphor, with providing input to release plans, and with writing and executing acceptance tests to ensure the evolving system is meeting functional requirements at adequate levels of quality and performance. |
| *Coding standards* | The pace of short release iterations and the collective ownership approach of XP requires that developers adhere strictly to agreed upon coding standards. This ensures that programming pairs can be formed without the added overhead of ad hoc standards development and that members of the development team can easily review any code unit for design simplifications or for reuse opportunities. |

A higher-level set of attributes is provided as part of the Agile Manifesto (www.agilemanifesto.org) and consist of the following four dichotomies:

"*Individuals and interactions* over processes and tools"

"*Working software* over comprehensive documentation"

"*Customer collaboration* over contract negotiation"

"*Responding to change* over following a plan"

These representative dichotomies can be summarized as describing software development as an innately people-oriented enterprise. Software system development for dynamic organizations is complex and so should focus on incremental delivery of software

releases that meet some pressing need with the simplest possible solution. Integration of customer representatives into the system development team is crucial to provide direct links to the source of system requirements as well as to ensure that accountability is correctly apportioned between the user group(s) responsible for identifying, clarifying, and prioritizing system requirements and the system development team responsible for realizing these requirements in working systems. Finally, agile software development focuses on the need for adaptability over that of the long-term predictability as suggested by detailed project plans.

One argument for the current popularity and on-going proliferation of agile methods is the success of the world-wide web and the fact that so many successful web applications appear to have been developed without many of the process constraints that typify corporate development of information systems. Many web sites have evolved page by page in response to customer and market demand. New pages can be developed in less than a day without the costly, analysis-design-build-test cycles that governed so much in-house information systems development in the past. Agile methods may be perceived as an attempt to codify and legitimize this system development approach with identified activities and the rationale for their success.

# Best Practices in Agile Software Development

The body of substantially detailed and rigorous studies of agile methods in practice is still relatively small and immature. Longitudinal studies of the sustained success (or lack thereof) of these methods over time is essentially non-existent. The many writings by industry commentators that do exist suggest that agile methods are in fact succeeding in practice, but these claims have yet to be substantially corroborated with reliable studies. Many of the empirical studies that have been done were performed using undergraduate or graduate students as the participant sample. The results of these studies, while interesting from the perspective of higher education, lack some of the ecological validity required to generalize their results to the domain of professional software development. There is however an emerging body of work that describes different short-term case studies of agile methods in practice. We reviewed this body of work to derive a framework of lessons learned that might be used to guide organizations setting off on the agile methods track. The resulting framework is intended as an aid to successful implementation of the agile methods approach.

The framework consists of a setoff practices ranging from the very high level, such as evolving an agile development culture, to more low level, detailed prescriptions, such as the consistency of coding standards. The framework is relatively simple and consists of the following practices:

- Evolve an adaptive development culture
- Ensure customer champion engagement
- Manage requirements
- Don’t ignore architecture…
- But do practice iterative development
- Paired programming and paired development
- Test, really, and practice test planning
- Develop coding standards, provide training in their use, and then enforce

We found this set of practices most pervasive in practice, and most compelling in terms of the effect their implementation may have on successful agile development. Each of these practices is described in more detail in the sections that follow.

### *An adaptive culture*

Much of the work reviewed suggests that creation of a successful agile development organization requires focus on creating a culture that embraces the tenets of ASD while at the same time working to adapt the methods and procedures to the organization's unique context (Lindvall, et al., 2002). As with anything new, aspects of agile development are often met with initial resistance, though this resistance fades as successes are achieved (Blotner, 2002). In at least one case researchers found that developers at first embrace ASD but then revert to old practices, e.g., failing to test their own code, when deadlines loom (Mueller & Borzuchowski, 2002). This suggests that early ASD projects be given extra slack time to allow practices to evolve and take hold within development teams. Managers should use early ASD projects to guide evolution of a methodology to fit the organization's culture (Blotner, 2002).

### *Customer champions*

Though not limited to agile development practices alone, customer representation on a software system development project is a crucial factor in their success (Boehm, 2002; Reifer, 2002; Elssamadisy & Schalliol, 2002). It's important that customer champions are assigned for the duration of the project and that they do not lose focus as the project progresses (Elssamadisy & Schalliol, 2002). Also essential is that customer champions be representative of the target customer/user base rather than proxies from groups less directly related to the application domain (Elssamadisy & Schalliol, 2002).

One study suggests using two customer champions: one who is a domain or subject-matter expert, and one a more senior person familiar with the high-level strategic priorities of the organization (Nawrocki, J., Jasinski, et al. 2002). The former is available to drive detailed requirements and answer queries related to the domain and the latter is responsible for mediating conflicts between customer groups and developers and for ensuring that the design and construction of the system tracks to higher level concerns within the organizations (e.g., infrastructure, strategy, etc.).

### *Manage requirements*

Central to the XP approach to ASD is that requirements documents consist primarily of user stories describing interaction scenarios. User stories are kept necessarily short and simple so that multiple user stories can potentially be delivered in a single development iteration (2-4 weeks). Early iterations of a project should include a special focus on building a productive customer-developer dialog. Developers focus on understanding the requirements of the customer and customers focus on understanding the practices and constraints of the developers (Martin, 2000).

The relatively 'loose' nature of the story card construct for requirements determination leads to dangers of miscommunication when story card granularity and level of analysis are not understood and specified consistently (Elssamadisy & Schalliol, 2002). Careful attention should be paid to development of a shared conception of what constitutes a story card and the level of detail at which one should be specified. A range of techniques can be used as the basis for an organization's story card writing method. Scenario-based design (Carroll & Rosson, 1992) is a technique close to the ethos of ASD that has been well-documented and successfully used in practice. The Unified Modeling Language includes use cases for modeling scenario classes and activity diagrams for specifying more detailed use scenarios in a widely accepted, graphical form.

The relative lack of documentation in ASD may result in losing key information related to requirements, despite the presence of an on-site customer. One study suggests that responsibility for requirements be assigned to one or more members of the test team (Nawrocki, J., Jasinski, et al. 2002). This provides a direct mechanism to verify that system releases meet the requirements specified for the development increment.

### *Don't ignore architecture*

Several works suggest that ASD's lack of focus on architectural issues can result in a range of project and organizational inefficiencies (Mueller & Borzuchowski, 2002; Glass, 2001; Kivi, et al., 2000; Newkirk & Martin, 2000; Elssamadisy & Schalliol, 2002). Developers and team architects should understand that story card implementations are *not* independent (Elssamadisy & Schalliol, 2002). Regardless of the lack of architectural focus, design of prior code units necessarily constrain design space options for

subsequent iterations. The need for true architectural work for large systems without stabile requirements presents a significant challenge to applying ASD to large projects (Boehm, 2002).

One case study resulted in the authors recommending that one member of the development team be responsible for the 'big picture' seeing the woods of system architecture for the trees of short, simple software releases (Mueller & Borzuchowski, 2002). However, it's important not to forget that too much planning for future, unsubstantiated requirements can radically increase project complexity and can offset some of the advantages gained from ASD (Hannula, 1999). When requirements are stabile, plan longer range than is normally dictated by the ASD. This allows the organization to take advantage of the economies gained through forward planning for architecture and reusability (Boehm, 2002).

Given the complexity of large scale system designs, expert developers should play the lead roles in architecture development. One study suggests that up to 80% of refactoring rework may be attributable to architectural mistakes caused by junior developers (Lindvall, et al., 2002). The same study suggests that for agile methods to succeed, 25%-33% of the team members must be "competent and experienced". Without an architectural focus, larger projects can end up with refactorings that are larger efforts than the small increments that initially led to the faulty design. Manage both the size of the software release AND the size of a refactoring to ensure teams don't pay later for rapid development today (Elssamadisy & Schalliol, 2002; Lindvall, et al., 2002).

### *Practice iterative development*

Short, iterative development cycles with small feature set releases is one of the tenets of ASD and one found to contribute much to the success of the method (e.g., Reifer, 2002). However, one study suggests that the pace of one week release cycles was too frenetic, resulting in low quality software deliverables. In this case the team managed the pace of development by increasing the release interval to two weeks. In addition, they found that specifying the activities within this two-week space using a "micro-waterfall" cycle helped ensure that all necessary steps (design-test writing-build-test implement) were followed for iteration (Blotner, 2002).

Though XP prescribes short intervals between software builds, in at least one case developers found that the overhead involved in integrating software modules into a build meant that these times needed to be lengthened. Training in and then enforcing of coding standards is one approach to simplifying the integrate-build task and for helping developers adapt to shorter release cycles (Mueller & Borzuchowski, 2002).

### *Paired programming & paired development*

Paired programming is a method of gaining the advantages of code reviews, advantages typically not realized because teams find it difficult to allot time to reviewing each others' code (Newkirk & Martin, 2000; Williams & Upchurch, 2001). Paired programming also helps to mitigate development staff turnover and knowledge management issues as at least two people have intimate knowledge of any unit of code (Beck, 1999). Experiments in academic programming suggest that pairs spend 42.5% fewer elapsed hours than individual programmers on the same task with only 15% increase in total person-time (Williams & Upchurch, 2001). The same study found 15% fewer defects in the code written by pairs. Though the lack of ecological validity in this study may mean that similar results are not achieved in industrial settings, it may suggest that especially novice programmers may prove more productive working in pairs.

Allow programming pairs to dictate how and the extent to which they pair for different tasks. One study has found a broad range of preferences among developers in terms of how they operationalize paired programming, which tasks they choose to work on as a team and which as individuals (Müller & Tichy, 2001). Some work together constantly while others divide tasks and synchronize their work as needed. Moving experienced developers to paired programming may result in failure if team members find the approach awkward (Wells, 1999).

Paired development goes beyond paired programming, and specifies that developers work on all aspects of the development task as a team. Although one of the most pervasive and controversial aspects of ASD, paired work is also one often identified as having a positive impact on project success, software quality, and organizational learning (Blotner, 2002; Greening, 2001; Hannula, 1999).

### *Testing and Test Planning*

A key tenet of XP is that developers write their test plans before writing code to implement the functionality tested by those plans. At least one study shows that this approach contributes to the development of higher quality systems (Newkirk & Martin, 2000). The tempo of ASD's short release cycle times can sometimes result in developers minimizing their time on testing (Mueller & Borzuchowski, 2002). This can result in the illusion of rapid progress but at the expense of high-quality software. Ensuring that developers test their code may involve providing them with specialty training and purchasing or building tools to support the testing process (Hannula, 1999). The unit test framework JUnit is often used to support test case development in agile environments (Müller & Tichy, 2001).

One practice found to contribute to testing success is when software components are not unit tested by their developers but are passed to another developer pair for testing to ensure quality (Blotner, 2002). This helps promote a culture of shared software ownership and ensures that development pairs deliver working code units for integration.

### *Coding Standards*

Lack of coding standards was identified as a significant impediment to short software build cycles (Mueller & Borzuchowski, 2002). This suggests providing developers with training and continuing to enforce standards that exist. Keeping coding standards as simple as possible is another method for ensuring that standards are followed. In one case the team adopted the simplest of standards: "make new code look like code that is already there" (Greening, 2001).

# Discussion

Our review of the agile software development literature leads us to suggest a set of focus points for agile developers and managers. These focus points consist of four high-level concerns that we think should form the basis for development methodology planning, implementation, and on-going management. These focus points are:

- *Adaptation*
- *Customer*
- *Tempo*
- *Architecture*

*Adaptation* refers to the idea of *evolving* a customized development methodology and the need to avoid dogmatism with respect to the organization's use of agile development methods. *Customer* refers to the critical issue of obtaining, sustaining, and leveraging customer champion representation on agile development projects. *Tempo* is the rate at which project activities progress. In many of the cases we reviewed, too fast a tempo resulted in a number of dysfunctional behaviors including dropping important aspects of the methodology (e.g., testing and test planning) and delivery of low quality software components and systems. *Architecture* refers to the importance of "seeing the wood for the trees", in other words, maintaining a strategic perspective on not only the immediate system under development but also has it fits with the organization's IT infrastructure and operating strategy.

### *Adaptation*

An important factor in adoption of ASD is being agile, or adaptive in the adoption process itself. It is increasingly recognized that methods should be tailored to various aspects of the development context (Fitzgerald, et al., 2003). Organizations should adopt an inclusive approach to the techniques and methods they package into a methodology. Strict adherence to a prescribed methodology only serves to discredit a development paradigm if it seen as too rigid or lacking a common sense perspective on what works. Some organizations have reported success incorporating techniques from traditional

software development methods, including tools and diagrams from UML, into an agile development setting (Greening, 2001).

Some projects and project components require more careful attention than others to ensure a high quality product. Assessment of how to apply stricter controls, in the form of additional methodology steps for example, should be driven by the risks posed both to and from a given software deliverable. In other words, difficult-to-design and difficult-to-build components, or those employing construction techniques unfamiliar to the development team, inherently involve greater risk than simpler, more familiar development tasks. However, because of architectural dependencies, both simple and complex components can pose a risk to both a project and to system users. In the case of safety-critical domains for example, architectural issues should be managed with special care to ensure high quality, reliable products of development. We propose that organizations adopting any methodology spend time considering how different aspects of the methodology relate to the risk profile of their operating environment and how these aspects can be tailored, relaxed, or enforced depending on the risk level of a given system project or project component.

Allowing the methodology to evolve to fit the organization's culture accomplishes a number of difficult objectives. First, it helps avoid the disenfranchising effect of dictating how expert knowledge workers approach their work. The personal software process (Humphrey, 1995), for example, goes so far as to suggest that each individual developer take responsibility for the procedures they use to produce quality software on time. However, to avoid complete methodological anarchy, we suggest employing *guided evolution* while adopting agile methods. This involves assigning one or more expert and respected members of the technical organization with the responsibility for ensuring that methodology evolution maps to organizational priorities and objectives. A dogmatic approach to methodology adoption seems rarely practical. Techniques from agile methods can be mixed in with existing methodological practice, especially when existing practices have proven useful within the organization. For example, ASD can be integrated into the Capability Maturity Model (CMM) to derive both the institutional and management benefits of the CMM while at the same time fostering the individual and small-team quality focus central to the approach (Paulk, 2001).

### *Customers*

Real customer involvement in requirements analysis, design, and implementation of management information systems has long been recognized as a significant contributor to project success. Effective customer participation means resolving a range of tensions that work against success including involving the 'right' customers, fostering customer-development team communication and trust, and especially sustaining customer participation when constraints of time work to disengage customers and developers from one another. Engaging the right customer for most projects probably means involving representatives from at least two groups. One customer represents the project at the executive or management level, ensuring the project fits with organizational priorities and working to gain and sustain resource commitments to the project. The other is drawn from the target user population and should be a subject matter expert in the application domain. If possible, both representatives should be information technology savvy so that they are able to act as effective mediators and translators between customer and development groups.

### *Tempo*

One of the more important findings derived from this study concerns the role of tempo in software development methodologies. By tempo we mean the pace at which development activities proceed. The issue is not that development should necessarily be fast-paced, as suggested by ASD, but that the pace be appropriate to the complexity of the project, to the expertise and comfort level of the development team, and to the rate at which requirements emerge and are specified using design stories. Managing tempo requires managing the grain size of system development iterations. Too small a grain size and the tempo may be too fast for team members and customer champions to maintain focus on quality development practices. Too large and many of the professed advantages of ASD are lost including the sense of accomplishment and the customer feedback that comes when system components are implemented in practice.

In systems development, too fast a tempo can result in disorientation with respect to critical objectives and priorities. This disorientation can affect both the development team and the customer, and is always counter-productive. One or the other may not be able to maintain a considered understanding of the course of a project, resulting in a

myriad of potential maladaptive behaviors such as reverting to older, more comfortable practices, not necessarily agile, that help to bring a project's pace under control.

A number of the studies reviewed suggest mismanagement of project tempo as a significant issue in the quality of project deliverables. In all cases the issue was a tempo that was too fast to maintain careful, quality oriented practice. One of the many sensible prescriptions that is abandoned as tempo exceeds team capabilities is adequate test planning and testing of incremental software releases. Managing ASD requires managing the tempo of the development project such that the pace of accomplishment is sufficient to maintain project momentum but at the same time is within the reasonable boundaries of the development and customer teams' cognitive and physical capabilities. This issue is implicitly addressed in the tenets of XP, which includes the 40 hour week prescription. What we mean by tempo here goes further than this however, is more difficult to achieve, and requires constant vigilance on the part of team managers. Too slow a tempo results in losing the focus on rapid delivery of working components to gain a sense of accomplishment, customer feedback, and customer trust. Too fast a tempo results in an inability to focus on enacting the proven practices and processes that are most likely to result in successful projects.

### *Architecture*

One of the most troublesome aspects of ASD is also one at the core of these methods, a focus on small, incremental deliverables without the detailed architectural planning seen as detrimental to getting working code completed quickly (Kivi, et al., 2000). Central to the agile approach is that design documents consist primarily of user stories describing interaction *scenarios*. User stories are kept necessarily short and as simple as possible so that multiple user stories can potentially be delivered in a single development iteration (2-4 weeks). This short term focus has many advantages including rapid delivery of working software components, but can lead to inefficiencies if taken to extremes. Despite ASD's focus on short release cycles, at least one expert member of the development team should be tasked with ensuring that the overall architecture of a system is coherent. This is an element of a federal approach that considers the need to evolve high quality, 'grass roots' communities of practice while at the same time not losing sight of the economies and technical coherence to be gained by understanding how

the system and its components fits into the overall organizational IT architecture and strategy.

Software architecture involves a deep understanding of the trade-offs associated with design and development decisions. This understanding comes from experience with large-scale systems projects and their outcomes in terms of meeting customer requirements, their cost to maintain and extend over time, and the reusable software components derivable from them. The importance of expert developers continues to be highlighted as a critical success factor in software systems development (Boehm, 2002) The lack of people qualified to play the role of system architect and the ramifications when junior developers are tasked with architecture presents significant challenges to software team managers. Make the title of architect something that junior people can strive to achieve through apprenticeship with more experienced developers. Explicitly acknowledge that it takes time and commitment to high quality development practices to gain promotion to architect status.

Issues with the turnaround time involved in creating a software build (Mueller & Borzuchowski, 2002) lead to several possible suggestions. First, development organizations should invest in people, tools, and processes to support the software module integration process. Second, organizations should be realistic about what constitutes a reasonable build cycle time. Software integration of short-cycle builds may result in significant time lost to preparation and procedures required for system builds, rather than on progress towards higher-level customer and organizational objectives.

Agile methods raise the question of what, if any, role is played by standard architecture-oriented modeling environments such as UML. In one case, a development team successfully replaced XP's story cards with UML use cases (Greening, 2001). Use cases may be further elaborated with activity diagrams that describe important variants of a given use scenario. An important point to remember is that graphical analysis and design representations such as the UML and associated tools (e.g., Rational Rose) are not simply documentation formats, they are also meant to serve as cognitive aids and the basis for shared understanding in the design problem solving process. As such, they play an important role regardless of the methodological commitments taken on by a

development organization. As with other aspects of the development process, teams should carefully consider what aspects of these tools are usefully suited to the context of the organization and adopt those that make significant contributions without being bound by any single methodological perspective or approach.

## Conclusion

Though dialog in the literature suggests a polarization between users of traditional, plan-based methodologies and agile software development adherents, an increasing number of researchers and practitioners are calling for more open-minded and compromising perspectives on the methods debate (Boehm, 2002; Glass, 2001). This seems sensible, since in most cases the approach an organization takes towards applying a software development methodology will depend on the specifics of the development project, the development team members, the organizations operating objectives and priorities, and a host of contextual factors.

One approach to implementation of a coherent software development methodology when faced with this dynamic range of factors is to adopt a risk-driven, *checklist* approach to applying elements of an organizational methodology. Methodological checklists are an approach to creation of an adaptive software development methodology that defines the tasks that must be performed as part of any development project without specifying exactly *how* each task is performed. This provides individual project managers technical leads, and developers with a set of reminders for what they need to consider while at the same time being flexible enough to allow team members to assess how a given checklist item is to be operationalized. It suggests what to do but allows individuals the freedom to decide how they do it.

A methodology checklist consists of perhaps as many as 100 items to consider. The set of checklist items that apply to a given development project is driven by an assessment of the risk level of the project. For example, a simple, relatively risk-free project may only require 40 of the 100 checklist items. Projects with intermediate risks require 70 of the 100. Projects with high risk employ the full set of 100 items. This risk-driven approach involves identification of a set of criteria for assessing the risk level of a given development project, and then using more or less structure and prescription in the methodology for a given project as appropriate for the level of risk associated with the project. Measures of risk may also be used to adjust tempo in response to project iterations of more or less complexity. Riskier project components require additional diligence to ensure that the pace of development coincides with the thoughtfulness



required to build a high quality complex component. Some examples of project risk assessment approaches and associated tools include the Constructive Cost Estimation Model II, or COCOMO II (see Boehm, et al. 1995), and various risk element checklists (e.g., Karolak, 1996; Software Engineering Institute, 1993).

Over 60% of software development projects that are begun are never completed (Grudin, 1996) One of the most compelling arguments for the agile approach to developing systems is that given this completion rate, developers should focus on delivering working functionality in the shortest possible increments of time so that project sponsors quickly receive some return on their development investment. Diminishing the time between analysis and fielding of a system also helps to manage the fact that organizational requirements change rapidly and that the best way to meet evolving requirements is through a development approach that is able to evolve in parallel. The challenge to developers and managers is to create a development team culture capable of adapting to shifting organizational priorities while at the same time maintaining a commitment to high-quality processes and delivery of sound software system products.

One approach to the development of cohesive product development teams capable of producing high-quality software systems at reasonable and predictable cost is by fostering communities of practice oriented towards common goals (Lave & Wenger, 1991). Especially important to building such communities is that all members participate and contribute to the identification of the objectives, priorities, and practices that define the community *ethos*. One way to begin building this shared culture is through a participatory approach to the risk-driven approach described above. All members of the development organization take part in identification of the methodology checklist and in assessment of which items are appropriate at different project risk levels. Building this culture also involves providing training, encouragement, and potentially project slack time as developers learn to adapt to new methods and ways of working. As with many collaborative activities, building a critical mass of committed individuals is an essential prerequisite for widespread adoption of proposed innovations (Grudin, 1994).

# References

Abrahamsson, P., Salo, O., Ronkainen, J., & Warsta, J., (2002). *Agile software development methods: Review and analysis.* Oulu, Finland: VTT.

Ambler, S. (2002). “Lessons in Agility from Internet-Based Development.” *IEEE Software*. March/April 2002, pp. 66-73.

Beck, K. (2000). *eXtreme Programming explained: Embrace Change*. Boston: Addison Wesley.

Beck, K. (1999). “Embracing Change with Extreme Programming.” *Computer*. October, pp. 70-77.

Blotner, J. (2002). “Agile Techniques to Avoid Firefighting at a Start-Up.” *Conference on Object Oriented Programming Systems Languages and Applications*, Seattle, Washington, ACM.

Boehm, B. (2002). “Get Ready for Agile Methods, with Care.” *Computer*. January, pp. 64-69.

Boehm, B., Clark, B., Horowitz, E., Madachy, R., Shelby, R., Westland, C. (1995)."An Overview of the COCOMO 2.0 Software Cost Model," *Software Technology Conference*, April, 1995.

Carroll, J.M. & Rosson, M.B. (1992). “Getting around the task-artifact cycle: How to make clains and design by scenario.” *ACM Transaction on Information Systems*, 10, 181-212.

DeMarco, T. & Boehm, B. (2002). “The Agile Methods Fray.” *Computer*. June, pp. 90-92.

DeMarco, T. & Lister, T. (1987). *Peopleware: productive projects and teams*. New York: Dorset House.

Elssamadisy, A. & Schalliol, G. (2002). “Recognizing and Responding to “Bad Smells” in Extreme Programming.” *Proceedings of the 24th international conference on Software engineering*, Orlando, Florida, ACM.

Fitgerald, B., Russo, N. L., & O’Kane, T. (2003). “Software Development Method Tailoring at Motorola.” *Communications of the ACM*, 46(4), April, pp. 64-70.

Glass, R. (2001). “Extreme Programming: The Good, the Bad, and the Bottom Line.” *IEEE Software*. November/December, pp. 111-112.

Grenning, J., (2001). “Launching Extreme Programming at a Process-Intensive Company.” *IEEE Software*. November/December 2001.

Grudin, J. (1996). “Evaluating Opportunities for Design Capture.” In J. M. Moran & T. P. Carroll (Eds.), *Design Rationale: Concepts, Techniques and Use*. Mahwah, NJ: Lawrence Erlbaum, pp. 21-51.

Grudin, J. (1994). “Groupware and social dynamics: Eight challenges for developers.” *Communications of the ACM*, 37(1), January, 92-105.

Highsmith, J. & Cockburn, A. (2001). “Agile Software Development: The Business of Innovation.” *Computer*. September, pp. 120-122.

Hannula, J. (1999). “Acxiom: Working toward a Common Goal.” *Computer*. October, pp. 74.

Humphrey, W. H., (1990). *Managing the Software Process*, Reading,MA: Addison-Wesley.

Humphrey, W. H., (1995). *A discipline for software engineering*. Reading, MA: Addison Wesley.

Karolak, D. W., (1996). *Software Engineering Risk Management.* IEEE Computer Society Press.

Kivi, J., Haydon D., et al (2000). “Extreme programming: a university team design experience.” *2000 Canadian Conference on Electrical and Computer Engineering*, Halifax, Nova Scotia, Canada, IEEE.

Lave, J. & Wenger, E. (1991). *Situated learning: Legitimate peripheral participation.* Cambridge, UK: Cambridge University Press.

Lindvall M., Basili V. R., Boehm B., Costa P., Dangle K., Shull F., Tesoriero R., Williams L., and Zelkowitz M. V., (2002). "Empirical Findings in Agile Methods", In *Proceedings of Extreme Programming and Agile Methods - XP/Agile Universe 2002*, Wells, D. & Williams, L., Springer,

August, pp. 197-207. Available at: http://fc-md.umd.edu/fcmd/Papers/Lindvall_agile_universe_eworkshop.pdf

Marchesi, M. (2003). "Which AM Should I Use?" In Marchesi, M, Succi, G., Wells, D, & Williams, L. (Eds.) *Extreme Programming Perspectives*. Boston: Addison-Wesley, pp.17-22.

Martin, R. (2000). "eXtreme Programming Development through Dialog." *IEEE Software*. July/August, pp. 12-13.

Mathiassen, L. & Stage, J., (1992). "The Principle of Limited Reduction in Software Design".. In: *Information, Technology and People*, Vol. 6, No. 2, 1992.

Moore, R. (2001). "Evolving to a "Lighter" Software Process: A Case Study." *26th Annual NASA Goddard Software Engineering Workshop*, Greenbelt, Maryland, IEEE.

Müller, M. & Tichy, W. (2001). "Case Study: Extreme Programming in a University Environment." *Proceedings of the 23rd international conference on Software engineering*, Toronto, Ontario, Canda, ACM.

Mueller, Gary and Borzuchowski, Janet (2002). Extreme Embedded: A Report from the Front Line. Proceedings of the 24th international conference on Software Engineering, Orlando, Florida, ACM.

Nawrocki, J., Jasinski, M., Bartosz, W., & Wojciechowski, A. (2002). "Extreme Programming Modified: Embrace Requirements Engineering Practices." *Proceedings of IEEE Joint International Conference on Requirements Engineering*, IEEE.

Newkirk, J. & Martin, R. (2000). "Extreme Programming in Practice". *Addendum to the 2000 proceedings of the conference on Object-oriented programming, systems, languages, and applications*, Minneapolis, Minnesota, ACM.

Paulk, M. C., (2001). "Extreme Programming from a CMM Perspective." *IEEE Software*. November/December, pp. 19-26.

Pelrine, J. (2000). "Modelling infection scenarios – a fixed-price extreme Programming success story." *Addendum to the 2000 proceedings of the conference on Object-oriented programming, systems, languages, and applications*, Minneapolis, Minnesota, ACM.

Reifer, D. (2002). "How Good Are Agile Methods?" *IEEE Software*, July/August, pp. 16-18.

Schuh, P., (2001). "Recovery, Redemption, and Extreme Programming." *IEEE Software*, November/December.

Shukla, A. & Williams, L. (2002). "Adapting Extreme Programming For A Core Software Engineering Course." *15th Conference on Software Engineering Education and Training*, Covington, Kentucky, IEEE.

Software Engineering Institute, (1993). "Taxonomy-Based Risk Identification," Software Engineering Institute, Technical Report SEI-93-TR-6.

Wells, D. (1999). "Ford Motor: A Unique Combination of Agility and Quality." *Computer*. October, pp. 77.

Williams, L. & Upchurch, R. (2001). "In Support of Student Pair-Programming." *Proceedings of the thirty second SIGCSE technical symposium on Computer Science Education*, Charlotte, North Carolina, ACM.