Extreme Programming from a CMM Perspective

Mark C. Paulk, *Software Engineering Institute*

Extreme Programming is an "agile methodology" that some people advocate for the high-speed, volatile world of Internet and Web software development. Although XP is a disciplined process, some have used it in arguments against rigorous software process improvement models such as the Software Capability Maturity Model.[1]

XP has good engineering practices that can work well with the CMM and other highly structured methods. The key is to carefully consider XP practices and implement them in the right environment.

In this article, I summarize both XP and the SW-CMM, show how XP can help organizations realize the SW-CMM goals, and then critique XP from a SW-CMM perspective.

The Software CMM

The Software Engineering Institute at Carnegie Mellon University developed the SW-CMM as a model for building organizational capability, and it has been widely adopted in the software community and beyond. As Table 1 shows, the SW-CMM is a five-level model that describes good engineering and management practices and prescribes improvement priorities for software organizations.

Although the SW-CMM is described in a book of nearly 500 pages, the requirements for becoming a Level 5 organization are concisely stated in 52 sentences—the 52 goals of the model's 18 key process areas (KPAs). The practices, subpractices, and examples that flesh out the model can guide software professionals in making reasonable, informed decisions about a broad range of process implementations.

The SW-CMM informative materials focus primarily on large projects and large organizations. With minor tailoring and common sense, however, the model can be applied in radically different environments, ranging from two- to three-person projects in small start-up companies to 500-person projects building hard real-time, life-critical systems.[2,3] The SW-CMM's rating components are intentionally abstract, capturing "universal truths" about high-performance software organizations. As a glance at Table 2 shows, the KPAs are clearly important to all types of software organizations.

With the exception of *software subcontract management*, which applies only to organizations that do subcontracting, the KPAs and their goals can apply to any software organization. Companies that focus on innovation more than operational excellence might downplay the role of consis-

0740-7459/00/$10.00 © 2001 IEEE

Table 1

An overview of the Software CMM

Level	Focus	Key process areas
5: Optimizing	*Continual process improvement*	Defect prevention Technology change management Process change management
4: Managed	*Product and process quality*	Quantitative process management Software quality management
3: Defined	*Engineering processes and organizational support*	Organization process focus Organization process definition Training program Integrated software management Software product engineering Intergroup coordination Peer reviews
2: Repeatable	*Project management processes*	Requirements management Software project planning Software project tracking and oversight Software subcontract management Software quality assurance Software configuration management
1: Initial	*Competent people (and heroics)*	

tency, predictability, and reliability, but performance excellence is important even in highly innovative environments.

Extreme Programming

The XP method is typically attributed to Kent Beck, Ron Jeffries, and Ward Cunningham.[4,5] XP's target is small to medium-sized teams building software with vague or rapidly changing requirements. XP teams are typically colocated and have fewer than 10 members.

XP's critical underlying assumption is that developers can obviate the traditional high cost of change using technologies such as objects, patterns, and relational databases, resulting in a highly dynamic XP process. Beck's book is subtitled "Embrace Change," and XP teams typically deal with requirements changes through an iterative life cycle with short cycles.

The XP life cycle has four basic activities: coding, testing, listening, and designing. Dynamism is demonstrated through four values:

- continual communication with the customer and within the team;
- simplicity, achieved by a constant focus on minimalist solutions;
- rapid feedback through mechanisms such as unit and functional testing; and
- the courage to deal with problems proactively.

Principles in practice

Most of XP's principles—minimalism, simplicity, an evolutionary life cycle, user involvement, and so forth—are commonsense practices that are part of any disciplined process. As Table 3 summarizes, the "extreme" in XP comes from taking commonsense practices to extreme levels. Although some people may interpret practices such as "focusing on a minimalist solution" as hacking, XP is actually a highly disciplined process. Simplicity in XP terms means focusing on the highest-priority, most valuable system parts that are currently identified rather than designing solutions to problems that are not yet relevant (and might never be, given that requirements and operating environments change).

Although developers might use many different XP practices, the method typically consists of 12 basic elements:

- *Planning game*: Quickly determine the next release's scope, combining business priorities and technical estimates. The customer decides scope, priority, and dates from a business perspective, whereas technical people estimate and track progress.
- *Small releases*: Put a simple system into production quickly. Release new versions on a very short (two-week) cycle.
- *Metaphor:* Guide all development with a simple, shared story of how the overall system works.

Table 2

The Software CMM key process areas and their purpose

Key process area	Purpose
	Maturity Level 2: Repeatable
Requirements management	Establish a common understanding between the customer and software project team about the customer's requirements.
Software project planning	Establish reasonable plans for software engineering and overall project management.
Software project tracking and oversight	Provide adequate visibility into actual progress so that management can act effectively when the software project's performance deviates significantly from the software plans.
Software subcontract management	Select qualified software subcontractors and manage them effectively.
Software quality assurance	Provide management with appropriate visibility into the product and the software process.
Software configuration management	Establish and maintain the integrity of software products throughout the project's software life cycle.
	Maturity Level 3: Defined
Organization process focus	Establish organizational responsibility for software process activities that improve the organization's overall software process capability.
Organization process definition	Develop and maintain a usable set of software process assets that improve process performance across the projects and provide a basis for cumulative, long-term organizational benefits.
Training program	Develop individuals' skills and knowledge so they can perform their roles effectively and efficiently.
Integrated software management	Integrate the software engineering and management activities into a coherent, defined software process based on the organization's standard software process and related process assets.
Software product engineering	Consistently use a well-defined engineering process that integrates all the software engineering activities to produce correct, consistent software products effectively and efficiently.
Intergroup coordination	Establish a way for the software engineering group to participate actively with other engineering groups so that the project can effectively and efficiently satisfy customer needs.
Peer reviews	Remove defects from the software work products early and efficiently. An important corollary effect is to develop a better understanding of the software products and the preventable defects.
	Maturity Level 4: Managed
Quantitative process management	Quantitatively control the performance of the software project's process. Software process performance represents the actual results achieved from following a software process.
Software quality management	Quantify the quality of the project's software products and achieve specific quality goals.
	Maturity Level 5: Optimizing
Defect prevention	Identify the cause of defects and prevent them from recurring.
Technology change management	Identify new technologies (such as tools, methods, and processes) and introduce them into the organization in an orderly manner.
Process change management	Continually improve the organization's software processes with the goal of improving software quality, increasing productivity, and decreasing the product-development cycle time.

- *Simple design*: Design as simply as possible at any given moment.
- *Testing*: Developers continually write unit tests that must run flawlessly; customers write tests to demonstrate functions are finished. "Test, then code" means that a failed test case is an entry criterion for writing code.
- *Refactoring*: Restructure the system without changing its behavior to remove duplication, improve communication, simplify, or add flexibility.
- *Pair programming*: All production code is written by two programmers at one machine.
- *Collective ownership*: Anyone can improve any system code anywhere at any time.
- *Continuous integration*: Integrate and build the system many times a day (every time a task is finished). Continual regression testing prevents functionality regressions when requirements change.
- *40-hour weeks*: Work no more than 40 hours per week whenever possible; never work overtime two weeks in a row.
- *Onsite customer*: Have an actual user on the team full-time to answer questions.
- *Coding standards*: Have rules that emphasize communication throughout the code.

These basic practices work together to cre-

Table 3

The "extreme" in Extreme Programming

Commonsense	XP extreme	XP implementation practice
Code reviews	Review code all the time	Pair programming
Testing	Test all the time, even by customers	Unit testing, functional testing
Design	Make design part of everybody's daily business	Refactoring
Simplicity	Always work with the simplest design that supports the system's current functionality	The simplest thing that could possibly work
Architecture	Everybody works to refine the architecture all the time	Metaphor
Integration testing	Integrate and test several times a day	Continuous integration
Short iterations	Make iterations extremely short—seconds, minutes, and hours rather than weeks, months, and years	Planning game

ate a coherent method. XP characterizes the full system functionality using a pool of "stories," or short feature descriptions. For the planning game and small releases, the customer must select a subset of stories that characterize the most desirable work for developers to implement in the upcoming release. Because the customer can add new stories to the pool at any time, requirements are highly volatile. However, volatility is managed by implementing functionality in two-week chunks. Having a customer onsite supports this ongoing cycle of two-week releases.

XP developers generate a metaphor to provide the project's overarching vision. Although you could view this as a high-level architecture, XP emphasizes design, while at the same time minimizing design documentation. Some people have characterized XP as not allowing documentation outside code, but that is not quite accurate. Because XP emphasizes continual redesign—using refactoring whenever necessary—there is little value to detailed design documentation (and maintainers rarely trust anything other than the code anyway).

XP developers typically throw away design documentation after the code is written, although they will keep it if it's useful. They also keep design documentation when the customer stops coming up with new stories. At that point, it's time to put the system in mothballs and write a five- to 10-page "mothball tour" of the system. A natural corollary of the refactoring emphasis is to always implement the simplest solution that satisfies the immediate need. Requirements changes are likely to supersede "general solutions" anyway.

Pair programming is one of XP's more controversial practices, mainly because it has resource consequences for the very managers who decide whether or not to let a project use XP. Although it might appear that pair programming consumes twice the resources, research has shown that it leads to fewer defects and decreased cycle time.[6] For a jelled team, the effort increase can be as little as 15 percent, while cycle time is reduced by 40 to 50 percent. For Internet-time environments, the increased speed to market may be well worth the increased effort. Also, collaboration improves problem solving, and increased quality can significantly reduce maintenance costs. When considered over the total life cycle, the benefits of pair programming often more than pay for added resource costs.

Because XP encourages collective ownership, anyone can change any piece of code in the system at any time. The XP emphasis on continuous integration, continual regression testing, and pair programming protects against a potential loss of configuration control. XP's emphasis on testing is expressed in the phrase, "test, then code." It captures the principle that developers should plan testing early and develop test cases in parallel with requirements analysis, although the traditional emphasis is on black-box testing. Thinking about testing early in the life cycle is standard practice for good software engineering, though it is too rarely practiced.

The basic XP management tool is the metric, and the metric's medium is the "big visible chart." In the XP style, three or four measures are typically all a team can stand at one time, and those should be actively used and visible. One recommended XP metric is "project velocity"—the number of stories of a given size that developers can implement in an iteration.

Adoption strategies

XP is an intensely social activity, and not everyone can learn it. There are two conflicting attitudes toward XP adoption. XP is gen-

erally viewed as a system that demonstrates emergent properties when adopted as a whole. As the discussion thus far shows, there are strong dependencies between many XP practices, such as collective ownership and continuous integration.

Nonetheless, some people recommend adopting XP one practice at a time, focusing on the team's most pressing current problem. This is consistent with the attitude toward change that XP is "just rules" and the team can change the rules anytime as long as they agree on how to assess the change's effects. Beck, for example, describes XP practices as "etudes": They help developers master the techniques, but experienced users can modify them as necessary.

XP and the CMM

The SW-CMM focuses on both the management issues involved in implementing effective and efficient processes and on systematic process improvement. XP, on the other hand, is a specific set of practices—a "methodology"—that is effective in the context of small, colocated teams with rapidly changing requirements. Taken together, the two methods can create synergy, particularly in conjunction with other good engineering and management practices. I'll now illustrate this by discussing XP practices in relation to the CMM KPAs and goals outlined in Table 2.

XP and Level 2 practices

XP addresses Level 2's *requirements management* KPA through its use of stories, an onsite customer, and continuous integration. Although system requirements might evolve dramatically over time, XP integrates feedback on customer expectations and needs by emphasizing short release cycles and continual customer involvement. "Common understanding" is established and maintained through the customer's continual involvement in building stories and selecting them for the next release (in effect, prioritizing customer requirements).

XP addresses *software project planning* in the planning game and small releases. XP's planning strategy embodies Watts Humphrey's advice, "If you can't plan well, plan often." The first three activities of this KPA deal with getting the software team involved in early planning. XP integrates the software team into the commitment process by having it estimate the effort involved to implement customer stories; at the level of two-week releases, such estimates are typically quite accurate. The customer maintains control of business priorities by choosing which stories to implement in the next release with the given resources. By definition, the XP life cycle is both incremental and evolutionary. The project plan is not detailed for the project's whole life cycle, although the system metaphor does establish a vision for project direction. As a result, developers can identify and manage risks efficiently.

XP addresses *software project tracking and oversight* with the "big visual chart," project velocity, and commitments (stories) for small releases. XP's commitment process sets clear expectations for both the customer and the XP team at the tactical level and maximizes flexibility at the project's strategic level. The emphasis on 40-hour weeks is a general human factors concern; although CMM does not address it, having "rational work hours" is usually considered a best practice. XP also emphasizes open workspaces, a similar "people issue" that is outside CMM's scope. XP does not address *software subcontract management*, which is unlikely to apply in XP's target environment.

While an independent *software quality assurance* group is unlikely in an XP culture, SQA could be addressed by the pair-programming culture. Peer pressure in an XP environment can achieve SQA's aim of assuring conformance to standards, though it does not necessarily give management visibility into nonconformance issues. Dealing with process and product assurance using peer pressure can be extraordinarily effective in a small team environment. However, larger teams typically require more formal mechanisms for objectively verifying adherence to requirements, standards, and procedures. Also, peer pressure might be ineffective when the entire team is being pushed, just as a software manager might be vulnerable to external pressure. This vulnerability should be addressed at the organizational level when considering SQA.

Although not completely and explicitly addressed, *software configuration management* is implied in XP's collective ownership, small releases, and continuous integration. Collective ownership might be problematic for large systems, where more formal com-

Taken together, the two methods can create synergy, particularly in conjunction with other good engineering and management practices.

Table 4

XP satisfaction of key process areas, given the appropriate environment

Level	Key process area	Satisfaction
2	Requirements management	++
2	Software project planning	++
2	Software project tracking and oversight	++
2	Software subcontract management	—
2	Software quality assurance	+
2	Software configuration management	+
3	Organization process focus	+
3	Organization process definition	+
3	Training program	—
3	Integrated software management	—
3	Software product engineering	++
3	Intergroup coordination	++
3	Peer reviews	++
4	Quantitative process management	—
4	Software quality management	—
5	Defect prevention	+
5	Technology change management	—
5	Process change management	—

\+ *Partially addressed in XP*
++ *Largely addressed in XP (perhaps by inference)*
— *Not addressed in XP*

munication channels are necessary to prevent configuration management failures.

XP and Level 3 practices

At Level 3, XP addresses *organization process focus* at the team rather than organizational level. A focus on process issues is nonetheless implied in adopting XP one practice at a time, as well as in the "just rules" philosophy. Because XP focuses on software engineering process rather than organizational infrastructure issues, organizations adopting XP must address this and other organization-level processes, whether in a CMM-based context or not.

Similarly, the various XP-related books, articles, courses, and Web sites partially address the *organization process definition* and *training program* KPAs, but organizational assets are outside the scope of the XP method itself. As a consequence, XP cannot address *integrated software management* because there may not be any organizational assets to tailor.

Several XP practices effectively address *software product engineering*: metaphor, simple design, refactoring, the "mothball" tour, coding standards, unit testing, and functional testing. XP's de-emphasis of design documentation is a concern in many environments, such as hard real-time systems, large systems, or virtual teams. In such environments, good designs are crucial to success, and using the refactoring strategy would be high-risk. For example, if developers performed refactoring after a technique such as rate-monotonic analysis proved that a system satisfied hard real-time requirements, they'd have to redo the analysis. Such an environment invalidates XP's fundamental assumption about the low cost of change.

XP's emphasis on communication—through onsite customers and pair programming—appears to provide as comprehensive a solution to *intergroup coordination* as integrated product and process development. In fact, XP's method might be considered an effective IPPD approach, although the software-only context ignores multidiscipline environments.

Pair programming addresses *peer reviews*, and is arguably more powerful than many peer review techniques since it adopts preventive concepts found in code reading and literate programming. However, pair programming's relative lack of structure can lessen its effectiveness. Empirical data on pair programming is currently sparse but promising.[6] To make informed trade-off decisions, we'll need more empirical research that contrasts and compares pair programming and peer review techniques, especially more rigorous techniques such as inspections.

Beyond Level 3

XP addresses few of the Level 4 and 5 KPAs in a rigorous statistical sense, although feedback during rapid cycles might partially address *defect prevention*. Table 4 summarizes XP's potential to satisfy CMM KPAs, given the appropriate domain.

Many of the KPAs that XP either ignores or only partially covers are undoubtedly addressed in real projects. XP needs management and infrastructure support, even if it does not specifically call for it.

Discussion

As the earlier comparison shows, XP generally focuses on technical work, whereas the CMM generally focuses on management issues. Both methods are concerned with "culture." The element that XP lacks that is crucial for the SW-CMM is the concept of "institutionalization"—that is, establishing a culture of "this is the way we do things around here."

Although implicit in some practices, such as the peer pressure arising from pair programming, XP largely ignores the infrastructure that the CMM identifies as key to institutionalizing good engineering and management practices. Table 5 summarizes XP's coverage of institutionalization in its domain.

The CMM's KPAs share common features that implement and institutionalize processes. Each KPA's institutionalization practices map to the area's goals; a naïve XP implementation that ignored these infrastructure issues would fail to satisfy any KPA. XP ignores some of these practices, such as policies. XP addresses others, such as training and SQA, by inference. It addresses still others—project-specific practices such as management oversight and measurement—to a limited degree. As an implementation model focused on the development process, these issues are largely outside XP's focus, but they are arguably crucial for its successful adoption.

Table 5

XP and institutionalization practices

Common feature (in each KPA)	Practice	Satisfaction
Commitment to perform	Policy	—
	Leadership and sponsorship	—
Ability to perform	Organizational structures	+
	Resources and funding	+
	Training	+
Measurement and analysis	Measurement	+
Verifying implementation	Senior management oversight	—
	Project management oversight	++
	Software quality assurance	+

\+ *Partially addressed in XP*
++ *Largely addressed in XP (perhaps by inference)*
— *Not addressed in XP*

Size matters

Much of the formalism that characterizes most CMM-based process improvement is an artifact of large projects and severe reliability requirements, especially for life-critical systems. The SW-CMM's hierarchical structure, however, is intended to support a range of implementations through the 18 KPAs and 52 goals that comprise the requirements for a fully mature software process.

As systems grow, some XP practices become more difficult to implement. XP is, after all, targeted toward small teams working on small- to medium-sized projects. As projects become larger, emphasizing a good architectural "philosophy" becomes increasingly critical to project success. Major investment in product architecture design is one of the practices that characterizes successful Internet companies.[7]

Architecture-based design, designing for change, refactoring, and similar design philosophies emphasize the need to manage change systematically. Variants of the XP bottom-up design practices, such as architecture-based design, might be more appropriate in large-project contexts. In a sense, architectural design that emphasizes flexibility is the goal of any good object-oriented methodology, so XP and object orientation are well suited to one another. Finally, large projects tend to be multidisciplinary, which can be problematic given that XP is aimed at software-only projects.

Why explore XP?

Modern software projects should capture XP values, regardless of how radically their implementation differs from XP's. Organizations might call communication and simplicity by other names, such as coordination and elegance, but without these values, nontrivial projects face almost insurmountable odds.

XP's principles of communication and simplicity are also fundamental for organizations using the SW-CMM. When defining processes, organizations should capture the minimum essential information needed, structure definitions using good software design principles (such as information hiding and abstraction), and emphasize usefulness and usability.[2]

For real-time process control, rapid feedback is crucial. Previous eras have captured this idea in aphorisms such as "don't throw good money after bad"; in a quantitative sense, we can view this as the soul of the CMM's Level 4. One of the consequences of the cultural shift between Levels 1 and 2 is the need to demonstrate the courage of our convictions by being realistic about estimates, plans, and commitments.

False opposition

The main objection to using XP for process improvement is that it barely touches the management and organizational issues that the SW-CMM emphasizes. Implementing the kind of highly collaborative environment that XP assumes requires enlightened management and appropriate organizational infrastructure.

The argument that CMM's ideal of a rigorous, statistically stable process is antithetical to XP is unconvincing. XP has disciplined processes, and the XP process itself is clearly well defined. We can thus consider CMM and XP complementary. The SW-CMM tells organizations what to do in general terms, but does not say how to do it. XP is a set of best practices that contains fairly specific how-to information—an implementation model—for a particular type of environment. XP practices can be compatible with CMM practices (goals or KPAs), even if they do not completely address them.

Most of XP consists of good practices that all organizations should consider. While we can debate the merits of any one practice in relation to other options, to arbitrarily reject any of them is to blind ourselves to new and potentially beneficial ideas.

To put XP practices together as a methodology can be a paradigm shift similar to that required for concurrent engineering. Although its concepts have been around for decades, adopting concurrent engineering practices changes your product-building paradigm. XP provides a systems perspective on programming, just as the SW-CMM provides a systems perspective on organizational process improvement. Organizations that want to improve their capability should take advantage of the good ideas in both, and exercise common sense in selecting and implementing those ideas.

Should organizations use XP, as published, for life-critical or high-reliability systems? Probably not. XP's lack of design documentation and de-emphasis on architecture is risky. However, one of XP's virtues is that you can change and improve it for different environments. That said, when you change XP, you risk losing the emergent properties that provide value in the proper context. Ultimately, when you choose and improve software processes, your emphasis should be to let common sense prevail—and to use data whenever possible to offer insight on challenging questions.

Acknowledgments

I gratefully acknowledge Kent Beck, Steve McConnell, and Laurie Williams for their comments.

References

1. M.C. Paulk et al., *The Capability Maturity Model: Guidelines for Improving the Software Process*, Addison-Wesley, Reading, Mass., 1995.
2. M.C. Paulk, "Using the Software CMM with Good Judgment," *ASQ Software Quality Professional*, vol. 1, no. 3, June 1999, pp. 19–29.
3. D.L. Johnson and J.G. Brodman, "Applying CMM Project Planning Practices to Diverse Environments," *IEEE Software*, vol. 17, no. 4, July/Aug. 2000, pp. 40–47.
4. K. Beck, *Extreme Programming Explained: Embrace Change*, Addison-Wesley, Reading, Mass., 1999.
5. "eXtreme Programming Pros and Cons: What Questions Remain?" *IEEE Computer Soc. Dynabook*, J. Siddiqi, ed., Nov. 2000; www.computer.org/seweb/dynabook/index.htm (current 24 Sept. 2001).
6. L. Williams et al., "Strengthening the Case for Pair Programming," *IEEE Software*, vol. 17, no. 4, July/Aug. 2000, pp. 19–25.
7. A. MacCormack, "Product-Development Practices that Work: How Internet Companies Build Software," *MIT Sloan Management Rev.*, no. 42, vol. 2, Winter 2001, pp. 75–84.

About the Author

Mark Paulk is a senior member of the technical staff at the Software Engineering Institute. His current interests are in high-maturity practices and statistical control for software processes. He was "book boss" for Version 1.0 of the Capability Maturity Model for Software and project leader during the development of Software CMM Version 1.1. He is also involved with software engineering standards, including ISO 15504, ISO 12207, and ISO 15288. He received his bachelor's degree in mathematics and computer science from the University of Alabama in Huntsville and his master's degree in computer science from Vanderbilt University. Contact him at the Software Engineering Inst., Carnegie Mellon Univ., Pittsburgh, PA 15213; mcp@sei.cmu.edu.