

Is Internet-Speed Software Development Different?

Richard Baskerville and Balasubramaniam Ramesh, *Georgia State University*

Linda Levine, *Software Engineering Institute*

Jan Pries-Heje, *IT University of Copenhagen*

Sandra Slaughter, *Carnegie Mellon University*

To compete in the digital economy, companies must be able to develop high-quality software systems at “Internet speed”—that is, deliver new systems to customers with more value and at a faster pace than ever before. However, applying quality software development practices on a widespread basis has met with serious obstacles. The Internet environment intensifies software development problems by emphasizing shorter cycle times.

Current software development methods and software process improvement approaches (ISO 9000, Capability Maturity Models, SPICE, and BOOTSTRAP, for example) are typically effective in large-scale, long-term development efforts with stable and disciplined processes.¹ In contrast, Internet-speed software development involves rapid requirement changes and unpredictable product complexity. Such environments require software development approaches that balance flexibility and disciplined methodology.²⁻⁴

High-visibility Internet software leaders such as Microsoft and Netscape have adopted agile development methods for this new arena. Whether these practices support traditional software engineering principles and will result in high-quality software isn't clear, however. On the basis of our multiphase study of 10 software development organizations over three years, in conjunction with findings from a Discovery Colloquium on best practices for fast-paced software development, we examine how and why practices used to develop software at Internet speed differ from traditional software development.

Internet-speed development practices emerge

Working at Internet speed changes how software development is organized. The

Developing software at Internet speed requires a flexible development environment that can cope with fast-changing requirements and an increasingly demanding market. Agile principles are better suited than traditional software development principles to provide such an environment.

Study Methods

Our study used a mixed-methods research design¹ involving the collection of multiple kinds of data. Case studies of Internet-speed software development in Phase 1 were complemented with a Discovery Colloquium on Internet-speed development in Phase 2.

Phase 1: Internet-speed development case studies

During the first phase in Fall 2000, we conducted detailed case studies of Internet software development at 10 companies in two major metropolitan areas. Our objective was to understand how and why Internet-speed software development differs from traditional software development.

We collected data through open-ended interviews and analyzed it using grounded theory.² With this methodology, we can develop a theory for a problem under investigation without prior hypotheses. The analysis identified core categories and their interrelationships, explaining how and why Internet-speed software development differs from traditional approaches.

Phase 2: Discovery Colloquium

Phase 2 objectives were to synthesize knowledge on best practices for quality and agility in Internet-speed software development. We held a one-day Discovery Colloquium on Innovative Practices for Speed and Agility in Internet Software Development³ using innovative open-forum search techniques to enable what has been called "creative abrasion."⁴ Colloquium activities were designed to expand understanding of the interface between business issues and Internet software development through dialog; explore promising practices for Internet software development; and engage participants in a vision-based approach to identifying emerging models, strategies, and directions addressing challenges in Internet software development.

Grounded in Kurt Lewin's action research model, search conference participants "bring the whole system in the room" to exchange views and learn from each other (<http://future-search.net/>). Like other forms of action research, participants share observations, engage in collaborative analysis, and logi-

cally test discoveries in an interactive debate. Search conferences produce data and findings coincidentally.

The colloquium benefited from the Phase 1 findings and included interviewees from Phase 1 companies as well as selected experts. Software practitioners from entrepreneurial small companies and large "brick and mortar" companies, Internet business strategists, and leading software development experts also participated. Methodological perspectives represented a broad spectrum: from adherents of agile methodologies to adherents of traditional software process disciplines.

Participants joined one of several breakout groups dedicated to exploring a core issue. The groups first identified observations relating to their core issue, and then developed hypotheses about possible associated factors and dynamics. The groups tested the hypotheses, identifying linkages, contradictions, and interdependencies among them. The groups then delved into underlying assumptions to further build dialog from generative ideas and to allow for divergence and convergence of insights with maximum cross-fertilization. They identified principles, promising practices, and other dynamics.

The Discovery Colloquium's foundations in action research are reflected in how data collection, analysis, and social action are conflated into one flowing exercise. We took part in the colloquium as participant-observers. All of the participants, operating in various groups, explicated the concepts from their practical experiences. The groups then operated analytically on these concepts in a socially situated forum. Both data and analysis emerged in the mode of action learning.

We based the results reported in this article on discussions in a group focused on agile software development.

References

1. A. Tashakkori and C. Teddlie, *Mixed Methodology: Combining Qualitative and Quantitative Approaches*, vol. 46, Sage Publications, 1998.
2. A. Strauss and J. Corbin, *Basics of Qualitative Research: Grounded Theory Procedures and Techniques*, Sage Publications, 1990.
3. L. Levine et al., *Discovery Colloquium: Quality Software Development @ Internet Speed*, SEI tech. report CMU/SEI-2002-TR-020, ESC-TR-2002-020, Software Eng. Inst., Carnegie Mellon Univ., 2002.
4. D. Leonard, *When Sparks Fly: Igniting Creativity in Groups*, Harvard Business School Press, 1999.

grounded theory analysis in Phase 1 of our study (see the "Study Methods" sidebar) uncovered three major causal factors:

- A desperate rush-to-market
- A new and unique software market environment
- A lack of experience developing software under the conditions this environment imposed

As a result, a new development process that depends on new software development cultures evolved. In this process, software prod-

uct quality becomes negotiable. Eight identifiable practices characterizing the Internet-speed software development process emerged from Phase 1.⁵

Develop in parallel

In a marketplace with faster and faster release cycles, companies must deploy new features quickly or competitors will get out there earlier and snatch away customers. One way to increase speed is to use overlapping parallel development. Releases can be totally developed in parallel or staged onto the market such that design, development, and quality as-

Software development involves achieving interoperability and integration among components developed elsewhere or purchased off the shelf.

insurance all occur simultaneously, but sequentially on different releases. Sometimes developers begin coding even before they fully understand a project's requirements. Development proceeds in anticipation of the features that will be required in the product's final version.

Release more often

"People have a perception of Internet speed. They expect it. So we've had to scope our delivery or deliver a smaller set of features, thereby releasing more often," said one manager we interviewed. Requirements can stay fluid when a company constantly monitors and prioritizes the features that will be included in successive releases of the product. Features that can't be completed in time can slip from one release to the next. Similarly, the company can introduce an important new feature rather late in the process when market conditions require it. The fast cycle time removes much of the trauma of slipping a feature.

Depend on tools

Many of the companies we visited make heavy use of development tools and environments that speed the design and coding process. The infrastructure and tools provided by new technologies offer much of the functionality that used to be custom-built in traditional software development. One developer estimated that "50 percent of development is already taken care of by the tools we use."

Implant customers in the development environment

Customer involvement in traditional software development is usually through reference groups or steering committees, which can slow development. When requirements are fuzzy and fast-changing, intimate access to customers slashes time delays and ensures that high-priority features are correctly identified. Customers move their normal working environment to the development environment to shorten cycle times. This "implantation" lets customers participate closely in all phases of development. Internet projects rely on this close involvement rather than a formalized requirements management process. It provides customers with immediate feedback on the costs and schedule implications of requirements changes and leads to good strategies for chunking requirements into logically cohesive releases.

Establish a stable architecture

A fixed architecture helps anchor a rapid development process, which is never quite stable. It allows similarity among releases and the largest possible reuse of components. A three-layer architecture—database, business logic, and user interface—is common.

Some companies we interviewed focused on developing a common architecture and standardizing it across applications. They viewed this as an investment that would pay huge dividends by facilitating development of scalable and maintainable systems. Other companies did not apply this practice because throwaway systems might not need stable architectures to operate. The effects of bad architecture can emerge far downstream in the product evolution. By that time, the product might be undergoing a complete redevelopment.

Assemble and reuse components

Internet-speed development can be achieved often only if developers maximize reusable components, rather than craft the software from scratch. "Internet speed needs reuse. We need to take components or assets and know how to put them together," one developer said during our interview. A manager from the same company continued, "The strategy is to acquire, integrate, and assemble components with wrappers—to get things done quickly."

Component reuse at all levels of the architecture (business logic, interfaces, and back-end infrastructure) was prominent among companies interviewed. Software development involves achieving interoperability and integration among components developed elsewhere or purchased off the shelf.

Ignore maintenance

The short life span of Internet-speed software means that developers rarely give maintenance serious consideration. One small software house said, "Products are not documented: no design document, no requirements specification. If the person who originally did it is gone, it takes a much longer time. Often we can start from scratch. It leads to a throw-away mentality." When software is retired quickly and replaced with newer versions developed from scratch, this cavalier attitude might avoid the serious maintenance problems that can occur in long-lived, tightly-coupled, and complex software.

Principles or Practices?

In a strict dictionary sense, a *principle* is a fundamental doctrine or assumption, while a *practice* is how users customarily apply the principle. The principles of software engineering continue to emerge, and distinguishing practice from principle

will always bring contention among experts. For example, as Table A shows, the Agile Manifesto's principles (<http://agilemanifesto.org/principles.html>) partly overlap practices and principles from our study.

Table A

Comparison of selected Agile Manifesto principles and Internet-speed practices and Discovery Colloquium agile principles

Agile Manifesto principles	Internet-speed practices and Discovery Colloquium agile principles
Satisfy the customer through early and continuous delivery of valuable software	Good software development methodologies engage the customer
Welcome late and changing requirements; harness change for the customer's competitive advantage	Methodologies must accommodate requirements change
Self-organizing teams develop the best architectures, requirements, and designs	Software processes require good teamwork regardless of methodology
Deliver working software frequently	Release more often
Business people and developers must work together daily throughout the project	Implant customers in the development environment
Routinely reflect on the team's effectiveness and tune behavior	Tailor the methodology daily

Tailor the methodology daily

The processes and methods the companies used in Internet software development varied considerably depending on project team composition and product type. Some companies developed an overall framework within which project teams could tailor their methodologies daily. With the intense demands for speed, many companies used just enough process to be effective. Often the tendency was to skip phases or tasks that might impede their ability to deliver the software on time, although this might be done at the risk of producing lower quality software.

Principles behind the practices

We can easily relate the practices found in Phase 1 of our study to similar practices in more traditional software development. Although no single characteristic distinguishes this new software process, the collection of practices and the way they are used is unique and remarkably common to Internet-speed software development processes. Traditional software development sometimes employs a select few of these practices, but seldom does it use several of them together. Further, the practices are often extreme when compared to seemingly similar practices observed in traditional software development. For example, the typical frequency of Internet-speed software releases is greater than packaged software by orders of magnitude.

Is Internet-speed software development truly different from traditional software development, and if so, how? For this question, we must consider the principles corresponding to the practices. In Phase 2, we held a Discovery Colloquium, aiming to understand the issues faced by organizations attempting to develop quality software at Internet speed (see the "Principles or Practices?" sidebar). Although the findings from the colloquium distinguished Internet speed as a set of practices, it denoted the underlying principles as principles of agility. Agility, which came to software engineering through agile manufacturing, invokes design principles that facilitate rapid redesign and reconfiguration of a system's components for entirely new applications. Internet-speed practices are a professional implementation of certain agile software development principles, which Phase 2 sought to discover.

Agile development principles

We first identified rival hypotheses and underlying assumptions about agile methodologies and their role in Internet-speed software development vis-à-vis traditional software development. Colloquium participants explicitly generated some hypotheses:

- Agile methods are more effective than traditional methods.

Table 1**Internet-speed versus traditional software development metaprinciples**

Discovery Colloquium on Internet-speed software development	Montreal Workshop on software development standards
Several equally valid approaches to accomplishing a development goal exist	Establish a flexible software process
Good software development methodologies engage the customer	Invest in understanding the problem
Methodologies must accommodate requirements change	Change is inherent to software; therefore, plan for and manage it
Current agile methods are a simple instantiation of a set of practices that seem to work; successful methods derive from the basic principles of software development	To improve design, study solutions to similar problems
All software processes require teamwork	
The project environment constrains what software development practices will be effective	
Mixing and matching methodologies can have unintended consequences	
	Apply and use quantitative measurements in decision making
	Build with and for reuse
	Control complexity with multiple perspectives and multiple levels of abstraction
	Rigorously define software artifacts
	Manage quality throughout the life cycle as formally as possible
	Minimize software component interaction
	Produce software in a stepwise fashion
	Set quality objectives for each deliverable product
	Make trade-offs explicit and document them
	Identify and manage uncertainty
	Implement a disciplined approach and continuously improve it

- Internet-speed development is fundamentally different from traditional development, so it requires agile methods.
- Agile methods are effective when the time horizon is short and not as effective over the long term.
- Agile methods aren't really new, but their implementation is extreme.

Key insights emerged when these hypotheses were discussed:

- Basic principles of software development exist, are known, and are immutable.
- You can implement a software development principle in many ways. It's important to identify the principles and discern how the practices differ from one methodology to another.
- Every agile method practice has an analog in traditional development.
- Environments dictate practices, not principles. Some combinations of practices are superior in specific environments. Thus, different environments require different implementations for effectiveness.
- Project size impacts software development practices.
- How a developer composes a product's life cycle over time changes how the developer attacks the problem. Choosing a software

development life cycle such as waterfall or spiral dictates the choice of practices.

Based on these insights, the group members generated a set of metaprinciples underlying agile software development. The first column in Table 1 lists these metaprinciples.

Traditional versus agile principles

In our search for a good traditional set of principles to compare with agile principles, we reviewed well-known publications on the topic, such as Barry Boehm's "Seven Basic Principles of Software Engineering,"⁶ and Alan Davis's "Fifteen Principles of Software Engineering."⁷ We adopted a set of principles, listed in column two of Table 1, rigorously developed in a workshop on software development standards held in Montreal using a multistage Delphi study involving well-respected researchers and practitioners.⁸ It exemplifies the best attempt to date to define general metaprinciples for traditional software development.

Table 1 shows the overlap resulting from our comparison of the Montreal workshop metaprinciples to the agile development principles that emerged from our colloquium. Each set also has some distinctive and nonoverlapping principles: 11 metaprinciples from the Montreal workshop with no comparable agile metaprinciple in the Discovery Colloquium, and

three agile metaprinciples with no comparable metaprinciple from the Montreal workshop. Five metaprinciples correspond. The one-to-one correspondence is not totally complete as Table 1 shows. Several principles from the Montreal workshop could be related, albeit more indirectly, to principles from the colloquium, and vice versa. For our analysis, the important principles are those that could not be related.

We consider this analysis in three parts:

- Overlapping set of traditional and agile principles
- Traditional principles with no equivalent agile principle
- Agile principles with no equivalent traditional principle

Agile principles prioritize speed, responsiveness, and improvisation rather than quality or cost as traditional principles do. Contrary to traditional software development's emphasis on control, discipline, formality, and rigor, agile principles stress informal knowledge exchange, collaboration, and experience, and acknowledge more sensitivity to tailoring project practices to environmental conditions. Agile principles are essential for managing software projects in volatile settings where fast-changing technologies and markets drive fast-changing skills and knowledge.

The overlapping principles—such as flexibility and responding to change—are essential for keeping processes aligned with changing requirements. Both agile and traditional principles stress the importance of developing toward a “just-right” set of requirements.

Internet-speed practices and development principles

Is Internet-speed software development different? Our analysis of software development principles reveals that whereas certain agile principles overlap traditional principles, others differ. However, how these differences in principle are reflected in differences in practice is unclear. We've identified eight Internet-speed development practices (described earlier in the “Internet-speed development practices emerge” section). To understand how these practices relate to principles, we analyze how they are compatible with agile principles and incompatible with the traditional development principles that have no counterpart in agile development.

Compatibility of Internet-speed development practices and agile principles

Each Internet-speed development practice can also be found in traditional software development. What distinguishes the practices is how Internet-speed developers combine and apply them. Although we'd like to be able to map practices to principles in a one-to-one alignment, the real world of software development is too messy to support such a crisp analysis.

One or more of the practices can be combined and used to implement each principle. However, some practices are clearly more prominent than others in enacting each principle. We consider the seven agile principles identified in Phase 2 in terms of how they are supported and enacted by the eight practices identified in Phase 1.

Accept multiple valid approaches. This principle is supported by the practice of “tailoring the methodology daily.” It becomes more successful when combined with frequent releases because different methodologies can be isolated when used for different releases. A stable architecture a tool orientation, and component-based development combine to enable this fluid view of methodology by providing a framework to constrain and contain the behavior of system components that might have been developed with different approaches.

Engage the customer. The “customer implantation” practice supports this principle. Engaging the customer and providing frequent releases gives customers immediate satisfaction as they see their ideas and requirements embodied in a new release.

Accommodate requirements change. Most practices are complementary in accommodating changing requirements. Frequent releases and parallel development permit last-minute changes in an imminent version of a product. Fixed architecture and fluid methodology interact to help manage the product while relaxing controls on the software process. Component-based development and tools speed the software creation process. Ignoring maintenance loosens concerns about the impact of last-minute changes on maintainability. Even customer implantation facilitates requirements change by improving communication.

Build on successful experience. Component-

Both agile and traditional principles stress the importance of developing toward a “just-right” set of requirements.

Good teamwork can result from the right mix of people and the right process framework.

based development facilitates successful component reuse, but agile practices highly value developers' knowledge as well. The "right" people can mean project success. In some parallel development projects, specialized teams—requirements, programming, quality control, and so on—roll releases out in stages. Many development tool environments have mechanisms to preserve information and notes on architecture, design, or implementation decisions. Customer implantation lets customers and developers exchange tacit knowledge and facilitates construction based on deep knowledge of software development and the application domain.

Develop good teamwork. Good teamwork can result from the right mix of people and the right process framework. Customer implantation helps achieve the right mix of people and knowledge. Methodological flexibility lets a team find their ideal working style given the mix of characters in their group. Frequent releases yield small chunks of well-defined problems that help teams more quickly understand the task and how to work together.

Conform to project environment constraints. Methodological flexibility permits agile developers to vary their approaches or improvise when environmental constraints change. A stable architecture that helps standardize and decouple product components makes this flexibility possible. Frequent releases also let developers quickly revisit and fix problems that arise because of unrecognized environmental constraints.

Prepare for unexpected consequences from software process innovation. Methodological flexibility's "mix-and-match" strategy of dealing with development issues as they arise can create problems. Certain Internet-speed development practices mitigate these problems by placing boundaries on process innovation and compensating quickly when unexpected consequences arise. A stable architecture, component-based development, and an established development tool environment constrain process innovation. Compensatory practices for quickly fixing innovation-driven problems are also available. Frequent releases, enabled by parallel development, let developers correct problems as they occur.

Incompatibility of Internet-speed development practices and "orphaned" traditional principles

Many "orphaned" traditional principles (that is, those that were ignored by the agile methods group in the Discovery Colloquium) appear to be motivated by the need to increase quality and lower software development costs.

In Internet-speed development, developers practice parallel development, frequent releases, customer implantation, and disregard for maintenance in ways that let them forego formal specification, design, and documentation overhead. At least seven traditional principles promote such formalisms and conflict with how these practices are applied in Internet-speed development. These traditional principles include

- Quantitative measurements
- Formal quality management throughout the life cycle
- Rigorous software artifact specifications
- Stepwise software production
- Formal quality objectives
- Explicit documentation of trade-offs
- Identification and management of uncertainty

(Participants in the Montreal workshop do acknowledge the difficulty of implementing their principles in some contexts, particularly in environments requiring agility.)

Other traditional principles, while more or less ignored by agile developers, only partly conflict with Internet-speed development practices. A good example is the principle of building with and for reuse. In general, agile developers try to build "with reuse," but less "for reuse." Developers might reuse components from component libraries wherever possible but—consistent with the practice of ignoring maintenance—rarely take the time to develop their own components other than for their own immediate applications.

Another partially served traditional principle is the minimization of software component interaction. Although agile development doesn't explicitly hold this principle, it does get served somewhat accidentally because Internet-speed development heavily uses component libraries whenever possible.

The traditional principle of continuous improvement to disciplined processes is also partly and accidentally served. Internet-speed

About the Authors

development approaches are continuously tuned to align the team and its objectives. In this sense, improvements are continuous. However, the degree to which agile developers value these approaches for their “disciplined” nature is negligible.

Our analysis suggests at least four implications for software management:

- Cost and quality do not drive Internet-speed software development. Rather, development speed is paramount. Development costs become more aligned with operating costs. Quality becomes negotiable, a notion of quality-in-use where the exact quality requirements are a moving target in play with functionality and product availability.
- Project management in Internet-speed development differs from project management in traditional development. Projects do not begin or end, but are an ongoing operation more akin to operations management. Development problems are chunked into small jobs that can be rolled out as small, tailor-made products.
- Maintenance in Internet-speed development is sometimes merged into the specification-build-release cycle along with new functionality, or maintenance cycles become small project cycles (maintenance releases) interspersed with larger project cycles (functional releases).
- Human resource management differs in Internet-speed development. Team members are less interchangeable, and teams require people with initiative, creativity, and courage as well as technical knowledge, experience, and drive. The right candidates are more difficult to describe, identify, and place on Internet-speed development teams.

Many Internet-speed development practices look deceptively similar to long-standing software development practices. However, a close examination of how Internet-speed development practices unfold, and the agile principles to which these practices respond, reveals that Internet-speed software development is a fundamentally new way to develop software. ☞



Richard Baskerville is a professor and chair of computer information systems at Georgia State University. His research interests are information systems development and security. He received a PhD from the London School of Economics. He is a member of the IEEE Computer Society, the ACM, and the British Computer Society. Contact him at baskerville@gsu.edu.

Balasubramaniam Ramesh is an associate professor of computer information systems at Georgia State University. His research interests include requirements engineering and traceability in systems development, knowledge management, data mining, and Web services. He received a PhD in information systems from the Stern School of Business, New York University. He is a member of the IEEE, the ACM, AAAI, and AIS. Contact him at bramesh@gsu.edu.



Linda Levine is a senior member of the technical staff at Carnegie Mellon University's Software Engineering Institute. Her research focuses on acquisition of software intensive systems, agile software development, system of systems interoperability, diffusion of innovations, and knowledge integration and transfer. She received a PhD in rhetoric from Carnegie Mellon University. She is a member of the Association for Information Systems, National Communication Association, and cofounder and vice chair of IFIP Working Group 8.6 on Diffusion, Transfer and Implementation of Information Technology. Contact her at ll@sei.cmu.edu.

Jan Pries-Heje is an associate professor at the IT University of Copenhagen. He is also a professor of software engineering and management at the IT University of Gothenburg. His research focuses on leadership and organizational issues relating to software development. He received a PhD in computer science and business administration from Copenhagen Business School. He is a member of AIS and the Danish Computer Society. Contact him at jph@itu.dk.



Sandra Slaughter is an associate professor of information systems at Carnegie Mellon University. Her research focuses on productivity and quality improvement in software development and management of development professionals. She received a PhD in information systems and business administration from the University of Minnesota. She is a member of the ACM, the Association for Information Systems, and the Academy of Management. Contact her at sandras@andrew.cmu.edu.

References

1. D. Harter, M. Krishnan, and S. Slaughter, “Effects of Process Maturity on Quality, Cycle Time, and Effort in Software Product Development,” *Management Science*, vol. 46, no. 4, Apr. 2000, pp. 451–466.
2. M.A. Cusumano and D.B. Yoffie, “What Netscape Learned from Cross-Platform Software Development,” *Comm. ACM*, vol. 42, no. 10, Oct. 1999, pp. 72–78.
3. M.A. Cusumano and D.B. Yoffie, “Software Development on Internet Time,” *Computer*, vol. 32, no. 10, Oct. 1999, pp. 60–69.
4. M. Iansiti and A. MacCormack, “Developing Products on Internet Time,” *Harvard Business Rev.*, vol. 75, no. 5, Sept./Oct. 1997, pp. 108–117.
5. B. Ramesh, R. Baskerville, and J. Pries-Heje, “Internet Software Engineering: A Different Class of Processes,” *Annals of Software Eng.*, vol. 14, no. 1–4, Dec. 2002, pp. 169–195.
6. B.W. Boehm, “Seven Basic Principles of Software Engineering,” *J. Systems and Software*, vol. 3, no. 1, Mar. 1983, pp. 3–24.
7. A.M. Davis, “Fifteen Principles of Software Engineering,” *IEEE Software*, vol. 11, no. 6, Nov./Dec. 1994, pp. 94–101.
8. P. Bourque et al., “Fundamental Principles of Software Engineering—A Journey,” *J. Systems and Software*, vol. 62, no. 1, May 2002, pp. 59–70.