

What happened to the promise of rigorous, disciplined, professional practices for software development?

BY IVAR JACOBSON AND ED SEIDWITZ

A New Software Engineering

WHAT HAPPENED TO software engineering? What happened to the promise of rigorous, disciplined, professional practices for software development, like those observed in other engineering disciplines?

What has been adopted under the rubric of “software engineering” is a set of practices largely

adapted from other engineering disciplines: project management, design and blueprinting, process control, and so forth. The basic analogy was to treat software as a manufactured product, with all the real “engineering” going on upstream of that—in requirements analysis, design and modeling, among others.

Doing the job this way in other engineering disciplines makes sense because the up-front work is based on a strong foundational understanding, so the results can be trusted. Software engineering has had no such basis, so “big up-front design” often just has not paid off. Indeed, the ethos of software engineering has tended to devalue coders (if not explicitly, then implicitly through controlling practices). Coders, though, are the ones

who actually have to make the software work—which they *do*, regardless of what the design “blueprints” say should be done.

Not surprisingly, this has led to a lot of dissatisfaction.

Today’s software craftsmanship movement is a direct reaction to the engineering approach. Focusing on the *craft* of software development, this movement questions whether it even makes sense to *engineer* software. Is this the more sensible view?

Since it is the code that has to be made to work in the end anyway, it does seem sensible to focus on crafting quality code from the beginning. Coding, as a craft discipline, can then build on the experience of software “masters,” leading the community to build better and better code. In addi-

tion, many of the technical practices of agile development have made it possible to create high-quality software systems of significant size using a craft approach—negating a major impetus for all the up-front activities of software engineering in the first place.

In the end, however, a craft discipline can take you only so far. From ancient times through the Middle Ages, skilled artisans and craftsmen created many marvelous structures, from the pyramids to gothic cathedrals. Unfortunately, these structures were incredibly expensive and time consuming to build—and they sometimes collapsed in disastrous ways for reasons that were often not well understood.


Modern structures such as skyscrapers became possible only with the development of a true *engineering* approach. Modern construction engineering has a firm foundation in material science and the theory of structures, and construction engineers use this theoretical foundation as the basis of a careful, disciplined approach to designing the structures they are to build.

Of course, such structures still sometimes fail. When they do, however, a thorough analysis is again done to determine whether the failure was caused by malfeasance or a shortcoming in the underlying theory used in the original design. Then, in the latter case, new understanding can be incorporated into the foundational practice and future theory.


Construction engineering serves as an example of how a true engineering discipline combines craftsmanship with an applied theoretical foundation. The understanding captured in such an accepted foundation is used to educate entrants into the discipline. It then provides them with a basis for methodically analyzing and addressing engineering problems, even when those problems are outside the experience of the engineers.

From this point of view, today's software engineering is *not really an engineering discipline at all*.

What is needed instead is a new software engineering built on the experience of software craftsmen, capturing their understanding in a foundation that can then be used to



Today's software craftsmanship movement is a direct reaction to the engineering approach. Focusing on the *craft* of software development, this movement questions whether it even makes sense to engineer software. Is this the more sensible view?



educate and support a new generation of practitioners. Because craftsmanship is really all about the practitioner, and the whole point of an engineering theory is to support practitioners, this is essentially what was missing from previous incarnations of software engineering.

How does the software community go about this task of “refounding” software engineering?

The SEMAT (Software Engineering Method and Theory) initiative is an international effort dedicated to answering this question (<http://www.semat.org>). As the name indicates, SEMAT is focusing both on supporting the craft (*methods*) and building foundational understanding (*theory*).

This is still a work in progress, but the essence of a new software engineering is becoming clear. The remainder of this article explores what this essence is and what its implications are for the future of the discipline.

Engineering Is Craft Supported by Theory

A *method* (equivalently, *methodology* or *process*) is a description of a way of working to carry out an endeavor, such as developing software. Ultimately, all methods are derived from experience with what does and does not work in carrying out the subject endeavor. This experience is distilled, first into rules of thumb and then into guidelines and, when there is consensus, eventually into standards.

In a craft discipline, masters, who have the long experience necessary, largely develop methods. In older times, the methods of a master were closely guarded and passed down only to trusted apprentices. In today's world, however, various approaches based on the work of master craftsmen are often widely published and promoted.

As a craft develops into an engineering discipline, it is important to recognize commonality between the methods of various masters, based on the underlying shared experience of the endeavor being carried out. This common understanding is then captured in a *theory* that can be used as a basis for all the different methods to be applied to the endeavor.

In this sense, *theory* is not the bad word it is sometimes treated as in our culture (“Oh, that’s just a theory”). As noted earlier, having a theoretical foundation is, in fact, the key that allows for disciplined engineering analysis. This is true of material science for construction engineering, electromagnetic theory for electrical engineering, aerodynamics for aeronautical engineering, and so forth.

Of course, the interplay between the historical development of an engineering discipline and its associated theory is generally more complicated than this simple explanation implies. Engineering experience is distilled into theory, which then promotes better engineering, and back again. Nevertheless, the important point to realize here is this: traditional software engineering did not have such an underlying theory.

One might suggest computer science provides the underlying theory for software engineering—and this was, perhaps, the original expectation when software engineering was first conceived. In reality, however, computer science has remained a largely academic discipline, focused on the science of computing in general but mostly separated from the creation of software-engineering methods in industry. While “formal methods” from computer science provide the promise of doing some useful theoretical analysis of software, practitioners have largely shunned such methods (except in a few specialized areas such as methods for precise numerical computation).

As a result, there have often been cycles of dueling methodologies for software “engineering,” without a true foundational theory to unite them. In the end, many of these methods did not even address the true needs of the skilled craft practitioners of the industry.

So, how to proceed?

The creation of a complete, new theory of software engineering will take some time. Rather than starting with an academic approach, we can begin, as already mentioned, by capturing the commonality among the methods that have proven successful in the craft of software development. This, in turn, requires a common way

of describing, understanding, and combining various software-development techniques, instead of setting them up in competition with each other.

To see how this might be accomplished, let’s take a closer look at methods and the teams of practitioners that really use them.

Agility Is for Methods, Not Just Software

The current movement to promote

agility in software development complements the recognition of software craftsmanship. As the name suggests, agile software development is about promoting flexibility and adaptability in the face of inevitably changing requirements. This is done by producing software in small increments, obtaining feedback in rapid iterations, and continually adjusting as necessary.

Agile software-development teams take charge of their own way of working. Such a team applies the methods

Figure 1. The kernel alphas.

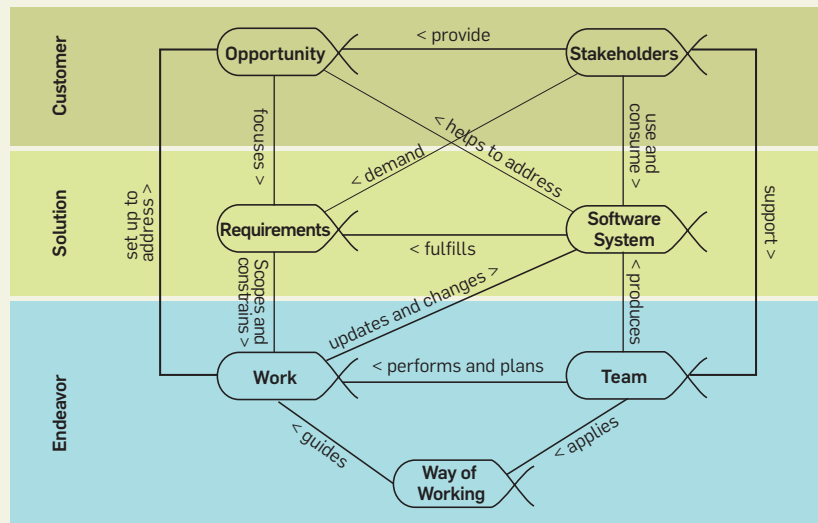


Figure 2. Tracking progress with alphas.

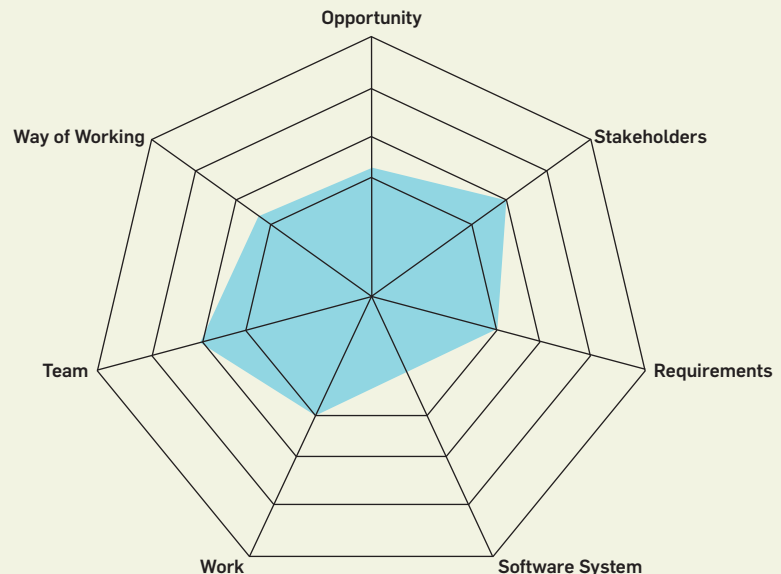
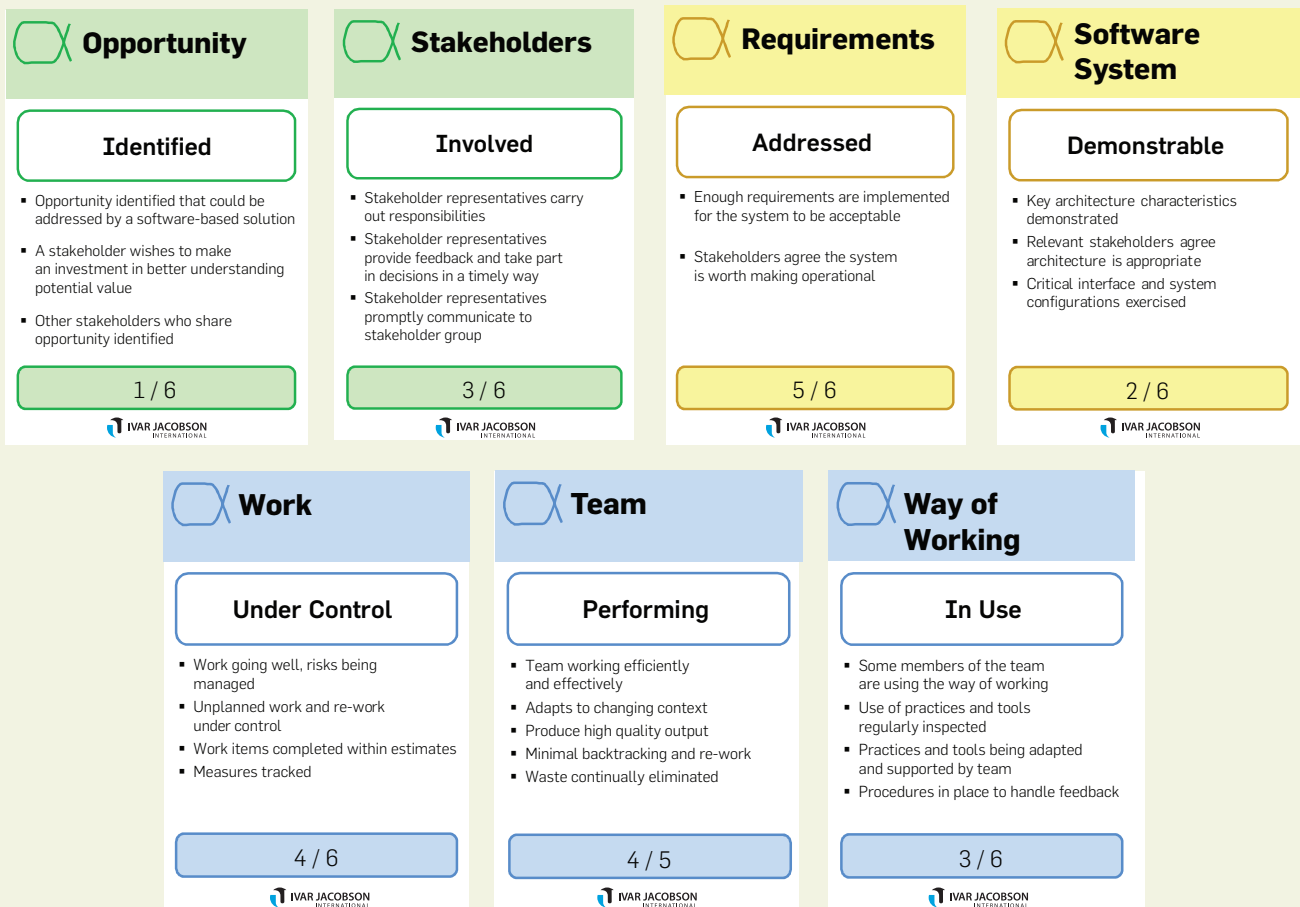


Figure 3. Alphas made tangible with cards.



it feels it needs for the project at hand as they are needed, adapting the development process throughout a project. In effect, an agile team needs to evolve and improve its methods in as agile a fashion as it develops its software.

A lack of agility in methods is a central failure of traditional software engineering.

Software is, by its very nature, malleable and (physically) easy to change. A complicated software system, however, can exhibit a kind of intellectual rigidity in which it is difficult to make changes correctly, with each change introducing as many or more errors as it resolves. In the face of this, the response of traditional software engineering was to adopt process-control and project-management techniques such as those used to handle similar problems with complicated hardware systems.

From an agile viewpoint, however,

the application of hardware-engineering techniques was a mistake. Agile techniques, instead, take advantage of the changeable nature of software, using quick feedback cycles allowed by continuous integration and integrated testing to manage complexity, rather than process control. As a result, agile development focuses on supporting the practitioner in building quality software, rather than requiring the practitioner to support the process.

So, how do you introduce agility into software-engineering methods? By looking at the basic things that practitioners actually do—their practices.

Methods Are Made from Practices

A method may appear monolithic, but any method may be analyzed as being composed of a number of *practices*. A

practice is a repeatable approach to doing something with a specific purpose in mind. Practices are the things that practitioners actually *do*.

For example, the agile method of Extreme Programming is described as having 12 practices, including pair programming, test-driven development, and continuous integration. The agile framework Scrum, on the other hand, introduces practices such as maintaining a backlog, daily scrums, and sprints. Scrum is not really a complete method but a composite practice built from a number of other practices designed to work together. Scrum, however, can be used as a process framework combined with practices from, say, Extreme Programming, to form the method used by an agile team.

That exemplifies the power of explicitly considering how methods are made up of practices. Teams can pull

together the practices that best fit the development task at hand and the skills of the team members involved. Further, when necessary, a team can evolve its method in not only small steps, but also more radical and bigger steps such as replacing an old practice with a better practice (without having to change any other practices).

Note how the focus is on teams and the practitioners in teams, rather than “method engineers,” who create methods for other people to carry out. Creating their own way of working is a new responsibility for a lot of teams, however, and it is also necessary to support a team’s ability to do this across projects. It is also useful, therefore, to provide for groups interested in creating and extending practices, outside of any specific project, so they can then be used as appropriate by project teams.

This can be seen as a separation of concerns: practices can be created and grown within an organization, or even by cross-organization industry groups (such as is effectively the case with Extreme Programming and Scrum); practitioners on project teams can then adopt, adapt, and apply these practices as appropriate.

What assurance do project teams have that disparately created practices can actually be smoothly combined to produce effective methods? This is where a new software-engineering foundation is needed, independent of practices and methods but able to provide a common underpinning for them.

The Kernel Is the Foundation for Practices and Methods

The first tangible result of the SEMAT initiative is what is known as the kernel for software engineering. This kernel can be thought of as the minimal set of things that are universal to all software-development endeavors. The kernel consists of three parts:

- ▶ A means for measuring the progress and health of an endeavor.
- ▶ A categorization of the activities necessary to advance the progress of an endeavor.
- ▶ A set of competencies necessary to carry out such activities.

Of particular importance is having a common means for understanding

how an endeavor is progressing. The SEMAT kernel defines seven dimensions for measuring this progress, known as alphas. (The term *alpha* was originally an acronym for abstract-level progress health attribute but is now simply used as the word for a progress and health dimension as defined in the kernel. Many other existing terms were considered, but all had connotations that clashed with the essentially new concept being introduced for the kernel. In the end, a new term was adopted without any of the old baggage.) The seven dimensions are: opportunity, stakeholders, requirements, software system, work, team, and way of working. These alphas relate to each other as shown in Figure 1.

Each alpha has a specific set of states that codify points along the dimension of progress represented by the alpha. Each of the states has a checklist to help practitioners monitor the current state of their endeavor along a certain alpha and to understand the state they need to move toward next. The idea is to provide an intuitive tool for practitioners to reason about the progress and health of their endeavors in a common, method-independent way.

One way to visualize the seven-dimensional space of alphas is using the spider chart¹ shown in Figure 2. In this chart, the gray area represents how far an endeavor has progressed, while the white area shows what still needs to be completed before the endeavor is done. A quick look at such a diagram provides a good idea of where a project is at any point in time.

The alphas can be made even more tangible by putting each of the alpha states on a card, along with the state checklist in an abbreviated form (see Figure 3). The deck of all such cards can then fit easily into a person’s pocket. Although more detailed guidelines are available, these cards contain key reminders that can be used by development teams in their daily work, much like an engineer’s handbook in other disciplines.

A more complete discussion of the kernel and its application is available in previous work.^{2,3} The kernel itself is formally defined as part of the Essence specification that has been standardized through the Object Management

Group.⁶ In addition to the full kernel, the Essence standard also defines a language that can be used both to represent the kernel and to describe practices and methods in terms of the kernel. Importantly, this language is intended to be usable by practitioners, not just method engineers; for basic uses, it can be learned in just a couple of hours (the alpha state cards are an example of this).

Of course, this ability to use the kernel to describe practices is exactly what is needed as a foundation for true software-engineering methods.

Practices Built on the Kernel Enable Agile Methods

A practice can be expressed in terms of the kernel by:

- ▶ Identifying the areas in which it advances the endeavor.
- ▶ Describing the activities used to achieve this advancement and the work products produced.
- ▶ Describing the specific competencies needed to carry out these activities.

A practice can also extend the kernel with additional states, checklists, or even new alphas.

The critical point is that the kernel provides a common framework for describing all practices and allowing them to be combined into methods. Bringing a set of practices into this common system allows gaps and overlaps to be more easily identified. The gaps can then be filled with additional practices and the overlaps resolved by connecting the overlapping practices together appropriately.

For example, consider two practices: one about using a backlog to manage the work to be carried out by a team (advancing the work alpha); the other about defining requirements using user stories (advancing the requirements alpha). The backlog practice does not prescribe what the items on the backlog must be, while the user-story practice does not prescribe how the team should manage the implementation of those stories. The two practices are thus complementary and can be used together—but, when so combined, they overlap. The two practices can be connected in a smooth and intuitive way within an overall method by identifying backlog

items from the one with user stories from the other, so that user stories become the items managed on the backlog.

Note, in particular, how the common framework of the kernel provides a *predictive* capability. A construction engineer can use material science and the theory of structures to understand at an early stage whether a proposed building is likely to stand or fall. Similarly, using the kernel, a software developer can understand whether a proposed method is well constructed, and, if there are gaps or overlaps in its practices, how to resolve those.

Further, through the separation of concerns discussed earlier, an organization or community can build up a library of practices and even basic methods that a new project team may draw on to form its initial way of working. Each team can then continue to agilely adapt and evolve its own methods within the common Essence framework.⁴

Ultimately, the goal will be, as an industry, to provide for the standardization of particularly useful and successful practices, while enhancing, not limiting, the agility of teams in applying and adapting those practices, as well as building new ones as necessary. And that, finally, is the path toward a true discipline of software engineering.

Conclusion

The term *paradigm shift* may be a bit overused these days; nevertheless, the kernel-based Essence approach to software engineering can quite reasonably be considered to be such a shift. It truly represents a profound change of viewpoint for the software-engineering community.

When Thomas Kuhn introduced the concept of a paradigm shift in his influential book, *The Structure of Scientific Revolutions*,⁵ he stressed the difficulty (Kuhn even claimed impossibility) of translating the language and theory of one paradigm into another. The software-development community has actually seen such shifts before, in which those steeped in the old paradigm have trouble even understanding what the new paradigm is all about. The move to object orientation was one such shift, as, in many ways,

is the current shift to agile methods.

In this regard, Essence can, indeed, be considered a paradigm shift in two ways. First, those steeped in the “old school” of software engineering have to start thinking about the true engineering of software specifically, rather than just applying practices largely adapted from other engineering disciplines. Second, those in the software craftsmanship and agile communities need to see the development of a true engineering discipline as a necessary evolution from their (just recently hard-won!) craft discipline.

In regard to the second point, in his foreword to *The Essence of Software Engineering: Applying the SEMAT Kernel*,³ Robert Martin, one of the SEMAT signatories, describes a classic pendulum swing away from software engineering toward software craftsmanship. Martin’s assessment is correct, but it is important to note that this proverbial pendulum should not simply swing back in the direction it came. To the contrary, while swing it must, it now needs to swing in almost a *90-degree different direction* from which it came, in order to move toward a new discipline of true software engineering.

There is, perhaps, hardly a better image for a paradigm shift than that. In the end, the new paradigm of software engineering, while building on the current paradigm of software craftsmanship, must move beyond it, but it will also be a shift away from the old paradigm of traditional software engineering. And, like all paradigm shifts before, this one will take considerable time and effort before it is complete—at which point, as the new paradigm, everyone will consider its benefits obvious.

Even as it stands today, though, using Essence can provide a team with some key benefits. Essence helps teams to be agile when working with methods and to measure progress in terms of real outcomes and results of interest to stakeholders. These progress measurements are not only on one dimension, but along the seven dimensions of the kernel alphas, all of which need to move along at some pace to reduce risks and achieve results.

Further, Essence can allow orga-

nizations to simplify governance of methods, using a pool of practices that may be adopted and adapted by project teams. Having Essence as a common foundation for this also allows practitioners to learn from one another more readily.

The real shift, however, will only come as teams truly realize the benefits of Essence today and as SEMAT builds on Essence to complete the new software engineering paradigm. A community of practitioners is now contributing their experience and becoming part of this “refounding” of software engineering—or, perhaps, really founding it for the first time. ■

Related articles on queue.acm.org

The Essence of Software Engineering: The SEMAT Kernel

Ivar Jacobson, Pan-Wei Ng, Paul E. McMahon, Ian Spence and Svante Lidman
<http://queue.acm.org/detail.cfm?id=2389616>

First, Do No Harm: A Hippocratic Oath for Software Developers

Phillip A. Laplante
<http://queue.acm.org/detail.cfm?id=1016991>

Software Development with Code Maps

Robert DeLine, Gina Venolia and Kael Rowan
<http://queue.acm.org/detail.cfm?id=1831329>

References

1. Graziotin, D. and Abrahamsson, P. A Web-based modeling tool for the SEMAT Essence theory of software engineering. *J. Open Research Software* 1, 1 (2013), e4; <http://dx.doi.org/10.5334/jors.ad>.
2. Jacobson, I., Ng, P.-W., McMahon, P., Spence, I. and Lidman, S. The Essence of software engineering: The SEMAT kernel. *ACM Queue* 10, 10 (2012); <http://queue.acm.org/detail.cfm?id=2389616>.
3. Jacobson, I., Ng, P.-W., McMahon, P. E., Spence, I. and Lidman, S. *The Essence of Software Engineering: Applying the SEMAT Kernel*. Addison-Wesley, Reading, PA, 2013.
4. Jacobson, I., Spence, I. and Ng, P.-W. Agile and SEMAT—Perfect partners. *Comm. ACM* 6, 11 (Nov. 2013); <http://cacm.acm.org/magazines/2013/11/169027-agile-and-semat/abstract>.
5. Kuhn, T. *The Structure of Scientific Revolutions*. University of Chicago Press, 1962.
6. Object Management Group. Essence—Kernel and Language for software engineering methods, 2014; <http://www.omg.org/spec/Essence>.

Ivar Jacobson is the founder and chairman of Ivar Jacobson International. He is a father of components and component architecture, use cases, aspect-oriented software development, modern business engineering, the Unified Modeling Language, and the Rational Unified Process.

Ed Seidewitz is the former CTO, Americas, for Ivar Jacobson International and is currently chair of the ongoing Essence Revision Task Force. With Ivar Jacobson International, he has led agile system architecture and development engagements in both the commercial and government sectors and participated in practice development.

Copyright held by owners/authors. Publication rights licensed to ACM. \$15.00.

Copyright of Communications of the ACM is the property of Association for Computing Machinery and its content may not be copied or emailed to multiple sites or posted to a listserv without the copyright holder's express written permission. However, users may print, download, or email articles for individual use.