Issues in Software Engineering of Relevance to Instructional Design

By Ian Douglas

"Many academic thinkers, when considering methodology, develop prescriptions for a methodological process. Few of these academic models are adopted on a wide scale. Rather than trying to define the ultimate standard methodology, another approach is to standardize the categorization of methodologies."

Software engineering is popularly misconceived as being an up-market term for programming. In a way, this is akin to characterizing instructional design as the process of creating PowerPoint slides. In both these areas, the construction of systems, whether they are learning or computer systems, is only one part of a systematic process. The most important parts of this process, analysis and design, precede the actual construction. In studies of software failure, the failure is more often traced to poorly stated or missing requirements than it is non-functional code (Standish Group International, 1999). Even when programs are functional, the interface design may prevent easy access to that functionality by end-users.

There is scope for instructional designers to use some of the body of research and experience in software engineering, especially as technology increasingly infuses learning systems. Goodyear (1995) and Bostock (1998) both refer to "courseware engineering," which represents the intersection of the fields of instructional design and software engineering. Other attempts to draw parallels between the two areas include Wilson, Jonassen and Cole (1993), who note how software engineering has largely moved away from the linear process model, still prevalent in instructional design, toward more iterative approaches utilizing prototyping. Also, an emerging concept in instructional design borrowed from the software engineering world is that of the learning object. In this article, I will introduce several software-engineering process issues of relevance to the development of methodological thinking in instructional design.

Categorizing approaches to methodology

Many academic thinkers, when considering methodology, develop prescriptions for a methodological process. Few of these academic models are adopted on a wide scale; instead, customized processes emerge within different organizations. Given the vast varieties of practice, rather than trying to define the ultimate standard methodology, another approach is to standardize the categorization of methodologies.

The Software Engineering Institute at Carnegie Mellon University developed the capability maturity model (CMM) to facilitate such an approach. The CMM is a system for measuring the quality of the processes used within a software development organization. The CMM provides a means for the qualitative evaluation of processes without the need to follow a specific methodology.

The CMM (Paulk, Weber, Curtis, & Chrissis, 1995) includes five levels of maturity: initial, repeatable, defined, managed and optimized. In the initial level, everything is done in an ad hoc manner with no formal organization

or process management. The process is unpredictable and dependent on individuals. It is not possible to give definite answers on the time and cost involved in a product development.

At the repeatable level, polices and procedures are established to ensure successful practices are repeated. Discipline is brought to development projects through a project management system. Project and product standards are defined and the organization ensures they are followed.

At the defined level, an organization will have a documented and well-defined standard process, which integrates project management and development methods. The process will be well defined in that it has inputs, outputs, standards (e.g., for documentation or modeling), completion criteria and verification mechanisms (e.g., peer review). A unit within the organization is assigned responsibility for the process and organization-wide process training is made available.

At the managed level, the organization has quantitative quality goals for both products and processes. An organizational database is use to collect and analyze the data from projects. This gives a measure of predictability to the process as metrics on a current project can be compared with those on past projects. An organization can look for trends and identify when new approaches (e.g., the use of a new development tool) lead to better results.

At the optimized level the entire organization has a clearly defined goal of continual process improvement and established numerical measures, and control techniques are used to guide the organization toward higher process quality and productivity. The organization actively works to identify new innovations that improve performance and transfers proven improvements throughout the organization.

The CMM model has gained widespread acceptance and many software development contracts, particularly in the government, specify that bidders must have reached at least level 3 on the CMM. An organization that is aware of the CMM levels recognizes the value of the higher levels, is objective about its own level and the need for improvement and has a good start on process improvement.

Is there room for a CMM tailored for the instructional design processes? Although the CMM specifically addresses software engineering processes, much of it can apply to any product development process. It would be interesting to speculate how many organizations involved in producing instruction would score the equivalent of a Level 5.

Heavy or agile?

Over the past several years, there has been a growing revolution among many software engineering practitioners against the traditional approaches to methodology, which are prominent in most textbooks (e.g., Pressman, 2000). Traditional models derive from what is called the waterfall approach, which is similar in structure to the ADDIE model that instructional designers are familiar with. Variations of this model have been developed and augmented since the early seventies. Some traditional models are now so complex and prescriptive that many software companies have large manuals describing their methodology in detail and even go to the extent of requiring forms to be filled out if anyone wishes to deviate from the prescribed method.

The criticisms expressed in the software world against traditional models are similar to those in the "Attack on ISD" article published in *Training* magazine (Gordon & Zemke, 2000). The main criticisms in this article — that ISD as a process is too slow and clumsy, assumes superiority without empirical evidence, can still produce bad solutions and clings to the wrong world view — would resonate with critics of traditional software design approaches. The difference is that the critics of traditional software engineering methodologies have proposed some well-developed alternatives.

The alternatives have been referred to as light, new or agile methodologies and have generally been developed by practitioners rather than academics.

"The main criticisms [of ISD] would resonate with critics of traditional software design approaches. The difference is that the critics of traditional software engineering methodologies have proposed some well-developed alternatives."

"The main difference between agile and traditional methods is that agile methods are adaptive and people-oriented rather than predictive and process-oriented."

The leading proponents of the agile approach have collaborated to state their combined vision for an alternative approach to software development in the Agile Manifesto (Agile Alliance, 2001). Fowler (2000) states that the main difference between agile and traditional methods is that agile methods are adaptive and people-oriented rather than predictive and process-oriented.

The Agile approach has some roots in the open source movement (Open Source Initiative, 2005), an earlier and still growing movement against traditional practices. One of the open source movement's criticisms of the traditional approach is that it results in software products where the code and the much of the design knowledge are hidden from the user. They advocate that code be open for inspection and modification by anyone who would wish to use or improve upon the design. This community-oriented approach has also been a theme in many of the new agile methodologies.

One of the best-known examples of an agile methodology is called extreme programming (Beck, 2000). Rather than focus on the process phases and their products, extreme programming begins with four values for the people involved: communication, feedback, simplicity and courage. It further specifies a set of guidelines for the production of software products:

- Work on short three-week cycles based around "stories." Stories describe a discrete functional unit of the product.
- Have small, frequent releases and testing of product components.
- Refactor (review the design structure) mercilessly.
- Design the test before the code.
- Conduct programming in pairs. Pair programming is seen to provide real time quality control and learning.
- Encourage collective ownership. This is a refinement of an earlier concept of ego-less programming, where there is collective responsibility for the system quality irrespective of individual work assignments.
- Continuously integrate the different components of the system.
- Maintain a 40-hour workweek to avoid burnout of creative talent.
- Have an on-site customer to provide continuous validation of requirements and functional units.

Extreme programming has gained popularity with a number of developer groups since it is based on the way they like to work and quickly generates demonstrable software products. It is also relatively easy to understand. A complaint against traditional methodologies is that a great deal of time is taken up with process artifacts (documentation) and that these must be completed and accepted before development can begin.

Extreme programming is just one example of several methodologies that come under the agile banner, and each of them borrows ideas from each other. A full review of the agile methods can be found in Abrahamsson, Salo, Ronkainen and Warsta (2002). Many of the guidelines of agile methods could be easily incorporated into an agile method for instructional design.

Critics of agile methodologies have claimed that they are too focused on the experience of small teams of highly qualified software engineers on relatively small projects. It is thought that they may be difficult to apply in large projects with fixed costs, deadlines and multidisciplinary teams. Traditional methodologies also tend to fit with traditional approaches to project management, and thus agile processes may not appeal to management in large organizations.

How do you design with objects?

The component approach to product development has evolved in physical products for centuries, gaining particular impetus with the industrial revolution. Prior to this, a craft-based approach was prevalent, where one or two individuals would create a complete product from the raw materials available to them. In software development, the transition

from a craft-based development has not been immediate.

The component approach has a number of benefits. The main benefit is in allowing reuse (i.e., a component design used on one product can be used to provide the same function for another product). This is possible if there is a standard way of connecting the components.

Additional benefits of components include ease of maintenance (i.e., if a problem arises in a product, it is possible to identify the faulty component and replace it). In addition, speed of new product development can be increased since products are assembled from different combinations of existing components. Incremental improvement is also possible as new and improved components become available.

This paradigm shift in computer software development is changing how people think about software design. Designers can first look for components that already exist for the functionality they wish to achieve rather than having to craft a whole application through their own efforts. The web has facilitated this through the creation of a number of sites where components can be obtained.

The idea of moving to a component model for instructional design has arisen relatively recently and been driven by the interest in the educational potential of the web. There are now a number of initiatives that seek to transfer the ideas and benefits of the component approach to the development and delivery of learning systems. The initiatives include efforts to establish technical standards required for learning components. These have strong support from the vendors of learning management systems and major users of learning technology such as the U.S. Department of Defense (Advanced Distributed Learning, 2005).

The concept of "software components" (also referred to as "web services") has emerged over the last decade, built upon the older concepts of object-oriented software engineering. In software engineering, a component is a self-contained mini-program that provides a distinct functionality. Component-based applications are assembled by connecting the components together (across the internet in the case of web services). This seems to be similar to most current conceptions for "learning objects." The concept of object-oriented programming, which shares the main aim of promoting reuse, is a little more complex than that of software components.

There is great interest in how object/component-based systems are organized into effective architectures (Yourdon & Constantine, 1979). Effective architectures ease maintenance (one of the biggest costs in software) and facilitate systems being extended to cope with changing requirements. Two key concepts that are often stressed are cohesion and coupling. Software system architects will strive to achieve high cohesion and low coupling in their systems.

A system is said to be highly cohesive if its components have a well-defined function to perform within a system; if a component has a number of different functions to perform, its lacks cohesion. If this concept were adopted in relation to learning objects, it could translate to each object being tied to no more than one learning objective. It might also translate to the separation of instruction and assessment into different objects. The idea here is that if an object has more than one distinct function, it is harder to maintain, replace or reuse.

A system is said to be highly coupled if its components are interlinked and dependent upon one another. In a highly coupled system it is difficult to reuse an individual component independent of the other objects to which it is coupled. The acceptance of a package of coupled components is often required, even when the function of only one component is needed in a new system. In relation to learning objects, this means it might be difficult to equate a section or chapter of a book directly with a learning object, since sections are often written on the assumption that the reader has access to the whole book. Sections and chapters of the

"The concept of 'patterns' has gained prominence as an alternative way to achieve reuse in the software world and could also be adapted to facilitate reuse in instructional design. Patterns are focused on the reuse of design knowledge rather than the reuse of artifacts produced in prior design efforts."

book will often cross-reference each other and this makes it more difficult to reuse specific material outside the context of the book. The low coupling approach has been adopted in the ADL SCORM (Advanced Distributed Learning, 2005) concept of objects, where there is a specific requirement to separate the navigation and sequencing of content from the content itself.

While it is seen as desirable to design software systems as highly cohesive and lowly coupled, achieving this goal is often difficult (Nandigam, Lakhotia, & Cech, 1999). It is more difficult to achieve if designers are not specifically taught and encouraged to apply these concepts.

Among the other related concepts that have arisen in component-based systems is tiered architectures. This approach essentially categorizes components for different purposes within a system, the main ones being boundary (interface to users and/or other systems), control (e.g., processing data) and data storage. More thinking is required along these lines for learning objects. How should objects be categorized and organized into functional systems? The currently amorphous concept of the learning object may have to evolve into a taxonomy of learning-related objects with at least a definite separation between learning content and its presentation.

Moving from objects to patterns

Some have argued that in the software world, reusable components have had limited success compared with the investment and hype they have generated. It can be argued that developers find it difficult to trust code produced by others or perceive acquiring and adapting existing components to a new context to be more time-consuming than building from scratch. Advocates of the open software movement, which requires that all source code be made available, would argue they have a solution to the trust issue.

The concept of "patterns" has gained prominence as an alternative way to achieve reuse in the software world and could also be adapted to facilitate reuse in instructional design. Patterns are focused on the reuse of design knowledge rather than the reuse of artifacts produced in prior design efforts. The concept first came to prominence in architecture, where Christopher Alexander (1979) argued that commonly occurring patterns could be identified in successful town/building/room designs. He began describing, rating and cataloguing these patterns. A pattern is a record of how a particular recurring problem has been solved successfully in the past. It is general enough to be adapted and reused in a way that matches a particular situation. A pattern attempts to provide the best solution to a problem by recognizing and recording principles that are practiced by the best designers. One of Alexander's own patterns related to the design of workspaces for optimal learning is presented in Table 1. The patterns would be illustrated by photographs or diagrams that help illuminate the solution explained in the pattern. Alexander rates his patterns according to their significance, with some patterns being "more true, more profound, more certain than others." Communities of software developers have come together to propose and evaluate patterns for software design problems, and there is an annual conference specifically devoted to patterns.

There is scope for a similar approach occurring in instructional design. Pedagogical patterns are collections of common learning design problem-solution pairings. One community of computer science educators has already begun to establish a collection of pedagogical patterns for computer education based on their collective experience (Pedagogical Patterns Project, 2005). The active involvement of instructional designers in such communities is likely to have a beneficial effect, and indeed a catalogue of abstract patterns (that could apply across a range learning domains) developed by the instruc-

Name: Master and Apprentices *

Problem: The fundamental learning situation is one in which a person learns by helping someone who really knows what he/she is doing.

Forces: learning from lectures and books is dry as dust…The schools and universities have taken over and abstracted many ways of learning which in earlier times were always closely related to the real work of professionals, tradesmen, artisans, independent scholars … An experiment by Alexander and Goldberg has shown that a class in which one person teaches a small group of others is most likely to be successful in those cases where the "students" are actually helping the "teacher" to do something, to solve some problem, which he is working on anyway — not when a subject of abstract or general interest is being taught. (Report to the Muscatine Committee, on experimental course ED. 10X, Department of Architecture, University of California, 1966).

Solution: Arrange the work in every workgroup, industry and office, in such a way that work and learning go forward hand in hand. Treat every piece of work as an opportunity for learning. To this end, organize work around a tradition of masters and apprentices and support this form of social organization with a division of the workplace into spatial clusters — one for each master and his apprentices — where they can meet and work together.

Table 1. Example of one of Alexander's patterns

tional design community would be a useful resource. Pedagogical patterns, in addition to providing a description of common problem-solution pairing, could refer to empirical studies to support the proposed solution.

Seeing the plan before you build

In recent years, one of the most prominent developments in software engineering is the widespread adoption of a standard set of notations, the unified modeling language (UML), which allows developers to model problem and solution domains. This follows the practice in other design domains such as architecture, which have standard visual communication languages. UML evolved from the fusion of notations from different development methods (Booch, 1999) and has been adopted as an industry standard. It contains several different diagramming methods for mapping the features, structure and information flows of a system.

The benefit of a standard notation is that it allows a design team to create and evaluate a detailed model of what it intends to build. Correcting flaws in an architectural plan for a building saves effort and cost over correcting flaws once the building has been constructed. In the same way, UML can save the costs of constructing software that has a design flaw or does not meet customer requirements.

Currently, the most commonly used modeling language in ISD is concept mapping. A UML-like modeling language for instructional design was created at Open University in the Netherlands and has been integrated into the Learning Design Specification of the IMS global learning consortium (2005), one of the main bodies involved in learning technology standards.

Computer-aided software engineering (CASE)

Automation in education has tended to focus on development and delivery tools for computer-based instruction. There are a number of commercial tools for development (authoring tools) and delivery (learning management systems). However, although research has been done on automating analysis and design (Goodyear, 1997; Spector & Muraida, 1997), there are relatively few fully developed and widely used software tools in this category. Spector and Muraida (1997) argue the need for such tools by noting that there is "a lack of ID expertise, pressures for increased productivity of designers, and the need to standardize products and ensure the effectiveness of products." The same can be said for software design.

Pressman (2000) notes that until relatively recently, there has been a similar lack of design tools in software engineering where practitioners constructed automated systems for others but used little automation themselves. Computer-aided software engineering (CASE) has matured to the level that there is now a range of commercial tools covering the entire process lifecycle.

CASE tools can be divided into high and low CASE, with the low CASE tools supporting programming and other development activities such as debugging and the high CASE tools supporting analysis and design. The UML modeling approach discussed in a previous section has been integrated into a number of high CASE tools. The tools are often marketed on the basis that they can generate some of the required software code from the design models and thus shortcut from design to development. IBM's Rational Software (2005) is a prominent example of a CASE tool built around the use of UML. Figure 1 illustrates an early version of the Rational Rose Case tool, in which a UML use case diagram has been created. In this particular example, a system whose boundaries are determined by the square has three "use cases" (ellipses), which are the essential functions of the system. The use cases are made available to external "actors" (stick figures) on the system. This kind of diagram would be used to analyze the requirements for a system; the use cases would have more detailed procedural descriptions

"Compared with the work in automated ISD, there has been much more success in developing commercially successful high CASE tools. The lack of an established standard modeling language (for ISD) may be part of the reason."

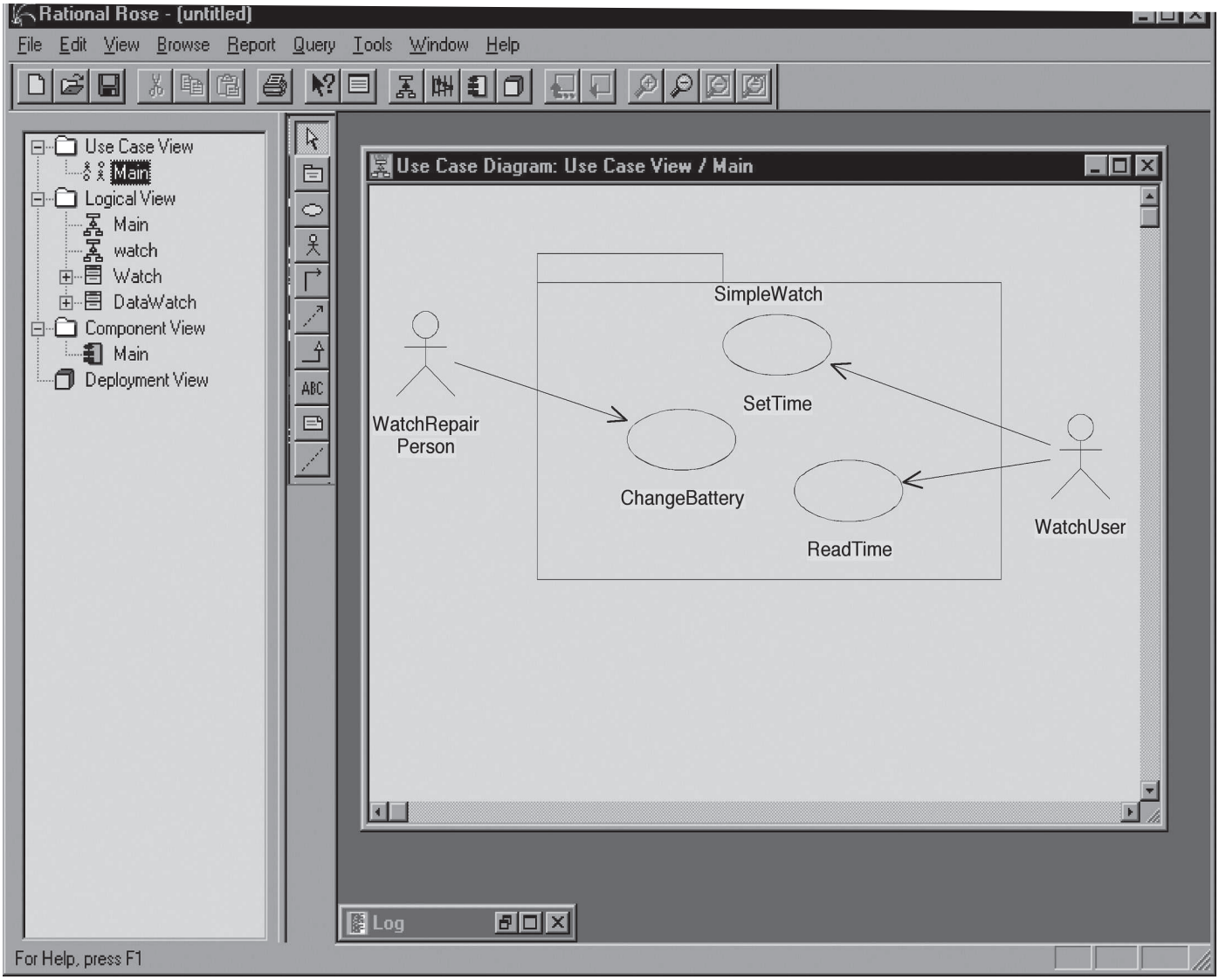

Figure 1. A UML model being constructed in a CASE tool

"If the barriers of jargon can be overcome, it may become apparent that many of the issues in design-related disciplines are the same, and there is great potential for sharing knowledge."

attached to them based on scenarios (stories describing the system's use). Other diagrams are created to illustrate different views of the system, e.g., how objects would connect together. Code can be generated from the most detailed structural view of the system, which, together with the diagrams, can ease the work of the programmer.

Compared with the work in automated ISD, there has been much more success in developing commercially successful high CASE tools. It is not easy to speculate why, but the lack of an established standard modeling language may be part of the reason. It is certainly true that the increasing acceptance of UML and its integration into software design tools had a positive effect on their popularity. There is scope for transferring much of the thinking in current CASE tools to the construction of tools to assist in the design of courses and content. This may be a direction for some of the many e-learning companies that are currently focused primarily on learning management systems and authoring tools.

Conclusion

This article has identified some areas of relevant knowledge that can be adapted and transferred from software engineering to instructional design. Software engineering is a large and diverse domain, and there are a number of other areas that could have been mentioned, such as software usability analysis and testing. There is

a definite parallel between the two areas, particularly as technology increasingly infuses learning systems.

Sharing knowledge across disciplines is often difficult due to the insular nature of much of academia and the communication silos through which knowledge must cross. However, if the barriers of jargon can be overcome, it may become apparent that many of the issues in design-related disciplines are the same, and there is great potential for sharing knowledge.

The development of better process thinking, design techniques and tools will provide an impetus toward better systems in general. I would urge instructional designers to look beyond their own journals and conferences for reusable "objects" of knowledge that exist in other domains. I would also urge software engineers to do as I have and look at instructional design for insights into how the human element can be better incorporated into technology-based systems design.

Ian Douglas has a PhD in computer science, an MA in psychology and an MSc in computing and cognition. He is a professor with a joint appointment between the Learning Systems Institute and the College of Information at Florida State University. Dr Douglas's research interests are in human-computer interaction, human performance technology, knowledge management and IT systems in support of systems design. He has a number of publications relating to training and technology and has received three awards for innovations in the use of educational technology. He is currently the principal investigator on a research project investigating software models for the management of human performance knowledge in the military.

References

Abrahamsson, P., Salo, O., Ronkainen, J., & Warsta, J. (2002). Agile software development methods: Review and analysis. Retrieved January 22, 2006, from http://www.inf.vtt.fi/pdf/publications/2002/P478.pdf

Advanced Distributed Learning. (2005). Retrieved January 22, 2006, from http://www.adlnet.org/

Agile Alliance. (2001). *The agile manifesto.* Retrieved January 22, 2006, from http://www.agilemanifesto.org/

Alexander, C. (1979). *The timeless way of building*. New York: Oxford University Press.

Beck, K. (2000). *Extreme programming explained: Embrace change*. Reading, MA: Addison Wesley.

Booch, G. (1999). UML in action. *Communications of the ACM, 44*(10), 26-28.

Bostok, S. (1998). Courseware engineering: An overview of the course development process. Retrieved January 22, 2006, from http://www.keele.ac.uk/depts/cs/Stephen_Bostock/docs/atceng.htm

Fowler, M. (2000, December). Put processes on a diet. *Software Development Online*. Retrieved January 22, 2006, from http://www.sdmagazine.com/documents/s=737/sdm0012a/

Goodyear, P. (1995). Infrastructure for courseware engineering. In R.D. Tennyson & A. E. Barron (Eds.), *Automating instructional design: Computer-based development and delivery tools* (pp. 11-31). Berlin: Springer-Verlag.

Goodyear, P. (1997). Instructional design environments: Methods and tools. In S. Dijkstra, N.Seel, F. Schott & D. Tennyson, (Eds.), *Instructional design: International perspectives* (Vol. 2) (pp. 83-111). Mahwah, NJ: Lawrence Erlbaum.

Gordon, J., & Zemke, R. (2000). The attack on ISD. *Training*, *37*(4), 42-56.

IBM Rational Software. (2005). New to Rational. Retrieved January 22, 2006, from http://www-128.ibm.com/developerworks/rational/newto/

IMS global learning consortium. (2005). Retrieved January 22, 2006, from http://www.imsglobal.org/learningdesign/

Nandigam, J. Lakhotia, A. & Cech C., (1999). Experimental evaluation of agreement between programmers in applying the rules of cohesion. *Journal of Software Maintenance: Research and Practice*, *11*, 35-53.

Open Source Initiative. (2005). Retrieved January 22, 2006, from http://www.opensource.org/

Paulk, C. V., Weber, C. V., Curtis, B., & Chrissis, M. B. (1995). The capability maturity model: Guidelines for improving the software process. Reading, MA: Addison-Wesley.

Pedagogical Patterns Project. (2005). Retrieved January 22, 2006, from http://www.pedagogicalpatterns.org/current/right.html

Pressman, R. S. (2000). *Software engineering: A practitioner's approach*. New York: McGraw-Hill.

Spector, J. M., & Muraida, D. J. (1997). Automating instructional design. In S. Dijkstra, N.Seel, F. Schott & Tennyson, D. (Eds.), *Instructional design: International perspectives* (Vol. 2) (pp. 59-81). Mahwah, NJ: Lawrence Erlbaum.

Standish Group International. (1999). CHAOS: A recipe for success. Research report. West Yarmouth, Massachusetts. Retrieved January 22, 2006, from http://www.standishgroup.com/sample_research/PDFpages/chaos199%9.pdf

Wilson, B., Jonassen, D., & Cole, P. (1993). Cognitive approaches to instructional design. In G. M. Piskurich (Ed.), *The ASTD handbook of instructional technology* (pp. 21.1-21.22). New York: McGraw-Hill.

Yourdon, E., & Constantine, L. (1979). *Structured design: Fundamentals of a discipline of computer program and systems design.* Englewood