

# LESSONS FROM RESEARCH WITH EDUCATIONAL SOFTWARE COMPONENTS

Jeremy Roschelle  
Chris DiGiano

SRI International

Manolis Koutlis

Computer Technology Institute,  
Greece

Alexander Repenning  
Jonathan Phillips

University of Colorado, Boulder

Nicholas Jackiw

KCP Technologies

Dan Suthers

University of Hawai'i at Manoa

**Contact:** *Jeremy Roschelle*, [Roschelle@acm.org](mailto:Roschelle@acm.org), (650) 859-3049  
*SRI International, 333 Ravenswood Avenue, Menlo Park CA 94025 USA*

## ABSTRACT

US schools spend \$5 billion annually on technology. However, despite the educational software market's size and potential impact on students, its fragmented nature makes it difficult for software producers to develop viable education business strategies based on traditional monolithic applications. Component-based software design radically changes this picture by supporting the design and dissemination of educational components that can be customized to specific needs. Such customization represents an important shift away from the highly generic software such as Web browsers and word processors found on computers in today's schools toward a model for educational software that provides fertile ground for cost-effective, innovative, and powerful learning tools. Although this article is centered on educational applications, it also raises general implications of component-based software design for other large but fragmented markets that require customized software.

## INTRODUCTION

Developers across many market segments are increasingly turning to component engineering techniques to cope with the growing complexity of software, the low efficiency of reinventing code, and the increasing desire to conform to standards. The most common examples of this trend involve building custom interfaces to business objects and databases using such technologies as Visual Basic, JavaBeans, and CORBA. In such examples, components have remained largely the province of full-time programmers. As evidenced by such technologies as OpenDoc, ActiveX, National Instruments' LabVIEW and Lotus's eSuite, component technologies are likely to expand their audience toward authors who are considerably less technical and more domain-oriented--authors whose job description typically does not include "software development."

One such audience is education. The demand for educational software is growing exponentially with the surge of interest in educational reform, the Internet, and distance learning. Educational applications must be very flexible because curriculums and teaching styles vary tremendously from school to school, state to state, and university to

university, and even among teachers at the same institution. On the supply side, educational software development is characterized by a wide array of small-scale, casual developers at universities, research labs, and small businesses with no dominant "solution" or supplier.<sup>1</sup> Smaller suppliers often cannot reach scale without dependency on other vendors' capabilities. Increasingly, these small developers are converging on the idea that they need a standard, common platform. These conditions are driving the evolution of a component marketplace that multiplexes creators of learning objects and authors of curricular materials so as to meet the tremendous demand for customized learning materials.<sup>2</sup> Indeed, for the educational software market to survive, component methodologies appear to be a necessity, not a luxury.<sup>3</sup>

**Table 1: Educational software projects that formed the basis for this article.**

<b>Project or Product Information</b>	<b>Description</b>
<b>E-Slate</b> Manolis Koutlis Computer Technology Institute, Greece <a href="http://E-Slate.cti.gr">http://E-Slate.cti.gr</a>	E-Slate is an end-user component-based environment for constructing Internet-aware learning activities of an exploratory nature. In addition, it provides a platform for the development of educational component libraries.
<b>ESCOT</b> Jeremy Roschelle, Chris DiGiano, Roy Pea, Jim Kaput SRI International <a href="http://www.escot.org">http://www.escot.org</a>	ESCOT is a testbed that seeks to enable interoperability and reuse of educational software components for middle school mathematics.
<b>AgentSheets</b> Alexander Repenning, University of Colorado <a href="http://www.agentsheets.com">http://www.agentsheets.com</a>	AgentSheets is an agent-based authoring environment for computer end users to build interactive simulations that can be turned in Java applets and JavaBeans.
<b>Java Sketchpad</b> Nicholas Jackiw KCP Technologies <a href="http://www.keypress.com/sketchpad/java_gsp/">http://www.keypress.com/sketchpad/java_gsp/</a>	JavaSketchpad is a Web-enabled incarnation of the Geometer's Sketchpad, a market-leading mathematics exploration tool for grades 6-14.
<b>Belvedere</b> Dan Suthers University of Hawai'i at Manoa <a href="http://lilt.ics.hawaii.edu">http://lilt.ics.hawaii.edu</a>	Belvedere enables students to construct diagrams of evidential relationships between hypotheses and empirical evidence, while receiving guidance from an automated advisor and collaborating via a network.

Explorations in the education domain stress issues that should be important to the component movement overall. Educational components require attention to their cognitive characteristics, not just purely computational ones.<sup>4</sup> As critics of component engineering argue, component reuse has overemphasized technical attributes of components at the expense of considering the cognitive aspects of their use, such as the fit of the component to the secondary designers' mental model and thought processes. Also, educational components are explicitly oriented to learning and often secondarily oriented toward student collaboration; helping component users learn and collaborate while they work is likely to be an increasingly important engineering objective across all component domains. Finally, educational components are often oriented toward interaction design; learning happens because of how students are able to interact with screen elements.<sup>5</sup> In particular, educators place high value on constructivism, a philosophy of learning through engaged inquiry. This philosophy in turn stresses the ability of a component system to be open to manipulation by students and teachers as they learn. Thus, the argument for component software in education comes to rest on a great deal more than merely efficiency of building closed applications; continual support for end-user manipulation of components becomes part of the learning process.<sup>6</sup>

The authors of this article have been working on educational components for several years, in the context of a set of related projects on both the commercial and nonprofit sides, and both in the US and internationally. These projects are described in Table 1. Over the past year, the authors have come together in several forums to identify ways to work more closely together. As part of this process, we have collectively identified commonalities in our evolving approaches to educational components. In this article we report (a) our common vision for educational components and (b) the lessons learned across our past component software projects.

## **VISION**

It is an often-expressed dream that teachers, professors, and other nontechnical educators could create compelling educational software if only given the right authoring tool. Despite the availability of many software packages with excellent authoring capabilities, our experiences with both commercial products and research prototypes suggest that very few educational software users (estimated at less than one percent) will ever author a reusable lesson. This raises the dilemma of who the audience for components really is: are components targeted primarily at software developers themselves, or at shifting the production burden from developers closer to end users? Neither answer is sufficient: developers lack the contact hours with learners to directly create excellent educational software, and yet there are few signs that nondevelopers will ever become efficient producers of educational software. Thus, a shift in the focus of the question may be required: how can a component strategy bring software development and curriculum authoring expertise more closely together?

In answering this reformulated question, a difficult issue is the nature of the technological and social climate that will enable nontechnical educators and educational researchers to collaborate effectively. Below, we envision a future in which this climate has been realized.

In this future, ideas for specific educational activities will be turned into software as easily as writing a document about them: the process of development will be replaced by that of editing, of creating and manipulating "editable applications" that consist of high-level computational objects, available as tangible building blocks. Domain-specific but pedagogy-neutral components will decouple the educational value of end products from engineering. Standards (technical, educational, curricular, conceptual) will guarantee plug and play operation among components, independently of origin or type: incorporating new components in an end-user constructions will be as simple as

copy/pasting images from various sources in a document. Users will have many choices in the educational-component market, just as they do when choosing a home stereo system, where they can decide on the modules of their preference no matter who the manufacturer is. Intercomponent synergy, combined with users' creativity and imagination, will result in functional configurations that were not possible before or at all intended by their developers. This will provide the unique opportunity to combine different genres of learning tools, which, although valuable, were insufficient in themselves alone, and to implement cross-subject, holistic learning approaches (e.g., the pyramids of Egypt in a combined historical, geometrical, geographical, and cultural perspective).

Libraries of interoperable educational components of all sorts will be available for use via the Web, categorized and arranged for easy search and access: graphers, mapping tools, agents, simulators and simulations, statistical tools, database manipulation tools, calculators, spreadsheets, image and video tools, dynamic graphics manipulation tools, symbolic manipulation tools, word processors, tools for musical synthesis and sound manipulation, interactive communication tools, UI gadgets, interfaces to hardware component counterparts like lab instrumentation, clocks, globes, keyboards, sensors, etc. Components and component-based constructions will be seamlessly published, shared, and communicated among users through common e-mailing or Web access.

Independent developers, small research teams, or software companies will focus on their domains of expertise producing narrow-scope, high-quality educational components ready to interoperate with others' components in larger contexts, rather than wasting development efforts in replicating two-thirds of existing functionality just to add one-third innovation in their products. In addition, proper software architectures will enable a smooth "componentization" of existing applications, providing incentives to their developers for transitioning to a component model. Finally, content providers will contribute to the component repositories by providing their materials in the form of active components rather than passive multimedia assets.

Given the fledgling state of component technologies such as JavaBeans, it is no surprise that the current technological climate favors programmers. Nonetheless, we can easily foresee the technological state of the art rapidly advancing to meet our vision. However, for educational software components to be truly revolutionary, the social and economic climate will need to shift, as well. In particular, in the future we imagine, software developers and other designers will develop a shared communication language to discuss component requirements. Further, discussion among educators about the value and utility of the component offerings will present clear metrics for improvement of component quality. Finally, the economic experiment of the Educational Object Economy (<http://www.eoe.org>) will have succeeded: there will be a range of business models, from freeware to commercial software, each of which draws on and gives reusable components back to a blossoming marketplace of educational components and activities.

## **LESSONS LEARNED**

In this section, we describe seven lessons that have been learned as our collective set of projects have struggled to realize aspects of this vision. Each of these lessons applies to multiple projects, and the description of each lesson reflects the contribution of multiple authors. These lessons form the basis for our group's moving collectively forward toward a component marketplace for education.

## Enable legacy applications to serve as component authoring tools

Substantial challenges complicate the processes of (a) developing a large library of interoperable components, for example, a library sufficient to cover the K-12 mathematics curriculum, and (b) encouraging teachers and students to adopt component-based products in place of familiar monolithic applications. On the production side, component development and lifecycle maintenance require engineering resources proportionate to the number of intended components. During development, each new component demands unique requirements and usability testing. Until the component library reaches a critical mass, in which it presents wider opportunities for reuse and new deployment than afforded by an equivalent investment in monolithic implementation strategies, development efforts run in the red.<sup>7</sup> On the classroom side, teachers and students have an investment in traditional monolithic applications that may bias them against adopting new technologies. In this setting, introducing a new component to play an effective role in the overall component library can require its own program of user "evangelism" and education in order to gain sufficient acceptance by the user community. Similar challenges will arise in transitioning any developer community and user base to a component strategy.

In our projects, we have attempted to limit the impact of these obstacles by enabling existing applications to serve as component-generating tools. In particular, we have relied heavily on two general-purpose mathematical modeling tools--AgentSheets (AgentSheets, Inc.) and The Geometer's Sketchpad (Key Curriculum Press)--that have the capacity to export individual models as JavaBeans. These modeling tools can each produce a very wide range of instances of models, and such instances are often at the right level of granularity to be considered an individual library component. The Geometer's Sketchpad allows us to construct arbitrary geometric models that are dynamically malleable, subject to simultaneously propagated constraints that are specified by the model's author. AgentSheets allows us to create arbitrary interactive simulations, subject to sequentially executed imperatives, again specified by the simulation's author.

The Bridge Builder (Figure 1) is an example of an educational simulation component generated by AgentSheets. The Bridge Builder allows students to playfully explore general bridge design issues from different perspectives, including physics (forces) and architectural history (Greek versus Roman bridge design approaches). One of the tasks given to learners is to remove the maximum number of bricks from the bridge without collapsing the bridge under the load of cars driving over it. The Bridge Builder can be used in combination with other components to create a richer learning experience.

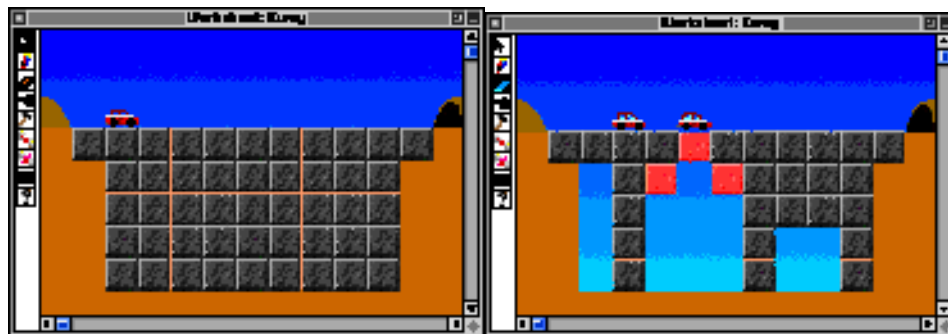


Figure 1. The Bridge Builder is a simulation component used to explore bridge design.

Using legacy modeling applications to specify and generate library components has several benefits over a strategy of developing each component from scratch. The user community has already established the pedagogic merits of these tools. Over the past eight years, The Geometer's Sketchpad has become one of the most widely used and respected software programs in middle school and secondary mathematics education. Though a more recent innovation, AgentSheets has likewise received broad critical acclaim and is rapidly expanding its user base. These pedigrees influence educators' initial interest and commitment to our component efforts. With enhancements to output Java classes, these modeling tools can become authoring tools with the ability to specify a new component and automatically generate the code. Furthermore, the exported JavaBeans can be structured to comply with intercomponent communication protocols even where the source applications did not: an AgentSheets simulation applet can communicate its data to a Java graphing component even though the original AgentSheets simulation did not have JavaBeans communication capabilities.

Because these authoring tools take care of the actual engineering aspects of component development, curriculum authors can focus their resources on designing and specifying educational content. More time can be spent developing mathematical lessons and less developing software programs. Indeed, by leveraging the installed base of AgentSheets and Sketchpad users, we dramatically expand the number of people who can create new components within our framework: teachers need not also be software developers if they wish to add a new component to the overall library. This coincidentally allows us to resolve granularity issues--how much functionality to address with a single component--by considering what sorts of configurations existing monolithic applications consider to be a coherent assemblage.

A final advantage to the authoring tool approach is that individual components generated by a single tool share a common user interface. This simplifies the task of exposing students to numerous new components simultaneously. Although the benefits of common user interfaces need to be weighed against an individual application's specific interface requirements, our approach allows us to decide these considerations on a case-by-case basis. We remain able to develop custom components from scratch if our authoring tools cannot adequately address specific needs. And in the more common occurrence that a component's interface stands to benefit from consistency with other components, it is reassuring to know that uniformity and consistency are being enforced automatically, at the programmatic level, by our authoring tools.

### **Develop cognitive supports to help activity designers manage the complexity of wiring**

In the JavaBeans component model, like many others, components interoperate by being "wired" together; wires provide the paths along which data and control can flow.<sup>8</sup> In many respects, synthesizing software by wiring components is analogous to the process of defining associations among entities while modeling in some notation (e.g., OMT, E-R, etc). For example:

- A Map component "hosts" Agent components (or Agents move on Map components).
- An Agent component "displays" its current view in a TV component.
- A Globe component "synchronizes" time and location with a Map component.
- A Database component "edits" feature attributes of a Map component.
- A Master-clock component "distributes" animation ticks to many other components.

In particular, our researchers have found traditional analysis techniques useful for classifying some of the kinds of associations that components can enter into. For example, intercomponent associations can be one-to-one, one-to-many, many-to-one, or many-to-many, depending on their semantics and the specific contexts. If an individual component implements many kinds of associations, it is important to clarify under which conditions those associations are valid. However, we have found that three additional complexities that arise in wiring components go beyond traditional analysis.

First, components are not always amenable to static analysis; indeed, the capabilities and connectivity options for a component may vary considerably, depending on the context and the component's state. From a static analysis, a designer might determine some component features that hold unconditionally over the component's lifetime. For example, a clock component may always be able to provide the current time in a simulation. A map component, however, may be able to provide time information for a particular locale, but only if a master clock has already been attached to it. Thus, a more dynamic analysis is needed to determine whether the map component can provide time information. Extending this example, the map component may provide additional data about a locale at a particular time, but only if an agent has been attached to the map so as to index a location. For example, if an agent component is placed on a map, the map may be able to provide information indexed to the time and location of the agent, like a traffic report.

To handle the complexity of dynamic wiring, we have used techniques that give components the ability to dynamically describe their current data manipulation and connectivity capabilities:

- ***JavaBeans Reflection***: JavaBeans provides a Reflection mechanism, which design tools can use to discover the wiring design patterns of a given component. The Reflection mechanism, however, does not cope well with components that dynamically change because Reflection uses a static analysis of the methods that a class implements to determine its capabilities.
- ***InfoBus***: In one project, we have explored the use of InfoBus to overcome this limitation; InfoBus allows a set of data producers to broadcast the dynamic availability of data by either name or type, and allows a set of data consumers to search for data by name or type. Further, InfoBus mediates the process of acquiring references to data channels and releasing them when no longer needed.
- ***Proprietary Mechanisms***: In another project, a proprietary mechanism called "E-Slate"<sup>10</sup> was designed that dynamically handles components' connection capabilities and mediates the process of establishing connections between components that have compatible capabilities. In both cases, standardized protocols reduce the cognitive difficulty to designers of making dynamic interoperability capabilities available.

A second wiring complexity concerns how a component's wiring possibilities can be made explicit to end users, such as teachers or curriculum designers. Conventional design tools (such as BeanBox) allow users to connect almost anything to anything else, and require that every connection be made manually by the user. We have found that in real educational applications, designers are easily overwhelmed by flexibility to specify arbitrary wiring and the need to create complex wiring diagrams for relatively simple tasks. In response, we have designed a variety of techniques to reduce the burden of wiring, such as:

- Making components' wiring capabilities explicit in a consistent and unambiguous way by using comprehensible symbolism or metaphors, such as puzzle pieces with colored pins.

- Automatically producing default wiring, such as connecting all time-driven components to a clock.
- Aggregating translation capabilities into components, so that less wiring is needed to mediate connections, for example, including plotters for a wide variety of data types into our graph component rather than requiring external plotting components.
- Guiding users through the wiring process according to templates for typical situations or according to underlying semantics of the components available in a layout, for example, guiding users to connect a mathematical model to various candidate views.
- Automatically ensuring the correctness of resulting wiring diagrams.

A third wiring complexity is mapping the meanings intended by users in establishing a wire down to the implementation by specific components. We discuss this issue further in the next section.

### **Focus interoperability on domain concepts and requirements**

The various projects in Table 1 center around K-12 math and science learning, though some include history and geography as well. For this educational subject matter, major breakthroughs in quality of learning are associated with the ability of computer displays to bring concepts to life through dynamic, visual presentations.<sup>9</sup> Thus, assembling components means assembling simulations, visualizations, animations, mathematical and geographical analysis tools, spreadsheets, and computer algebra systems. An important wiring issue involves linking multiple representations such that a change in one representation is immediately reflected in another representation; if a student drags the plot of a function, the algebraic and tabular representations of the function should change simultaneously with the graph.

Researchers in each of our projects have found it necessary to go beyond the available interoperability mechanisms to support dynamic interaction at the level desired. For example:

- File or stream-based communications are too slow for dynamic user interfaces.
- CORBA is too heavyweight for educational applications, which in most cases need very little by way of distributed computing.
- The full generality of COM/ActiveX interfaces or Java Reflection puts too much burden on the educational component developer.
- One-way notification semantics of event models such as JavaBeans are awkward for establishing linked multiple representations.
- Standard scripting interfaces to components are too closely tied to user interface or presentation actions, rather than meaningful actions on an underlying conceptual model.

Instead, each project has found it necessary to implement a form of model-view-controller architecture on top of existing standards. In this architecture, a conceptual model is clearly separated from its presentation. Object references are shared multiple views, and controllers can act directly on the object's public methods. In the ESCOT

project, for example, we implemented a model-view-controller architecture for mathematical models on top of the InfoBus architecture supplied by Sun.

One wiring architecture, however, is not enough. We have found that intercomponent associations span to a broad conceptual range, and the underlying semantics vary accordingly. In addition to the model-view-controller architecture, which is ideal for dynamically linked representations, other educational component connections have an "event" semantics, and yet others have "data flow" or "data sharing" semantics. The E-Slate project, for example, implemented a custom communication architecture based mostly on data flow semantics. We are seeking ways to smoothly combine different wiring semantics in an architecture without overburdening either the programmer or the author-designer with too much complexity.

### **Support end-user programming for just-in-time adaptation of components**

In addition to the intercomponent issues described in the preceding two sections, there are also intracomponent issues. In the quickly growing component-based software economy, prefabricated components often only partially fulfill the needs of designers and end users. Scripting languages or component builders allow casual programmers to connect components, but stop short of allowing casual programmers to adapt components to new requirements. Our experience shows that without the ability to adapt components to new needs, sometimes in elementary ways, functional components can be rendered useless.

End-user programmable component generator tools can support users in adapting existing components. Our experience suggests that four kinds of adaptability are made possible by end-user programming languages:

- Developing small components to fill gaps in the provided component library.
- Customizing or modifying the behavior of a component to fit an unanticipated circumstance.
- Connecting components where the need for a common interface was not anticipated.
- Adding logic that integrates behavior across several components, often specific to a single application.

One approach to enable adaptation is the framework of open sources in which a community of users collects and cultivates components in a shared repository. The Educational Object Economy (EOE, <http://www.eoe.org>) is currently the largest repository of educational Java applets. Many of the applets in the EOE enable adaptation by including their Java sources. However, most teachers and students do not have the required skills, time, tools, or interest to modify existing components at the Java source level.

Our solution to this problem is to use end-user programmable tools that elevate the process of programming to the level of manipulating problem domain concepts. The AgentSheets<sup>11</sup> project has developed a number of new end-user programming paradigms, including graphical rewrite rules, rule-based functions, and Tactile Programming,<sup>12</sup> that are geared toward, and have been successfully tested with casual computer users lacking a traditional programming background. A different approach has been taken by the E-Slate project, which has expanded the Logo program language from its "Turtle Graphics" roots to become a full-fledged component scripting language handling domain-specific primitives defined by each component. (Logo is a high-level programming language derived from Lisp, and has been popular among educators for almost 30 years.)

We have found that another important form of adaptation can be provided through specially designed tools that have the ability to break components into finer-grained subcomponents that can be comprehended, shared, and reassembled. In the case of the Bridge Builder simulation component discussed above, bricks, cars, and tunnels are AgentSheets subcomponents called agents. A Web-based forum called The Behavior Exchange (<http://www.agentsheets.com/behavior-exchange.html>) can be used to exchange such agents. Users can find other kinds of building materials, such as steel-based construction blocks for the bridge. Users can not only download agents from the Behavior Exchange and put these agents into their simulation but can also open up these agents to see how they are programmed. Simulations modified by adding new agents from the Behavior Exchange can be wrapped up again as JavaBeans. This mechanism provides users a lot of control over components without the need to create or modify programs in Java.

### **Determine component granularity iteratively**

Determining a component's boundaries so that it is a reusable, meaningful building block for the task at hand is a difficult job: our cognition is very flexible in defining "objects" on the spot according to the personal interests and focus of the moment. So how can engineers provide prefabricated objects that will be aligned with the spontaneous needs of end users? The key issue is how to maintain a right balance between two extremes. On the one hand, components can become too general purpose and low level (e.g., the simple user interface elements usually found in authoring tools). On the other hand, components can become too feature rich and inflexible. To resolve this problem, component designers need to identify an appropriate trade-off for the curriculum designers.

Our experience shows that the traditional linear development cycle (user requirements, specifications, design, implementation) is not sufficient for determining the appropriate trade-off. Needs most often cannot be shaped until the educational designers have the opportunity to "play" with a rough prototype and conceive the new potential capabilities. In particular, software and curriculum often co-evolve; as the technology changes, teachers' ideas of what and how students should learn change as well. An iterative process of rapid prototyping in close synergy between component and curriculum developers works best.

### **Use translators and wrappers to adapt existing resources**

In any realistic domain of applications, programmers will not have the resources to create from scratch all the components necessary to match a particular component framework. To the contrary, programmers will often have legacy models with inadequate communication modules or different communication protocols. Yet, our experiences show that there is often a great deal of value of in combining such large-scale modules.

To obtain interoperability with these modules, we developed a translator approach. A translator is a small component that knows how to change the representation suitable for one component into that suitable for another. Translators allow developers of component systems to make their own representational decisions. Once these decisions have been made, developers can get together to identify the shared data types and specify translators to implement them. In a design experiment conducted by one of our authors (Suthers), translator components were critical to the relative ease with which three systems were composed.

A similar concept to translators is wrappers; whereas a translator adapts an existing communications protocol to a desired standard, a wrapper adapts an existing containment protocol to allow the component to be embedded in a

desired containment hierarchy. A commercial example of wrappers is the ability to use a JavaBean as an ActiveX control and vice versa. We have found that wrappers dramatically reduce the costs and risks for developers of porting existing components to a new standard. Since profit margins are very thin in educational software, we have found that developers are much more likely to support a new component standard if they can become compatible via a wrapper strategy and thus avoid the cost and risk of a rewrite.

### **Help users track objects across components**

Efforts toward achieving plug and play component-based software tend to address primarily the need for a common communication protocol or API. Where data semantics are addressed, the focus is usually on solving the problem of sharing information between the component applications. However, we have found that for a component system to remain comprehensible to teachers and students, it is crucial that key domain objects retain their identity for the users as they move between representations and/or components. A user should be able to tell at a glance which components are presenting information about the same domain object. Moreover, a user often needs to understand how cause-effect relationships propagate among components, that is, how a change made to a domain object in one component affects a view in another component. Persistence of identity, in turn, requires coordination in how components present domain objects to their users. A shared standard for the visual presentation of objects (e.g., name, color, description, icon) or a means for objects to carry presentational information along with them may be necessary. Further research is needed to identify strategies for making the identities and relationships of objects obvious to end users across a wide variety of components that present different forms of displays.

## **CONCLUSION**

Our emphasis in this article has been on synthesizing a small number of high-level findings across a number of research and commercial projects in educational software. In general, we find that the component software model has high appeal to educational technologists and their clients, especially where component models ride on the coattails of widely deployed Internet standards. The principal challenge is moving the conception of component software from a developer-centric view toward a domain expert (i.e., curriculum author) view. Specific aspects of this challenge include enabling the domain-experts' favored tools to become component generators, reducing the cognitive complexity of wiring, focusing interoperability on domain-specific needs, supporting end-user programming, using translators to enable communication, and determining granularity through an iterative, participatory design process. In addition, unresolved obstacles exist in the market, at the user interface, and in supporting an appropriate authoring culture.

We speculate that these issues are not unique to education, but will resonate in other domains of component use where requirements are hard to specify or evolving rapidly and nontechnical designers have a major role in implementing whatever is considered a "final product." Despite the overall attractiveness of a componentware vision, these domains will be risky markets for initial component implementations. An overall goal for future component software research should be to reduce these risks, in particular (a) reducing the time to scaling a component marketplace to critical mass and (b) reducing the time interval at which individual products can realize benefits from adopting a component approach. Some risks may be mitigated technically, through design frameworks that reduce the cost of developing, understanding, and adapting components. But others will require close attention

to economic and social structures that would foster reuse in a heterogeneous community of small-scale developers and designers.

## Acknowledgments

Thanks to Ken Koedinger for reading early drafts and providing suggestions.

E-Slate has been funded by the following projects: (a) Project "IMEL," EC DGXXII, Socrates 25136-CP-1-96-1-GR-ODL (b) Project "YDEES" (EU Support Framework II, Greek Ministry of Industry Energy and Technology, General Secretariat for R&D, Measure 1.3, Project 726), and (c) Project "ODYSSEUS" (EU Support Framework II, Greek Ministry of National Education and Religious Affairs). AgentSheets has been funded by NSF DMI-9761360, RED925-3425, and DARPA CDA-940860.

The DARPA CAETI program also funded Belvedere. ESCOT is funded by National Science Foundation grant REC-9804930. JavaSketchpad has been funded in part by National Science Foundation awards DMI-9561674 and 9623018. Opinions presented here are those of the authors, and may not reflect the views of the funding agencies.

## References

1. J. Roschelle and J. Kaput, "Educational Software Architecture and Systemic Impact: The Promise of Component Software," *Journal of Educational Computing Research*, 14(3), 1996, pp. 217-228.
2. J. Roschelle, B. Henderson, J. Spohrer, and J. Lilly, "Banking on Educational Software: A Wired Economy Unfolds," *Technos*, 6(4), 1997, pp. 25-28.
3. B. Henderson, *The Components of Online Education. Higher Education on the Internet*, University of Saskatchewan, Saskatoon, Canada, 1998.
4. J. Kaput, "Technology and Mathematics Education," *Handbook of Research on Mathematics Teaching and Learning*, D. Grouws (Ed.), Macmillan, New York, 1992, pp. 515-556.
5. J. Roschelle, "Designing for Cognitive Communication: Epistemic Fidelity or Mediating Collaborating Inquiry," *Computers, Communication & Mental Models*, D.L. Day and D.K. Kovacs (Eds.), Taylor & Francis, London, 1996, pp. 13-25.
6. C. Rader, G. Cherry, C. Brand, A. Repenning, et al., "Principles to Scaffold Mixed Textual and Iconic End-User Programming Languages," *Proceedings of the 1998 IEEE Symposium of Visual Languages*, Nova Scotia, Canada, Computer Society, 1998.
7. R.L. Leach, *Software Reuse: Methods, Models, and Costs*, McGraw-Hill, New York, 1997.
8. E.R. Harold, *JavaBeans*, IDG Books, Foster City, Calif., 1998.
9. D.N. Gordin, and R.D. Pea, "Prospects for Scientific Visualization as an Educational Technology," *Journal of the Learning Sciences*, 4(3), 1995, pp. 249-280.
10. M. Koutlis, P. Kourouniotis, K. Kyrimis, N. Renieri, "Inter-component Communication as a Vehicle towards End-User Modeling", *ICSE'98 Workshop on Component-Based Software Engineering*, Kyoto, Japan, 1998.
11. A. Repenning and T. Sumner, "AgentSheets: A Medium for Creating Domain-Oriented Visual Languages," *IEEE Computer*, 28, 1995, pp. 17-25.

12. A. Repenning and A. Ioannidou, "Behavior Processors: Layers between End-Users and Java Virtual Machines," *Proceedings of the 1997 IEEE Symposium on Visual Languages*, Capri, Italy, Computer Society, 1997, pp. 402-409.

## **Tutorial 1: Determining component granularity and connectivity**

*By Manolis Koutlis & Jeremy Roschelle*

How can the right-sized building blocks for a given domain be identified?

We have found a two-stage process to be the best practice. In the first stage, we try to start by collaboratively designing a prototype piece of software with curriculum authors. Because curriculum authors are not component engineers, in the design we stay focused on overall functionality. But in implementing the prototype, we attempt to generalize to a family of similar (in functionality, characteristics, requirements, nature) pieces of software that might be asked for. Criteria for decomposing that family into components include:

- Stay close to the curriculum designers' cognitive and subject matter domains (i.e., objects that model familiar entities, concepts, phenomena, relationships, behaviors, etc.).
- Favor powerful, large-granularity components that can fulfill a major role in the curriculum designers' concept of the application.
- Connectivity among components should give curriculum designers some key advantages, but not shift much programming burden to them.

The goal of this stage is to help the curriculum designers reconceptualize their problem domain in terms of components. Based on a decomposition of the prototype application, we design components and synthesize an integrated whole close to the prototype originally specified. We then demonstrate to curriculum authors how the component kit can realize a family of applications. If the right conceptual level and component granularity have been successfully achieved, the authors will begin to think in terms of the components at hand, building configurations that have not been predicted.

This first stage, however, is unlikely to yield a collection of components with exactly the right levels of granularity and connectivity. Thus, in a second stage we proceed with a user-centered redesign, based on observational and participatory studies of the curriculum designers using the component kit to develop student learning activities. In particular, as the curriculum designers play with these components in a variety of prototyping situations, we observe the kinds of flexibility the curriculum designers actually use and desire, the places where routine and repetitive component configurations could be automated, and the previously unidentified and unmet needs. We allow the user-centered analysis to drive realignment of the component granularity and connectivity to meet their needs. This will often involve breaking some components into smaller parts, adding some new components, introducing wizards and other tools to automate routine configuration tasks, and redefining connectivity options to focus on the actual connectivity demands. Thus, the goal of the second stage is to tune the component collection for maximum flexibility and minimum complexity in actual design use.