Developing Educational Software Components

Having component developers collaborate with domain experts to build applications may be the future of software development. A group of component developers discuss what they've learned in collaborating with educators on 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 Hawaii at Manoa

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 curricula and teaching styles vary tremendously among institutions, locations, and even among instructors at the same institution.

To meet these needs, a wide array of small-scale, casual developers at universities, research labs, and small businesses develop educational software, and no dominant solution or supplier has emerged.[1] In this market, smaller suppliers often cannot produce full solutions without depending on the capabilities of other vendors. For example, producing software for a grade 6-8 mathematics curriculum might require more than 30 different tools, such as random number spinners, dice, tables, graphs, spreadsheets, and notebooks. As a consequence, small educational software publishers must either limit their product to a particular topic in the curriculum, or they must team with others who have already built some of the components. Because narrowly limited products are not very desirable in education, small developers are recognizing that they need a standard, common platform.

These conditions are driving the increased used of components in educational applications. An evolving educational-component market combines the efforts of fixed-component developers and educators to meet the tremendous demand for customized learning materials.[2] Indeed, for the educational software market to survive, component methodologies appear to be a necessity, not a luxury.[3]

Until now, components have remained largely the province of full-time programmers. However, component technologies are likely to expand toward an audience that is considerably less technical and more domain-oriented—users whose job descriptions typically don't include software development. Thus, the lessons we've learned about how to collaborate with nontechnical domain experts on educational software could become increasingly important to developing good software for any application domain.

SPECIAL NEEDS OF EDUCATIONAL COMPONENTS

In contrast to more conventional uses of components, educational components require attention to their cognitive characteristics rather than just the purely computational ones.[4] Critics of component engineering argue that component reuse has overemphasized the technical attributes of components at the expense of the cognitive aspects of their use. Cognitive aspects include the component's fit with the mental model and thought processes of the secondary designer—the person who uses the components to build an application.

Today's educational components are explicitly oriented toward learning and often only secondarily oriented toward student collaboration. In the future, helping application users learn and collaborate while they work will likely be an increasingly important engineering objective across all component domains.

Finally, educational components are often designed for interaction: Learning occurs because of how students interact with screen elements.[5] In particular, educators place high value on *constructivism*, a philosophy of learning through engaged inquiry. This philosophy in turn favors a component system that students and educators can manipulate as they learn; such manipulation is part of the learning system and demands continual support.[6] Components could be helpful in providing that support. Thus, the argument

0018-9162/99/$10.00 © 1999 IEEE

Table 1. Educational-software projects that are the basis for this article.

Project or product	Developer and organization	Description
E-Slate http://e-slate.cti.gr	Manolis Koutlis Computer Technology Institute, Greece	E-Slate is a component-based environment for constructing Internet-aware learning activities for exploration. In addition, it provides a platform for the development of educational component libraries.
ESCOT http://www.escot.org	Jeremy Roschelle, Chris DiGiano, Roy Pea, and Jim Kaput SRI International	ESCOT is a test bed that seeks to enable interoperability and reuse of educational software components for middle-school mathematics.
AgentSheets http://www.agentsheets.com	Alexander Repenning University of Colorado, Boulder	AgentSheets is an agent-based authoring environment for educators to build interactive simulations that can be turned into Java applets and JavaBeans.
Java Sketchpad http://www.keypress.com/sketchpad/java_gsp	Nicholas Jackiw KCP Technologies	JavaSketchpad is a Web-enabled incarnation of The Geometer's Sketchpad, a market-leading mathematics exploration tool for grades six through 14.
Belvedere http://lilt.ics.hawaii.edu `	Dan Suthers University of Hawaii at Manoa	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.

for writing educational software with components rests on a great deal more than mere efficiency in building the initial applications.

We have been working on educational components for several years in related commercial and nonprofit projects, both within the US and internationally. Table 1 describes these projects. Over the past year, we have been identifying ways to work more closely together. As part of this process, we have collectively identified commonalities in our evolving approaches to educational components.

COMMON DILEMMA

Despite the availability of many software packages with excellent authoring capabilities, our experiences with both commercial products and research prototypes suggest that very few educators (estimated at less than one percent) will ever write a reusable lesson.

This fact raises the dilemma of who the audience for educational software components really is: Are components targeted primarily at traditional software developers or at shifting the production burden from developers to educators? Neither answer is sufficient: Developers lack the contact hours with students to directly create excellent educational software, and there are few signs that educators will ever become efficient producers of educational software. Thus, it may be better to ask, "How can a component strategy bring software development and curriculum-authoring expertise more closely together?"

In answering this question, we first envisioned a technological and social climate that would enable technical researchers and nontechnical educators to collaborate effectively.

OUR VISION

In the future, ideas for specific educational activities will be turned into software as easily as writing a document: The development process will be replaced by a process of editing, creating, and manipulating "editable applications." Such applications will consist of high-level computational objects that are available as tangible building blocks. Domain-specific but pedagogy-neutral components will decouple the educational value of end products from their engineering.

Standards—technical, educational, curricular, and conceptual—will guarantee plug-and-play operation among components, independent of origin or type. Incorporating new components in specific applications will be as simple as copying and pasting images from various sources into a document. Educators will have many choices in the educational-component market, just as they do when choosing a home stereo system, where they can choose the modules they prefer from various manufacturers.

Intercomponent synergy combined with an educator's creativity and imagination will result in functional configurations that were not possible before or weren't anticipated by the components' developers. This will provide the unique opportunity to combine different genres of learning tools that are more valuable in combination than individually. Such a combination of tools will more effectively implement cross-subject, holistic learning approaches. For example, they could present a subject like the pyramids of

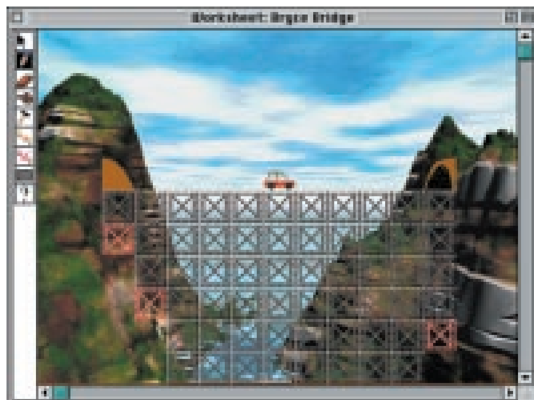

Figure 1. Bridge Builder is a simulation component for exploring bridge design.

Egypt from a combined historical, geometrical, geographical, and cultural perspective.

Digital libraries of interoperable educational components that are categorized and arranged for easy search and access will be available via the Web. Such libraries could include components for a multitude of functions, such as graphing, mapping, agents, simulators and simulations, statistics, and database manipulation. New components and component-based constructions will be seamlessly published and shared among educators through common e-mail or Web access.

Independent developers, small research teams, or software companies will focus on their domains of expertise, producing high-quality educational components of narrow scope. Component developers will write their components to interoperate easily with those of other vendors, rather than wasting efforts replicating two-thirds of existing functionality just to add one-third innovative functionality to their products. Proper software architectures will enable a smooth componentization of existing applications, providing their developers an incentive for transitioning to a component model. Finally, content providers will contribute to component repositories by providing their materials in the form of active component tools (such as spreadsheets, graphs, and simulations) rather than as canned multimedia elements (such as movies or graphics).

To enable the future we envision, software developers and educators who have become application designers must develop a shared communication language to discuss component requirements. Discussions among educators about the value and utility of the component offerings will present clear metrics for improving component quality. Finally, the economic experiment of the Educational Object Economy (http://www.eoe.org) will have succeeded: A range of business models—from freeware to commercial software—will draw on and return reusable components to a blossoming marketplace of educational components and activities.

WORKING TOWARD THE VISION: LESSONS LEARNED

Our projects have struggled to realize aspects of this vision, and along the way we've learned seven significant lessons. Each lesson applies to multiple projects, and the description of each lesson reflects the contributions of several of us. These lessons form the basis for our group's collective move toward a component marketplace for education.

Use legacy applications as component authoring tools

Substantial challenges complicate the processes of

- developing a large library of interoperable components (for example, one sufficient to cover the K-12 mathematics curriculum), and
- encouraging educators and students to adopt component-based products in place of familiar monolithic applications.

Component development and life cycle maintenance require engineering resources proportionate to the number of intended components. During development, each new component demands testing for requirements and usability. Until the component library reaches a critical mass—that is, presents wider opportunities for reuse and new deployment than an equivalent monolithic implementation—development efforts will run in the red.[7]

In the classroom, educators and students have an investment in traditional monolithic applications that can bias them against adopting new technologies. In this setting, introducing a new component to the overall component library can require a program of "evangelism" and education to gain sufficient acceptance. Similar challenges arise in transitioning any developer community and user base to a component strategy.

In our projects, we attempted to limit the impact of these obstacles by letting existing applications serve as component-generating tools. In particular, we relied heavily on two general-purpose mathematical modeling tools: The Geometer's Sketchpad (Key Curriculum Press) and AgentSheets (AgentSheets Inc. and University of Colorado, Boulder), which can export individual models as JavaBeans. These modeling tools can each produce a wide range of model instances, and such instances are often at the right level of granularity to serve as an individual library component. The Geometer's Sketchpad can construct arbitrary geometric models that are dynamically malleable, subject to author-specified geometric constraints. AgentSheets helps create arbitrary interactive simulations, subject

to sequentially executed imperatives, which the simulation's author specifies.

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

Using legacy modeling applications to specify and generate library components has several benefits over developing each component from scratch. For one, educators have 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 the number of educators using it is expanding rapidly.

This previous experience influenced educators' initial interest and commitment to our component efforts. With enhancements that output Java classes, these modeling tools can become authoring tools that can specify a new component and automatically generate the code. Furthermore, such tools can structure the exported JavaBeans to comply with intercomponent communication protocols even where the source application does not: An AgentSheets simulation applet can communicate data to a Java graphing component even though the original AgentSheets simulation did not incorporate JavaBeans communication.

Because these authoring tools handle the engineering aspects of component development, educators can focus their resources on designing and specifying educational content. They can spend more time developing mathematical lessons and less time developing software programs. Indeed, leveraging the installed base of AgentSheets and Sketchpad users dramatically expands the number of people who can create new components within our framework. Educators don't also need to be programmers to add a new component to the overall library. This coincidentally resolves granularity issues—how much functionality to address with a single component—by considering what sorts of configurations current monolithic applications consider 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, which simplifies the task of exposing students to numerous new components simultaneously. In this way, authoring tools enforce uniformity and consistency automatically at the program level.

As developers of the authoring tools, we must weigh the benefits of common user interfaces against an individual application's specific interface requirements. Our approach lets us consider this trade-off on a case-by-case basis. It also permits development of custom components from scratch if authoring tools cannot adequately address specific needs.

Educators can spend more time developing mathematical lessons and less time developing software programs.

Educators need cognitive support to manage wiring complexity

In JavaBeans (like many other component models), components interoperate by being "wired" together; wires specify the paths along which data and control can flow.[8] In many respects, synthesizing software by wiring components is analogous to defining associations among entities by modeling with notations such as object model transformations (OMT) or entity-relation diagrams. In these examples, the way these components are used models the following relations (in italics):

- a map component *hosts* agent components (or agents move on map components);
- an agent component *displays* its current view in a television component;
- a globe component *synchronizes* time and location with a map component;
- a database component *edits* feature attributes of a map component; and
- a master clock component *distributes* animation ticks to other components.

In particular, our researchers have found traditional analysis techniques useful for classifying associations in which components participate. For example, intercomponent associations can be one-to-one, one-to-many, many-to-one, or many-to-many—standard relationships in classic relational database theory—depending on their semantics and specific contexts. If an individual component implements many types of associations, it is important to clarify the conditions under which those associations are valid.

However, we have found that two additional complexities—which go beyond traditional analysis—arise in wiring components.

Providing for dynamic 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 component developer might determine some component features that hold unconditionally over the component's lifetime. For example, a clock compo-

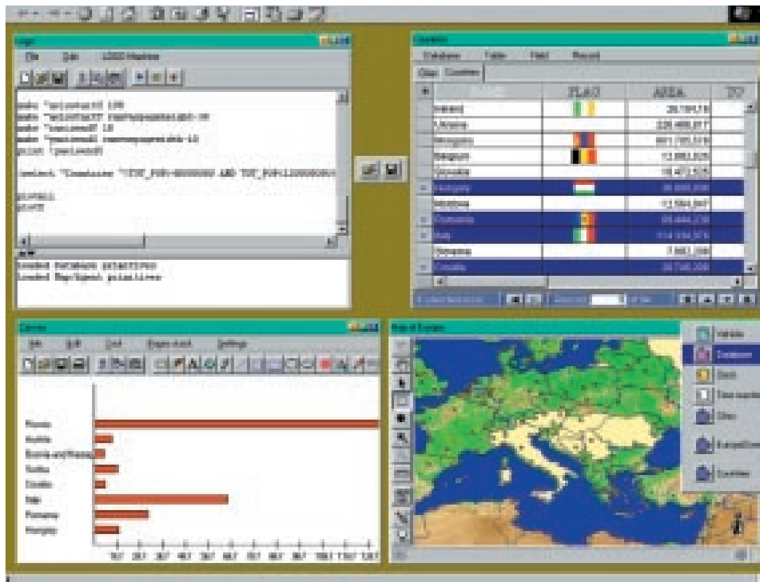

Figure 2. E-Slate is a proprietary mechanism for assessing a component's connection capabilities.

nent 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 the component is already attached to a master clock. Thus, we need a more dynamic analysis to determine whether the map component can provide time information.

Extending this example, the map component could provide additional data about a locale at a particular time, but only if it is attached to an agent that indicates a location. For example, if an agent component is placed on a map, the map could provide information, like a traffic report, which is indexed to the agent's time and location.

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

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 component implements to determine its capabilities.

In one project, we used InfoBus to obtain a more dynamic analysis. 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.

E-Slate, shown in Figure 2, is a proprietary mechanism that dynamically assesses a component's connection capabilities and presents a user interface to allow educators and students to make connections among components that have compatible capabilities.[9] In both E-Slate and InfoBus, standardized protocols reduce an educator's cognitive difficulty in using dynamic interoperation capabilities.

Making wiring options clear. A second wiring complexity concerns how tool developers make a component's wiring possibilities explicit to application designers. Conventional design tools (such as BeanBox) connect almost any component to any other, but they require the educator to make every connection manually. In real educational applications, educators are easily overwhelmed by this flexibility and the accompanying need to create complex wiring diagrams for even relatively simple tasks. In response, we have developed techniques to reduce the burden associated with wiring; these include

- making component wiring capabilities explicit in a consistent and unambiguous way by using comprehensible symbols 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 they need less wiring for connections (for example, incorporating plotters for several data types into a graph component);
- guiding application designers through the wiring process with templates for typical situations or according to the components' underlying semantics (for example, guiding designers to connect a mathematical model to various candidate views); and
- automatically ensuring the correctness of wiring diagrams.

Focus interoperability on domain concepts and requirements

The various projects in Table 1 focus on K-12 math and science learning, though some include history and geography as well. For this subject matter, major breakthroughs in learning quality derive from the computer's ability to bring concepts to life through dynamic, visual presentations.[10] 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 so that a change in one representation is immediately reflected in another representation. For example, if a student can alter the plot of a function by dragging the cursor across it, the function's algebraic and tabular representations should change simultaneously.

Researchers in each of our projects have found it

necessary to go beyond the available interoperability mechanisms to support dynamic interaction at the desired level because

- file- or stream-based communications are too slow for dynamic interfaces;
- the Common Object Request Broker Architecture is too heavyweight for educational applications, which in most cases aren't distributed;
- the flexibility of Component Object Model/ActiveX interfaces or Java Reflection puts too much burden on application designers;
- one-way notification semantics of event models such as JavaBeans are awkward for linking multiple representations; and
- standard scripting interfaces are too closely tied to user interface or presentation actions, rather than meaningful actions based on an underlying conceptual model.

Instead, each project has implemented a form of model-view-controller architecture on top of existing standards. This architecture clearly separates a conceptual model from its presentation. Object references are shared by multiple views, and controllers can act directly on the object's public methods. In the Educational Software Components of Tomorrow (ESCOT) project, for example, we implemented a model-view-controller architecture for mathematical models on top of the InfoBus architecture.

However, one wiring architecture is not enough. Intercomponent associations span 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, some educational-component connections have event-based semantics, and others have semantics based on data flow or data sharing. The E-Slate project, for example, implemented a custom communication architecture based mostly on data flow semantics. We thus seek ways to smoothly combine different wiring semantics without overburdening either the component developer or educator with too much complexity.

Support user programming for just-in-time adaptation of components

In addition to the intercomponent issues, there are also intracomponent issues. In the quickly growing component-based software market, prefabricated components often only partially fulfill the needs of educators and students. Scripting languages or component builders allow casual programmers to connect components but stop short of letting them adapt components to new requirements. The inability of components to be adapted to new needs—sometimes in elementary ways—can render them useless.

User-programmable component-generator tools help application designers to adapt existing components. These tools incorporate programming languages that let designers

- develop small components to fill gaps in the standard component library,
- customize or modify a component's behavior to fit an unanticipated circumstance,
- connect components in situations where the need for a common interface was unanticipated, and
- add logic that integrates behavior across several components that are often specific to a single application.

Prefabricated components often only partially fulfill the needs of educators and students.

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

Our solution is to use user-programmable tools that elevate programming to the level of manipulating problem domain concepts. The AgentSheets[11] project has developed several new user-programming paradigms, such as graphical rewrite rules, which have been successfully tested with casual programmers who lack a traditional programming background.

E-Slate takes a different approach. It is based on Logo, a high-level programming language derived from Lisp, which has been popular among educators for almost 30 years. E-Slate expands Logo into a full-fledged component scripting language that can call the public scripting interface defined by each component. Specially designed tools provide another important form of adaptation. These tools break components into finer-grained subcomponents that educators can comprehend, share, and reassemble.

In the Bridge Builder simulation component, bricks, cars, and tunnels are subcomponents called agents. Educators can use a Web-based forum called The Behavior Exchange (http://www.agentsheets.com/behavior-exchange.html) to exchange such agents. The Behavior Exchange provides other kinds of building materials, such as steel-based construction blocks, for the bridge. Educators can download these agents and put them in their simulation; they can also see how the agents are programmed. The modified simulations can be rewrapped as JavaBeans. This mechanism provides educators control over components without the need to create or modify Java programs.

Iterate to determine component granularity

Defining a component's boundaries so that it is a reusable, meaningful building block is a difficult job: Human cognition is very flexible in defining "objects" on the spot according to personal interests and the moment's focus. So how can component developers provide prefabricated objects that will be aligned with the spontaneous needs of application designers? The key issue is how to maintain a balance between two extremes. On the one hand, components can become too general and low-level, like 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 developers need to identify an appropriate trade-off for the application designer.

Our experience shows that the traditional linear development cycle (requirements, specifications, design, and implementation) is not sufficient for determining the appropriate trade-off. Needs are often not apparent until educators play with a rough prototype and comprehend the potential capabilities. Software and curriculum, in particular, often co-evolve; as the technology changes, educators' ideas of what and how students should learn change as well. An iterative process of rapid prototyping in close synergy between component developers and application designers works best. The "Determining Component Granularity and Connectivity" sidebar describes our two-stage process for identifying correct component size.

Use translators and wrappers to adapt existing resources

In any realistic application domain, component developers lack the resources to create—from scratch—all the components necessary to match a particular component framework. Instead, developers will often start with legacy models that have inadequate communication modules or different communication protocols. However, there is often value in combining such large-scale modules with each other and other, smaller components.

To help build interoperable modules, Steve Ritter and Ken Koedinger[12] developed a translator, a small component that component systems developers use to make their own representational decisions. Once developers make these decisions, they can identify the shared data types and specify translators to implement them.

Wrappers, another way to provide interoperability, are similar in concept to translators. Whereas a translator adapts an existing communications protocol to a desired standard, a wrapper provides a communications layer around a component so that it can communicate effectively with other components.

Determining Component Granularity and Connectivity

Manolis Koutlis and Jeremy Roschelle

How can component developers identify the correct size of component building blocks for a given domain? We have found a two-stage process to be the best practice.

In the first stage, we start by collaboratively designing a prototype piece of software with educators. Because educators are not component engineers, we focus on overall functionality in this design. But in implementing the prototype, we attempt to generalize it to a family of software that is similar in functionality, characteristics, and requirements. Criteria for decomposing that family into components include

- Stay close to the educator's cognitive processes and subject matter domains (that is, objects should model familiar entities, concepts, phenomena, relationships, and behaviors of the domain).
- Favor powerful, large-granularity components that can fulfill a major role in the educator's concept of the application.
- Component connectivity should give educators some key advantages, but not shift much programming burden to them.

The goal of this stage is to help the educators reconceptualize their problem domain in terms of components. Based on a decomposition of the prototype application, we design components and synthesize an integrated whole that is close to the prototype as originally specified. We then demonstrate to educators how the component kit can realize a family of applications. If we've achieved the right conceptual level and component granularity, educators will begin to think in terms of the components at hand, building previously unanticipated configurations.

This first stage, however, is unlikely to yield a component collection with exactly the right levels of granularity and connectivity. Thus, in a second stage we proceed with a student-centered redesign, based on the educators' observational and participatory studies, which use the component kit to develop student learning activities.

In particular, as the educators apply these components in a variety of prototyping situations, we observe the kinds of flexibility they desire and will actually use, the places where routine and repetitive component configurations could be automated, and previously unidentified and unmet needs. We allow this user-centered analysis to drive realignment of the component granularity and connectivity to meet user needs. This often involves 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 complexity in actual design use.

A commercial example of wrappers is the ability to use a JavaBean as an ActiveX control and vice versa. Wrappers can dramatically reduce an application designer's costs and risks in porting existing components to a new standard. Because profit margins are very thin in educational software, designers are more likely to support a new component standard if they can use a wrapper to make applications compatible, which avoids the cost and risks of a rewrite.

Help users track objects across components

Efforts toward achieving plug-and-play component-based software tend to address the need for a common communication protocol or an application programming interface. Where such efforts address data semantics, the focus is usually on solving the problem of sharing information between component applications. However, we have found that for a component system to remain comprehensible to educators and students, key domain objects must retain their identity as they move between representations and/or components.

Educators and students should be able to tell at a glance which components present information about the same domain object. Moreover, both groups often need 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 educators and students.

A shared standard for the visual presentation of objects (relative to name, color, description, and icon) or a means for objects to carry presentational information could be necessary. Further research is needed to identify strategies for making the identities and relationships of objects across a wide variety of components obvious to application designers.

For component-based techniques to influence these new software areas, the principal challenge is moving the conception of component software from a developer-centric viewpoint toward a domain-expert-centric viewpoint. This involves not only technical considerations, but also addressing unresolved obstacles 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 in which requirements are hard to specify or are evolving rapidly, and those in which nontechnical designers have a major role in implementing the 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, research should look for ways to reduce the time it takes a component marketplace to reach critical mass and the time it takes individual products to benefit from adopting a component approach.

Some risks may be mitigated technically through design frameworks that reduce the cost of developing, understanding, and adapting components. Managing other risks will require close attention to economic and social structures that foster reuse in a heterogeneous community of small-scale product designers. ❖

Using a wrapper to make applicaions compatible avoids the costs and risks of a rewrite.

Acknowledgments

We thank Ken Koedinger for reading early drafts and providing suggestions.

E-Slate has been funded by projects IMEL, EC DGXXII, Socrates 25136-CP-1-96-1-GR-ODL; YDEES (EU Support Framework II, Greek Ministry of Industry Energy and Technology, General Secretariat for R&D, Measure 1.3, Project 726); and Odysseus (EU Support Framework II, Greek Ministry of National Education and Religious Affairs).

AgentSheets was funded by the National Science Foundation DMI-9761360, RED9253425, REC 9804930, and DARPA CDA 940860. The DARPA CAETI program also funded Belvedere. ESCOT is funded by NSF grant REC-9804930. JavaSketchpad has been funded in part by NSF DMI-9561674 and 9623018.

This article presents the authors' opinions and may not reflect the funding agencies' views.

References

1. J. Roschelle and J. Kaput, "Educational Software Architecture and Systemic Impact: The Promise of Component Software," *J. Educational Computing Research*, Vol. 14, No. 3, 1996, pp. 217-228.
2. J. Roschelle et al., "Banking on Educational Software: A Wired Economy Unfolds," *Technos*, Vol. 6, No. 4, 1997, pp. 25-28.
3. B. Henderson, *The Components of Online Education: Higher Education on the Internet*, Univ. of Saskatchewan, Saskatoon, Canada, 1998.
4. J. Kaput, "Technology and Mathematics Education," *Handbook of Research on Mathematics Teaching and Learning*, D. Grouws, ed., Macmillan Co., New York, 1992, pp. 515-556.
5. J. Roschelle, "Designing for Cognitive Communication: Epistemic Fidelity or Mediating Collaborative Inquiry?," *Computers, Communication and Mental Models*, D.L. Day and D.K. Kovacs, eds., Taylor & Francis, London, 1996, pp. 13-25.
6. C. Rader et al., "Designing Mixed Textual and Iconic Programming Languages for Novice Users," *Proc. 1998 IEEE Symp. Visual Languages*, IEEE CS Press, Los

Alamitos, Calif., 1998, pp. 187-194.
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. M. Koutlis et al., "Inter-Component Communication as a Vehicle Towards End-User Modeling," ICSE 98 Workshop on Component-Based Software Engineering, 1998, http://www.sei.cmu.edu/activities/cbs/icse98/papers/p7.html.
10. D.N. Gordin and R.D. Pea, "Prospects for Scientific Visualization as an Educational Technology," *J. Learning Sciences*, Vol. 4, No. 3, 1995, pp. 249-280.
11. A. Repenning and T. Sumner, "AgentSheets: A Medium for Creating Domain-Oriented Visual Languages," *Computer*, Mar. 1995, pp. 17-25.
12. S. Ritter and K.R. Koedinger, "An Architecture for Plug-in Tutoring Agents," *J. Artificial Intelligence in Education,* Vol. 7, No. 3-4, 1997, pp. 315-347.

Jeremy Roschelle *is a senior cognitive scientist at SRI International's Center for Technology in Learning. His research interests include educational software design, math and science education, collaborative learning, and video analysis methodology. He received a PhD in cognitive science and educational technology from the University of California, Berkeley. Contact him at Jeremy Roschelle@sri.com.*

Chris DiGiano *is a research computer scientist at SRI International's Center for Technology in Learning. His research interests include human-computer interaction, informal learning environments, virtual learning communities, and end-user modifiable software. He received a PhD in computer science from the University of Colorado, Boulder. Contact him at chris.digiano@sri.com.*

Manolis Koutlis *is a senior engineer-researcher at the Computer Technology Institute, Patras, Greece. His research interests are educational technology, component-oriented software engineering, and visual programming systems. He received a BS from the Computer Engineering and Informatics Department of the University of Patras and is currently a PhD student. Contact him at koutlis@cti.gr.*

Alexander Repenning *is a computer science professor at the Center of LifeLong Learning & Design at the University of Colorado, Boulder; he is also the director and a founder of AgentSheets Inc. His research interests include end-user programming, computers in education, agent-based simulations, component software, and visual programming. He received a PhD in computer science from the University of Colroado. He is a member of the IEEE, the ACM, and the AACE. Contact him at ralex@cs.colorado.edu.*

Jonathan Phillips *is a doctoral student at the University of Colorado, Boulder. He is also a senior programmer for AgentSheets Inc. His research interests are end-user programming and communities of learning. He received a BS in computer science from the University of Colorado. Contact him at phillipj@colorado.edu.*

Nicholas Jackiw *is the chief technology officer of KCP Technologies Inc. His research focuses on interactive visualization technologies for mathematics education. Contact him at njackiw@keypress.com.*

Dan Suthers *is an assistant professor of information and computer sciences at the University of Hawaii at Manoa. His research interests are in applying human-computer interaction and artificial intelligence perspectives to designing software that synergizes with minds and social systems, with a current focus on representational tools that support collaborative learning. He received a PhD from the University of Massachusetts. He is a member of the IEEE, the ACM, the Cognitive Science Society, and the International AIED Society. Contact him at suthers@hawaii.edu.*