Roschelle, J., Kaput, J., Stroup, W. and Kahn, T.M. (1998). *Scaleable Integration of Educational Software: Exploring The Promise of Component Architectures.* Journal of Interactive Media in Education, 98, (6) [www-jime.open.ac.uk/98/6]

Scaleable Integration of Educational Software: Exploring the Promise of Component Architectures

Jeremy Roschelle[1], Jim Kaput[2], Walter Stroup[3], and Ted M. Kahn[4]

[1]SRI International
333 Ravenswood Avenue
Menlo Park, CA 94025
U.S.A.
Roschelle@acm.org
www.slip.net/~jeremy

[2]University of Massachusetts Dartmouth
285 Old Westport Road
N. Dartmouth, MA 02747
U.S.A.
JKaput@umassd.edu
www.simcalc.umassd.edu

[3]University of Texas
406 Education Building
Austin, Texas 78712
U.S.A.
WStroup@mail.utexas.edu
http://student.edb.utexas.edu:8001
/ci/faculty/facvitas/Stroupvita.htm

[4] DesignWorlds for Learning
1116 Little John Way
San Jose, CA 95129
U.S.A.
Ted@designworlds.com
www.designworlds.com

Abstract: Technology-rich learning environments can accelerate and enhance core curriculum reform in science and mathematics by enabling more diverse students to learn more complex concepts with deeper understanding at a younger age. Unfortunately, today's technology research and development efforts result not in an richly integrated environment, but rather with a fragmentary collection of incompatible software application islands. In this article we ask: how can the best innovations in technology-rich learning integrate and scale up to the level of major curricular reforms? A potential solution is component software architecture, which provides open standards that enable plug and play composition of software tools produced by many different projects and vendors. We describe an exploratory effort in which four research groups produced software components for the mathematics of motion. The resulting prototypes support (a) integration of the separately produced tools into the same windows, files, and interfaces, (b) dynamic linking across multiple representations and (c) drag and drop activity authoring without programming. We also summarize an extended Internet discussion which raised critical issues regarding the future of component software architecture in education, and speculate on the future need for components for devices other than the desktop computer and for virtual communities that coordinate design teams.

Keywords: architecture, component software, standards

Commentaries: All JIME articles are published with links to a commentaries area, which includes part of the article's original review debate. Readers are invited to make use of this resource, and to add their own commentaries. The authors, reviewers, and anyone else who has 'subscribed' to this article via the website will receive email copies of your postings.

1. Introduction

Technology has the potential to enable core curriculum reform in science and mathematics education (SME) by democratizing access to powerful ideas. Powerful software for designing, experimenting, modelling, simulating, visualizing and communicating can allow students to learn ideas at a younger age and with deeper understanding (Kaput, 1992). Despite the current sense of progress and hope, we are concerned that our community's proposed solutions may not scale to the level of the implementation problems in SME.

The paradigmatic research and development project in SME examines short-term conceptual development. Typically, a project investigates a single concept or small cluster of concepts with one uniform age group over the course of days to weeks. The software involved normally consists of a single program, usually developed over months to a year by a handful of programmers. Our agenda in this article is not to question their value of this paradigm—such design and teaching experiments produce extremely insightful results—but to ask: how can focussed, localized, distributed innovations in technology-rich learning integrate and scale up to the level of major curricular reforms?

Software today is locally effective, but globally fragmentary. Hence, to date, it has had limited impact in systemic curriculum reform. For example, it is awkward to combine software tools that are each valuable in their own niche, and theoretically complimentary in ensemble. In science education, it is natural to want to compare data gathered with microcomputer-based labs (MBL, Mokris and Tinker, 1987; Thornton, 1992) with conceptually-motivated simulations (Snir, Smith and Grosslight, 1993). But because data sensors and simulations are presently authored by different research groups, as different software applications, with different data formats and screen layouts, such comparisons are nearly impossible. Similarly, in mathematics it could be nice to use a video analysis tool (e.g. CamMotion, Boyd and Rubin, 1996) with simulations written by students, say in Logo (Papert, 1980). But as yet, there is no way to construct CamMotion in Logo, and no way to bring Logo into CamMotion. A further schism is between electronic communication tools (such as Internet browsers) and the whole world of dynamic representations and notations. How can the many innovations of educational research become integrated into a practical suite of tools that support large-scale curricular reform?

Answering this question will require transcending the application island architecture that is taken for granted in both the Mac OS and Windows 95. By the term "application island", we mean that software is organized into programs which run independently with their own collection of resources such as windows, menus, and files. Application island architecture provides weak mechanisms for integrating independent innovations. Consequently educational software succeeds in small scale design experiments but fails in large scale systemic reform (Roschelle and Kaput, 1996a). Indeed, examination of recently produced curriculum materials reveals that technology remains at the margins of most innovations and reform (Bork, 1995).

In a progressive technological discipline, each innovation and experiment should accumulate not only as so many grains of sand on a beach, but also as the structures and systems of a functioning whole. Just as internal combustion engines, electronics, and mechanical linkages all fit together to form an automobile, so should graphs, tables, MBL, video, simulations, notebooks, journals, e-mail and collaboration tools all fit together to form a vehicle for 21st century school (and home) learning. In order to focus energy where it is needed most, on students' learning outcomes and changes in teaching practice, we need integration of

independent innovations to become as easy as publishing a web page or producing a newsletter. Thus, a critical challenge for technology in SME is scaleable integration.

In this article, we report on a collaboration among four projects that explored the potential of component software architecture to meet this challenge. Component software architecture is an alternative to application island architecture which addresses the educational needs identified above: it enables composition of modular software objects into larger scale products. The work reported below represents a first exploration of this promising alternative.

We begin by setting the emergence of component architecture into historical context; arguing that component software is a natural successor to four long-standing lines of innovation in educational technology. Next we describe how component software architecture overcomes the problem of fragmentary, incompatible software applications by allowing individual software modules to co-exist, smoothly sharing the space inside a window, the storage inside a file, the resources in the processor, and the user interface. The four collaborating research groups utilized OpenDoc[1] to produce a set of interoperable objects for learning simple MCV concepts. The target of this effort was to demonstrate three critical educational features: (1) live linking across multiple representations, (2) the integration of data gathering and simulation tools, (3) authoring activities by mixing and matching core components in varied containers. We followed this experiment with an extended internet electronic-mail-based discussion of the potential advantages and difficulties of component software, which we will summarize for the reader. Finally, we close the article with two speculative generalizations of our notion of scaleable integration to alternative devices and social knowledge networks.

2. A Short History of Educational Software Architecture

Software architecture means the design of a framework to enable diverse functionalities to co-exist and share resources on a computer. The design of a particular learning tool always takes place inside an architecture, although usually that architecture is taken for granted. For example, most educational software programs use the stand-alone application architecture that is standard on both the Mac OS and Windows 95. Industry standard architectures have evolved from time-shared mainframes, to client-server workstations, personal computer operating systems (such as Windows and the Mac OS), and now possibly towards hetrogenous webs of network computers and low-cost hand-held devices. In this section we introduce a historical perspective in order to show that emerging industry-standard component software architectures are well-aligned with long standing, research-based principles for educational software architecture. The demonstration project that we later present is thus contextualized both by long-standing principles as well as present problems.

Four lines of prior investigation have developed architectural concepts: First, educational programming languages have allowed educators to compose high-quality learning activities in software without requiring extensive technical backgrounds. Second, designers of intelligent tutoring systems have emphasized modularity as a means for controlling the growing complexity of software development. Third, the runaway success of the World Wide Web has brought attention to the value of open standards and decentralized systems. Fourth, educational technologists have long sought authoring tools for teachers

---

1 OpenDoc, Apple Computer Inc. <http://opendoc.apple.com>

and students, emphasizing the importance of a division of labor and team effort in producing high quality educational content.

Below, we briefly review these prior efforts, showing that each raises an critical architectural issue, but also has encountered serious obstacles. In the following section, we introduce Component Software Architecture and show that it has the potential to capture the insights of each line of research, while overcoming the obstacles.

2.1 Educational Programming Environments

Educational programming environments have a long history, beginning with Logo and BASIC. Here we start with "Dynabook" (Kay and Goldberg, 1977; Goldberg 1979), which provided the earliest conceptualization of an entire computer architecture, comprising hardware, operating system, and software, all designed specifically to match learners' needs. In the 1970's, Kay and the Learning Research Group at Xerox PARC conceived and designed a portable, personal computer that would allow children to construct, explore, and extend simulated worlds of their own imagination, while learning about school subjects like the dynamics of motion. The Dynabook hardware explicitly addressed children's needs: portability, ruggedness, high-resolution graphic displays, user-friendly interfaces (e.g. mice and menus) and low cost. The operating system was open-ended, in-line with the expansive possibilities Kay imagined, but would directly support simulations, drawing, word processing, and communications (including a networked electronic library). The software included a child-centered programming language to enable students to construct software and new tools to express their own ideas. The Xerox team sought to design an architecture in line with Papert's (1980) admonition that children should control computers, not be controlled by them.

The cornerstone of Kay's architectural insight was recognizing the need to create a medium in which children could compose ideas without extensive technical training. Kay's method for supporting composition centered on Smalltalk, the first extensible, object-oriented programming language that was designed specifically for children. Smalltalk was designed to support student-designed simulations. It was based on a new programming metaphor, one of defining classes of graphical objects that communicated through passing messages to one another (Learning Research Group, 1976). As diSessa and Abelson (1986) later argued in the Boxer project, computers were to become "reconstructable computational media" and simplified programming languages would become the basic tool for composition. For this architecture to work, learning to program would have to become as easy as learning to use a pencil.

Although early Smalltalk experiments generated many local stories of success, Smalltalk and the Dynabook concept gradually parted ways: The Dynabook led to the Xerox Alto and Star systems (Smith, Irby, Kimball, and Verplank, 1982) and later to the Macintosh, and modern graphical user interfaces. Along the way, the concept of a simple programming language as the core architectural building block for all software was dropped. Instead Applications Programmer's Interfaces (APIs) were instituted, enforcing a separation between professional programmers and "the rest of us."

Today, non-technical people routinely compose ideas on their desktop computers but few use a programming language. Ironically, today Smalltalk is used predominantly by corporate management information systems (MIS) departments, such as those in large investment banks, as well as by a group of professional software engineers—but only rarely by children. Other programming languages for children

have been invented. Logo (which preceded Smalltalk) is the most famous and long-lived (Papert, 1980). More current examples are KidSim (Smith, Cypher, and Spohrer, 1994; now known as "Cocoa"[2]) and AgentSheets (Repenning and Sumner, 1995), which provide a graphical if-then rules rather than linguistic codes in an effort to make programming more comprehensible to children. Yet while Kay's vision of computers as personal dynamic media for creative expression, composition, and simulation has prevailed; programming as a ubiquitous skill has not (Nardi, 1993).

On a more technical level, the idea of a programming language as system architecture has proved problematic. Boxer, in particular, has made a virtue out the fact that every object in the reconstructable medium must be written in the same programming language (diSessa, 1985). On one hand, this does make it possible for learners to inspect, modify, and extend any object in the system, a principle diSessa calls "naive realism." On the other hand, this requires "detuning" (diSessa, 1985, p. 5) and "shallow structuring" (diSessa, 1985, p. 6) of the programming language, in order to control the complexity presented to the novice. Consequently it is easy to learn Boxer, and simple to build a wide variety of useful education interfaces. But it is difficult for Boxer to keep pace with rapidly advancing interactive technology. For example, there is no tool with the depth and power of Geometer's Sketchpad (Jackiw, 1988-97) for dynamic geometry in Boxer, because it would be burdensome to program with Boxer's detuned and shallow structures. As computer systems have evolved, programming languages have become specialized: some languages are better for professional programmers (e.g. Lisp, C++, Java), while others are targeted for educational end-users (e.g. Logo, HyperTalk, AppleScript, JavaScript). Speed, efficient data structures, and low-level communications require one class of programming languages, while ease of use, rapid prototyping, and automation of tasks require another class. Building a component like CamMotion in the Boxer programming language would not be feasible; Boxer does not allow the low-level control needed to control QuickTime, for example.

The metaphor of computers as reconstructable computational medium remains highly compelling—ease of constructing and expressing powerful ideas should be a key criteria for all educational environments. But basing the medium on a single programming language that spans the gamut from operating system to educational scripting has yet to become pragmatic.[3] It appears that no single computer language can be both simple to learn and rich enough for to build professional quality dynamic curricula. As we will see later, component software architecture retains the goal of a reconstructable computational medium, but allows for diversity in programming languages.

2.2 Intelligent Tutoring Systems

Intelligent Tutoring Systems (ITS) research provides a second line of educational technology with explicit architectural concerns. A classic ITS uses artificial intelligence techniques to formulate a model of a student's knowledge and a model of expert knowledge, and then intervenes with tutorial advice when differences become evident (Wenger, 1987). The earliest ITS projects recognized the complexity of the

---

2 Cocoa, Apple Computer, Inc. <http://cocoa.apple.com>

3 Java is the most recent incarnation of the language-as-system concept. Yet Java has quickly stratified into HTML, JavaScript, Java, and "native" levels, each which is appropriate for a different range of authoring problems. The long-term success of Java will largely depend on the implementor's ability to support a coherent community with quite varied authoring abilities and needs.

necessary code, and developed architectures that emphasize modularity. For example, Clancey's (1987) Guidon system explicitly presents a set of components for expert knowledge, student knowledge, interpreting knowledge relative to a task, and apply rules for tutoring the student.

The ITS emphasis on using modularity to control complexity continues to be a central theme in current ITS research, and is a well-established principle in object-oriented design (Booch, 1994). Yet ITS systems have not achieved much re-use, integration, or scalability (Murray, 1996). One critical flaw in ITS architecture is that the inherent modularity of ITS systems is only available to the developers. Once a classical ITS program leaves the shop, it is closed and monolithic. This prevents re-use, extension, or modification of the system except by its original developers. (ITS presents other difficulties, too, such as the profound challenge of constructing useful student and expert domain models. But our purpose here is to identify contributions of ITS to educational software architecture, not to provide a full-scale critique of competing approaches.)

More recently, ITS researchers are seeking to enable non-programmers to compose educational activities by designing authoring systems (e.g. Munro, 1995). These systems, however, are susceptible to the same critique as the Smalltalk and Boxer: every object must be constructed out of a uniform language. This language may be graphical (e.g. RIDES[4]) or may be a knowledge representation language (Murray, 1996). In either case, a closed, proprietary language becomes the feature bottleneck that limits the possibilities of the architecture.

An important emerging trend in ITS design leads directly to component software architecture by emphasizing the use of open standards to integrate modules. Standards have been proposed for knowledge representation languages (e.g. KQML, Finin, Fritzon, McKay, and McEntire, 1994). Also Ritter and Koedinger (1995) are developing a technique for building modular tutors that communicate via industry standard scripting languages. As we argue below, combining modularity with open standards can enable scaleable integration.

2.3 Open Standards and Decentralized Systems

The theme of open standards has been developed by a third line of research, focussed on indexing and retrieval of documents. Starting with Bush's anticipation of hypertext, "Memex" (Bush, 1945), visionaries such as Doug Engelbart (Engelbart, 1962; Engelbart and English, 1968) and Ted Nelson (Nelson, 1987) have imagined interoperable hypermedia document systems and machines for creating, finding, and linking documents and selections, as well as speedily displaying them. Engelbart's and Neslon's early hypertext and hypermedia systems anticipated the importance of interoperable applications and hyper-indexing to support distributed knowledge and intelligence in knowledge-building communities--now a basic feature of 21st century knowledge work.

Weyer first integrated hypertext and information retrieval systems within the object-oriented Smalltalk environment (Weyer, 1982). Nelson's (1987) Xanadu system first proposed open standards for formatting and citing hyperlinked documents. The World Wide Web ("the Web"), which is driven by the HTTP (file transfer) and HTML (document format) standards, provides the best example of scaleable integration to

---

4 The RIDES Project, University of Southern California <http://btl.usc.edu/rides/>

date. The Web has grown exponentially to millions of servers and billions of documents in just a few short years, while retaining well-integrated modes of navigation, cross-referencing, and display.

The Web is widely recognized as an important educational resource, and we expect that as it matures the challenge enabling students and teachers to easily navigate its huge repositories will be met. Another limitation, however, is more worrisome: unlike Kay's Dynabook or ITS tutors, the Web still provides few opportunities for learners to compose or construct their own ideas, and the tools and format for doing so (the "web page") are fairly arcane and constrained. In contrast, educational research, particularly in math and science education, emphasizes the need for students to construct and explore dynamic representations, visualizations, simulations, and animations, using appropriate content-specific analysis tools. Web browsers draw a harsh line between composing and reading; the tools for constructing web pages are separate application islands from the tools for using them. The limitations on dynamic content and composition prevent the current Web from living up to Kay's vision of a child's medium for constructing ideas.

2.4 Authoring Tools

Elements of the fourth research them have already appeared in the previous three themes. Since the early history of educational history, systems such as PLATO (Alpert and Bitzer, 1970; Molnar, 1997) have recognized that high quality didactic content is the result of teamwork: subject matter experts, teachers, pedagogical experts, programmer, and (sometimes) students join to together to produce effective courseware. Developers have sought to support the contributions of diverse team members by creating authoring systems which support effective division of labor among team members with different skills.

Some of the early authoring systems (such as Boxer and Smalltalk, as discussed above) were based around programming languages. Most of the early experimentation in the Xerox PARC Smalltalk project focused on supporting kids as designers using computers as dynamic media: students as young as 10-12 years old were able to program their own interactive games, create animations, compose music and even design new kinds of painting tools (Kay and Goldberg, 1977; Goldberg, 1979). However, few teachers have the time or inclination to become programmers (Goldberg and Kahn, 1997). Rather, a shift has taken place towards a paradigm of authoring and construction by design, rather than programming, including the development of powerful direct manipulation tool kits (Gould and Finzer, 1984; Borning, 1977; Kahn, 1981; Goldberg, 1979).

A watershed event was the introduction of HyperCard (Apple Computer, 1987), which dramatically increased the number of professors and teachers who could produce their own interactive educational applications and courseware. Hypercard succeeded by introducing the stack metaphor for sequencing screens of information, direct construction of fields, buttons, and graphics, and a simple scripting language. Hypercard was also extensible through a modular plug-in architecture ("XCMD") although use of this feature was quite difficult and awkward. Since HyperCard, the trend towards direct manipulation has continued with an increasing diversity of tools (some market leaders as AuthorWare, Director, mTropolis, HyperStudio, and Digital Chisel).

Looking at authoring tools over time, one can find a recurrent trend and counter-trend. As technology power has increased, authoring tools have attempted to support more complexity through an appropriate division of labor, and interfaces which capture generalities in the patterns of production. For example,

today's multimedia tools often support specific editors for different media types, libraries of re-usable media elements, and a structured interface for specifying the flow of information. The counter-trend has emphasized the constraints upon pedagogical variety imposed by these tools, and the limits in subject-matter specificity. For example, an early debate centered on the authoring tools support of "instructionalist" pedagogy, which emphasized traditional didactics versus a more "constructionist" pedagogy (Papert, 1991). With the growth in popularity of constructionist thinking, it has been increasingly important for authoring tools to support alternative pedagogies. Moreover, generic tools such as HyperCard provide little support for representations specific to a subject matter, such as Dynamic Geometry (Jackiw, 1988-97) or Newtonian simulations (White, 1993), which have proven educational value. Subject matter specific tools, however, often provide less generic structure for planning educational experiences. An unfortunate consequence is a considerable fragmentation of the authoring community around particular tools (rather than learning objectives), with limited ability to share innovations across different authoring tools.

Most recently, products from both sides are trying to bridge these gaps. For example, authoring tools like SK8[5] (Spohrer, 1995) can support the construction of subject-matter specific representations. And subject-matter specific tools such as Geometer's Sketchpad (Jackiw, 1988-97) and SimCalc MathWorlds (Roschelle and Kaput, 1996b) allow authoring through drag and drop construction. MathWorlds, in particular, provides AppleScript support, allowing users to write scripts that control any aspect of the interface, record scripts while manipulating the interface, and attach scripts to the interface (Roschelle, Kaput, and deLaura, 1996). Although this is a powerful combination, which can enable considerable flexibility in authoring with mathematical representations, we have found some important limitations to the scripting paradigm for combining functionalities:

- If students' work spans multiple applications, the state of each application must be saved in a separate file. This makes it hard for students to return to a previous state of their work.
- Screen space is controlled by application layers, and students' attention easily gets disrupted as layered windows replace each other.
- AppleScripts are too slow to operate while the simulation is running, limiting there usefulness to moving data before or after a simulation run.

Continuing effort to resolve the dialectic tension between authoring and subject-matter representation essential, for authoring of high quality educational content is clearly a critical mass phenomena, and fragmentation of the communities around specific authoring tools obstructs the accumulation of re-usable innovations.

3. Component Software Architecture

The key principles drawn from the history of educational software architecture (table 1) emphasize a reconstructable medium to support composition, modularity to control complexity, open standards and decentralized systems to scaffold a broad scale implementation, and authoring tools to support teamwork

---

5 SK8, Apple Computer Inc. <http://sk8.research.apple.com>

and division of labor. These principles are echoed by analyses of the software industry in general. Morris and Ferguson's (1993) analysis of the high technology industry shows that architecture is the major factor in long-term, large scale success. They emphasize the need for open standards allow software to be flexibly adapted and extended to meet diverse needs. Cox (1996) argues that modular architectures are required to encapsulate and coordinate the complexity inherent in modern software systems, and emphasizes the transition of the computer from a technical means of computation to a widespread medium for composing, expressing, and communicating ideas.

	Architectural principle	**Architectural obstacle**
Dynabook and Boxer	composable medium	uniform programming language
Intelligent Tutoring Systems	modularity	closed system
World Wide Web	open standards and decentralized systems	sharp distinctions between browsing and composing
Authoring Tools	teamwork and distribution of labor	limited pedagogical diversity, forms of interaction, subject matter specificity

Table 1: Principles and Obstacles in Prior Architectures

A software architecture which adopted all four principles could radically increase the potential for meeting the challenge of longitudinal curricular reforms. Research and development products could accumulate in a digital library of useful learning tools. Open standards could supply a unified target platform on which all the necessary software capabilities would operate, while not restricting implementations to use any particular language or development method. Modularity could allow the natural boundaries in educational objects (e.g. calculators, tables, graphs, simulations, etc.) to be the basis for division of labor among research and development efforts. A composable medium could allow the resulting plethora of powerful tools to be assembled to diverse combinations for particular children, grade levels, curricular goals, etc. Curriculum authors, teachers, or students could easily compose projects from a suite of standard math and science components, favored containers, and internet-savvy collaboration tools. The educational community could leap forward towards scaleable integration by adopting these principles.

To explore the potential advantanges and pitfalls of component software architecture in education we undertook a design experiment using OpenDoc, one of several emerging component software architectures. As we will describe, OpenDoc allows educators to compose objects into a single window, store them in a unified file, and arrange them to support a focussed task. It supports simple programming to automate sequences of behaviors ("scripting"), but does not require it. OpenDoc enables separately developed modules to plug and play in a single process. In contrast to monolithic applications, the resulting "compound" windows are open to inclusion of computational objects from multiple developers. OpenDoc's user interface is carefully tuned to support authoring by typical non-technical users. Moreover, OpenDoc provides open standards which encourage many projects to coordinate their efforts to achieve common goals.

Since the time of our design experiment, OpenDoc's support has been suspended by Apple and IBM, and attention is turned to two alternative infrastructures for Component Software Architecture (CSA) are Microsoft's ActiveX[6] and Sun's JavaBeans[7], a Java-based component architecture. Nonetheless we believe the lessons learned here remain valuable, because all CSAs support similar capabilities (Orfali, Harkey, and Edwards, 1996). These capabilities include:

- sharing interface resources such as windows and menus among components
- embedding components inside other components
- storing a layout containing multiple components in a single file
- linking and updating data dynamically among components
- scripting languages for controlling behavior across components
- interoperating with internet services

In contrast, most current educational software uses an application island architecture, simply because this has been the only available possibility on classroom computers. CSA shares some attributes with modern applications islands, but also provides some important differences. In both CSA and application islands, a programmer constructs a component as a modular object. A typical component in education, for example, might be a graph, a table, or a calculator. But in contrast to stand-alone applications, CSA produces open rather than closed systems. In open system, new objects can be added to an on-going project without the help of a programmer. Indeed, under CSA, a non-technical person can compose a graph and a calculator from different developers into the same window, by simple drag and drop operations. Under the traditional architecture, the original team of programmers must write, compile, and link code to add software from another development group into their program. CSA permits assembly of compound windows from standard parts, whereas older architectures require a programmer to laboriously fabricate each new combination with hand-crafted code (Cox, 1996).

4. EduObject Testbed

Beginning in December, 1995, four National Science Foundation (NSF) projects formed the EduObject testbed to explore the potential of CSA. Table 2 describes the participating projects, which represent both commercial and academic institutions, and diverse NSF directorates. The participants primarily collaborated through the internet, supplemented by a few face to face meetings. The sections below discuss the goals, techniques, and outcomes of the testbed through June 1996.

---

6 ActiveX, Microsoft Corporation <http://www.microsoft.com/activex/>

7 JavaBeans, Sun Microsystems, Inc. <http://splash.javasoft.com/beans/>

Project and Leaders	Institution and Website
SimCalc[8] Jim Kaput Jeremy Roschelle	University of Massachusetts, Dartmouth
MacMotion Ron Thorton Steve Beardslee	Tufts University[9]
DataSpace Bill Finzer	Key Curriculum Press[10]
Constructing Physics Understanding (CPU) Fred Goldberg Arni McKinley	San Diego State University[11] with a subcontract to MetaMind, Inc.

Table 2: Participants in the EduObject Testbed

4.1 Goals

This testbed established 3 goals:

1. Integration of independently developed math and science modules such as simulations, MBL data collection, graphs, and tables into unified displays, interfaces, and files.

2. Dynamic linking of data across multiple representations, such that all representations update synchronously.

3. Authoring of activities by mixing and matching core subject matter components (as in goal 1), containers (such as page layout or word processing), and collaboration tools (such as e-mail and web browsers).

We established these particular goals because of their relevance to educational software needs, and the difficulty of satisfying these goals in a traditional application island architecture. Integration among software modules, and between software and curriculum is crucial for the wide-scale adoption of technology in schools (Bork, 1995; Goldman, Knudsen, and Muniz, 1995). Dynamic linking across simulations, tables, graphs, and other representations of data has proven to be an important feature in many educational technology designs (Kaput, 1992; Kozma, Russell, Jones, Marx and Davis, 1996).

---

8 SimCalc, University of MA, Dartmouth <http://www.simcalc.umassd.edu/>

9 MacMotion, Tufts University <http://daffy.csmt.tufts.edu/>

10 Key Curriculum Press <http://www.keypress.com/>

11 CPU Project, San Diego State University <http://cpuproject.sdsu.edu/CPU/>

Whereas traditional architectures prohibit dynamic linking between application layers, CSA potentially enables synchronous updating. Finally, each project in our group expressed the importance and difficulty of supporting authoring. For example, the CPU project aims to enable teachers who are distributed across the country to easily build their own computer-based activities from stock containers, subject matter parts, and collaboration tools.

Within these goals, we selected the physics and mathematics of motion as our target curriculum, on pragmatic grounds. Three of the four projects include the physics of mathematics of motion in their core agenda, so this target allowed us to draw upon substantial resources. Moreover, extensive research has been performed on the use of simulations, MBL, multiple representations, and dynamic linking in teaching about motion. As a consequence, we could focus on the problems of scaleable integration with confidence that the underlying learning paradigm is sound. (Indeed, this report does not cover empirical work with students. Each of the collaborators will be producing their own evaluations of learning, according to their own project goals.).

Each project in this collaborative had already selected OpenDoc on the Macintosh for its own reasons, making OpenDoc the most convenient platform for the testbed. Nonetheless, lessons learned from our experience should apply equally well to ActiveX, or JavaBeans, and to Windows or other operating systems, as we did not exploit any features unique to one platform.

Components produced to the OpenDoc specification are called "LiveObjects." Using the OpenDoc platform, we decided to develop a variety of LiveObjects that would be useful for learning about velocity and acceleration. As Table 3 illustrates, each project agreed to produce some of the necessary components. In addition, third party commercial vendors produced other LiveObjects that we were able to use.

	Content	**Containers**	**Collaboration**
SimCalc	simulation table graph equation editor digital meter 3D visualization		
MacMotion	MBL data collection vector visualizer		
CPU	vector meter simulation	Dock'Em(tm)	
Dataspace	graph database		
Apple Computer or Shareware	voice recorder drawing tools QuickTime movies	ClarisWorks(TM) Word Processor Outliner	E-mail Newsreader Web browser

Table 3: Kinds of Components Produced By Each Partner

4.2 Linking Multiple Representations

OpenDoc provides most of the infrastructure required to integrate the components listed in Table 3 and thus achieve Goals 1 (listed above). Our primary challenge was live updating of multiple representations specified in Goal 2. In particular, we required that the graphs, visualizations, and tables produced by different projects should all update simultaneously. Moreover, we wanted the same graphs, tables and meters to display data regardless of its source (simulation, MBL, or database).

To achieve live updating of multiple representations, we created an open standard called EduObject. This standard specifies three things:

1. an interface to a shared object that represents a particle (position, velocity and acceleration vectors)

2. a change notification protocol (for synchronizing updates)

3. persistent storage routines (for maintaining cross-component links when saving, closing, and opening windows)

The standard is specified in CORBA IDL (Orfali. et al., 1996) a platform-independent, open standard for describing the interface to a shared object. It is implemented as a shared code library that implements a set of CORBA object classes. Each LiveObject viewer must follow the EduObject standard. In practice, this means the four projects agreed upon the EduObject shared library, and linked their specific components (simulations, graphs, tables, meters, etc.) to it. We negotiated this standard by e-mail discussion, and shipped the shared library to each site via the internet. Because the shared library uses IBM's System Object Model, we avoided the classic "fragile base class problem" in which changes to a shared object require every developer to produce a new version of their module (Orfali, et al., 1996). We were able to revise the EduObject standard and the library many times without breaking any of the existing software modules.

The standard follows the classical model-view-controller architecture for synchronizing views of shared data (e.g. Krasner and Pope, 1988). In our testbed, the model was a collection of particles each with its own position, velocity, and acceleration. Various other components, such as graphs and tables, display views of the model. Each view registers its interest in the model by creating and registering an object that listens for changes. As the simulation updates the model, it notifies each listening view of the changes. A controller is a user interface component that can change the model (and hence the views). We designed controllers driven by the mouse, or by real-time MBL data collection. Full technical details are described elsewhere (Roschelle, 1996) and sample code is available upon request.

Figure 1 is a schematic diagram of how a simulation, graph, and meter component cooperate to synchronize multiple views of an data collected from an MBL device. Importantly, each display view is an separate component. These views can be authored by independent programming teams, and yet they integrate immediately upon being dropped into into the same window.

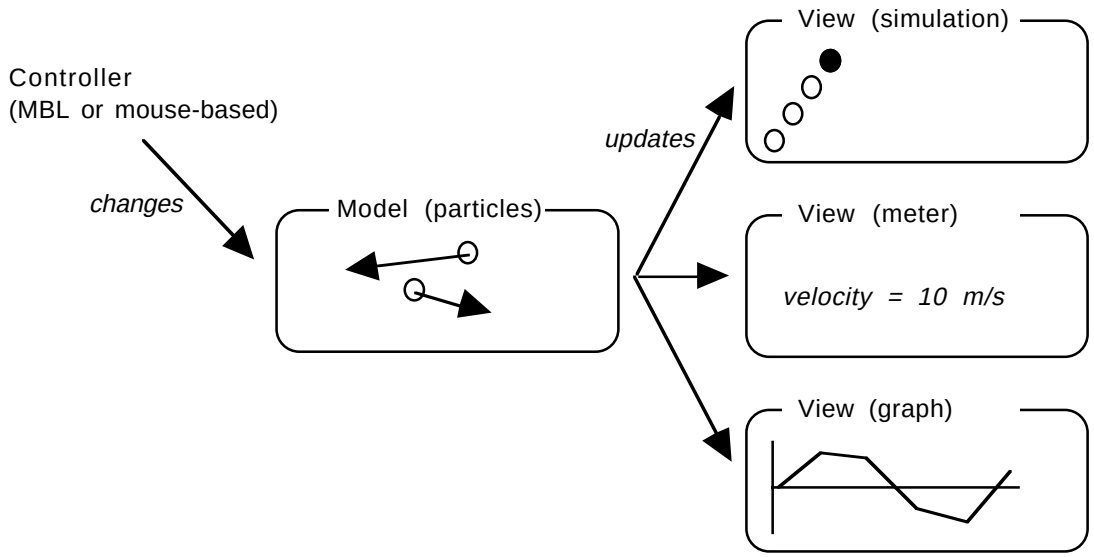

Figure 1: Linking multiple representations a la Model-View-Controller

We started in December, 1995. By early April, 1996, with only part-time effort on behalf of each group, we were able to demonstrate educational activities that were composed by mixing and matching LiveObjects from each of the four collaborating projects. Rarely do software components from different research projects work together at all. Yet this testbed was able to demonstrate sophisticated interoperability in a relatively short time. Below we discuss the range of features this testbed demonstrates.

Figure 2 shows a sample activity in which a student explores a race between an accelerating particle and a constant acceleration particle. The outer container for this activity is Dock 'Em, which supports composing activities as a stack of pages and also supports typing text and drawing shapes. Within Dock 'Em, we have placed a simulation and a position graph, as well as a table of data. When the simulation runs, the graph draws its plot from left to right, and the table highlights successive rows. The activity also includes a voice recorder component, which the student can use to record her thoughts. This activity consists of 5 separate software modules, written by several different authors, which nonetheless behave as one integrated activity. Moreover, by click on the arrow in Dock 'Em, the student can flip to the next activity in the sequence, which can use a different mixture of components, as appropriate.

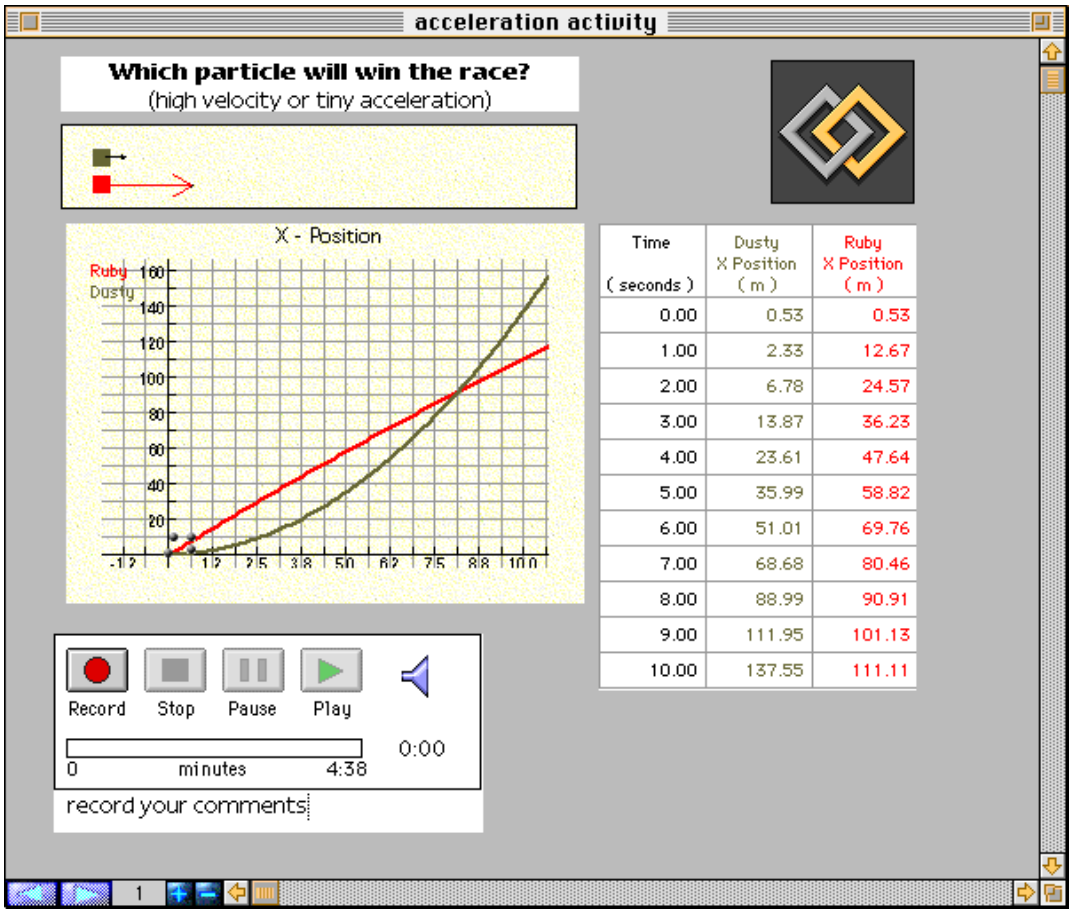

Figure 2: An activity composed of many independently developed components

4.3 Authoring

Our third goal was to enable teachers and students to compose their own workspaces and activities with the familiar ease of desktop publishing. Almost all of the authoring support in EduObject testbed is inherited directly from OpenDoc, which was designed with the needs of non-technical authors in mind. To construct an activity or workspace, a teacher or student can start from "stationery" that provides an appropriate template document. A teacher can place subject matter specific content in the template by drag and drop: the teacher (or student) selects the content on the computer desktop, and moves it into place using the mouse. The components developed by the testbed all interoperate and share data. Once components are on the same page, to establish a link the student or teacher drags a connection to the desired graph, table, or meter. Immediately the view will begin to update as the particle changes. The same views work with data sources that can be in a simulation, MBL data collection, or database.

Moreover, the content components can be embedded in a variety of containers besides Dock 'Em. For example, a student can keep a notebook by dragging and dropping components into an OpenDoc-savvy word processor. Many students and teachers currently use ClarisWorks, an integrated "office" package that combines word processing, drawing, spreadsheets, and other features. In an experimental pre-release

version of ClarisWorks, a teacher or student can drag any of our components (a running simulation or vector visualizer) into their own document. Other containing formats such as draw programs, outliners, and stacks are available from other vendors. Our experience with these containers suggests that authoring activities and lessons that use components may become as easy as authoring activities in lessons that use only text and pictures.

A teacher or student can add additional content to an activity in a variety of media types such as movies, sounds, pictures, 3D renderings, or virtual reality scenes. These content types are supported by LiveObjects provided by Apple. Similarly, Apple's Cyberdog suite allows e-mail, web browsers, file transfer, and newsgroup readers to be added to any activity window (Figure 3). We have explored many of these potential combinations in the testbed, and all work smoothly with our components. Teachers and other activity authors in some of the contributing projects are now actively using these capabilities to compose classroom activities and curricula.

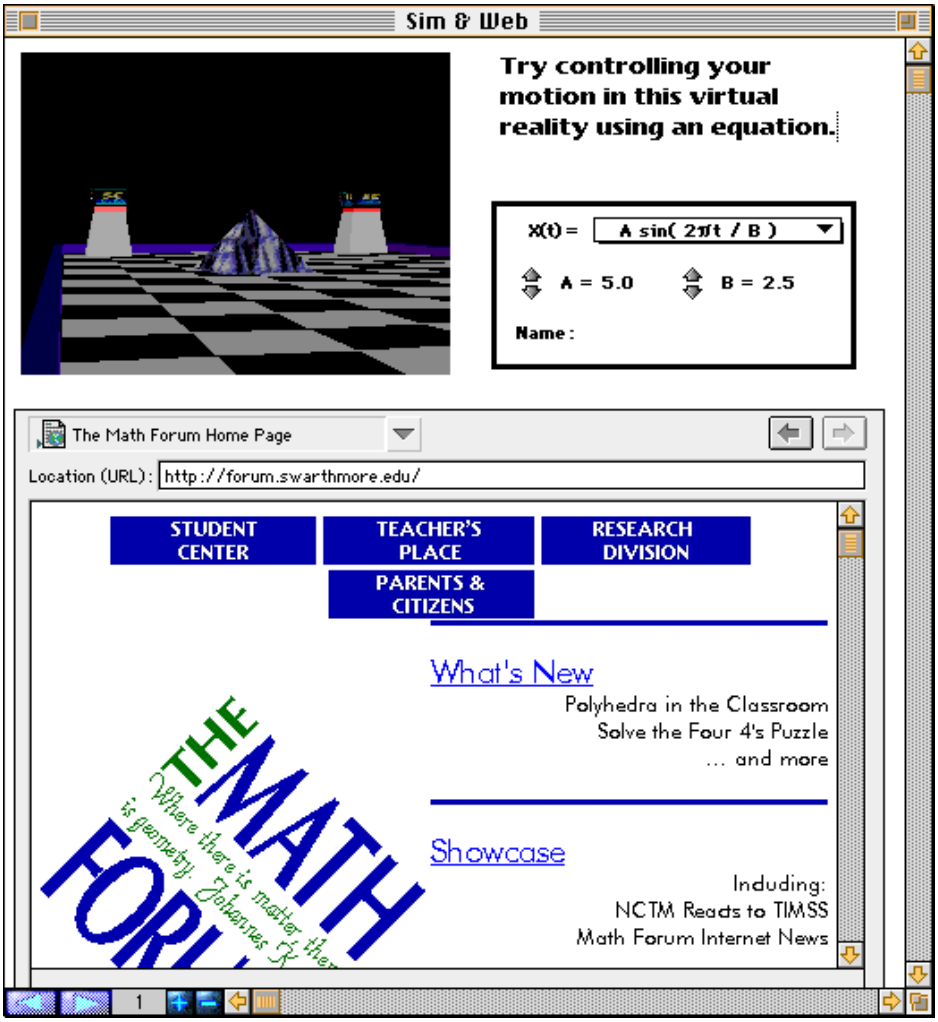

Figure 3: A simulation and a web browser combined in the same window

In separate work, we have explored the use of scripting languages to support user programming, where desirable (Roschelle, Kaput and DeLaura, 1996). Scripting languages can enable many of the desirable features of Dynabook and Boxer without requiring ubiquitous programming skills or a singular programming language. Ritter and Koedinger (1995), for example, have developed a modular ITS agent that interacts with our MathWorlds software via a scripting language. Koutlis (1996) has enabled Logo to be used within OpenDoc, providing teachers and students with a familiar programming language for controlling computational objects. Scripting offers a finer degree of control over the environment, but for a relatively high price—learning to program. Based on extensive, informal conversations with colleagues in industry, we expect that only 2-3% of teachers will be comfortable with any form of programming. In contrast, almost all teachers will be comfortable with drag and drop authoring of word processing documents that contain components. The teachers may however work in teams with programmers to add scripting where their activities require additional customization. Thus, we expect that both drag and drop and scripting will be necessary to support educational authoring.

4.4 Outcomes

Our experiences within this testbed suggest that emerging CSA platforms such as OpenDoc, ActiveX, and JavaBeans, will offer important opportunities for educational projects to achieve scaleable integration. Using CSA, we were able to readily integrate kinds of software that normally fall in distinct application islands. For example, our prototypes can integrate simulations and MBL, exploratory tools and multimedia, and dynamic graphs and communications. We were able to link multiple representations and enable live updating across representations. We were able to support teachers and students in composing their own activities and workspaces. Indeed, non-programmers in several of the cooperating projects are developing OpenDoc-based activities for students to use, and will be evaluating the success of those activities in schools.

Based on our experiences in the testbed, we also believe that CSA provides much stronger support for scaling up than traditional application island architecture. Under application island architecture, larger scale would require a programmer to compile, link, and build a monolithic program that grows larger and more complex with each new feature. In our testbed, however, programmers each worked on a limited component that encapsulated one level of complexity. It was relatively easy for us to collaboratively produce a large tool kit in a rather short time period by integrating innovations from many separate research projects. Moreover, the final form of these innovations readily supported authoring by a much broader community than the programmers who were directly involved.

5. Discussion: Assessing the Potential and Pitfalls

Following the successful design experiment discussed above, we initiated an internet-based electronic mail discussion group to further evaluate the potential of CSA to solve the problem of scaleable integration in educational technology. A complete set of discussion archives is currently available at the Math Forum web site.[12] This discussion list was widely announced and drew approximately one hundred participants during the 3 months of peak activity. Interestingly, the discussion spanned groups that

[12] EdComponents mail list archives, The Math Forum, Swarthmore College <http://forum.swarthmore.edu/edcomponents>

traditionally have been mutually exclusive: representatives of the Intelligent Tutoring System community and Mathematics Education researchers, and both academic projects and commercial publishers participated. CSA is clearly an issue that interests a broad range of educational technologists.

During the discussions, three general issues were raised. Each presents a potential challenge to be met in order for CSA to be productive in educational research and development.

1. **Project Management and Intellectual Property.** Component software implies greater cross-project dependencies. Hence researchers are concerned about how they will cross-license software, and manage schedules that require delivery of components from other projects. When larger scale products (such as a curriculum) are assembled, there needs to be a system for ensuring that each contributor gets appropriate financial and intellectual credit.

2. **Data-linking Standards:** In order to build software that dynamically links multiple representations, the community needs standards for linking data throughout components such as simulations, graphs, tables, and equations. At the moment, there is no process in place for arriving at technical standards within the educational community, although there are ample models in industry, government, and non-governmental standards groups.

3. **Appropriate Component Platform Choices:** Several component-like foundations are emerging from industry (including VisualBasic, Java, OpenDoc/LiveObjects, OLE/ActiveX, and Netscape plug-ins). Each serves some purposes better than others, and researchers and developers need a pragmatic strategy for the short-term that will enable creation of needed knowledge, without engaging in short-sighted platform wars or investing in development inappropriate to their real needs.

Since the time of these discussions, several groups have raised and started to address additional challenges. For example, CSA for education is based on a public network infrastructure, issues of security will arise, both with respect to distribution of components and educational content and with respect to student records and identities. Some of these issues are now being considered within the IMS Project[13] and the IEEE Learning Technology Standards Committee.[14] In addition, members of these two groups are at work at additional levels of component interoperability. Steve Ritter and colleagues have been working on modular interfaces connections between "tools" (such as described in this article) and "tutors" as represented in the ITS tradition with in the context of the IEEE effort. The IMS project is working towards interoperability between components and distance learning administration systems. It will be a challenge to bring all these levels of componentization together in a functional system.

The challenge of authoring educational content spans several levels of components, frameworks and standards. In this article, we have primarily been concerned with components at the user-interface level and the data types they manipulate. Education also has components, frameworks and standards at the level of curriculum—a component might be a lesson plan, state and local school districts often have subject matter specific frameworks, and national organizations are presently setting learning standards (such as

---

13 The Instructional Management System (IMS) Project, EduCom <http://www.imsproject.org/>

14 Learning Technology Standards Committee, IEEE <http://www.manta.ieee.org/p1484/>

the National Council of Teachers of Mathematics[15]). The authors of this proposal, along with several other partner institutions, expect to soon receive funding from the National Science Foundation for a new project that would address these interrelations. The new project is called "Educational Software Components of Tomorrow (ESCOT): A Testbed for Sustainable Development of Reusable, Interoperable Objects for Middle School Mathematics Reform." This project will seek to bring together interoperable JavaBean components to explicitly address the needs of five new middle school mathematics curriculum.[16] We will also coordinate with E-Slate project in Greece[17], which has similar goals for slightly different curriculum. We can imagine that the same JavaBean components will be reused across many different classroom lesson components, and these may be reused across different curricula and classrooms. The introduces opportunities to conceptualize and implement scaleability at multiple levels.

Another important dimension for future work would investigate the social architecture required for scaleable integration: how can we anticipate and support an emerging community of practice (Lave and Wenger, 1991, Wenger, 1998) around component software and customizable curriculum? As we have argued above, authoring often involves teamwork among participants with diverse, specialized skills. In the future, such teamwork might be supported by virtual communities (Hagel and Armstrong, 1997) which grow around the creation, use, modification and maintenance of a shared library of re-usable software objects. The community might include members with expertise in subject matter and teaching, as well as members with technical expertise in customizing software components. These communities will also need tools to support these communities' design and reflection practices, such as new Web-based hypermedia versions of applications such as Xerox PARC's Instructional Design Environment (IDE), (Russell, Burton, Jordan, Jensen, Rogers, and Cohen, 1990). A social architecture for such a community would provide standard mechanisms for teams to form, execute a focussed project, and then contribute new or improved software back into the public repository. We are presently exploring such ideas for an Educational Object Economy[18] with our colleagues.

6. Speculations: Scaleable Integration Beyond Computers

So far in this article, we have been considering the problems of integration among educational software tools running on a single desktop computer. Educational technology, however, is not limited to computers. A device of particular interest in mathematics and science is the graphing calculator. Graphing calculators have become nearly ubiquitous in high school and university math, science and engineering course. They are also making significant inroads into mathematics education at the middle and even at the late elementary school level.

Graphing calculators have many advantages over desktop computers. Calculators are inexpensive and thus can be personally owned by most students. They are portable, and thus travel from mathematics class to science class, and to the students' home. Many calculators are now user programmable, and thus can run software beyond the burned-in mathematics calculations. Indeed, in the not-so-distant future, hand-held

---

15 National Council of Teachers of Mathematics <http://www.nctm.org/>

16 ESCOT Project <http://wise.sri.com/escot/>. Contact Roschelle@acm.org for more information.

17 The E-Slate Project, Computer Technology Institute, Greece <http://www.cti.gr/RD3/EduTech/E-Slate.htm>

18 The Educational Object Economy, EOE Foundation <http://www.eoe.org/>

devices will be able to run programs written in the Java programming language, and thus calculators and computers will be able to exchange small Java programs. The idea of scaleable integration could therefore be extended to cover interoperability across different kinds of hardware, including calculators and a range of so-called "Personal Digital Assistants" such as Apple's Newton and US Robotic's PalmPilot, through wired or wireless network connections.

Assuming the development and implementation of sufficiently robust local network protocols we envision the day when each student's calculator becomes an fully accessible object to a more powerful computer acting as the server for a classroom of networked calculators. Students could download projects and assignments from the workstations to their calculators, perhaps by an infrared beam. Later they could upload results, examples and ideas to a workstation. The workstation might have a display projector so that the whole class could see ideas from any source, and could easily compare results emerging from individual student's calculators. A workstation could also support assessment tools such as a portfolio that could maintain a long term record of students work. This kind of communication and computational interoperability would allow both for close teacher and peer engagement with the activities of individual learners. Based on preliminary work in sixth through tenth grade math classes in Boston, we have reason to believe that this "one computer, many calculator" model of classroom integration can result in significant learning gains related to the national standards in mathematics and science.

7. Conclusions

Two decades of educational technology research and design has produced a large collection of prototype tools which have a proven ability to improve mathematics and science education. Yet sadly, these programs currently exist as fragmentary application islands, each which holds only a piece of the solution. The learning technology employed in research centers has not yet achieved widespread availability and use (Chipman, 1993), nor impact on mainstream teaching and learning (OTA, 1988). The lack of a mechanism for accumulating and integrating independent innovations is at least partially at fault.

Emerging infrastructures such as OpenDoc, ActiveX, and JavaBeans offer a platform that contains the needed mechanism in the form of component software architecture. CSA is consonant with the principles of reconstructable media, modularity, open standards and division of labor. And these principles have an extended history in research on educational programming environments, intelligent tutoring systems, digital libraries, and authoring systems. Component software architecture thus has the potential to deliver on the promise of scaleable integration of educational software: integration because CSA supports plug and play composition, and scale because CSA supports additive, layered composition of complementary components.

CSA has some disadvantages, too. It requires more coordination across organizations, and equally important, a higher level of trust among developers working in different organizations. CSA raises challenging intellectual property licensing issues, which must be resolved in order to commercially distribute works that aggregate multiple components, while protecting each owners rights and ability to make a profit. CSA platforms and tools are less mature than corresponding platforms and tools for monolithic applications, requiring compromises in performance and ease-of-use. Nonetheless, we expect that the economic realities of educational software RandD (Roschelle and Kaput, 1996a)—low profit margins, very small development teams, difficulties in capturing high market share—will increasingly push developers and publishers to overcome these obstacles.

Indeed, given our own experience, we believe the advantages of CSA will become increasingly attractive to software developers and researchers. In our role as developers, we cannot afford build high quality versions of each component we need from scratch. In fact, some components, such as a computer algebra, are so expensive to build that we cannot afford to build them at all. And yet in our role as researchers, we need an ability to compose different combinations of dynamic representational tools, to see which combinations and sequences help students learn. CSA could allow our development efforts to focus on narrow niches where we can make a unique contribution while allow our research efforts to draw upon a much wider collection of standard educational components.

CSA is also likely to be attractive to funders, such as the National Science Foundation, who would like a interoperable collection of technologies to aid in systemic reform, along with the research to guide their effective use. Instead of funding replicated efforts to build many independent application islands that each only cover part of science and math education, the NSF could aggregate its production of innovative components across projects, resulting in considerable efficiencies and a more useful end product. Publishers could re-use this collection of MCV components with different authors and produce electronic curriculum targeted at different state frameworks and local needs. Importantly, the collection would never be closed to innovation—a new and improved graph, for example, could always be substituted for a older model without needing to rebuild the entire collection. Moreover, commercial and research-based innovations could easily be combined.

Our experiments with CSA suggest that the technical infrastructure for achieving scaleable integration is largely available. Still, significant research and innovation will be needed to realize educational visions on top of industry standard architectures. We know little, for example, about the appropriate granularity of educational objects, or what teachers will need by way of support and training to use them effectively in real classrooms. Many combinations of components will suddenly become possible, and educators will need to know which combinations and sequence work best. The potential of a suite of modular educational tools needs to be properly articulated with standards-based curricula and modernized assessment practices.

Yet the biggest obstacles to achieving scaleable integration may be social and not technical. CSA will require a community of practice in educational technology research and development (RandD) that emphasizes cooperation and coordination, in place of independent, autonomous research activity. Supporting authoring will have to become a major concern through the RandD enterprise. The scope of research will have to include not just short term, localized learning experiences, but also the articulation between these and larger systemic effects. Indeed, the nature of the learning sciences enterprise might need to grow to include a higher objective: to understand how a virtual community can pool resources and engage design teams to provide high performance learning experiences at the scale of whole curricular strands, major school districts, and diverse populations of teachers and students.

Acknowledgements

We first thank our collaborators: Steve Beardslee, Bill Finzer, Fred Goldberg, Ken Koedinger, Arni McKinley, Steve Ritter, and Ron Thornton. We're also grateful to the SimCalc "swamp" team, Rich DeLaura, Jim Correia, and James Burke, for their efforts. Jim Spohrer and colleagues in the Educational Object Economy provided helpful feedback on earlier versions of this article, as did the two reviewers. The work reported in this article was supported by the National Science Foundation (Awards: RED-9353507 and REC-9705650). The opinions presented are the authors, and may not reflect those of the funding agency.

References

Alpert, D. and Bitzer, D. L. (1970). *Advances in Computer-Based Education.* Science, 167, 1582.

Apple Computer (1987). *HyperCard* [computer software]. Cupertino: Apple Computer, Inc.

Booch, G. (1994). *Object-Oriented Analysis and Design with Applications*. Redwood City, CA: Benjamin-Cummings.

Bork, A. (1995). *Why Has the Computer Failed in Schools and Universities?* Journal of Science Education and Technology, 2(4), 97-102.

Borning, A. (1977). *ThingLab—An Object-Oriented System for Building Simulations Using Constraints.* Fifth International Joint Conference on Artificial Intelligence, Cambridge, MA, Aug. 22-25, 497. (Also in Xerox PARC Report SSL-79-3, 1977).

Boyd, A. and Rubin, A. (1996). *Interactive Video: A Bridge Between Motion and Math.* Interational journal of computers for mathematical learning, 1(1), 57-93.

Bush, V. (1945). *As We May Think.* Life, September 10, 112-124. (Adapted from article in Atlantic Monthly, July, 1945.)

Chipman, S.F. (1993). *Gazing Once More Into the Silicon Chip: Who's Revolutionary Now?* In S.P. Lajoie and S.J. Derry, Computers as Cognitive Tools. Hillsdale, NJ: Erlbaum, 341-367.

Clancey, W.J. (1987). *Knowledge-Based Tutoring.* Cambridge, MA: MIT Press.

Cox, B. (1996). *Superdistribution: Objects as Property on the Electronic Frontier.* New York: Addison-Wesley.

diSessa, A.A. (1985). *A Principled Design for an Integrated Computational Environment.* Human-computer interaction, 1, 1-47.

diSessa, A.A. and Abelson, H. (1986). *Boxer: A Reconstructible Computational Medium*. Communications of the ACM, 29(9), 859-868.

Engelbart, D.C. (1962). *A Conceptual Framework for Augmenting Human Intellect* [Summary Report, Contract AF49(638)-1024]. Menlo Park: SRI International.

Engelbart, D.C. and English, W.K. (1968). *A Research Center for Augmenting Human Intellect.* AFIPS Proceedings, Fall Joint Computer Conference, 33, 395-410.

Finin, T., Fritzon, R., McKay, D., and McEntire, R. (1994). *KQML as Agent Communication Language.* In the Proceedings of the Third International Conference on Information and Knowledge Management (CIKM '94). New York: ACM Press.

Goldberg, A. (1979). *Educational Uses of a Dynabook.* Computers and education, 3, 247-266.

Goldberg, A. and Kahn, T. (1997). *Personal communication*.

Goldman, S., Knudsen, J., and Muniz, R. (1995). *When Promise Outweighs Problems: Technology Integration in Math Classrooms.* < http://www.ndec.sesp.nwu.edu/ndec/ProjectPages/mmap/mmap.html>

Gould, L. and Finzer, W. (1984). *Programming by Rehearsal* [Technical Report SCL-84-1]. Palo Alto: Xerox Palo Alto Research Center (PARC).

Hagel, J. and Armstrong, A.G. (1997). *Net Gain: Expanding Markets Through Virtual Communities*. Boston: Harvard Business School Press.

Jackiw, N. (1988-97). *The Geometer's Sketchpad* (Software; various versions). Berkeley, CA: Key Curriculum Press.

Kahn, T. (1981). *An Analysis of Strategic Thinking Using a Computer-Based Game.* Unpublished doctoral dissertation, Dept. of Psyschology, University of California, Berkeley, CA.

Kay, A., and Goldberg, A. (1977). *Personal Dynamic Media.* IEEE Computer, 10(3), 31-41.

Kaput, J. (1992). *Technology and Mathematics Education.* In D. Grouws (Ed.) A handbook of research on mathematics teaching and learning. NY: MacMillan, 515-556.

Koutlis, M. (1996). *(Personal Conversation).*

Kozma, R., Russell, J., Jones, T., Marx, N., and Davis, J. (1996). *The Use of Multiple, Linked Representations to Facilitate Science Understanding.* In S. Vosniadou, E. De Corte, R. Glaser, and H. Mandl (eds.), International perspectives on the design of technology-supported learning environments. Mahwah, NJ: Erlbaum. 41-60.

Krasner, G.E. and Pope, S.T. (1988.) *A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80.* Journal of Object-Oriented Programming, 1(3), 26-49.

Lave, J. and Wenger, E. (1991). *Situated Learning: Legitimate Peripheral Participation.* Cambridge: Cambridge University Press.

Learning Research Group (1976). *Personal Dynamic Media* [Technical Report No. SSL-76-1]. Palo Alto: Xerox Palo Alto Research Center (PARC).

Mokris, J.R. and Tinker, R.F. (1987). *The Impact of Microcomputer-Based Labs on Children's Ability to Interpret Graphs.* Journal of research in science teaching, 24(4), 369-383.

Molnar, A. (1997). *Computers in Education: A Brief History.* T.H.E. Journal, 24, June 11, 1997, 63-68.

Morris, C.R. and Ferguson, C.H. (1993). *How Architecture Wins Technology Wars.* Harvard Business Review, 86-96.

Munro, A. (1995) *Authoring Interactive Graphic Models.* In T. de Jong, D.M. Towne, and H. Spada (Eds.), The use of computer models for explication, analysis, and experiential learning. New York: Springer-Verlag.

Murray, T. (1996). *From Story Boards to Knowledge Bases: The First Paradigm Shift in Making CAI "Intelligent."* In Proceedings of Ed-Media 96 - World Conference on Educational Multimedia and Hypermedia, American Association of Computers in Education. Charlottesville, VA: AACE.

Nardi, B. (1993). *A Small Matter of Programming.* Cambridge, MA: MIT Press.

Nelson, T. (1987). *Computer Lib/Dream Machines.* Redmond, WA: Microsoft Press.

OTA (1988). *Power On! New Tools for Teaching and Learning.* Office of Technology Assessment, Washington DC: US Government Printing Office.

Orfali, R., Harkey, D., and Edwards, J. (1996). *The Essential Distributed Objects Survival Guide.* New York: John Wiley and Sons.

Papert, S. (1980). *Mindstorms.* New York, NY: Basic Books.

Papert, S. (1991). *Situating Constructionism.* In I. Harel and S. Papert (Eds.), Constructionism. Norwood, NJ: Ablex. 1-11.

Repenning, A. and Sumner, T. (1995). *Agentsheets: A Medium for Creating Domain-Oriented Visual Languages.* IEEE Computer, 28, 17-25.

Ritter, S. and Koedinger, K.R. (1995). *Towards Lightweight Tutoring Agents.* Proceedings of the the 7th World Conference on Artificial Intelligence in Education, August 16-19, 1995, Washington, DC, USA. AACE: Charlottesville, VA.

Roschelle, J. (1996). *Updating Parts Dynamically with SOM.* MacTech Magazine, 12(8), 73-89.

Roschelle, J. and Kaput, J. (1996a). *Educational Software Architecture and Systemic Impact: The Promise of Component Software.* Journal of Educational Computing Research, 14(3), 217-228.

Roschelle, J. and Kaput, J. (1996b). *SimCalc MathWorlds for the Mathematics of Change.* Communications of the ACM, 39 (8), 97-99.

Roschelle, J., Kaput, J, and DeLaura, R. (1996). *Scriptable Applications: Implementing Open Architectures in Learning Technology.* In P. Carlson and F. Makedon (Eds.), Proceedings of Ed-Media 96: World Conference on Educational Multimedia and Hypermedia, American Association of Computers. Charlottesville, VA: AACE. 599-604.

Russell, D.M., Burton, R., Jordan, D., Jensen, A-M., Rogers, R., and Cohen, J. (1990). *Creating Instruction with IDE: Tools for Instructional Designers.* Intelligent Tutoring Media, 1, 1, 3-16.

Smith, D.C., Cypher, A. and Spohrer, J. (1994). *KidSim: Programming Agents without a Programming Language.* Communications of the ACM, 37(7), 55-67.

Smith, D.C., Irby, C., Kimball, R, and Verplank, B. (1982). *Designing the Star User Interface.* Byte, April, 242-282.

Snir, J. Smith, C. and Grosslight, L. (1993). *Conceptual-Enhanced Simulations: A Computer Tool for Science Teaching.* Journal of Science Education and Technology, 2(2), 373-388.

Spohrer, J.C. (1995). *Site Description: Apple Computer's Authoring Tools and Titles R&D Program.* Artificial Intelligence Review. Netherland: Kluwer Academic Press.

Thornton, R. (1992). *Enhancing and Evaluating Students' Learning of Motion Concepts.* In A. Tiberghien and H. Mandl (Eds.), Physics and learning environments [NATO Science Series]. New York: Springer-Verlag.

Wenger, E. (1987). *Artificial Intelligence and tutoring Systems.* Los Altos, CA: Morgan Kaufman.

Wenger, E. (1998). *Communities of Practice: Learning, Meaning and Identity.* Cambridge: Cambridge University Press.

Weyer, S. (1982). *Searching for Information in a Dynamic Book* [Technical Report No. SCG-82-1]. Palo Alto: Xerox Palo Alto Research Center (PARC).

White, B. (1993). *Thinkertools: Casual Models, Conceptual Change, and Science Education.* Cognition and Instruction, 10, 1-100.