Interactive Learning Environments
2003, Vol. 00, No. 0, pp. 000–000

Taylor & Francis
Taylor & Francis Group

Issues in Component Computing: A Synthetic Review*

Andrea A. diSessa, Flavio Azevedo, and Orit Parnafes
Berkeley Boxer Project, Graduate School of Education, University of California, Berkeley, CA, USA

ABSTRACT

This paper provides a review of the rhetoric behind the component movement in educational software, and a critical analysis and synthesis of issues underlying the movement. We draw on case studies of several significant recent component projects in order to assess claims and to uncover and examine issues that are less often considered. While our empirical base cannot definitively answer all the questions raised, we hope to bring some clarity and some empirically based judgments to bear on how a promising technological innovation can best serve educational ends.

Our study led to a focus on three critical issues: (1) the nature of the environment in which components are configured and used; (2) the extent of modifiability that is necessary for effective re-use of components; (3) how the work of designing components and component configurations is distributed among people with different competencies.

INTRODUCTION

Over the last decade or more, accelerating in the last 5 years, a potentially very important technological trend has grown in the development of software – a trend reflected vividly in the work of many creating software for educational purposes. Broadly speaking, this trend aims at developing many smaller pieces of software, in comparison to traditional practice of creating large (in many cases, huge) and essentially monolithic programs or applications.

*This work was supported by a grant from the National Science Foundation, number REC-9973156, to Andrea A. diSessa. The conclusions and interpretations drawn here are those of the authors, and not those of the NSF. The PI of the sponsored work reported here has a financial interest in PyxiSystems LLC, which is the owner of the Boxer software.
Address correspondence to: Prof. Andrea A. diSessa, Graduate School of Education, University of California at Berkeley, EMST, 4533 Tolman Hall # 1670, Berkeley, CA 94720-1670, USA. E-mail: disessa@dewey.soe.berkeley.edu

1049-4820/03/0000-000$16.00

"Component software,"[1] according to the rhetoric of advocates, has a number of advantages:

- Re-use: Smaller functional elements means components should be re-usable in a number of contexts. In contrast, large applications are an all-or-none proposition, and potentially useful pieces of them – say, a nice graphing module as part of a physics simulation – are not separable from the main application and cannot communicate with other software, such as a new simulation. A new application that requires graphing must "reinvent the wheel," which wastes time, money, and, likely, sacrifices quality.
- Rapid Development: With a library of prefabricated components, development can proceed much more rapidly, and at a higher semantic level. For example, one does not have to think about how a graphing module should be implemented; just take an off-the-shelf component that does what is needed.
- Economics: Re-usability and rapid development would seem obviously to contribute to lower development cost. New social structures (below) may also contribute to reduced cost by allowing educators to do a fair amount "development" on their own.
- Accessability: While not necessarily the case, component software is often linked to Internet strategies for distribution. Small components can be quickly downloaded from the Web. On-line libraries pool the efforts of many developers.
- Adaptability: With the level of programming moved upward toward familiar semantic elements, it may well be true that re-arranging and reconfiguring components is an easier task than "programming" usually connotes. Subpopulations involved in education, even schools or individual teachers, may change and adapt component software to their local circumstances. Although not widely appreciated, local adaptability is a fundamental attribute of good and likely-to-be-successful innovation.
- New Social Structures of Production and Consumption: Production can be more distributed with regard to (1) level of technological expertise (non-programmers can become more involved) and (2) institutional affiliation (development does not need to happen all at the same physical location) compared to traditional models of software development.

[1]In order to encompass a range of educational work, we use a deliberately encompassing definition of "component" – a reusable unit of code with specified embedding and connecting protocols. A prototypical component in the work reviewed here also has an associated visual presence, typically a screen region with its own IO properties.

With the exception of the last, new social structures, all of these characteristics are fairly prominent in the public rhetoric of advocates of component computing. The first three are nearly universal claims. Re-use is listed first for a good reason; it is a cornerstone property that forms the basis for several other claims, especially rapid development and economics. The fourth claimed advantage, accessibility, seems only slightly less prominent.

The expectation that putting components together is not only faster, but also technically easier than building components is an implicit foundation for several of these claimed advantages. Roschelle, Pea, DiGiano, and Kaput (1999) use the analogy of a home entertainment setup, where few can create components such as amplifiers, receivers and speakers, but everyone can hook them together to construct a suitable system. The expectation that configurations should be easy to change, as well as to produce, provides adaptability; users and other less technically oriented individuals may be able adjust pre-made configurations independently. If experts are needed to adjust a system, still they may be able to do it much more easily. For reference, we call this property ''easy configuration,'' and treat it as a part of adaptability, although it might well appear explicitly in a longer list of core claims concerning components.

''New social structures'' has been less often cited as an advantage in component computing, at least until a few years ago. However, it has been implicit in the way many real-world projects worked, if not in their rhetoric. In any case, we feel this is an important part of the possible advantages of component computing. All of the projects featured in our study offer important lessons concerning new social structures.

Easy configuration is a part of a theme of ''democratizing technology'' that one finds frequently expressed in discussions of component computing. The theme is often explained in terms of *social factoring*. Social factoring specifically involves the expectation that programmers or other technology experts apply their special skills in producing components, and educators will apply their expertise in assembling and using component systems (see EOE, undated; Roschelle et al., 1999). Together, easy configuration and social factoring underlie many intuitions about new social configurations.

EMPIRICAL AND ANALYTICAL STRATEGIES

In this section we describe the empirical basis for our synthetic comments, and something about how we came to think about the diverse issues involved in

component computing. Naturally, our project team read widely in the realm of educational component computing, and somewhat less outside of educational circles. Component computing seems to have different properties outside education (stemming from, e.g., less standardized and stereotyped requirements for education, the multi-disciplinary expertise needed in education, limited financial resources, etc. See Spalter, 2002). In any case, our focus here is specifically educational uses.

Empirical Strategies

The most important empirical work we did involved constructing fairly intimate case studies of three current projects that were accessible to us: The E-Slate Project (University of Athens, and CTI; Greece; www.cti.gr/RD3/Eng/EduTech/); The ESCOT Project (SRI International; www.escot.org/), and the Educational Object Economy (www.eoe.org/). In addition, we used some of our own experiences with component computing, ''we'' being the Boxer Project at the University of California, Berkeley (www.soe.berkeley.edu/boxer). All of the projects include a significant focus on K-12 mathematics and science instruction. The EOE and E-Slate are substantially broader in their overall content focus, and, at the other end of the spectrum, ESCOT is highly targeted at middle school mathematics.

Obviously, we are most familiar with our own project work. In addition, Boxer is the oldest of the projects here considered, which means a more extensive and diverse base of experience. These two facts mean that our own experience may appear somewhat more prominently in some analyses and judgments compared to that of other projects. However, we tried hard to retain an objective balance of views – including separating off more ''editorial'' comments in a sidebar article (The Boxer Project's Perspective on Component Computing).

In preparing case studies, we relied on the following sources of data:

- Reading – Including websites, papers from the projects, and internal documents that were made available to us.
- Visits and joint meetings – When feasible, we visited projects, including sites where their products were used, attended working meetings, and engaged in informal discussion with project members. Different projects were differentially accessible, given that one project is located in Greece (although we managed one intensive ''site visit'' and a shorter, more recent visit), and one project is nearly completely distributed.

- Examining products, mainly those accessible on the Web.
- Engaging users in discussions.
- Discussions with outsiders who had experience with the projects and their products.

In planning our work, we hoped to concentrate on teachers and their experiences with component computing. Unfortunately, several factors intervened. First, component computing is still an emerging technology, and school sites where it is used are fairly rare. In addition, the sites that exist are widely separated geographically, and we could not travel to many of them. Some visits were possible, and electronic communication helped fill in. However, overall, we were less successful in engaging teacher perceptions than we had hoped.

In addition to our study of component projects, we organized a Web-based survey concerning issues that emerged in our work. The survey was small and, by design, involved mainly experts and those now using components in K-12 educational work.

The case studies, or *profiles*, that resulted from our work are available on the World Wide Web (Azevedo, 2001; diSessa, 2001a, 2001b; Kynigos & Friedman, 2001; Parnafes & diSessa, 2001). All of the profiles can be found at: http://soe.berkeley.edu/boxer.html/webcomp/reports.html. In order to keep this article manageable in length, we do not report details of data or interpretation here that are reported in the profiles. The Web survey has not yet been reported separately.

Analytical Strategies

Our primary analytical strategy was systematically to collect, group, and sort for importance issues having to do with component computing. So, for example, we kept a database of issues as these were encountered, and sought to develop a better sense of those issues, how they played out in various contexts, and any resolution that we could see, given our empirical data. The database itself does not find explicit representation in this synthesis, except that we used it to make sure that important items we had encountered were not left out. Issues we considered fall, broadly, into two classes:

- issues concerning the social patterns of production and consumption;
- issues concerning the technology.

The first two main sections, following this one, concern these two classes. Questioning the validity of the ''core claims'' listed above, constituted an important sublist of issues, which we treat later in its own section.

Student learning is an important part of patterns of production and consumption. The bottom line for many involved in educational computing is, after all, whether students learn more or better through the technology. On the other hand, we dealt only with a small number of very different projects, each of whose work depended on many particular factors (the insight of developers in designing good learning activities; the quality of teachers; the preparation of students, etc.), and each of which focused on different curricular areas. So, comparative evaluation of learning was not possible.[2]

Models

One of the most challenging aspects of our work arose from the incredible diversity of ways of going about doing ''component computing.'' The projects we studied were very different in their social organizations, in their technology, and in the relations with the world of education. How, then, can we approach generalities about component computing, without unreasonably suppressing special features of these projects? A partial solution is to talk about the projects, not in terms of individualities, but in terms of ''models'' that are intended to be abstract or ideal types. The models we chose to develop are not prototypes of some full specification of a way of going about doing component computing, for example, simplifications of what any of our studied projects actually did. Instead, these models are deliberately *partial specifications*, a collected set of attributes that ''naturally'' go together, and which have substantial and fairly uniform consequences across projects that might implement that particular model.[3] As in any modeling enterprise, there is no guarantee that one can find insightful models of this sort, nor that, once found, assumptions made about them are valid. Nonetheless, models helped our thinking about and communicating our findings.

[2]For convenience of readers, we provide some entry-ways to projects' individual studies of student learning: For E-Slate, see Kynigos and Yiannoutsou (2002). For ESCOT, some study of student learning was done by the Math Forum's (www.mathforum.org) internal evaluation process. However, we are not aware of published results. Boxer has done a great deal of micro-analysis of student learning. For some overviews, see diSessa (1995, in press).

[3]In as much as they are partial specifications, this type of model differs from prototypes, caricatures (Hoyles, Noss, & Sutherland, 1991), or composites (Wenger, 1998), which constitute relatively complete specifications of a realizable, if idealized, type.

One of the advantages of models as partial specifications is that they have a ''mix and match'' character. That is, a real project might be well characterized by several of these models, not just one ''closest match.'' This allows, in principle, better coverage of the whole space of possible ways of doing component computing, and it allows us to extrapolate to ways that are not directly represented in our data, yet might be particularly ''good combinations'' of the strengths of different projects.

We offer here a preliminary taxonomy of models. Some issues we discuss at length later are exposed in a preliminary form here.

1. **The educationally adapted model**
   The prime characteristic of this model is serious and consistent attention to the educational use of components. This may entail (a) explicit descriptions of pedagogical use and style, (b) linking to established curricular topics or standards, (c) involving pedagogical experts in development, (d) attention to issues of real-world use by teachers, and (e) built-in or add-on assessment. Note that projects with very different philosophical orientations might still be described as ''educationally adapted.'' Each of our profiled projects aimed at being educationally adapted, but in very different ways.

2. **The rich container model**
   Although the motivation for this model is to attend to some social and educational issues, it is defined by a technological issue. The technological issue is the richness of resources in the ''container'' environment. The *container environment* is the ''place'' where components are assembled and presented to users. While containers are typically viewed mainly from the developer's perspective (where the name ''authoring environment'' is appropriate), the perspective relevant to rich containers involves resources that are available to all, developers and users alike. The rich container idea is to embed components within a broadly flexible environment with substantial resources of its own. In this way, any user or developer will have a lot of generic resources (text processing, hypertext, perhaps even programming) available from the container so as: (a) to fill in needs that components do not supply; (b) to allow easy interconnection and reconfiguration of components; and possibly (c) to allow easy modification and adapting of components. The rich container model aims mainly to give general computational resources to less technologically sophisticated individuals, including teachers and students, so that they can carry out tasks that otherwise would require specific design and implementation by

experts for any particular application. The issue of what technical contributions can or should be expected of less technologically sophisticated individual, and what benefits accrue from that, is a critical issue that we will consider at some length. A rich container is a partial solution to the problem of appropriate technology in the case that technical contributions by teachers and students are judged valuable.

3. The co-development model

User participation in design and construction of component-based educational software has many possible advantages, including (a) to offer face-value validity of products for the eventual audience, (b) a larger developer pool, and (c) the possibility of serious adaptation to local circumstances and concerns. There are, however, many different ways to engage in co-development. In addition, there is a fairly widespread distrust that many teachers can participate effectively in the design of technology. Thus, co-development is implausible to many, especially when teachers play roles beyond simply testing and providing feedback. Each profiled project undertook a different form of co-development. We will develop more refined distinctions in a taxonomy of different co-development models (in the section on social configurations). The various roles assigned to different communities – that is, different definitions of social factoring – account for a substantial part of the variation in these models.

4. The toolset model

An obvious assumption, and explicit commitment of many, is that generic components (like a graphing component), should evolve, but then standardize in order to serve all similar uses (e.g., all graphing needs). The toolset model is based on a slightly different idea, that the ideal component for a particular purpose (e.g., physics of motion, or biological simulations) will, necessarily, be adapted to its particular context of use. Thus, while generic components may be useful, they will need to be adaptable, and developers will need to make adaptations before a set of components (a toolset) will readily serve a particular need. In addition to adapting components to particular uses, toolsets might imply fairly strong co-configuring of components to make sure they work together well in the particular context. The toolset model interacts strongly with the social issue (one of our three critical issues) of who gets to develop and modify software. Just as the rich container model is a partial solution to the problem of technology access for non-specialists, the toolset model is a

partial solution to the problem of appropriate technology in the case that components systematically need substantial adaptation to contexts of use.[4]

The remainder of this report will consist of (a) an essay on advantages and disadvantages of various social configurations; (b) a review of the technology used by each project; (c) an essay of our choice of three most crucial issues facing component computing; (d) a review our general finding concerning the ''core claims'' of component computing, and (e) summary and concluding comments.

SOCIAL CONFIGURATIONS IN THE PRODUCTION OF COMPONENT SOFTWARE

How people collaborate in the production of software and how software is selected and drawn into educational practice constitute a critical family of issues that are easily marginalized within a technologically-oriented movement such as the component movement. Like most people involved in the component movement, we believe a good component-based product, like any other technology-based educational product, is likely better served by a collaboration among people having various expertise, including technical, educational and design-related expertise.[5] From the earliest days of educational computing, developers sought to involve those with educational expertise productively in the process. Still, there are better and worse ways of organizing collaboration, and any collaboration may be difficult to manage. Collaboration between educational and technical people may be difficult for many reasons, including:

- Divergent views: Different people, especially if they hold different kinds of expertise, may have different values, priorities and ''lenses'' to view ''how things should be done,'' and, particularly, what constitutes high quality and the likelihood of achieving that quality in a particular case.

[4]The toolset model is very similar in motivation to the product line view of enterprise software development (Clements & Northrop, 2002). Emphasis on community practices in that view parallels in some degree our highlighting of social configurations.

[5]Collaboration among communities is not universally advocated as a best method for developing educationally adapted software. The Education Division at MIT, which ''fathered'' the MIT's Media Laboratory's educational work, was founded on the presumption that creating a new cadre of interdisciplinary experts is better than interdisciplinary collaborations in many ways. Echoes of this idea appear when we talk about what it takes to allow teachers to be good technology-developing partners.

- Social hierarchy: Whether institutionalized or not, differential power and authority may develop in collaborations. Technologists tend to have high status (or, in a self-fulfilling manner, assume they have high status) compared to educators, especially teachers. Degrees, salaries, assumptions about articulate argument, and so on, can systematically favor some participants over others.
- Community-specific practices: Participants in collaborations usually retain a main affiliation with their "home" community. Patterns of practice, rewards and sanctions, and so on, all differ, which can cause subtle or explicit misalignments. For example, technologists gain approval in their community for designing and making systems; long periods of waiting for feedback from educational field trials, may cause at least tension, if not disruption.

All of the projects we profiled explicitly advocate one version or another of the co-development model, in which teachers or other educators join in the work of development. On the other hand, we found a remarkable and fortuitous divergence in how these projects went about collaborating. We next schematize these ways of collaborating by introducing partial specifications, models, specifically with respect to how collaboration is organized. These models constitute an empirically motivated taxonomy of submodels of the co-development model.

All of the projects seem to have managed workable collaborations, in some cases rather remarkable ones. Behind the scenes, our impression is that a huge amount of effort and cleverness went into achieving their results. What we can manage here is only to expose the tip of the iceberg with respect to the "black magic" of successful collaboration in the production of component software. In component computing, as in other educational matters, there is often an abundance of optimism with respect to collaboration, but less outright success, and still less articulation of how good collaborations can be managed. In that context, it is especially important to develop a balanced view of the strengths and weaknesses of various models of collaboration.

The Integration Team Model

The integration team model combines members of different communities in relatively small, product-oriented design groups.[6] Most typically, technologists (such as programmers or software designers) and educators (typically

[6]The idea of integration teams is quite old, going back at least to the 1970s work of Al Bork at the University of California, Irvine.

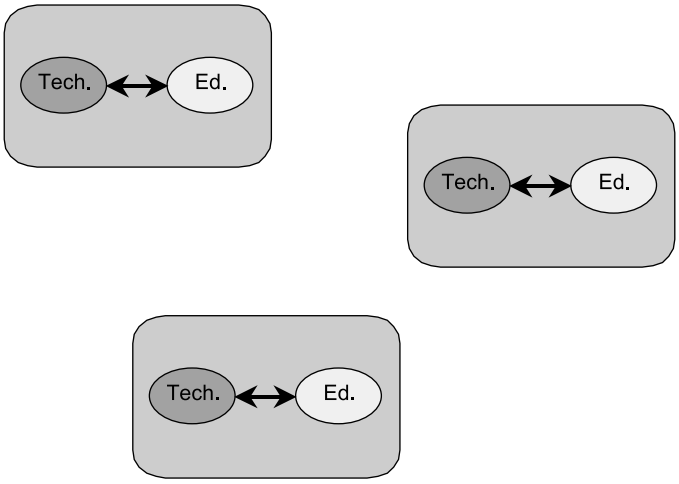

Fig. 1. Integration teams are small, product-oriented groups composed of members of different communities, notably technologists and educators.

teachers) collaborate. As other co-development models, this configuration is guided by the assumption that teachers and educators may have a much larger role in designing component computing software than typically assumed, and that they should add their educational point of view to the development process. Social factoring is prominent within the integration team model: It is assumed that teachers draw on their own teaching practices, bring considerations of pedagogy, curriculum, classroom and school reality, and contribute their pedagogical ideas. In turn, technology experts guide achievable design, and usher designs into suitable electronic forms. Such collaboration is claimed to produce higher quality and more educationally adapted activities and supporting software. Among our profiled projects, ESCOT advocated and used the integration team model.

Opportunities

The involvement of educators and teachers in design should promote meaningful connections to school reality and to the curriculum. Activities should better match teachers' and students' needs. Products are enriched by the contribution of multiple and different perspectives. Moreover, professional growth, broadly conceived, may be enhanced. That is, integration teams promote a dialog between teachers and technology developers so both sides better understand each other's perspectives. Technology developers get a good acquaintance with

educational considerations, which may enhance their ability to develop better educational software. Teachers get the not-so-common opportunity of participating actively in innovative, likely-to-be reform-oriented technological design, and become better prepared to offer technology-adapted suggestions.

Difficulties and Limitations

The promising characteristics of this model do not preclude most of the generic difficulties that characterize ''cross-cultural'' collaborations. In our empirical work, both divergent views and hierarchy (see descriptions, above) were fairly prominent,[7] at least in the eyes of some participants. Some teachers found it difficult and sometimes intimidating to participate as equal contributors in a technology-based development process. Technological developers involved with educational implementation often have, in addition to technical competence, considerable experience with instructional design and in mathematics and science as well. This may put teachers in a weaker position in which they do not have authority in technology-related issues, but neither can they act with clear authority with respect to content and educational issues. Especially at the beginning stages of teamwork, teachers may feel intimidated and thus less able to contribute, even where their view is especially valuable. Over longer periods of time, different perceptions may be bridged and merge into a productive team effort. The ESCOT project reported difficulties in their first year of integration teamwork, but redesign of team relations and accountability made things much better – in fact, they reported that later integration teams ran excellently. Our independent view confirmed significant improvement, but many issues, especially issues of hierarchy, remained salient in some participants' views. Overall, these results anticipate one of our general conclusions concerning collaborative models: that managing collaboration is as or more important than which basic model of co-development is chosen. Some details of our observations concerning the ESCOT project can be found in Parnafes and diSessa (2001).

The Two-Legged Model

The two-legged model implements collaboration between two main organizations: a technical development team and an education research team

[7]In contrast, we saw little indication of systematic conflict in community practices (the third of our three general loci for problems in collaboration). Perhaps this is because of the limited duration and very focused work of integration teams we observed.

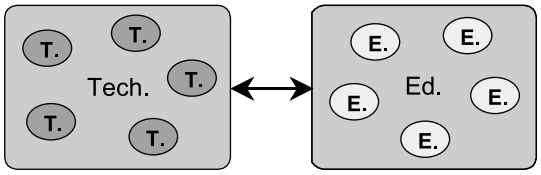

Fig. 2. The two-legged model involves systematic interaction between two different teams, typically oriented toward technology and educational matters, respectively.

(Kynigos, 2002). Once again, social factoring is assumed. The technical development team focuses mainly on component software design and development, and the education research team drives the educational considerations of components, activity design and formative trials. Cyclical development is a good method for this model, where the software is co-designed by the two teams working in conjunction, and tried and tested by the educational team as early as possible in realistic situations. The E-Slate project implemented a two-legged model.

Opportunities

This model may be particularly easy to start up. Separate teams may already exist – such as within universities or research and development institutes – so forming the model may be simply ''gluing together'' already functioning teams. Each team enjoys community-specific coherence of goals, shared beliefs and language, and therefore collaboration within a team should be relatively easy and smooth. Educational research participants do not need to get their hands dirty with highly technical tasks, and they are liberated to focus on educational issues such as learning research, working in classrooms with teachers, organizing teacher communities and developing activities from given components.[8] Technology development participants also focus on the things in which they are expert. They get input from the educational team and try to address the educational teams' needs in developing and refining the component software. The basic social factoring idea applies to the two-legged model: Presuming good collaboration and mutual understanding between the

[8]In the E-Slate project, the educational team also took on substantial responsibility for educational aspects of technology design, including advocacy of basic system properties, such as easy scriptability. We suspect, but cannot know, that this level of participation in technical matters would be exceptional across many implementations of the two-legged model.

teams, the development process can be efficient since each team is concentrating with what its participants do best.

Because the technology team works in an almost ''business as usual'' mode, this model may be particularly apt for larger software development. Among our profiled projects, the E-Slate two-legged model did, in fact, manage by far the largest software development. Infrastructural software (such as E-Slate or Boxer, as systems) may require the coherence of a specifically technological team, and may benefit less from educators who are not deeply steeped in the constraints and affordances involved in large systems design.

Difficulties and Limitations

Coordinating two big organizations that ''do their own thing'' is never an easy task. Generally, then, misalignments of community-specific practices may be prominent in two-legged designs. This contrasts with the relatively sparse appearance of such misalignments in the integration team model. Among these, difficulties in time coordination were highlighted in reports we got from the E-Slate project. Educational and technological processes have different natural time-scales and different modes of operation. Interruptions of one team's work to accommodate needs of the other can be destabilizing. For example, one cannot tell a development team to wait for a year while formative trials assess the educational affordances of an existing implementation. In complementary manner, the educational team may have trouble keeping up relations with schools and teachers while development undergoes inevitable delays and buggy prototypes.

Looser coupling between educators and technologists, compared to integration teams, has its disadvantages as well as advantages. The fact that participants are more insulated within their home communities means mutual understanding may be more difficult to come by. Discoordinations such as educators making unachievable demands and developers producing general resources that are not crafted specifically enough to serve educational need may be more common than with some other models.[9] Relative cultural insulation may mean more difficulty managing day-to-day relations and negotiating longer-term goals, and the intimate cross-fertilization of designs typical of integration teams is reduced. Long-term professional development

[9]See discussion of the ''lemon tree'' phenomenon in Kynigos and Friedman (2001).

that crosses disciplines is also likely reduced. Consult Kynigos and Friedman (2001) for details of our empirical observations of the E-Slate project, and Kynigos (2002) for additional discussion.

Member-Sustained Community Model

A member-sustained community brings together individuals with varied backgrounds, such as teachers, developers and researchers around the task of developing and distributing educational software components. The key idea is a symbiotic bartering of expertise. Developers distribute and get feedback on their creations; educators can download components, and they provide a service to other educators in documenting the educational properties of the software. Others have used the term ''knowledge network'' to describe this organization: ''[An] Internet-based community of experts – teachers, researchers, developers, and others – that self-organizes to publish, share, find, critique, and improve software resources and associated materials'' (Roschelle, Pea, et al., 1999, p. 2).

Social factoring is generally assumed. A teacher participating in such a community need not understand anything about technology design or implementation. Rather, she or he may use an available component, contribute evaluations or descriptions of effective use, suggest educationally relevant modifications to the component, or suggest a new component altogether. The Web site library serves a much more important function than simply to hold components or software; it is a medium of intellectual exchange, and must be

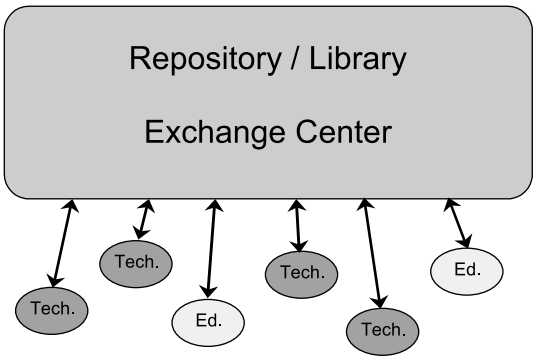

Fig. 3. A member sustained community involves different people with different interests and expertise, all participating in a symbiotic relationship that is typically mediated by a Web-based library or exchange center.

designed with that in mind. Documentation of both educational and technical properties (including evaluations) are vitally important. The Educational Object Economy (EOE) first conceptualized and implemented this model. ''Economy'' refers to the ''bartering of expertise'' of which we spoke. In the considerations here and below, it important to note that we refer mainly to the original model of operation for the EOE organization, which now operates under substantially different models featuring smaller, more local communities. In higher education, the Merlot Project (www.merlot.org) adheres to a member-sustained community model.

Opportunities

This model requires minimal financial and organizational commitment since the community of interested people should maintain and develop it. Member-sustained communities are a natural way of combining different expertise (a form of co-development) with minimal organizational overhead. Member-sustained communities also provide an opportunity for people with different expertise and points of view to contribute and engage in a dialog, and they appear to be naturally scalable.

Difficulties and Limitations

The loose organization that accounts for this model's ''low cost'' – even looser than the two-legged model – also seeds a number of difficulties. Lack of strong central management and goal-setting can lead to chaotic development of the group and its resources. For example, we observed that ''coverage'' in terms of components of the EOE is uneven, with a lot of redundancy in some areas, and lack of coverage in others. Quality, which is supposed to be kept up by community critique, may wane if the participants are not up to, or are not interested in critical assessment. Furthermore, without management, the balance in the community may be less than ideal. In particular, since technology is prime currency, technologists tend to dominate. We observed in the main EOE collection that components are many; educational commentary is sparse. Once tilted, new members and contributions tend to perpetuate the imbalance. Educational adaptation of offered software, supposedly ensured by educators' contribution, suffers.

On-line communities are simply difficult to manage and keep up. This problem is exacerbated by the fact that teachers as a group tend to be isolated in current practice, and often lack both interest in, and skills to understand how to use and critique software. Their primary job is to teach, not to provide

feedback or to help others learn pedagogic concerns. Generation of good software ideas by ''average teachers'' is rare.

Unless the community actively uses and critiques components, and unless developers are responsive, re-usability may suffer. With limited commitment, developers are likely to post components and be done with them. Re-usability in new combinations is minimal in the current EOE library due in part to: (a) inaccessible code; (b) a general tendency not to document code; (c) no strong cultural impulse toward re-use. We hasten to add that the EOE technological base (mainly black-box Java applets) does not foster re-usability; hence limited re-use may not be inherent in the member-sustained community model.

For details concerning our observations relative to this model, see the EOE profile (Azevedo, 2001). As mentioned, newer practices of the EOE, which we did not profile, tend to focus on smaller communities, which likely allows better management, more focused goals, and better community communication (channels other than the library, itself), alleviating many of the problems here cited. See Gaible (this issue).

The LaDDER Model

The LaDDER model (Layered Distributed Development of Educational Resources) involves four ''layers'' of participants: students, teachers, secondary developers, and primary developers (or simply developers), all working together to develop learning materials over an extended period of time. Figure 4 illustrates, using three layers for simplicity. The core goal of the model is to empower levels of participants with less technological expertise, especially teachers and students, to solve as many of their own problems as possible. Thus, the model works well in conjunction with the toolset model, creating flexible resources, in addition to software solutions.

Typically, developers will initiate construction of a toolset and will provide model configurations for educational use. Yet, a lot of work – modifying, trying out new configurations and activity structures – can happen among teachers, likely in conjunction with members of a helping class, what we call a secondary developer. A secondary developer might be, for example, a member of a university community or a teacher with more-than-usual technical competencies, but with more pedagogical expertise and local community connection than a primary developer. One function of the secondary developer is to provide leadership and liaison, both up and down the technical-expertise ladder. Face-to-face groups, say, in a particular school or school system, might

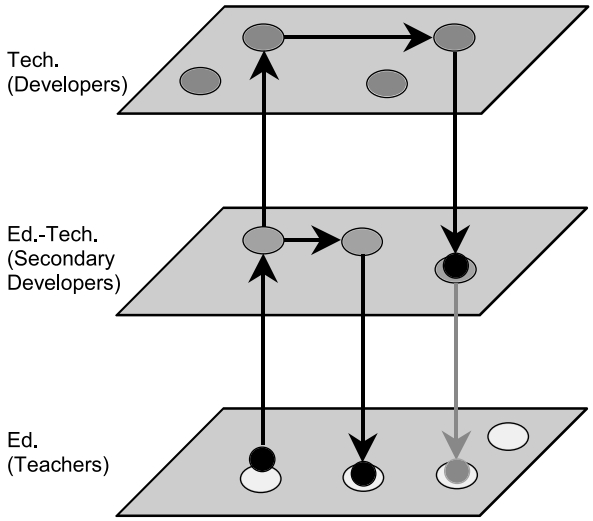

Fig. 4. In the LaDDER model, problems percolate up the Hierarchy of expertise (uplinks) from teachers, to secondary developers, to primary developers; know-how and resources to solve problems (downlinks) propagate downward.

be headed by a secondary developer. Primary developers may not need to have as much direct contact with the project. Secondary developers serve a similar function to what Bonnie Nardi calls ''gardeners'' in her work on end-user programming (Nardi, 1993).

The characteristic pattern of work in the LaDDER model is that technical needs or problems propagate up the technical competence hierarchy (students to teachers, to secondary developers, to primary developers) to the point that they can be addressed. Then, however, instead of solutions, new or modified resources are created and propagated as far as feasible back down the hierarchy before converting those resources into solutions. The Boxer Group has experimented with the LaDDER model (diSessa, 2001b).

Opportunities

The LaDDER model, in enhancing educational adaptedness by empowering educational participants, aims at enhancing the education side of the education/technology balance. This may be particularly felicitous if, as is suggested by some of our data, technology generally holds the upper hand within the component movement. Flexibility and local adaptation should be

enhanced at the same time. At least some teachers can act relatively autonomously, seeking help from ''up the LaDDER'' only occasionally.[10]

Because of its emphasis on producing autonomy and because of the helping, secondary developer level, professional development of teachers may be enhanced. If it works well, responsibilities of primary developers to be involved in every technological change can be lessened. In principle, misalignment of community-specific practices may be minimized by the looser coupling between technology and educational participants, and difficulties of hierarchy are addressed by emphasizing teacher and student empowerment. Our own experiences with the LaDDER model have been highly positive, especially with respect to limiting the responsibilities of technologists and promoting local adaptation.

Difficulties and Limitations

The LaDDER model is much less familiar and less common than the other social models, and there is less data on which to generalize. Examples studied within the Boxer project may have worked for idiosyncratic reasons, including the people involved and the domain investigated. (See diSessa, 2001b, for details.) A given presumption in this model is that everyone, notably including teachers, has some degree of technical expertise, which is not a situation to be taken for granted, nor a likely prospect in the estimation of many. See Chapter 9 in diSessa (2001) for extended discussion. The LaDDER model is also dependent technologically on something like the rich container model, where users have a lot of building resources available to them. A trade-off for empowering teachers is that the model may tend to be educationally conservative. That is, if ''average teachers'' develop materials, ''average (and not particularly innovative) outcomes'' may result. However, several management strategies are available, such as selecting innovative teachers to work with and using innovative technology as a ''Trojan mouse.''[11]

[10]Because of the technology they used, teachers could not work independently in ESCOT integration teams. E-Slate's technology was explicitly designed to advance independence in teachers. Unfortunately, our primary work finished before that project began serious work with teacher autonomy. Readers should consult newer E-Slate references, such as Kynigos, Trouki, and Yiannoutsou (2002).

[11]The term ''Trojan mouse'' refers to an attractive and seemingly unthreatening artifact that, nonetheless, seeds major change.

While it may reduce burden and, especially, time-coordination problems with respect to developers, the model does depend on developers' involvement over an extended period of time, albeit at reduced levels. Finally, the model may be better adapted to fairly local projects, and less adapted to large-scale, say, ''national curriculum'' development, where a single product is expected. LaDDER is not adapted to major infrastructural software innovations: Boxer, itself, was not developed using this model.

We close this section by noting, again, that collaboration is a complex theoretical and practical matter about which we can have no pretensions concerning complete analysis. In particular, we emphasize that on-the-ground strategies of managing collaborations may overcome intrinsic weaknesses, and, in any case, such strategies must account for a significant part of variations in success. More generally, we feel there is a great need for understanding of management issues specific to these kinds of collaboration. Such understanding is very difficult to develop by projects, themselves, at the same time that they actually do development. Thus, it may make for a good supplemental focus for research in future projects.

TECHNOLOGIES: REVIEW OF PROJECT ARCHITECTURES

The second principal perspective we entertain is technology. What are the range of relevant possibilities and issues concerning the technological bases of component computing? We begin concretely with a brief review of the technologies underlying the four profiled component projects.

Component Construction

Three of the four projects we profiled (EOE, ESCOT, E-Slate) used Java as a component construction language. Java is an evolved, modern, reasonably high-level programming language that, importantly, has as a central aim to be platform-independent: Constructions should, in principal, run equally well under any operating system. Java has generic standards for components (''beans''). ESCOT and E-Slate add further specifications for components that participate in their architectures.

Boxer does not currently use Java. Instead, Boxer is an autonomous, integrated medium. That is, it provides a rich collection of resources, including text processing and programming, to any user; hence, it adheres to

the rich container model. A programmer works in the Boxer medium by using text processing to create text-and-boxes documents that are programs. To create a component, the programmer typically puts all relevant programs in a box, and creates a ''boxtop'' for that box, which is the presentation and interface he wants users to see. Users can use the component directly (as one would just use a Java applet), but there is nothing that prevents the user from ''opening the box,'' looking at, modifying or extending the programs contained inside. If a component is not feasibly constructable in Boxer for speed or memory efficiency reasons, it can be implemented in Lisp (Boxer's implementation language), at the cost of becoming ''black box'' from a user's point of view.

Component Hosting, Configuring, and Connecting

In creating a particular learning application from components, one selects from available components and creates a particular screen configuration. In addition, one must interconnect the components to make them work together properly for the purpose one has in mind. One might select a simulation component, a graphing component, a text component (for instructions and documentation) and a few control components, such as slider controls and buttons, to form a control panel. The graphing component has to be connected to the simulation, and the control components likely need to be connected to both the simulation and graphing component.

ESCOT and the EOE use Web browsers to host their components (provide the screen space and interaction protocols). The EOE does not demand or highlight combinable components, so configuring and connecting are not particularly relevant operations. ESCOT components are Java beans, and they use some standard Java technology for interconnection as well as some proprietary protocols and graphical techniques to simplify interconnection in common cases. ESCOT delivered black box applets (configurations of components) to students.

One of the non-negotiable commitments of the E-Slate project has been to make components configurable by non-experts, in particular, by teachers. Hence, they developed a proprietary container environment. The principal paradigm of interconnection is ''wiring,'' and components can be connected by using type-coded plugs and sockets. In addition, one of the components available with E-Slate is a Logo programming system. Components can export component-specific commands to the Logo component, and Logo can thus control some aspect of the component. Scripting is therefore available as an

inter-component connection method that is more adaptable and more general (if sometimes more complex) than wires.[12]

Boxer has a ''one environment for all purposes'' commitment. The environments in which one generally creates components, in which one combines and configures them, and in which configurations are used are one and the same. Boxer's always-available ''text'' editor has a sense of geometry and objects (components may simply be boxes), so layout is encompassed. The most natural interconnection method is scripting, but a host of more specialized methods are feasible. See diSessa (2001b). In particular, a simple wiring protocol (implemented in Boxer) allows one to connect components with a ''drag and drop,'' plug-to-socket gesture. Boxer does not have a built-in wiring editor, in contrast to E-Slate.

The Roles of Programming

Programming plays a number of roles, discounting infrastructure building such as creating the E-Slate container, ESCOT interconnection protocols, or Boxer environment. First, obviously components need to be created. Boxer is the only system of the four that uses a general-purpose language that is intended to be user accessible to implement most components. Somewhat along these lines, however, ESCOT used two *component generators* (in addition to Java programming) for the purpose of creating some components, dominantly the Geometer's Sketchpad (/www.keypress.com/sketchpad) for geometric constructions, but also AgentSheets (www.agentsheets.com), which is a grid and agent-based programming system. We consider these to be specialized programming systems that have the intent of end-user accessibility.[13] They are the equivalent of Boxer toolsets[14] (see elaboration below).

For component interconnection, ESCOT uses generally non-user-accessible programming. E-Slate allows programming for interconnection, and it is user

[12]Whether E-Slate should be described as having a rich container or not may be a matter of semantics. It certainly is enriched (by connectivity editing) compared to a browser host. Programming does not reside in the container, but can be added to any configuration of components. How important is always-available hypertext processing, ala Boxer?

[13]Most people use Sketchpad as a gesture-based construction system, hence would not consider it to be programming. We do not feel it is appropriate to restrict ''programming'' to text-based systems. In our view, any system that allows the construction of complex, dynamic and interactive objects should be classified as programming.

[14]Indeed, one Boxer toolset allows gesture-based construction of dynamic, geometric objects. See Sherin (2002).

accessible. Newer work of the E-Slate Project on teacher creativity and professionalization depend on interconnect (and other) functionality of programming in the hands of teachers. Boxer reverses the priority and depends mostly on user-accessible programming for interconnect, but also can support wiring, if not currently as elegantly as E-Slate.

Boxer and E-Slate also explicitly support student programming as an expressive practice with important pedagogical value in its own right. This heritage from the Logo tradition (Papert, 1980) manifests itself as student activities that involve programming. So, for example, students in E-Slate learn about topics in mathematics, particularly geometry, by programming (Kynigos, Koutlis, & Hadzilakos, 1997), and students in Boxer learn about motion via programming representations (diSessa, 1995; Sherin, 2001).

THREE CRITICAL ISSUES

The astute reader will have noticed that we did not define models of technology (''architectures'') for each project in the way we did for social configurations. The reason is the technologies involved in this study do not seem to be classifiable in a way similar to the simple topological scheme that applied to models of collaboration. Technology for components is more diverse and multidimensional than our top-level organization of collaborations, and these projects do not appear to us to represent any pure forms.[15] In addition, the main issues we want to talk about cut across projects in more complex ways than the tradeoffs for each model, which organized our discussion of social configurations. On that basis, we proceed directly to discussion of our three main clusters of issues, subsuming technological issues into a more general discussion mainly as a space-saving strategy. We start with a relatively simple issue, and progressively move to more complex ones.

[15]Consider wiring. ESCOT has wiring, but only for experts. E-Slate wiring is prominent in the system, and intended for users, not just developers. Wiring is possible in Boxer, but it is not a uniform, explicitly represented system property. Similarly, user programming is possible in E-Slate and Boxer, but it may be (and we will argue) that it has different properties in the two systems. Of the four projects, Boxer may best represent a ''pure'' model – a ''computational medium'' where all users have access to all generic computational resources (diSessa, 2000). This is a particular instantiation of the rich container model.

The Critical Container

The first of our three critical issues emerges directly from our discussion, above, concerning component hosting, configuring and connecting. Web browsers have some strong advantages, and a few possibly critical disadvantages as container environments. On the one hand, they are free and ubiquitous. A lot of industry effort goes into developing and supporting the technology. One of the key downsides, which was amply reported to us by the ESCOT project, is that industry standard technology is still not very good for supporting rapid and stable component interconnection and re-use. Among other things, browser independence seems to be an illusion; more commonly, endless multiple-platform and multiple-browser testing and tuning are required. Unfortunately, technology that is owned by the industry widely is very difficult to affect, and there are no particularly clear indications that the needs of re-usable components in education will be met any time soon.

Using a browser as a container has one other serious downside. Browsers are *one-way media* (diSessa, 2000). That is, the mechanisms of constructing web pages (and, more critically for our purposes, interconnecting components) are reserved for relative experts. Wide-spread construction and modification of component-based systems by ''ordinary'' folks (including even technologically sophisticated teachers and designers who are not programmers) is nearly foreclosed. ESCOT early on hoped to make some configuration and connection possibilities accessible to less technologically sophisticated individuals, but the task of creating an appropriate container was judged beyond the means of the project.

An appropriate container, then, constitutes a critical dilemma for the field. Even if one is willing to use only expert-appropriate technology, cross-platform and cross-browser issues are daunting. If one is not willing to use expert-only technology, one is apparently faced with the much larger scaled prospect of designing and building an appropriate container from scratch. E-Slate and Boxer have taken on that challenge, but both suffer from treading a ''non-industry standard'' path: For example, it is difficult to break into the mainstream of computing, and it is difficult for one project to keep up a complex piece of software for an extended period of time.

Component Modifiability

Re-use is the lynchpin assumption behind component computing, but we feel it needs to be reconsidered. In short, we believe the image of a fixed component library out of which new educational products can be quickly

assembled (without significant modification) is more illusion than reality.[16] Instead, our contention is that component re-use depends on the capability to modify and adjust them extensively according to context. This is the second of our critical issues, and we call it *component modifiability.*

ESCOT produced the best documentation of re-use of any of the projects. But even here, in the end, we feel ''code re-use,'' rather than component re-use, best describes what was accomplished. (Code re-use doesn't presume that the units or boundaries of re-use are established in advance.) The distinction between code re-use and component re-use often comes down to the programmer/development team's anticipating the important variations of use for a component, and providing easy customizations (such as preferences and adjustable attributes) for them. In fact, despite initial hopes, ESCOT programmers wound up doing much more changing of components than they originally intended.[17]

It is possible that poor design, in some sense, lies behind limitations in strict component re-use. Certainly it is true that if designers can foresee re-use needs, relevant features can, in principle, be designed in as customizing options. To some extent, this is a sensible position in that extensive use by a community may well lead to components that are easily adjustable in ways that are important to the community. On the other hand, components with too many adjustments suffer from well-known difficulties such as inherent complexity and dependence on documentation, which is often lacking or difficult to interpret.

An elaborated form of component modifiability might be phrased as follows:

- Context dependence: Components that work adequately or optimally in a certain context are actually much more sensitive to the context than is generally expected.
- Diversity: Since educational materials development constantly innovates new contexts of use (new lessons, new subjects), we conjecture that context sensitivities and appropriate adjustments are, in principle, not anticipatable in many cases.

[16]Obviously ''quick assembly'' and ''significant modifiability'' need calibrating, which will be partially addressed in later discussion.

[17]These are somewhat delicate judgments, since rhetoric and commitment are often not well-captured in published documents. In addition, shifts in strategies are often reported only informally, or not in the context of prior expectations. Nonetheless, we feel we have adequate formal and informal documentation to substantiate this judgment.

Our experiences with Boxer components can be summarized as follows: *Every significant use of a component in a new context has required adjusting the behavior of the component*. Here, we content ourselves with producing two of examples that illustrate the general phenomena.[18] The examples are meant to argue: (a) Components are sensitive to the contexts in which they are used. (b) It is not possible and does not make sense to anticipate all such contexts in the construction of a component. (c) Modification via programming might be the best option, and often such is not difficult at all. The first two of these directly reflect elements of our elaborated version of component modifiability, above.

Figure 5 shows two components from an image processing toolset in Boxer. The astronomical image on the left is a graphical rendition of an array of numerical values representing light intensity in a CCD array (from the Hubble telescope). The graphing component at right was based on a generic graphing component that we used for years, with little modification, in the context of

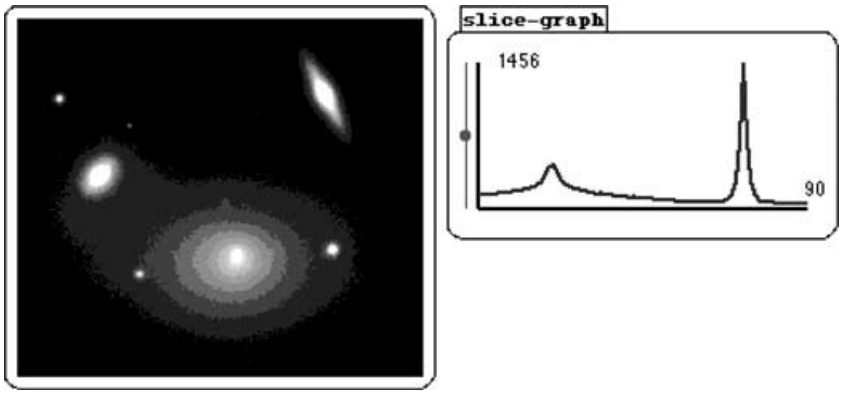

Fig. 5. Components in an image processing toolset.

[18]Our use of Boxer for examples does not mean that we think the phenomenon is not general. Instead, it reflects the difficulty in getting information about how much modification was actually done in order to re-use a component. Obviously, we are in a better position to know what modifications we made in components within our own project. Finally, Boxer has had a long life, and thus more opportunity to experience the need to change apparently stable components.

students' explorations of motion. In this configuration, the user drags out a straight-line segment on the image, and the graph shows the light intensities across this slice. For image processing, we needed to make at least the following modifications of the graphing component:

1. The physical size of the graph automatically adjusts to the length of the slice.
2. Only maximum values are shown on x and y axes, and no tick marks are displayed. (The graphs are used mostly for qualitative analysis.)
3. The vertical bar with ''knob,'' on the left of the graph, can be dragged across the graph, which then ''lights up'' the entire corresponding slice in the image, and places an ''χ'' at the position corresponding to the indicated x coordinate. Simultaneously, the precise y value is numerically indicated on the graph.
4. The graph keeps track of the image from which a particular slice comes, in case of multiple images.
5. The grapher is capable of producing a miniature thumbnail of itself (for documenting explorations). When one drags and drops a thumbnail onto the grapher, it responds by reinstating the graph that was present when the thumbnail was created, including the link that highlights the relevant slice.

While some of these are plausibly generic adjustments that one might like to have available (e.g., size and tick mark display), others seem quite idiosyncratic to our needs in this context. For example, the vertical control bar and its display properties, internally keeping track of the image from which a slice was drawn, and the response to graph-thumbnail drag and drop just do not seem plausibly like optional features of a general graphing component, unless that component were to become baroquely complex. We note in passing that we deliberately chose not to use wiring as a way of connecting graph to image. Instead, any image responds to a slice gesture by communicating its data automatically to the slice graph. No single connection protocol is ideal for all circumstances. With respect to the issue of how much effort modification takes, it took about an hour or two of programming to make the modifications listed above.

Figure 6 shows a collection of components from a ''plant growth'' modeling toolset, which contains a second modified graphing component. In the top row are output devices, left to right: a plant with three input parameters, corresponding to height, width, and depth of branching; a graph allowing three independent parameters to be graphed in different colors; a data

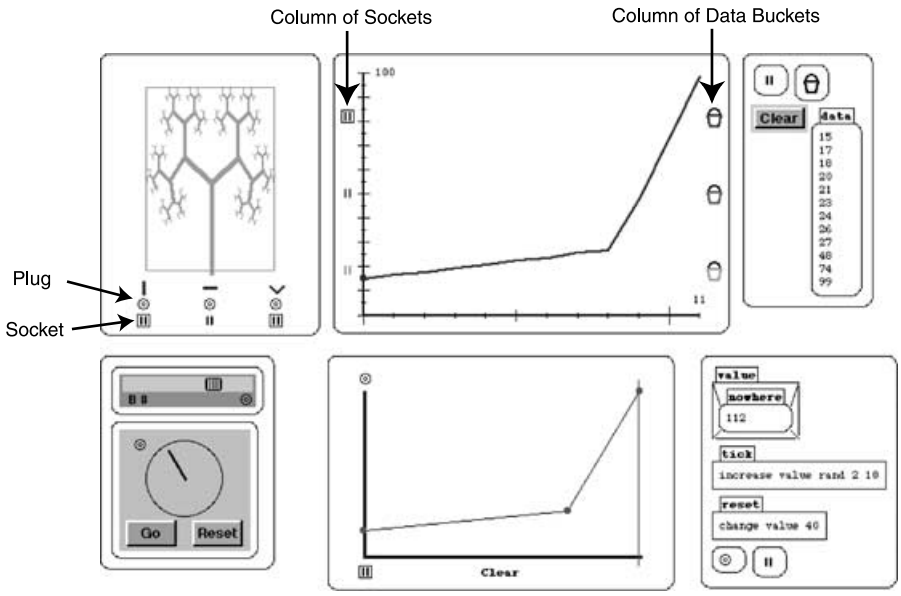

Fig. 6. Components in a ''plant growth'' modeling toolset. In the top row are ''output'' components, and the bottom row contains ''input'' components that control various parameters of the plant. Components are wired together by drag and drop gesture, from plugs (target icons) to sockets (a pair of vertical ''slots'').

display component that shows a sequence of numerical data points, as might also be displayed in the graph. The bottom row are controllers, left to right: a slider that can directly control, for example, the parameters of the plant; below the slider, a clock that can control other components that create a sequence of numerical output data; a graph serving as an input device to specify a growth pattern; a programmable controller, which produces an output according to any simple program – in this case, increment the current value by a random number between 2 and 10.

These components can be cut, copied and pasted to configure a model of plant growth. Typically, one would wire, plug to socket, the clock to one's choice of controllers (a graph or program – middle and right components, bottom row), and then the controller would be wired to the appropriate parameter of the plant. The plant's parameters could, in turn, be wired to a graph output device or numerical display of the data.

In this case, the modifications of our generic grapher included:

1. three input sockets for different-colored graphs (left-most column in the graph component);
2. three data buckets (right-most column in the graph component), from which one can ''pour'' data from each of the colored graphs into another component, such as the numerical data display;
3. internal state of the grapher to ''remember'' what components it is connected to, and methods to respond to new messages be sent from connected components.

In addition to reinforcing the implications of our first example, this one especially highlights mutual configuration that may be necessary with toolsets. In the case of this plant growth modeling toolset, we judged wiring to be an appropriate connection method. So we had to add plug-and-socket interfaces for each component. In addition, each component needs to respond appropriately to messages sent along connecting wires, and pass on appropriate messages to other components. Further, we wanted to have plugs and sockets visible on the components themselves, rather than available (only) in a separate wiring editor, since wiring is such an intimate aspect of this use of components; wiring is not ''designing the application,'' but is an expressive action for users. Once again, we argue that specializations such as these do not make sense as part of the options of generic components, and are best handled by modification via programming.

Examples, of course, do not prove that component modifiability is a critical issue. But they illustrate a broader trend we saw in our data, and show something about why and when it is critical. It is particularly important to monitor the issue because of fairly wide-spread unqualified rhetoric concerning the value and feasibility of component re-use. It is worth noting that these claims are fairly directly relevant to intentions concerning local adaptability. Easy configuration (not including serious component modifiability; see discussion in the introduction) may not be sufficient for genuine adaptability.

The need for component modifiability does not signal a death knell for component computing. However, as a community, component advocates may need to develop strategies to deal with the consequences. A list of compensating strategies will be presented later. In closing this section, we remind readers that our graphing component had stood essentially unmodified for years, as long as our work rested in the context of motion. When we turned to image processing and plant growth models, however, a more intimate connection between the

grapher and other components of the toolset were required. This illustrates what might be expected – that the need for component modifiability may not show up when components are used only for a few, similar purposes.

Who Gets to Do What?

Our third key issue really constitutes a complex net of both social and technological issues. Furthermore, that fact that the issue is relatively unaddressed in much of the component literature, we feel, reflects a basic techno-centric orientation of the component computing community. The issue therefore warrants more extended treatment.

A Cultural Gap

As a prelude, we describe a cultural gap that we discovered in analyzing our survey of important issues for component computing. Recall this was a Web-based survey. It involved only a small number of individuals (12), all of whom volunteered from a larger group of individuals we contacted, who were involved in component computing. Most of the surveyed individuals were from the projects discussed here, although no members of the Boxer group participated. No claims of representativeness or statistical significance are made. Still, some of the results are provocative.

On part of the survey, we asked respondents to assign a value on a scale of importance, from 1 to 5, for two lists of issues. The first list concerned the most valuable features of component computing, and the second list concerned the most important issues that need to be addressed to make further progress with components. The choices we offered were as follows:[19]

Best properties

- Cumulativity – Components avoid ''reinventing the wheel'' and provide a base for continued improvement and use.
- Adaptability – Components allow schools and teachers to adapt to their local circumstances.
- Social Factoring – Components allow teachers to do what they do best, and technologist to do what they do best.
- Quality – Components can support higher quality software.

[19]Free responses were also allowed. There were relatively few of these, and we do not analyze them here.

- Speed – Components allow quicker development of software.
- Cost – Components reduce the cost of software development.
- Accessibility – Components allow easy wide-spread access to software.
- Teacher Autonomy – Components allow teachers creative expression and independent empowerment.

Issues for the future

- Hospitality for Teachers – A collection of issues concerning involvement and empowering of teachers, including autonomy and teacher professionalism.
- Hospitality for Students – A collection of issues concerning involvement and empowerment of students, including ability of students to configure components themselves.
- Improved Technology – Better technological infrastructure for components.
- Authoring Models – Developing better ways to involve different kinds of people in the production of component-based software.
- Curriculum Match – Better matching of software to the curriculum.
- Curriculum Coverage and Access – Wider curriculum coverage and easier access to the software.
- Economic Models – Models that allow ''profit motive'' to work successfully with components.

Among our analyses, we looked for issues that engendered the most disagreement. In ''Best Properties,'' Teacher Autonomy had the greatest standard deviation. Among the ''Issues for the Future,'' three issues were outliers in standard deviation: teacher independence and autonomy, student ability to configure and adjust component software, and economics. With the exception of economics, it seemed that creative options, ''who gets to do what,'' involving teachers and students systematically engendered most disagreement. With this in mind, we looked at two groups, which were split about the median in ''teacher autonomy.''[20] We call the resulting groups ''Teacher-'' and ''Technology-oriented'' respondents.[21] Tables 1 and 2 are the rankings for our two sets of issues, by group. Ties are marked in brackets.

[20]In order to enhance contrast, we also omitted three subjects closest to the median.

[21]Although participants had the option of remaining anonymous, most chose to identify themselves. Interestingly, a couple of respondents who would likely be classified as technologists fell into the teacher-oriented group, and at least one teacher respondent fell into the technology-oriented group.

Table 1. Ranking of Importance of Various Properties of Component Computing, Based on Average Ratings on a 5 Point Scale. Ties Marked With Brackets. Highly Contrasting Rankings are Indicated by Arrows.

Components' Best Properties	
"Teacher-oriented" respondents	"Technology-oriented" respondents
[Quality	Speed
Cumulativity]	Cost
[Adaptability	Social Factoring
Social Factoring]	Cumulativity
[Teacher Autonomy	Adaptability
Accessibility	[Accessibility
Speed]	Quality]
Cost	Teacher Autonomy

Table 2. Rankings of Importance of Various Issues to the Future of Component Computing, Based on Average Ratings on a 5 Point Scale. Ties Marked With Brackets. Highly Contrasting Rankings are Marked With Arrows.

Issues for the Future	
"Teacher-oriented" respondents	"Technology-oriented" respondents
Hospitality for Teacher	[Hospitality for Teachers
Hospitality for Students	Economic Models]
Improved Technology	Improved Technology
[Curriculum Match	Authoring Models
Authoring Models]	Curriculum Match
Economic Models	Hospitality for Students
Curriculum Coverage and Access	Curriculum Coverage and Access

In terms of best properties, teacher-oriented respondents ranked cost and speed last and tied for next to last, while technology-oriented ones ranked them first and second, respectively. (See arrows in Table 1) Quality was tied for first in the teacher-oriented group, and tied for next to last in technology-oriented respondents. Teacher autonomy, the most disparately ranked item and the basis for the group split, showed up last for technologists, but only in the middle of the rankings for teacher-oriented respondents. This reflects the extremely low rating for teacher autonomy scored with technologists. We suspect the middle-of-the-pack rating for teacher-oriented respondents was

skewed low by the fact that, with the technology involved in most of these projects, teacher autonomy was seen as somewhat implausible, even if it was considered important.

With respect to issues for the future, there was firm agreement across the board that teachers are important and need to be involved. But, after that, the most striking feature is the reversal – from second to next-to-last, and from next-to-last to second – of economics and hospitality for students. (See arrows in Table 2) Especially regarding free-form responses of teacher-oriented respondents, it was clear to us that involvement and creative expression for students, in particular in configuring components, marks a strong disagreement.

To sum up, the most visible trend in this analysis is that some people consider the ability of students and teachers to control the technology on their own extremely important, while others do not see this as important at all. This cultural gap sets the stage for a wider discussion of our third critical issue for component computing – who gets to do what? Here, we focus mainly on teachers, why they might profit from control of the technology, and what that means about component technology.

Teachers and the Control of Technology

Let us start by reviewing the technological bases for the component projects studied in this work. The major relevant distinction is what controllable aspects of the technology are made available to teachers and students. Without argument, we assert (and we think most will stipulate) that Java programming is, in general, simply not accessible to teachers or students. With this in mind, a fairly straightforward hierarchy of possibilities is demonstrated by the projects. The EOE – for technological and historical reasons, rather than matters of ideology – does not allow teachers or students any direct access. ESCOT, while it hoped for some teacher and student adjustability, mainly in wiring components, finished with a technological base that was largely off-limits for teachers.[22] E-Slate aimed at a scalable teacher and student accessibility, including wiring and also programmability. Boxer's uniform architecture, inside and outside components, represents an extreme in the continuum in that, to the extent that teachers and students learn any programming, they can in principle configure components, open them up to modify them, or even construct new components from scratch.

[22]An exception is that the ''component generators'' (e.g., Sketchpad) could, in principle, be used by teachers, although teachers themselves reported almost no use of this possibility.

Let us examine why it might be useful for teachers to adjust component configurations.[23]

1. Local adaptability

Classroom practices vary greatly across individual teacher's styles, across school cultures (say ''open'' or ''regimented'' styles), in response to different students' needs, and so on. A software component or configuration of components that fits one teacher's style and pedagogical goals is unlikely to fully satisfy another teacher's needs and preferences. Even repeated uses of a component or set of components within the ''same'' context (i.e., the same teacher and students working on an extended curricular unit, or the same teacher in subsequent years) are likely to require modifications to the component's behavior and/or appearance in view of any attempt to follow student interest or level of competence, year-to-year variability of students, professional growth of teachers, and so on.

A counter argument to this claim for the need for adaptability is that textbooks are uniform, and teachers can and do adapt their own styles to textbooks. In response, although we cannot make a compelling argument here, one might contend that activity structures in the classroom are much more sensitive to software than to textbooks. Teachers can always have a complex, open discussion in a class, no matter what the textbook (or software). But open exploration is absolutely curtailed by regimented software.

What sort of modification and adjustment might teachers make? First, teachers might want to *change the specification of the task*, or of the *sequencing of tasks* for students to accomplish. In experiences reported to us by members of the ESCOT project, this might only require changes to the text accompanying the software, describing what students are to do. In recent work, we developed software involving image processing (see Fig. 5) in collaboration with a teacher (Friedman & diSessa, 1999). By and large, the development process involved jointly discussing activities that might be valuable for students. Then, graduate students (serving as secondary developers) configured components for those purposes. The

[23]Once again, there is a ''familiarity effect'' in the selection of examples in this section. Most of the examples are Boxer partly because we know Boxer best, and partly because only Boxer and E-Slate make teacher modification a priority. We have seen apparently impressive constructions by teachers in the E-Slate Project, but these have been too recent for our careful consideration. Consult recent publications from E-Slate.

final pass was reserved for the teacher, who sometimes resequenced ''things to do,'' but almost always extensively edited the text of instructions and explanations to his taste and to his understanding of students' level of comprehension.

Beyond this simple, but potentially important kind of customization, any reasonably open kind of software creates possibilities for task invention. Tool-like software (for example, a simulation or an image processing toolset) creates the possibility for teachers to invent new tasks, which often can benefit from customization of the software. For example, in the Boxer profile (diSessa, 2001b), we report how a teacher was inspired to invent a new task concerning student-constructed patterns of colored cells in a numbered 10×10 array. The task required a modification of the then-available tool, which was accomplished with the help of a secondary developer.

Another example of task invention – or, more precisely, substantial task modification – emerged in an earlier Boxer project that involved developing a year-long curriculum on motion for sixth graders (diSessa, 1995). One of the culminating projects for students was to take stroboscopic image of a thrown ball, and make a simulated ball travel so as to exactly match the sequence of images. Literally the day before the task, the teacher told us that she felt the way we had conceived of the task, in which students merely adjusted a single acceleration vector, hid (by assuming) the essential fact – that acceleration is constant. On-the-spot, the task software was reconfigured to allow students to adjust acceleration for each stroboscopic image.[24] See Figure 7.

The extent to which activity design and customization requires, or could benefit from, small or larger changes in the underlying software is an issue with no clear consensus in the field. Our impression is, however, that the value of software tailoring, even of broadly useful tools, is much underappreciated. The rise in prominence of scaffolding resources (task specific resources) in educational software does hint that more recognition is emerging in the educational software development community.

Can teachers actually develop components, in addition to adjusting text and configurations? This is a complexly situated issue that depends on

[24]While we do not show all aspects of the change, the main part of it was merely to paste in five copies of the vector component. It took about two minutes to create the working new task configuration.

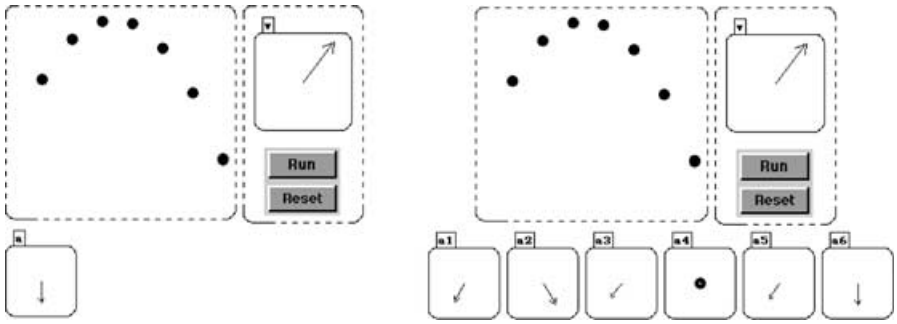

Fig. 7. Task: Match the stroboscopic images of a tossed ball by adjusting acceleration and initial velocity. Left: Original configuration with one acceleration vector, assumes constant acceleration. Right: Modified task – acceleration may be adjusted individually for each ''hop''; Constant acceleration is the ''surprise'' result.

teacher preparation, expectations of the educational community, and the complexity and easy of use of relevant software. We can state unequivocally, however, that several good examples exist. See Sherin (2002). Furthermore, the argument above – that entering into the interior of components is a much more important and likely-to-be-needed capability than might be expected – also suggests that shutting teachers out of components' insides might be too restrictive.

2. Professional Development

Beyond local adaptation, there are potentially important issues concerning professional development and assumed professional responsibilities of teachers. To begin, consider the contributions of teachers in co-development. First, allowing teachers to contribute to technology (at their own level of expertise) may well curb some of the sharp social boundaries that can disrupt good collaboration, for example, in integration teams. Furthermore, a reasonable degree of teacher technological competence, obviously, is an assumption of the LaDDER model. Also within the LaDDER model, allowing less technologically expert people to work directly with components is a requirement at the secondary developer level. Considering the needs of secondary developers emphasizes the fact that opening up technology for teachers also expands possibilities for others competent in education, but less so with technology. Co-development can become an apprenticeship process that leads to a long-term innovative spirit for teachers.

It is not a trivial matter that, once separated from team contexts and without more open architectures, teachers will be closed out of further pursuits. See the case study of ''Sarah'' in Parnafes and diSessa (2001).

There is a subtle, but we think important effect concerning the ability of teachers to contribute to technology design, independent of their actually constructing part of the technology product. Technology-naïve teachers have little sense for interactivity, and for the limits of what is possible with technology. Therefore, their suggestions for improvement may be, for example, overly optimistic about what can be done with modest resources, or do not take advantage of what is actually easy to do. There are other routes to a reasonable level of technological literacy, but direct involvement in technology construction can be an excellent means to that end.

Summing Up

The direct involvement of teachers (and others who are conventionally considered to have little or nothing to do with innovations in technology) with component technology is a complex and contentious issue. But there are three reasons it is particularly important to consider opening technology to teachers, beyond the suggestive examples and brief arguments we have given.

1. Teacher empowerment is a phenomenon whose importance has been extensively documented. McLaughlin and Talbert (2001), for example, identify local teacher communities who work among themselves as a critical locus of reform. If such groups are to harness component technology's power for their own efforts, the most direct and probably most sustainable route is to put the technology under their control.
2. The fact that teacher empowerment runs against the grain of many current trends and intuitions (e.g., massive accountability testing, rigid state and local standards) means that more, rather than less, resources may be necessary to counter such trends. That teachers now may be less than optimally prepared to take on the challenge and promise of component technology for themselves may be one of the best reasons to foster such a trend.
3. The very fact that empowering teachers with open technology seems least important and least feasible to technologists who dominate public discourse on component technologies (compared to judgments of teachers and other educators – recall our survey disparities) suggests that more balanced discourse and careful attention to possibilities are appropriate.

Our three critical issues, exposed in the last three subsections, constitute our best judgments for the most important considerations for the advancement of component computing in education. In the following section, we return to ''core claims'' for a broader review of the status of component computing. The cost of this breadth, as we anticipated early on, is that some of these issues are not well addressed by the data we have.

REVIEWING COMPONENT COMPUTING'S CORE CLAIMS

In spite of a manifest variety in component projects' philosophies and organizations, a common set of critical advantages has been attributed to component technologies. Over the years, and given local circumstances, individual projects have evolved to focus on particular aspects of the claimed advantages of components. But an overall guiding vision remains as a strong motivator for the pursuit of educational component computing. In what follows, we briefly assess – as much as our data allow – how much the field has progressed towards realizing that vision. Most sections briefly review the logic of advocacy and then follow with a brief assessment.

Re-Use

Rhetoric

In contrast to large, monolithic applications, component-based software is made up of smaller functional elements (i.e., components), which can be individually re-used across a range of applications. A graphing component, for example, can be used as part of an algebra lesson, in conjunction with a simulation in the biology of species population, or in teaching the physics of motion.

From the inception of the educational computing movement, component re-use has been perhaps the most central attribute justifying continued work on the technology. In terms of educational software production, a ''building block'' approach to development, it is said, will lead to improved quality, with concomitant gains in speed of development.

Assessment

Our observations, including experiences with component-style development within our group, convince us that there is, indeed, real value in

component re-use. There is no question this core intuition often plays out positively. On the other hand, there are complexities and hidden issues that cloud the extent of savings that may be achieved, and also cloud the conditions under which advantages may accrue. In this report we highlighted the problematic nature of assumptions of re-use without modification. Beyond our own cautionary stance, a component project at Brown University (Spalter, 2002) has been most vocal about how difficult it appears to be in practice to reap the ''obvious'' rewards of components. Spalter (2002) also investigates why this problem appears to be especially acute in education.

Although skewed pessimistically by a number of factors,[25] re-usability has been surprisingly muted in our focal projects. In the best cases, component re-use has been realized essentially in the form of code re-use. Only a small subset of components implemented within specific projects have been re-used according to the ''building block'' paradigm, as far as we have been able to ascertain. Truly large-scale reuse – say in dozens of significant new uses across multiple instructional domains – simply has not happened.

The best documented re-use among our profiled projects is the ESCOT project. Yet, ESCOT's experience also suggests problems with assuming limited modifiability, along with some important details and nuances. ''Widgets'' such as slider controls, buttons, and other interface elements reportedly have seen multiple, as-is uses. However, re-use of more complex components has been a different matter. ESCOT's developers eventually settled into a development mode in which component code was fairly freely modified and specialized to particular contexts. While code re-use, per se, is reported to be substantial, component re-use in a strict sense is more limited. Code re-use appears to have been facilitated by a confluence of factors. First, ESCOT component developers are masters of the programming language (Java) in which components are implemented. Second, often those programmers work on their own code, making code interpretation and modification easier. Third, ESCOT components have been used inside a

[25]For example, the EOE's early work did not emphasize component re-use. In addition, their distributed social organization was not particularly conducive to the emergence of common components. By the time of our profile, the E-Slate project had not reached a state of maturity of its architecture and components that serious re-use was tested. Again, the reader is referred to more recent work (e.g., Kynigos et al., 2002).

restricted community that strongly supports and foster code sharing.[26] Fourth, the ESCOT project developed a technique of using somewhat specialized, but still quite general ''programming'' systems (such as the Geometer's Sketchpad) to develop symantically specific components to anchor particular applications. This last point suggests to us that establishing the core semantics of an educational application may require more flexibility – perhaps enough to implicate ''re-programmability'' – compared to more fixed-purpose elements, such as interface elements.

Rapid Development

Rhetoric

Re-usable components can, in principle, drastically reduce the development time of educational software because programmers can readily use existing components stored in public/community libraries. By deploying a component as-is, development can also proceed at a higher semantic level. Similar advantages hold in case a pre-fabricated component provides many of the sought after functions, yet needs some sort of adjustment. To follow on the example of the graphing component, one need not implement a graphing tool from scratch. Rather, one can simply borrow an existing graphing component and extend or change its functionality. In either case, programmers start from an existing code base, thus saving time and resources.

Assessment

Again, the best-documented examples of rapid development came from the ESCOT project (Roschelle & DiGiano, this issue), although the Boxer Project, also, has experienced many cases of very rapid development based on existing components. (See, e.g., the discussion of image processing in Parnafes & diSessa, 2001.) More broadly, we do feel that many developments proceeded much more rapidly across projects studied than would have been the case without component technology.

[26]We got some confirmation that the ''local community of developers'' effect in re-use is important. An expert programmer who knows ESCOT and Java well reported to us that, in failing to understand how an ESCOT component worked (so as to adjust it slightly), he resorted to reimplementing it, rather than re-using it. Our interpretation is that this programmer's status as outsider to the ESCOT community, not his general expertise, undermined re-use.

On the other hand, there are caveats and complexities. Because rapid development depends on re-use, some of the same qualifications that we developed concerning re-use and modifiability are relevant. In short, rigid reuse is much less likely than many hope. Yet, accepting modifiability may not have a huge impact on rapid development. Providing that modification is easy (such as when it is facilitated by being done in a small, expert community, with tight, organized coding practices), development need not be much slowed, even if rigid re-use is limited.

A more serious threat to rapid development has to do with the big picture of designing educational software. The educational aspects of design, we feel, generally take much longer than current development practice assumes. Several substantial cycles of use and refinement may be extremely salutary, especially for larger elements of software or for components and toolsets themselves. So our favored scenario for future development of materials is not necessarily that development itself becomes shorter. Instead, *the technical aspects of development* can become a smaller and smaller part of the full development process. This is consonant with our description of LaDDER model collaborations, where educators can spend the time they need to develop and refine software and learning activities iteratively, without the coordination problems of tight coupling with primary developers.

Economics

Assessment

We have little data on the economics of component computing to report, and are hesitant to speculate. The ESCOT project is the only one to document specific claims of economic advantage (Roschelle & DiGiano, this issue), and their conclusions are highly positive. However, even if the economics of coding are advantageous, the ''bigger picture'' considerations above – that the biggest component of development may be educational design, not ''coding'' – suggest that the overall economic advantage of components may be more muted than advocates hope for.

The real economic issues begin mainly in terms of sustainable commercial models. None of our projects have seriously broken into this regime, although E-Slate and Boxer are making efforts in this direction. A central tension we can anticipate, however, is that components would seem to work best if they are widely and inexpensively available, and are documented and modifiable.

These qualities obviously run counter to the proprietary ownership that is typical in profit-making software.

Access

Rhetoric

Although a variety of distribution channels may be used, component software is often tied strongly to Internet strategies for distribution. In a nutshell, the Web makes libraries easily feasible, and small components can be quickly downloaded from the Web. While component production has been carried out under different social configurations, the existence of a central repository of components has been an idea common to the majority of projects. This central repository would serve as a clearinghouse for components, addressing a number of issues of economic nature (see below).

Assessment

The EOE is the only project that we judge has achieved critical mass in terms of the number of components it makes available and in implementing a central, Web-based repository with worldwide access. In preparing our profile of the EOE, for example, we studied the practices of a middle and high school in Australia that seemed to have succeeded in integrating substantial EOE use into their regular classroom practices.

Empirical results across all projects on distribution are thin and concentrated on distribution for users, as opposed to distribution of components for developers' purposes.[27] Even the EOE seems to have had limited effect via its central repository. Our Australian ''informant'' school, for example, admitted their use of EOE was more peripheral than their original claims seemed to imply. Certainly we can say that component software has yet to affect the core practices of instruction in a substantial number of schools. Of course, technology was a limiting issue for EOE, but other issues are important. Fetching a component from a repository is only a small part of its effective use. For instance, given a large enough set of components, how does one find that which best fits one's needs? What kinds of pedagogical guidance exist to help teachers use components/component-

[27]The ESCOT project did distribute its products via the Web. However, this was a very particular social context – products of component development fed into a well-organized social process of Problems of the Week. It did not approximate a library model of distribution.

constructions well?[28] How can educationally sophisticated, but technologically non-expert people really participate effectively in a component- (i.e., technology-) oriented community? To oversimplify the data, while the EOE intended to develop an educationally adapted model, it seems to have failed in that regard, and thus did not supply sufficiently adapted applications nor the cultural support needed for valid, wide-spread educational use.[29] There are further issues that are not specifically educationally-related. How does one manage redundancy and quality in a public library? See Azevedo (2001) and Spalter (2002) for more extensive treatment of some of these issues.

In a nutshell, while Internet distribution seems like a sound idea in principle, we found little data and more than a few in-principle worries about how this may work in practice. The fact that there are many web sites selling commercial components (e.g., www.componentsource.com) suggests that there may be specifically educational considerations at issue (e.g., cultural support for new instructional practices, and inherent diversity of educational products).

Local Adaptability

Rhetoric

We found a tension in rhetoric concerning local adaptability. On the one hand, component projects often seem keen to acknowledge that contexts, teaching practices, and pedagogical styles may vary across classrooms and school sites. Such variations suggest the need for mechanisms that allow local customization of individual components or of composite constructions. On the other hand, there is a clear refrain concerning limited capability and willingness of teachers, by themselves, to design, redesign, or do any implementation (e.g., EOE, undated; Roschelle, Pea, et al., 1999, p. 3). E-Slate and Boxer are exceptional in this group, and in the broader terrain of educational computing, in anticipating and designing for significant adaptability for a broad population of educators.

Among our profiled projects, three levels of mechanisms are advocated for adaptability: (1) tailoring of presentation and behavior, such as editing text and adjusting preferences (Roschelle, Pea, et al., 1999, p. 3); (2) authoring and

[28]Our Australian informant school developed a novel social structure to deal with the difficulty of finding relevant components. Students were assigned the task to find and justify the educational use of components, from which teachers selected components to be integrated more widely into instruction.

[29]Again, newer work of the EOE, which we do not report, may address many of these issues.

editing through simple wiring (Roschelle et al., 1999, and the E-Slate Project, generally); and (3) programming. Programming splits into two levels in consideration of whether the internals of most components are open to non-experts (Boxer) or not (E-Slate).

Assessment

By and large, locally adaptable technology using components has not been realized enough to test the concept seriously. The EOE did not have technology conducive to adaptation, especially adaptation by less technological sophisticated people such as teachers, and neither did ESCOT seek to develop such technology.[30] E-Slate's user-accessible wiring and programming (scripting) were not used in any local development/adaptation modes until after our study of component projects began to wind down.

Boxer has an open technological base (rich container model, one-level system) that is conducive to adaptability. Our emphasis on student creativity has led to many examples of students' adapting pieces or wholes of Boxer materials written by others. See, for example, examples of student work in chapters 3 and 8 of diSessa (2000), and Azevedo (in press). Concerning teachers, we have documented a few teachers carrying out significant adaptations (diSessa, 2001b; see also the examples in the ''Teachers and The Control of Technology'' section, above). Most of these have been in the context of a supportive collaboration, such as LaDDER. Furthermore, a few exceptional teachers have relentlessly pursued innovation and adaptation, largely autonomously (e.g., Picciotto, 1997) using Boxer. These examples are existence proofs that some teachers, under some circumstances can productively adapt and innovate. They motivated us to point out the possibilities and importance of local adaptation earlier in this article. But more work needs to be done to convince a generally skeptical educational technology community, and to show the way for those who feel local adaptation is important.

It is important to note that adaptability depends critically on the cultural assumptions about the role of non-experts in adjusting software to local circumstances, in addition to how well component architectures support the initiative of people who are not technology experts. Technology provides a baseline, but whether teachers feel their job can productively include

[30]In addition, the context of use for ESCOT constructions was as occasional enrichment activities (see Parnafes & diSessa, 2002), which is not conducive for teachers to think about or to be prepared to adapt the materials.

adaptation (or creation), whether they are adequately supported in learning how to do this, and whether they are sufficiently rewarded for their efforts are indispensable concerns. Future work on adaptation will need to attend to the social and cultural side of the equation. The LaDDER model is important in this regard as at least one example of how cultural and technical support may be provided to teachers on an extended basis.

New Social Structures of Production and Consumption

One of the more diverse, interesting and relatively unanticipated (e.g., in ''standard component rhetoric'') aspects of component computing that we sought to open up in this review is the many ways that groups of people with different affiliations and expertise can collaborate in the production of software. We exposed a complex set of tradeoffs for the various models of collaboration used in our profiled projects. What all these models share is the attempt to build better software by an appropriate melding of expertise from different communities.

The core problem to be managed by each of these models is that of two (or more) cultures coming together to serve a higher goal, the production and distribution of excellent educational software, which neither community by itself is likely to accomplish well. The member-sustained community model is optimistic (perhaps over-optimistic) that enlightened self-interest, along with the modest coordinating infrastructure of an Internet library and exchange center, is up to the task of bridging cultural differences. The two-legged model localizes the coordination problem at a somewhat macro-level, coordinating two teams, which in many ways operate autonomously. In contrast, cultural coordination may become more individualized and personalized in the relations of members of integration teams. The LaDDER model aims to manage a ''soft interaction'' of the two communities, buffered by intervening levels, such as secondary developers, and by technology that is aimed at increased autonomy for educators.

There are no easy, simple conclusions regarding the complexities of cultural coordination. We found these complexities hiding in some sharp differences of opinion in our Web survey and in reports of participants in each of the projects that we profiled. Certainly we are not in a position to say that one of these models is best, nor that others could not be substantially improved by innovative management of their ''intrinsic'' properties. In fact, if there is one generalization in which we have most confidence, it is that the local, innovative management of the difficulties of any of these models may well be

as or more important than their intrinsic tendencies. The ''black magic'' of cultural coordination in this management is a critically important topic, about which we have said little here.

THREE CENTRAL ISSUES, REVISITED

As a way of synthesizing a very complex exploration, we return to our three central issues and try to set them in the larger context. Aside from the idea of a model as a partial specification, and the characteristics of and trade-offs inherent in particular models (reviewed briefly in the section above), these three issues are the best, most compact ''take home'' results of our study.

In what senses are our three issues ''most important''? First, these are issues, we believe, that affect the very choice of avenues to pursue in realizing the promise of component computing, not so much ''how we proceed'' once we have selected a direction. Relatively common rhetoric in the movement hides a complex and diverse family of choices, including highly contrasting social models of production and consumption, and diverse underlying technical architectures, as illustrated by our profiled projects. Thus, our work should be more relevant to future component projects than those who have already chosen a path and need more detailed help.

Secondarily, we have chosen to highlight issues that are somewhat out of the limelight. Thus, while insiders know the problems related to browsers, new-comers and outsiders to component computing tend to assume a browser is obviously the right way to proceed. Similarly, the core rhetoric of re-use tends to obscure limits of strict re-usability and the need for strategies (e.g., modifiability) to enhance re-use. Finally, the technologist dominated component movement tends to ignore critical social issues that we have tried to highlight in ''who gets to do what?'' We felt we were very lucky to have projects to study that considered social organization important, and troubled to make it visible in their reports.

Here, then, is a brief reprise of our three core issues:

1. **The problematic container**
   We discussed what may appear to be local problems with available container environments and industry support for educationally-adapted component construction, in particular, in web browsers. Education has little leverage in turning browsers into a true two-way medium, allowing, for example, easy reconfiguration by non-experts. On the other hand,

building proprietary containers, like E-Slate and Boxer, has its own implausibilities in achieving the kind of standardization and industry-wide support that would be optimal.

2. Modifiability of components

We raised the possibility that components, to be broadly useful, need to be much more adjustable and flexible than most people believe. Indeed, our own experiences surprised us with respect to the importance of modifiability, which we argued might arise on an in-principle basis either because we can never anticipate well the specific needs of multiple contexts, or because designing components with adjustable preferences for many contextual uses might lead to components as complex and baroque as the large-scale applications that components are meant to replace.

The implication of modifiability go to the heart of component work since re-usability is, arguably, the core principle of the movement. The image of permanent libraries of stable components is put at risk. And yet, there are multiple strategies that can adapt to this ''fact,'' should it prove robust.

- Opening the internal architecture of components, as components open the architecture of applications: E-Slate is exploring hierarchical component systems, and Boxer already has a uniform system across levels that allows components as elements of components, and modification-via-programming of components on the same basis as they are combined into usable systems.
- Software developers can write for modification in re-use, including architectures and documentation aimed at that much more systematically than they do now.
- Libraries might be hierarchical, with generic components forming a certain core, about which satellites (''toolsets''), adapted to particular areas, are built.
- Social configurations of development might be encouraged that better support long-term development across multiple contexts, and adaptation to specific needs.

3. Who gets to do what?

The biggest cultural rift we perceived within the component movement – as revealed to us from our studies of four component-based projects and supplementary studies – concerns the technological empowerment of teachers, students, and other less technologically adept participants in the educational enterprise. On the one hand, some people view teacher

participation as critical, not just as commentators and critics, but, to the extent that it is sensible, teachers should be technology users in the full sense of being able to modify, adapt, and even create software. Indeed, components enter this line of thinking precisely by adding domain specificity and a lot of pre-existing functionality to the resources one has going *into* design.

On the opposite side, real teacher involvement with technology may seem to subvert the advantages of pure social factoring. Teachers, generally, are not competent to program, or, possibly, even wire components together productively. They are not likely to want to, let alone be capable of producing or adapting software.

Part of the reason the picture is unclear is that an emerging professionalism of teachers with respect to technology simply is not solidly in place. No one really knows what will happen in terms of expectations, desires and possibilities among the teacher community. The critical point, however, is that architectures that do not at least make room for the possibility of less-technical contributors absolutely foreclose the possibility before we have a chance to find out how far such empowerment may proceed.

Final Words

Component software is an exciting contemporary movement in software development, especially in educational software. Overall, we remain quite positive, despite the fact that rhetoric frequently outstrips reality, or it ignores issues that need attention. One of the most positive features of component computing is that it can, in many ways, democratize aspects of software development, leading to more educationally adapted software and, in the view of some, an increased autonomy and professionalism for teachers and other educators with less technical competence. Despite this promise, the component movement is still dominated by technologists, even if others with more educational expertise are being invited to the table.

One of the most evident things in our analysis is that there are many, many ways that component computing may be organized. We hope we have laid out a map, if partial and rough, so that others can proceed a little more expeditiously in exploration. We hope we have laid out a range of possibilities in mix and match models, a range of issues that deserve monitoring and further study, and a focus on some problems and impediments that may be removed with due diligence.

ACKNOWLEDGMENTS

We wish to acknowledge the openness and collaborative spirit shown to us by the ESCOT, E-Slate and EOE projects and their members. Without their cooperation, our empirical work would have been greatly diminished. Needless to say, the judgments and conclusions offered here do not necessarily reflect any of their views. We are also grateful to reviewers, who drew our attention to some errors, to problems in the organization of the report, and to additional issues that needed attention.

REFERENCES

Azevedo, F. (2001). *Educational Object Economy Profile*. Web-posted technical report. University of California, Berkeley: The Boxer Project. ftp://soe.berkeley.edu/pub/boxer/Distribution/EOE_Profile.pdf

Azevedo, F. (in press). Personal excursions: Investigating the structure and dynamics of student engagement. *International Journal of Computers for Mathematical Learning*.

Bijker, W.E., & Law, J. (2000). General introduction. In W.E. Bijker & J. Law (Eds.), *Shaping Technology/Building Society: Studies in sociotechnical change* (pp. 1–14). Cambridge, MA: MIT Press.

Clements, P., & Northrop, L. (2002). *Software product lines: Practices and patterns*. Boston, MA: Addison-Wesley.

diSessa, A.A. (1995). The many faces of a computational medium. In A. diSessa, C. Hoyles, R. Noss, & L. Edwards (Eds.), *Computers and exploratory learning* (pp. 337–359). Berlin: Springer-Verlag.

diSessa, A.A. (2000). *Changing minds: Computers, learning, and literacy.* Cambridge, MA: MIT Press.

diSessa, A.A. (2001a). Overview of component project profiles. Web-posted technical report. University of California, Berkeley: The Boxer Project. ftp://soe.berkeley.edu/pub/boxer/Distribution/Overview_of_Profiles.pdf

diSessa, A.A. (2001b). *Boxer profile: Component computing within a computational medium*. Web-posted technical report. University of California, Berkeley: The Boxer Project. ftp://soe.berkeley.edu/pub/boxer/Distribution/Boxer_Profile.pdf

diSessa, A.A. (in press). Meta-representation: Native competence and targets for instruction. *Cognition and Instruction*.

EOE (undated). *The EOE Info Pages*. http://www.eoe.org/foundation/info.htm.

Friedman, J., & diSessa, A.A. (1999). What should students know about technology? The case of scientific visualization. *International Journal of Technology and Science Education*, *9*(3), 175–196.

Hoyles, C., Noss, R., & Sutherland, R. (1991). *Final report of the microworlds project: 1986–1989*. London: Institute of Education, University of London.

Kynigos, C. (2002). Generating cultures for mathematical microworld development in a multi-organisational context. *Journal of Educational Computing Research*, *1–2*, 183–209.

Kynigos, C., & Friedman, J. (2001). *E-Slate Profile*. Web-posted technical report. University of California, Berkeley: The Boxer Project. ftp://soe.berkeley.edu/pub/boxer/Distribution/E-Slate_Profile.pdf

Kynigos, C., Koutlis, M., & Hadzilakos, T. (1997). Mathematics with component-oriented exploratory software. *Journal for Computers and Mathematical Learning*, *2*, 229–250.

Kynigos, C., Trouki, E., & Yiannoutsou, N. (2002). Generating communities of practice for educational innovation: Experience from an institutionally distributed integrated authoring community. *Proceedings of the 3rd international conference on communication and information technologies in education* (pp. 191–202). Rhodes, Greece: University of the Aegean (Kataniotis).

Kynigos, C., & Yiannoutsou, N. (2002). Seven year olds negotiating spatial concepts and representations to construct a map. *Proceedings of the 26th psychology of mathematics education conference* (Vol. 3, pp. 177–184). Norwitch, England: University of East Anglia.

McLaughlin, M.W., & Talbert, J.E. (2001). *Professional communities and the work of high school teaching*. Chicago: University of Chicago Press.

Papert, S. (1980). *Mindstorms: Children, computers and powerful ideas*. NY: Basic Books.

Parnafes, O., & diSessa, A. (2001). *ESCOT Profile*. Web-posted technical report. University of California, Berkeley: The Boxer Project. ftp://soe.berkeley.edu/pub/boxer/Distribution/ESCOT_Profile.pdf

Picciotto, H. (1997). *The turtle in the age of the mouse: Why I still teach programming*. http://www.picciotto.org/math-ed/t-and-m/turtle-and-mouse.html.

Roschelle, J., DiGiano, C., Koutlis, M., Repenning, A., Phillips, J., Jackiw, N., & Suthers, D. (1999). Developing educational software components. *IEEE Computer*, *32*(9), 50–58.

Roschelle, J., Pea, R., DiGiano, C., & Kaput, J. (1999). Educational software components of tomorrow. In M*/SET 99 Proceedings* [CD ROM], Charlottesville, VA: American Association for Computers in Education. Available at http://www.escot.org/escot/External/MSET_ESCOT.html.

Sherin, B. (2001). A comparison of programming and algebraic notation as expressive languages for physics. *International Journal of Computers for Mathematics Learning*, *6*(1), 1–61.

Sherin, B. (2002). Representing geometric constructions as programs. *International Journal of Computers for Mathematics Learning*, *7*(1), 101–115.

Spalter, A. (2002). Problems with using components in educational software. In *Proceedings of ACM SIGGRAPH 2002*.

Wenger, E. (1998). *Communities of practice: Learning, meaning, and identity.* Cambridge, England: Cambridge University Press.

APPENDIX A: OTHER ISSUES

We used two primary principles to generate issues in component computing: (1) core rhetoric of the community; (2) a short set of, in our judgment, most important issues. These principles obviously produce an incomplete list. In this complex area, we felt it better to concentrate, rather than list – much less consider – all the issues we or others might consider important. This appendix mainly marks that incompleteness.

Below are a few of the other issues most frequently mentioned by commentators on this article and by participants in our work. We briefly extrapolate the discussion contained in the text concerning these directions.

1. Component grain-size: How encompassing or minimalist should components be? There are innumerable trade-offs that affect this decision. In particular, smaller components are farther from usable educational materials, yet when components become too big, they often become too specialized and complex, defeating the main purposes of components. Our take is that a fluid technology is best, which allows nesting and re-drawing of levels. Beyond that, this seems certainly to be an enduring core computer science issue, not new to components, nor settled by their use.
2. The challenge of designing truly re-usable components: Many reported to us how difficult it was to produce truly re-usable components. Our discussion of modifiability suggests one direction of lessening constraints on ''truly re-usable,'' and at the same time introduces a classic set of old and new issues concerning re-usable elements of computing (e.g., the vast literature on object-oriented computing). One of the ideas we introduced, low intensity but extended duration development (e.g., LaDDER), can help find the right form for components, as well as the right size (above).
3. Indexing and library management: How do we make it possible for users or developers to find relevant and high-quality components? In our judgment, there are prior issues, such as to what extent large libraries can really work (e.g., in providing cultural support for educational use, not just components), or whether more local communities are more likely to build their own stock of components, lessening some of these problems.