

Reusability and Interoperability of Tools for Mathematics Learning: Lessons from the ESCOT Project

Jeremy Roschelle, Chris DiGiano, Mark Chung
SRI International, 333 Ravenswood Ave, Menlo Park CA 94025 USA

Contact: Jeremy.Roschelle@sri.com

Phone: (650) 859-3049 Fax: (650) 859-4605

Alexander Repenning
University of Colorado, Boulder

Seth Tager
Arrowhead Software

Marissa Treinen
Stanford University

Abstract

Increasing the interoperability and reuse of mathematical tools is important within the overall process of K-12 mathematics education reform, as linked tools can enable better learning, and reusable tools make the most of limited developer resources. In this paper we describe our experience in creating interactive mathematics activities that incorporate interoperating tools from multiple sources. We highlight 3 key findings: (1) the importance of identifying a small set of powerful, generative components; (2) the value of simple interoperability conventions that can be adopted easily by developers; and (3) the need to extend these techniques to accommodate more dynamic behavior, more advanced data types, and the persistence of component configurations.

Introduction

Tools such as calculators, spreadsheets, and graphing calculators have a potentially profound role in enabling many more students to learn complex mathematical ideas (Kaput, 1992). Tools can reduce some of the cognitive complexity of calculating with mathematical algorithms, allowing students to focus on conceptual understanding. Tools can present mathematical ideas in visual and interactive modes that are better tuned to students' learning capabilities. Tools can make mathematical tasks more authentic; for example, no business person would calculate a what-if situation without a spreadsheet. And in some cases, such as the mathematics of chaos and complexity, it is hard to work with mathematical ideas in the absence of tools.

The most useful tools for education, however, have finely tuned interfaces and fairly complex internal representations and computations, and thus are fairly expensive to construct. For higher education, some companies have produced large, complex packages that suit undergraduates and professionals (such as Maple, MatLab and Mathematica). These tools don't scale down well to the K-12 setting, as K-12 tools must support teacher-authors with a range of technical skills along a

design-use continuum (Repenning et al., 1999). Many educational projects, especially in K-12 learning, have small scale funding, and cannot on their own construct a full suite of appropriate mathematical tools for K-12 users on their own (Roschelle & Kaput, 1996). Reuse is a natural means to aggregate the efforts of many small-scale projects, where each provides part of an overall suite of mathematical capabilities.

Reuse alone, however, is insufficient— interoperability is necessary. In many pedagogically worthwhile activities, multiple tools are used together, requiring interoperability both in terms of presentation in a shared space (a screen or web page), storage (a unified "state of the student's work") and in terms of data sharing (the same data appearing in a spreadsheet and a graph). With regards to data sharing, many research efforts have highlighted the usefulness of "linked multiple representations" where any of the representations may be operated on, with changes propagating dynamically to other linked views (e.g. Kozma, et al, 1996). For example, a student could scale a function either by tweaking the form of its equation or by dragging its graph with an appropriate tool. To be useful in K-12, reusable components must be interoperable components (see Roschelle, DiGiano, et al., 1999)

The Educational Software Components of Tomorrow Project (<http://www.escot.org>), funded by the U.S. National Science Foundation, has created a testbed of multiple tool developers (as well as teachers and application service providers) to work towards reusable, interoperable components for middle school mathematics. Our testbed includes universities, nonprofits, and commercial K-12 vendors, all who are developing Java-based tools for mathematics learning. Over the past year and half, testbed members have produced a series of web-delivered "Problems of the Week," each of which reuses components from multiple sources to present students with a challenging problem to solve. An example "ePOW"—as we have come to call them—appears in Figure 1.

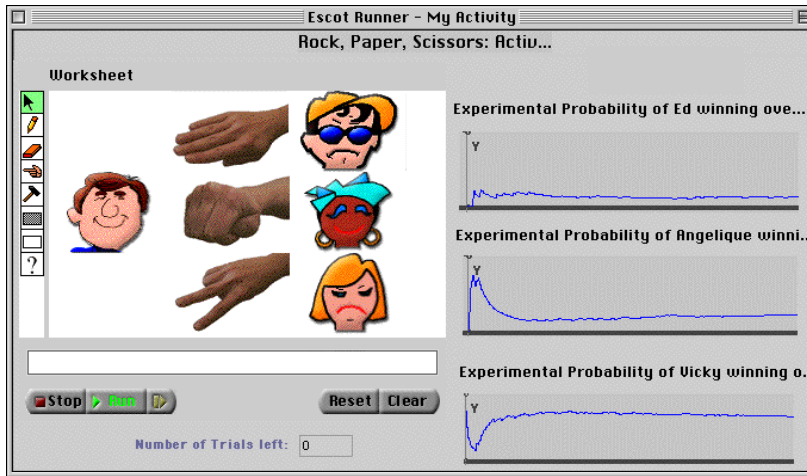


Figure 1. Problems of the Week are delivered as component-based activities. The Rock, Paper, Scissors activity combines 3 SimCalc Chart components (right side) with an AgentSheets generated simulation component (left side). Students first predict and then simulate a three-player version of a Rock, Paper, Scissors game. The charts show the experimental probabilities of each player winning.

Large numbers of students have responded to ePOWs through a web mentoring service run by our partner, The Math Forum (<http://www.mathforum.com>), giving us extensive feedback on the problem's success in engaging students in mathematical reasoning. We are field testing web-based, reusable interoperable components from multiple organizations every week.

In this paper, we report what we have learned from this effort. We focus on three lessons:

1. the difficulty of converting “found” tools to well-behaved components and the greater importance of focussing on a few, powerful generative tools.
2. the role of conventions and design patterns in recruiting developers to make their powerful tools reusable
3. the need to extend conventions and design patterns in the Java community to meet pedagogical needs

ESCOT in Context

Reuse and interoperability mean many things to many people within educational technology. Four specific areas of reuse are:

1. **descriptions** of educational objects, e.g. IMS metadata (<http://www.imsproject.org>)
2. **databases** of student records, e.g. the School Interoperability Framework (<http://www.schoolsinterop.org/>)
3. **contents** of lessons, such as paragraphs, images, assessment items (e.g. “Courseware Conversion

Factory”,
http://www.tekknowledge.com/education_training/)

4. **tools** such as spreadsheets, graphs, simulations, geometry visualizations.

ESCOT's focus is squarely on the last area, tools, although we aim to be compatible with work in the other areas. In an eventual synthesis, our tools would be incorporated into the contents of courses and lessons, which would be described by appropriate metadata, and would be deployed in conjunction with databases of student records.

Within our tool focus, we are strongly concerned with improving the quality of student learning. Hence we have spent considerable effort analyzing the highest leverage opportunities to use technology in a lesson-by-lesson analysis of five recent reform-oriented textbooks. Moreover, we include master teachers as core members of our team, and involve developers and teachers together in creating ePOWs through an “integration team” model (see Roschelle, et al., 1999). As mentioned above, we continuously test our lessons through a combination of the web mentoring service, classroom field observations, and interviews of participating teachers.

ESCOT is a five-year proof of concept project. We structured it carefully as a research and evaluation testbed to reduce risk to participants, who take pride in their intellectual property and are reluctant to give away rights to a proof-of-concept organization. As a testbed, our focus is on creating trust among members, giving them an opportunity to demonstrate the feasibility and value in working together, and encouraging them to go forth under other arrangements to a wide impact in the marketplace. ESCOT is open to volunteer members; information is provided on our web site (<http://www.escot.org>).

Because of our limited funding, we have chosen to focus on middle school mathematics, although we foresee expanding this scope in the future. We choose Java for a simple reason: it was the only common denominator of the projects we wished to bring together. Fortunately, Java also has a component architecture (JavaBeans), strong dynamic loading and binding capabilities (through ClassLoaders and reflection), and runs on both platforms that are used in American K-12 schools (Windows and MacOS). Another expected benefit of Java was the availability of a wide range of free or open source educational software. Because of problems running Java in a browser, ESCOT 1.0 used a specialized web application,

ESCOT Runner, instead of a conventional browser, although our next version of the architecture overcomes these limitations.

Methodology

Guided by a review of five modern curricula for middle school mathematics, we have searched for existing Java tools that would help students learn difficult mathematical ideas. In some cases, we deliberately recruited important, well-tested tools familiar to us. These included SimCalc, a set of tools for graphing and simulation; Java Sketchpad, a Java version of the single most popular tool among math teachers, The Geometer's Sketchpad; and AgentSheets, a powerful simulation building environment. But we also searched the Web for existing Java applets, beans and source code that might fit our needs. Important sources in our search included the Educational Object Economy (<http://www.eoe.org>), the IBM alphaWorks site (<http://alphaworks.ibm.com/>) and several JavaBeans repositories. To date, we have created 37 interoperable components, most of which have been tested in ePOWs.

The source material for these components has been quite varied, including:

1. Existing Open Source Tools: large, open source "applications" such as the TurtleTracks Logo programming language and the Image/J image analysis tool (based on NIH Image);
2. Component Generators: desktop applications which effectively "output" JavaBeans such as The Geometer's Sketchpad, AgentSheets, and PEN (a graphical authoring language);
3. Public Domain Applets: Java applets such as SimCalc, and found applets from university web sites and online repositories;
4. JavaBean Repositories: parts of large JavaBeans collections such as OpenMath (<http://pdg.cec.m.sfu.ca/openmath/> over 75 individual components);
5. Shareware Library Functions: small engines such as a shareware class library that evaluates strings representing functions such as " $x^2 + \cos(x)$ ";
6. GUI Component Packages: small GUI components, such as buttons and fields found, for instance, in Sun's Swing collection;
7. Components off the Shelf (COTS) well-written "commercial" JavaBeans such as those found at the alphaWorks site.

Our context for assembling these components is an authoring tool we built called "EscotBuilder," based on the BeanBox reference container for JavaBeans provided by Sun, which we extended to better meet our needs.

BeanBox is a Java application that imports JavaBean configuration files, allows visual placement of instantiated components in a layout, and allows connections to be wired among the components. We augmented this to use XML as a storage format, to offer better authoring control, and to support some pedagogically necessary features (as we discuss later).

A geographically distributed and diverse team of approximately a dozen part-time programmers has worked on these adaptations, and we have kept engineering logs. The findings reported in this paper arise from our analysis and synthesis of our observations across the many programmers and components.

Finding 1: De-emphasize "found" tools; focus on a few powerful, generative tools

Many naïve rationales for reuse and interoperability focus on the promise of discovering huge collections of tools on the internet. In contrast, computer science research on reuse has found that (Poulin, 1999, p. 98):

1. Favored collections are "small libraries of greatly used, well-designed, domain-specific, high-quality components." Libraries contain on the order of 30 to 250 components.
2. "The key to high levels of reuse comes from building collections of components that all work together and that many applications will need."

Researchers describe this as a "product line" approach, arguing that, similar to stereo consumers, developers desire component collections that are well-conceived, organized, compatible groups of products that coherently address their needs.

We have generally been disappointed in our efforts to find useful applets in public repositories and repurpose them for our own contexts. Many of the applets we have found are poorly written and tested (they are often class assignments or first efforts of new Java programmers). In addition, most components are poorly modularized, few bother to declare Application Programmer Interfaces (APIs), and many cannot be disaggregated easily to select a particular capability. These barriers to adapting found code can call into question the value of reuse, especially for the simpler components. Often the effort of understanding someone else's work can be greater than the effort of rewriting the code from scratch.

When we have found applets worth reusing, most of the time spent in adapting them has had very little to do with the high-level interoperability we are interested in, e.g. linked multiple representations. Most of our work involved

converting to a common denominator Java platform, which in our case was Java 1.1 with the Swing GUI framework. With “found” components, we spent a lot of time adapting code to:

1. use the Java 1.1 event model instead of the 1.0 event model
2. use lightweight (pure Java) screen rendering, instead of heavyweight, native OS-dependent rendering
3. use only features of Java that run reliably on both of our platforms (MacOS and Windows)

Many of these issues will arguably decline with better support and standardization of Java across platforms, but it is not clear when this will occur.

Another set of time-consuming issues with “found” components has more to do with the original programmers not anticipating their code’s eventual use in a component environment. For example, some of the code we have worked with:

1. assumes it will uniquely own the window or menus in its surrounding context;
2. uses static variables and assume there will be only one instance of the component at a time;
3. declares non-modifiable “final” methods for optimization reasons, which makes it impossible to adapt the component without access to the source code;
4. fails to expose or document a public API;
5. fails to instrument key data models for change notification, necessary for data sharing;
6. has untested paths in the code that become apparent in the switch from the original to a component context;
7. is part of a larger library with its own interoperability mechanisms, often having strongly interlocking dependencies that are hard to untangle.

We have had greater, more straightforward success in three areas:

1. Adapting engines that have no GUI interface (and hence are much more likely to run everywhere without modification). As engines, these often have better public APIs and documentation, often achieve fairly complex functionality, hence are worth the effort of adapting for reuse.
2. Adapting applications that generate JavaBeans. The key code needed for reuse and interoperability can often be implemented for these in a wrapper to the ‘normal’ output of the application. Adapting generator tools can have high pay-off, since they typically can output a large variety of components. End-user programmable JavaBean generators such as

AgentSheets even allow non-programmers to build their own components (Repenning 2000).

3. Adapting “commercial-quality” tools that have already been carefully architected to follow JavaBeans conventions and design patterns, and that have been tested in BeanBox-like environments.

Based on our analysis of the mathematics curricula, we have also learned that there are only on the order of 30-50 core components that are needed to cover the core curricula. Hence it makes more sense to focus on purposeful integration of a smaller set of powerful, flexible, generative tools than to maintain the dream of easily reusing “found” components.

Finding 2: Conventions and Design Patterns Attract Developers

In our first iteration in trying to achieve component reuse and interoperability, we used the standard Sun BeanBox as a bean assembly tool, and used the emerging InfoBus (<http://java.sun.com/beans/infobus/>) standard to achieve interoperability. InfoBus is a set of interfaces and a supporting class library that supports defining components which dynamically publish and subscribe to data flows. Components connect to a “bus”, and notify each other of mutual interest in specific flavors of data. The InfoBus standard seemed to meet our needs with regard to linking multiple views of the same data.

Our experiences with InfoBus were mixed. On one hand, we were able to demonstrate components from different vendors publishing and subscribing to data. On the other hand, we quickly found that developers were turned off by the complexity of the InfoBus APIs, and the amount of new support code they needed to write. In practice, to develop an InfoBus-aware component, a developer had to implement many new interfaces, each with many new methods. In addition, data had to be wrapped to comply with InfoBus standard data definitions. Our developers (especially those with an already shipping product) were reluctant to build dependencies in their core tool upon a complex API for which there was no current market demand. They were equally reluctant to maintain forked versions of their tool, one for ESCOT and one for their existing consumers.

As we gained more experience with InfoBus, we found it failed to meet our needs in other ways too. In particular, it was hard in InfoBus to determine the state of connections among components, so as to record inter-component dependencies in a file, and restore them later. More generally, we gradually found that each developer was writing repetitive code to connect to InfoBus that could better be centralized in the “bus” itself, making the bus

more complex and the reusable components simpler. Such centralization of complexity can be a valuable means to make it easier for outside developers to produce compatible components.

Hence, we dropped InfoBus and built our own “ESCOT Broker” with a metaphor of active brokering of data objects among components. Our active broker uses reflection to introspect a JavaBean component and determine its interoperability characteristics. As a consequence, if developers merely follow established JavaBeans conventions and design patterns, we can “discover” their capabilities and adapt those capabilities as necessary. For example, JavaBeans introduces the convention of naming a property “X” with `setX()` and `getX()` method. It introduces an event-based notification pattern, `PropertyChangeEvent`, for allowing interested observers to know when the data has been updated. We have found that developers *are* willing to follow such design patterns and conventions, because these often amount to good coding practice anyway, and furthermore do not entangle their core tool with external APIs, which may or may not be necessary. While external APIs may be “standards” they may not be in widespread use; sensible de facto conventions are more pragmatic than unused standard APIs.

Our ESCOT 1.0 implementation used a very “thin” API that relied on design patterns; in a forthcoming 2.0 version, we plan to allow components to be interoperable without implementing any required API, by simply following known conventions and design patterns. Components that desire more interaction within the ESCOT context can supplement these conventions with descriptors that describe the meaning of the conventions they implement (through extensions to BeanInfo-based Feature Descriptors). Components intended to directly control their containing ESCOT environment can go one step further, and directly implement ESCOT interfaces. In focus group meetings with our developer community, we have found that they greatly appreciate a tiered scheme that allows them to progress through degrees of interoperability as needed.

Based on our experience, we recommend that if the goal is to entice talented developers to adapt their existing production tools to a new reuse specification, the specification should rely as much as possible on well-known, easily understood, easily implemented patterns and conventions, and only require implementing APIs when highly specialized interactions are needed.

Finding 3: Existing Conventions and Design Patterns Are Not Enough

As we have progressed in building learning materials with the ESCOT architecture, using a wide variety of

components, we have found several places where existing conventions and design patterns do not fully support pedagogical needs. As we have argued elsewhere (Roschelle, et al., 1999), it is important to tune reuse and interoperability to domain needs. Some specific needs we have identified, and have begun to resolve are:

- 1. Dynamic publishing and subscribing.** Some components only know their interoperability characteristics at runtime. For example, ESCOT’s Java Sketchpad component can publish different geometrical measurements depending on the particular diagram it is displaying. Another example is a graphing component that can subscribe to varying numbers of external mathematical functions. Because introspection only examines a component’s compile-time features, it cannot detect dynamically varying publishing capabilities.
- 2. Coordination Patterns.** In standard JavaBeans event wiring, an event source is connected to a listener in one direction only. Connecting a pair of components bi-directionally often results in infinite circularities, as each component notifies the other that its data has changed. Moreover, for very simple data models, such as a number, it is easier for each component to maintain its own state than it is to create a proper model-view-controller separation, with each component agreeing to all share the same data model. To solve both problems, we have introduced a “change coordination” pattern (DiGiano, 2000), where each component maintains its own model and coordinates the value of the model with an external data source, without propagating circular notifications.
- 3. Function data types.** A mathematically important data type is a continuous function, represented by a method that takes a numeric argument and returns a numeric value. In addition, other attributes of a function are important, such as the domain over which it is defined. Functions can be viewed by multiple components, such as graphs, visualizations, and tables. JavaBeans does not have a convention for declaring that a component publishes or subscribes to a function, so ESCOT has had to define one.
- 4. Component Persistence.** The built-in mechanism for storing the state of an activity, or students’ work on an activity in Java is serialization. Serialization writes a low-level, binary representation of a tree of Java objects. We have found that serialization is poorly supported by existing components, is fragile under tool updates and substitutions, and results in large files. In its place we have been exploring an XML-based textual format for storing and retrieving the state of an aggregation of components and their

interconnections. This has been very successful in ESCOT 1.0, and in future work we are examining using the emerging “archiver” package from Sun which uses a similar technique and may become a supported standard (java.sun.com/products/jfc/tsc/articles/persistence/).

Conclusion

Increasing the interoperability and reuse of mathematical tools is important within the overall process of K-12 mathematics education reform, as linked tools can enable better learning, and reusable tools make the most of limited developer resources. The emergence of Java offers a potentially powerful platform for building educational tool components that are reusable, despite the many pragmatic difficulties with deploying Java user interfaces today. Our experience with ESCOT shows that it is practical to create activities (such as ePOWs) that incorporate multiple tools from multiple sources, and interoperate effectively. However, component techniques are not a panacea.

We have found it quite difficult to effectively appropriate components publicly available on the Web for our context. Instead, we have found it more worthwhile to focus on a small number of powerful, generative, much needed components and applications that output components. Hence, we believe software reuse efforts should focus on a product line approach, which allows for evolution and substitution within fairly granular and well-defined tool categories, rather than focussing on enabling mixing and matching of “found” tools, whatever their heritage and purpose.

In building a consortium of largely volunteer members, we have found it easier to encourage contributions that leverage well known, easy-to-implement conventions and standards, rather than requiring conformance to an unfamiliar and complex API. However, existing conventions and standards do not fully meet pedagogical needs, so we advocate strategically extending existing practice in a way that allows developers to incrementally adopt APIs as necessary for their particular component.

In the future, we anticipate growing the domain of ESCOT from middle school mathematics education to more general topics, and to wider reaching services. As we do, the compatibility of ESCOT with other layers of reuse and interoperability will become important challenges.

Acknowledgements

This work in this paper was supported by the National Science Foundation (Award: REC-9804930, DMI-9761360). The opinions presented are the authors, and may

not reflect those of the funding agency. We thank all members of the ESCOT team for their contributions to our effort.

References

- DiGiano, C., Roschelle, J. (2000), Rapid-Assembly Componentware. To appear in Proceedings of IWALT 2000. IEEE Computer Society Press.
- Kaput, J. (1992). Technology and mathematics education. In D. Grouws (Ed.) A handbook of research on mathematics teaching and learning. NY: MacMillan, 515-556.
- Kozma, R., Russell, J., Jones, T., Marx, N., & Davis, J. (1996). The use of multiple, linked representations to facilitate science understanding. In S. Vosniadou, E. De Corte, R. Glaser, & H. Mandl (eds.), International perspectives on the design of technology-supported learning environments. Mahwah, NJ: Erlbaum. 41-60.
- Poulin, J.S. (May 1999) Communications of the ACM, Vol. 42, No. 5, 98-100.
- Repenning, A., Ioannidou, A., & Phillips, J. (1999). Collaborative use and design of interactive simulations. In Proceedings of Computer Support for Collaborative Learning, Stanford University, Stanford CA, 475-487.
- Repenning, A., & Perrone, C. (2000). Programming by Analogous Examples. Communications of the ACM, 43(3), 90-97.
- Roschelle, J. & Kaput, J. (1996). Educational software architecture and systemic impact: The promise of component software. Journal of Educational Computing Research, 14(3), 217-228.
- 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
- Roschelle, J., DiGiano, C., Koutlis, M., Repenning, A., Jackiw, N., & Suthers, D. (1999). Developing educational software components. IEEE Computer, 32(9), 50-58.