

Rapid-Assembly Componentware for Education

Chris DiGiano and Jeremy Roschelle

SRI International

chris.digiano@sri.com

Abstract

In this paper, we provide an overview of our work in mining design patterns from the Educational Software Components of Tomorrow project. By identifying, crystallizing and organizing design patterns, we aim to address problems of reusability and interoperability that currently present critical bottlenecks for the rapid assembly of educational technology. We describe 4 categories of patterns: instantiation, interoperability, control cooperation, and screen cooperation and outline 3 illustrative patterns: Parceled Publication, Observer, and Replicated Model.

1. Introduction

The notion of software created from pre-fabricated components has long been a holy grail of software engineering. The benefits of "componentware," as we have come to call it, are obvious: a particular functionality need only be created once as a component, and then it can be reused over and over in innumerable contexts. With componentware, instead of wasting time reinventing the wheels of a new software vehicle, designers can focus on innovative new engines, brake systems, or interior controls.

In education, we are in desperate need of the efficiencies promised by componentware. Effective use of computers in the classroom demands highly interactive applications employing multiple representations that individual teachers can configure to their particular students, teaching styles, and local standards [1]. Such complex and customized applications currently require huge software development efforts—efforts that have largely been limited to government-supported proofs of concept [2].

Despite the great need for and promise of componentware, progress towards this goal has been slow and halting. Researchers point to some software reuse success stories [3], but the search continues for the technical and organizational conditions that will make them replicable. A central challenge is giving software development teams the tools that will enable reuse without over-constraining the development approach.

We propose that educational software developers turn their attention to a useful middle ground in reuse techniques: identifying and applying software design patterns[4] employed in prior componentware efforts.

What makes design patterns so appropriate as middle ground seek is that they do not specify an implementation for any particular programming language or platform. Design patterns follow a general form that includes a memorable name, a real-world example, an abstract description of the problem the pattern addresses, the fundamental solution principle underlying the pattern, and a discussion of its benefits and liabilities.

In this paper, we provide an overview of our work in mining design patterns from our Educational Software Components of Tomorrow (ESCOT) project. Below we describe 4 categories of patterns we have identified and outline 3 sample patterns. As background, we begin with a brief description of our project, and offer a definition of the kinds of componentware needed for educational content, which we term "Rapid Assembly Componentware."

2. The ESCOT Project

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. Large numbers of students have responded to each problem 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.

ESCOT's componentware focus is mathematical tools. Tools such as calculators, spreadsheets, graphing calculators, and animated simulations have a potentially profound role in enabling many more students to learn complex mathematical ideas [5]. 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 [6]. 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 [2]. 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). In ESCOT we are investigating how to design and connect such interoperable components (see [1]), and put them into practice as part of Problems of the Week.

3. Rapid-Assembly Componentware For Learning Tools

So far we have used the term componentware fairly loosely. Indeed, one could conceivably apply the term in a variety of contexts from distributed applications coordinating various processes using CORBA to software generated with the assistance of rapid application design (RAD) environments. We believe that educational content calls for a particular type of reusable software, which we call Rapid-Assembly Componentware (RAC). The ability to rapidly assemble educational software is critical when content providers such as the MathForum need to efficiently produce an on-going series of activities, or when curriculum designers seek to quickly adapt existing learning tools for their particular classroom environments.

RAC is distinguished from pure component libraries, which lack facilities for quickly combining components from their collections to form finished applications. RAC is also distinguished from distributed and multiprocess component efforts, which involve configuring and maintaining a collection of independent programs potentially running on multiple computers. Instead, RAC is about combining reusable parts to form applications or applets that can run in a single process on a personal computer.

Our emphasis on single-process applications is particularly appropriate for education, as learning tools will continue for the foreseeable future to be dominated by stand-alone software that can be launched in a single step on inexpensive classroom computers. Conveniently, this focus also helps simplify component interoperability issues, since requests and data do not have to be marshaled and unmarshaled between components and implementation languages, an unfortunate but necessary distraction of distributed componentware projects such as CORBA.

A RAC System consists of:

- a Kernel (RACK) providing component brokering and persistence facilities. Metaphorically, a RACK provides the structure onto which one mounts components. Within the same componentware system,

the RACK may have two different manifestations: a design-time version used during assembly and a run-time version used for application deployment.

- a Design Environment where an application is assembled through direct manipulation of visual components.
- a Library of components. Although the term component is used in many ways in the literature software design, in the context of RAC a component is a self-contained visible object with a public interface that other components can use either directly or indirectly through the Kernel.

It is necessary for the component to have some visible representation at least at design time, so that it can be configured and connected. At runtime, the component may be invisible if it serves only to process information on behalf of other components. From an information access perspective, no other component, nor the Kernel, should have direct access to a component's private variables or methods.

Existing examples of RAC systems include Microsoft's VisualBasic, Sun Microsystem's Forte, the educational software project E-Slate (<http://e-slate.cti.gr>), and our own system focused on mathematics education, ESCOT. Two RACs that had once shown great promise but are no longer supported are OpenDoc and Sun Microsystem's JavaStudio. The patterns in this article draw heavily from our two years of experience with ESCOT, but many of the above systems have also influenced the pattern language.

4. Categories of Patterns

4.1. Instantiation Patterns

This category covers what happens when a new component is introduced into an assembly and how the Kernel is made aware of its capabilities.

Instantiation patterns handle the arrival of a component on the scene. The central activity is establishing component “publications” and “subscribers”. The publishing metaphor is used to capture all varieties of component interoperability, including components monitoring changes in another and components sharing a central data source. This is in contrast to Buschmann et al [7] whose Publisher-Subscriber pattern is devoted solely to change monitoring. Names of patterns in this category include Parceled Publication, Parceled Subscriber, Capability Discovery and Capability Reflection.

4.2. Interoperability Patterns

This category covers how specific types of components connect and what protocols enable them to exchange data.

Interoperability patterns elaborate on the basic publishing and subscribing concepts of instantiation patterns. Some of these patterns make reference to a “model,” a key role of the Model-View-Controller pattern. Names of

patterns in this category include: Observer, Replicated Model, Shared Model, Action Access, and Scripted Bridge.

4.3. Control Cooperation

This category covers how components work together so they effectively enable the user to manipulate the same data. Names of patterns in this category include Change Coordination and Mutable Shared Model.

4.4. Screen Cooperation Patterns

This category covers how components can be good citizens in an assembly by being economical with screen space and accommodating of overlapping components. Names of patterns in this category include: Minimizable UI, Controllable Features, Glass Shadow and Composite.

5. Example Patterns

Below we outline examples, problems, solutions, and structure for three sample patterns: Parceled Publication from the instantiation category and Observer and Replicated Model from the interoperability category.

5.1 Parceled Publication

Example: A slider tool provides a public interface that includes the current integer value of the slider thumb, and the top and bottom of the slider range. A graphing tool can receive floating point values to change its scale. One ought to be able instantiate both tools in an assembly so that the slider could be used to control the scale of the graph. The Kernel could give the graph component direct access to the slider's entire public interface, but this would be overkill, since only the current slider value is needed. Furthermore, if the graph had direct access it would be solely responsible for addressing the floating point-integer type-mismatch. A mechanism is needed for the Kernel to interpose some access protection and type conversion.

Problem: There are two related issues that arise when a Kernel is facilitating component connections:

1. Limiting inter-component access to just the needed set of capabilities.
2. Coupling components in a sufficiently loose manner so that the Kernel can insert bridges to overcome mismatches

Solution: A data-providing component offers one or more self-contained objects that are published to other components. Subscribing components gain access to only the "parceled" publications they need instead of the publishing component's entire public interface. For certain types of mismatches between two components, the Kernel can generate a proxy parcel that acts as a bridge between

the original publication and the subscriber. The subscribing component is under the illusions that it is communicating directly with the publishing component.

5.2 Observer

Also Known As: Publisher-Subscriber [7]

Example: A simulation component models the spread of a virus through a community. While running, the simulation computes the total number of infected individuals. A thermometer component can display values within a bounded range. A curriculum designer wants to combine these two component in an assembly, with the thermometer used as a display for the number of infected. The thermometer needs to monitor changes in the simulation data and display the current number of infected.

Problem: A situation often arises in which data changes in one component, but one or more other components depend on this data. We could solve the problem by introducing direct calling dependencies along which to propagate the changes, but this solution would require custom-coded components and make it difficult for the designer to later change the connections between components. We are looking for a more general change-propagation mechanism that can be established rapidly in a design environment and is applicable in many contexts.

Solution: All dependent components subscribe to the component with the data. When the data changes, the publishing component sends a notification to all subscribers.

Structure: In the context of componentware, the Observer design pattern involves components in the following roles:

- A publishing component that offers an observable object called a subject. The subject may be the component itself.
- One or more subscribing components that each offer an observer object as a subscriber. The observer may be the component itself.

The subject maintains a registry of observers. The Kernel registers the observers with the subject at rendezvous. Whenever the subject changes state, it sends notification to all observers. The observers in turn retrieve the changed data at their discretion.

Variants: Subject receiver. Instead of the subscribing component offering an observer, it offers a subject receiver object. The interface for this object provides method that the Kernel can invoke to pass a reference to the subject at rendezvous. Unlike the standard Observer pattern, it is the responsibility of the subject receiver to register the subscribing component with subject. If a subject is later revoked, it passes a null value to the method.

5.3 Replicated Model

Example: Consider a variation of the Observer pattern example, where the thermometer component is not designed to receive change notification. Instead, it simply provides a method for setting the current value being displayed.

Problem: As in the Observer problem, a component may seek data from another component. We could solve the problem by using the Observer pattern. However, this assumes that the dependent component has the ability to accept change notification messages and properly retrieve the changed data. Many existing components are not designed this way, especially if they were never intended to interoperate in a componentware environment. Alternatively, components may provide a public interface that exposes models that can be changed externally. We seek a solution that can take advantage of the public mutable models of a component to affect a data conduit that another component can use to pass in values.

Solution: The publishing component and subscribing components each maintain their own copy of the data. The Kernel monitors changes in the published data and invokes corresponding change requests on the subscribing component.

Structure: The Replicated Model design pattern involves the following roles:

- A publishing component that offers an observable subject. The subject may be the component itself.
- One or more subscribing components that each offer a subscriber in the form of a mutable model.
- The Kernel interposes a Bridge [8] to transform change notices coming from the subject into change requests on the mutable model. The bridge is created at rendezvous when the components involved in the data exchange are connected.

The subject maintains a registry of observers. At design time a mutable model of a subscribing component is selected for connection with the subject. The Kernel generates a bridge and registers it as an observer. Whenever the subject changes state, it sends notification to all observing bridges. The bridges then retrieve the changed data and invoke corresponding change requests on their associated mutable models.

6. Conclusion

Most of the patterns in our RAC pattern language are found in our the ESCOT framework, which runs in Java 1.1.8 across multiple hardware and OS platforms. In the 1999-2000 school year, ESCOT was used to host over 30 mathematical tools for problems of the week. We plan to produce 16 more activities in the 2000-2001 school year. ESCOT is an open project, and new members may join through our Volunteer Program (see www.escot.org).

Design patterns are clearly tools for software reuse in that they offer problem-solving recipes based on reuse the knowledge in the heads of experienced designers. However, they do not by themselves support the componentware vision of composable units of prefabricated code. By providing patterns specifically for the componentware context, we aim to offer direct support for those either creating components, adapting code for the componentware environment, or developing componentware shells or "kernels." Our hope is that by recording componentware experiences in a pattern form we might begin to form a common vocabulary that helps brings coherence and maturity to our efforts. Ideally, the patterns will inspire greater participation in componentware by providing guidelines for developers to write their code in a form that can be connected to other components and for those interested in creating kernels for entirely new platforms.

7. Acknowledgements

This work in this paper was supported by the National Science Foundation (Award: REC-9804930). The opinions presented are the authors, and may not reflect those of the funding agency.

8. References

- [1] J. Roschelle, C. DiGiano, M. Koutlis, A. Repenning, N. Jackiw, and D. Suthers, "Developing educational software components," *IEEE Computer*, vol. 32, pp. 5-58, 1999.
- [2] J. Roschelle and J. Kaput, "Educational software architecture and systemic impact: The promise of component software," *Journal of Educational Computing Research*, vol. 14, pp. 217-228, 1996.
- [3] J. S. Poulin, "Reuse: Been There, Done That," *Comm. of the ACM*, vol. 42, pp. 98-100, 1999.
- [4] J. O. Coplien and D. C. Schmidt, "Pattern languages of programming design," Reading, MA: Addison-Wesley, 1995.
- [5] J. Kaput, "Technology and mathematics education," in *A handbook of research on mathematics teaching and learning*, D. Grouws, Ed. New York, NY: Macmillan, 1992, pp. 515-556.
- [6] A. Repenning, A. Ioannidou, and J. Phillips, "Collaborative use and design of interactive simulations," presented at Computer Support for Collaborative Learning, Stanford, CA, 1999.
- [7] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture: A system of patterns*. Chichester: John Wiley & Sons, 1999.
- [8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns*. Reading, MA: Addison-Wesley, 1995.