E-Slate: a software architectural style for end-user programming

George Birbilis, Manolis Koutlis, Kriton Kyrimis, George Tsironis, George Vasiliou*
Computer Technology Institute
Riga Feraiou 61
Patras 262 21, Greece
+30 61 273496
{birbilis, koutlis, kyrimis, tsironis, vasiliou}@cti.gr

ABSTRACT
This paper describes E-Slate (http://E-Slate.cti.gr), an exploratory learning environment that builds on a component-based approach, to enable end-users to create educational software constructions themselves by wiring components, using the Plug and Synapse metaphors.

Keywords
Component software architectures, educational software, end-user programming

1 INTRODUCTION

E-Slate is an exploratory learning environment that builds on a component-based approach to fulfill two core requirements stemming from the educational context: a) synchronized alternative representations of phenomena and concepts, and b) ability for end-users to create software constructions themselves. E-Slate, largely supporting the constructivistic learning approach ([1]), provides an end-user "workbench", within which educators, learners, activity authors, curriculum designers, or publishers can create Microworlds (pieces of software that model activity situations conforming to specific educational objectives) to experiment with concepts, relationships, and hypotheses.

From a software engineering point of view, E-Slate defines an architectural style for synthesizing software "systems" out of components. Under this perspective, end-users' software constructions, although of a size that would typically be called "programs", are viewed as systems, each with its own architecture determined by the structural and functional interrelationships among the building blocks—the components—as they are defined by end-users.

E-Slate's targeted uses and intended audience, typically consisting of "end-users" who are not computer experts, have particularities that have direct implications on E-Slate's design and key features:

- There is no a priori system to be built. Ideas are tried out by end-users until what has been built is satisfactory. Using a microworld often entails manipulating its "system architecture" (e.g., altering associations among participating components). In other words, an inherent characteristic of the design and use of microworlds is their dynamic nature.
- The large majority of users cannot—and do not want to—be engaged in any kind of "programming" other than simple scripting. However, they *are* able to assemble components together by "wiring" them. This means that components should provide encapsulated chunks of behavior in the form of "probes" for structural coupling with other components, and the supporting environment should provide the means for "translating" the structural inter-component relationships to overall microworld behavior [3, 5].

The E-Slate architectural style defines (according to [2]):

i A design "vocabulary", consisting of a) the nature, characteristics, and basic common behavior of E-Slate components, b) the inter-component connection types, and c) the notation used for entities and relationships.

ii The design "rules" according to which the synthesis of constructions is realized. These are *guidelines* for component developers, and usage *constraints* for end-users.

iii An infrastructure for facilitating a) the synthesis of component-based constructions by end-users, and b) the development of components by developers. The former comes in the form of a set of *core components* that provide end-users with powerful visual tools to build with. The latter comes as a set of *core services* (APIs and class libraries) that E-Slate components can deploy to implement basic portions of their behavior.

Core components and services together implement the logic that guarantees the validity and correctness of component-based constructions, by applying and maintaining the architectural design rules and constraints. Although these are currently implemented in Java, the E-Slate specification is

* Names appear in alphabetical order.

independent of implementation language and adopted component model and technology.

2 E-SLATE VOCABULARY AND DESIGN RULES

Components

E-Slate components are visually manipulatable entities that can be selected, dragged around, etc., representing—but not restricted to—one of the following:

- "View-controller" type of entities in Model-View-Controller (MVC) terminology, or else desktop "applications" that are given input (content) upon which to act.
- "Model" type of entities in MVC terminology, or else desktop "documents". Although not purely "passive" data, they can be regarded as such by components of the previous type.
- Functional entities targeted to supplement the functionality of some larger context in which they are plugged in. They may have a user interface (UI) (e.g., in the case of a clock component displaying the current time) or not.

Through the E-Slate Workbench (see Section 3), all of the above types of entities are provided to the user as "active" components with a common and consistently available behavior, such as creation and destruction, direct manipulation, inspection and programming, storage and retrieval.

Inter-component Synapses

Components are capable of forming associations with other components. Established associations are called *Synapses*[1]. Inter-component synapsing provides users with the unique ability to manipulate the structure and behavior of Microworlds at a high conceptual level.

A Synapse relates two components, each of which plays a specific *role* in the context of that Synapse. Establishing a Synapse means that the two components engage in a series of transactions that realize the—by nature abstract—association. Transactions are performed according to a communication protocol—a Synaptic Protocol—that "translates" the semantics of an association type to message exchanges between the associated components. "Performing" a role, means that components implement the logic handling these message exchanges according to a synaptic protocol.

For a Synapse to be established, two "coupling" roles are needed. By convention, one of these roles is designated as "male" role and the other as "female". It is up to the component designers to identify which is the male and which is the female role and which component implements the corresponding "logic").

Inter-component Synapses, in the order that they were established, are stored persistently as companions to the persistent state of the components.

[1] Derived from the Greek *synaptein*, "to join together".

Plugs

A component exposes its associativity capabilities to the mechanisms that handle establishing of Synapses and to end-users through *Plugs*. A Plug is the *descriptor* of a component's role and has the following characteristics:

i. A *name*, describing the component's function that the Plug represents. Plug names target the end-user and should be as meaningful as possible in this respect.
ii. A *type*, directly corresponding to a pair of two sets: a) the Java interfaces implementing the logic of the specific component's role and b) the corresponding Java interfaces that are expected to be implemented from the "other" Synaptic end. Each Plug type has a visual representation consisting of a unique combination of shape and color (reflecting among other things, a male or female role).
iii. A *reference* to the component internal object(s) implementing the particular synaptic interface.
iv. A *cardinality* attribute, denoting the number of distinct plugs (1 or MANY) to which the plug can be connected.

The information conveyed by Plugs is used by E-slate to guide the process of establishing Synapses. Once a Synapse is established, components interact on their own, independently of the existence of Plugs.

To summarize, Plugs carry the "syntax" rules that govern the inter-component association process. For end users, the rule is simple: Synapses may be established between Plugs bearing the same color and matching shape. For the component developer, proper definition of a component's Plugs at design time is essential for guaranteeing the establishment of "correct" Synapses at authoring time.

Establishing Synapses

Inter-component Synapses may be established and broken by end-users either by visual methods (drag-dropping Plugs) or via the scripting mechanism (by means of scripting commands of the form "connect thisPlug of thisComponent with thatPlug of thatComponent"). In both cases, the responsibilities of the supervising process include the following:

i. The logic that performs the eligibility checks based on Synapse syntax rules that are carried by the Plugs (pre-association stage).
ii. The actions that realize a Synapse, i.e., setting the mutual component references.
iii. The actions that should be taken by the components when establishing Synapses (post-association stage), such as reflecting the effects of an association on the display accordingly.

E-Slate provides a Synapse manager as one of its core services that performs all these actions and handles the complex details. The scripting mechanism uses this service while supervising the establishment of Synapses.

Actions (ii) and (iii) may have side effects on the components' state, behavior, and appearance. This means that the order in which Synapses are established in a multi-component configuration may affect its final state.

Component Scripting Interface

E-Slate components provide a scripting interface to the E-Slate scripting mechanism, consisting of handlers to language primitives that manipulate component properties and/or trigger component-specific functionality. E-Slate's scripting language is a component-oriented flavor of the Logo language, specifically designed for E-Slate [4]. Component Logo primitives are registered on the fly with the scripting mechanism upon component instantiation.

Event Handlers

Components may implement callback handlers for various events. Handlers are called from the E-Slate core when the corresponding events are fired. In addition, using the E-Slate Workbench, event handlers written in either Logo or Java can be attached to the events generated by the components themselves.

Component Properties

Component properties have a name and a data type. Their values can be interactively set/retrieved via scripting and/or the Workbench Property Editor. Component properties can be synchronized with the properties of other components through data-flow Synapses, implemented by corresponding roles/interfaces and manifested by Plugs.

Component persistency

Components use E-Slate's structured storage core services to save and retrieve their state in compound files. A component's persistent state typically includes its properties, its attachable scripts, and its event handlers. Save and Load operations are typically initiated by the E-Slate Workbench, and are applied on the entire Microworld, saving/loading a) the state of each component and b) the inter-component Synapses in the order they were established.

3 E-SLATE TOOLS AND CORE SERVICES

As mentioned in the introduction, E-Slate provides:

a) Tools (which are components themselves) for end users to build educational Microworlds out of components, namely the Workbench and the Plug Editor.

b) The infrastructure for component developers to develop E-Slate aware components, namely the E-Slate Core Services and the E-Slate API.

The E-Slate Workbench

Within the context of an Operating System, the E-Slate Workbench is an application used for creating, editing and "running" E-Slate Microworlds (i.e., an authoring tool and runtime environment together). Within the context of the E-Slate environment, the Workbench is the OS desktop equivalent: just as desktops provide a surface for documents and folders, the E-Slate Workbench provides the space in which "live" components can perform.

In a way, the E-Slate Workbench provides a functionality similar to that of the Windows and MacOS desktops. Specifically, it provides the means for creating, laying out and customizing components, editing component properties, defining scripts as event handelrs, and saving and restoring the state of Microworlds. Finally, E-Slate Workbench manages the sequencing of events and the appearance of components, so that Microworlds perform the intended functions.

The Plug Editor

The Plug Editor visualizes the graph of inter-component associations within an E-Slate session and allows the visual editing (creation and destruction) of Synapses. For each individual component, it displays its plug hierarchy. Selecting a plug highlights all plugs that are connected or can be connected to the selected plug, and dragging a plug and dropping it on another establishes or breaks a connection.

E-Slate Core Services

E-Slate provides the following core services that can be used by all components at the programming level:

- Synapse Manager: Mediates the process of establishing or breaking a Synapse between two components, making all necessary checks and ensuring properly established Synapses.
- Namespace Manager: Each E-Slate component has a name that can be referenced unambiguously by end-users in operations such as scripting, where internal E-Slate ids cannot be used. This service ensures that components have unique names within their immediate context, so that a component can always be identified by its name.
- Persistent storage: This service provides for storing and retrieving microworlds from files. Stored microworlds contain information about both the state of the comprising components and the established Synapses among them. A structured storage scheme is used, so that saving each component's state may be delegated to the component, without requiring the component to be aware of the format of saved microworld files.
- Scripting environment: This service is built as a set of extensions to the Logo language, and provides the ability to control the function of the microworld from Logo procedures. These extensions take the form of three kinds of additional Logo primitives: a) primitives implemented by the E-slate platform on behalf of each component (e.g., establish a Synapse between two plugs), b) primitives implemented by the E-Slate Workbench (e.g., load a microworld from a file), and c) an optional set of primitives implemented by the components themselves (e.g., "start" and "stop" primitives implemented by a chronometer).

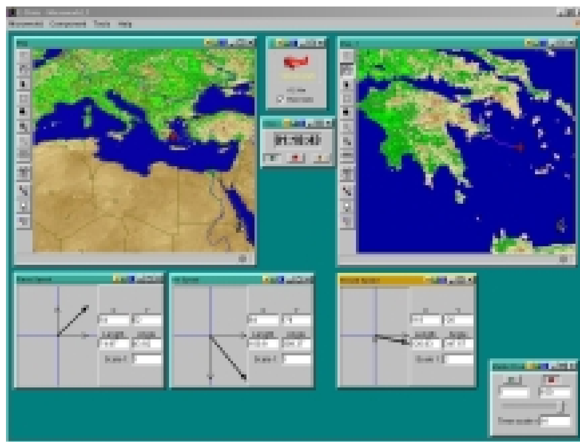
The E-Slate API

E-Slate components are true Java beans and can function autonomously inside the E-Slate Workbench. To cooperate

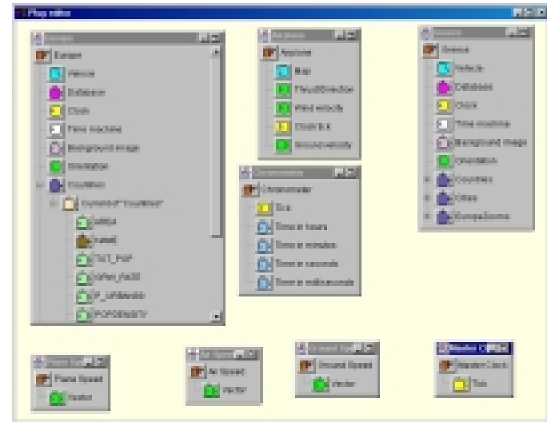
with other components, they have to be made E-Slate aware. This is accomplished in a few simple steps via the E-Slate API. First, the component obtains an E-Slate handle by invoking the `registerPart` method of class `ESLate`. This handle provides access to the E-Slate core functionality. The component then constructs plugs initialized with the internal objects and interfaces that it needs to share as part of the synaptic interfaces that it wants to implement, along with a message handler for receiving data through the synapse. Finally, these plugs are added to the handle using its `addPin` method. For components requiring fine control of their operation within E-Slate, the E-Slate platform provides a rich API that controls all aspects of its functionality.

4 E-SLATE MICROWORLD EXAMPLE

To build a Microworld, authors use E-Slate to lay out components, set their properties, define their associations by establishing the corresponding Synapses, and, possibly, write scripts for various events. The following example shows the construction of a simple microworld. The idea here is to create a small game for playing with the concept of "vector". Specifically we want to set up a scenario in which pupils have to navigate an airplane towards specific targets around Europe, by taking into account side winds.



E-Slate is started and a new Microworld is created within it. Two Map Viewer components are created and are assigned the name "Europe" and "Greece", respectively. Next, a specific map is loaded in "Europe" and passed to "Greece" through the components' Map Plugs so that the same map is rendered in both viewers simultaneously. Next, a new Airplane component is created. The airplane's "Map" plug is connected to the Map's "Vehicle" plug, so that the plane appears to be flying in the map. Note that dragging the plane on any of the two Map Views leads to the same result since they both render the "same" Map. Next, two Vector components are created. One is named "Air Speed", and its "Vector" plug is connected to the Plane's matching plug. The same process is repeated to connect the "Plane Speed" vector to the Airplane. Finally, a third vector is added to the Microworld which reflects the sum of the previous two vectors, that is, the Airplane's ground speed and is connected to the Airplanes' output Plug accordingly. Since we want to create a real-time simulation, we embed a Master Clock component and a Chronometer that is used to display the simulated time. We connect the Master Clock's "Tick" output to the chronometer and also to the Airplane. Inter-component connections are realized by the Plug-editor, as depicted below.



Now we're ready for take off! After we start the Master Clock, we can observe the Chronometer counting the simulated time and the plane flying on the map, navigated by the thrust or the air speed vectors.

5 DEVELOPMENT STATUS

E-Slate is under continuous design and refinement. This paper describes the features implemented in the current (IV) release, which is available at http://E-Slate.cti.gr.

REFERENCES

1. Report to the President on the Use of Technology to Strengthen K-12 Education in the United States (March 1997). Available at <http://www.whitehouse.gov/WH/EOP/OSTP/NSTC/PCAST/k-12ed.html>.
2. R.T. Monroe, "Modeling and Analyzing Software Architectures", *Tutorial for the 1999 International Conference on Software Engineering*, Los Angeles, CA, May 18, 1999. Available at <http://www.cs.cmu.edu/~able/talks_online/icse99_tutorial/>.
3. M. Koutlis, P. Kourouniotis, K. Kyrimis, N. Renieri, "Inter-component communication as a vehicle towards end-user modeling", *ICSE'98 Workshop on Component-Based Software Engineering*, Kyoto, Japan, April 26–27, 1998. Available at <(http://www.sei.cmu.edu/ activities/cbs/icse98/papers/p7.html)>.
4. M. Koutlis, Ch. Kynigos, A. Oikonomou, G. Tsironis, "Empowering Logo through a Component-Oriented Approach", *Proceedings of the 7th Eurologo Conference*, Budapest, Hungary, August 1997, pp. 166–175.
5. J. Roschelle, M. Koutlis, A. Reppening, et al.: "Developing Educational Software Components", *IEEE Computer 10,9, special issue on Web based learning and collaboration*, September 1999, pp. 50–58.