Interactive Component-Based Software Development with Espresso*

Ted Faison
tedfaison@msn.com
Faison Computing Inc.
Irvine, CA 92606
Sept 1997

Abstract

There are a number of competing component models in use today. Most are language-independent, but also platform-dependent and not designed to support a tool-based development paradigm. Espresso is a new platform-independent component model whose primary goal is to make it easy for tools to manipulate and compose Espresso components interactively during the software development process. This paper describes the salient features of Espresso, including the Espresso Notation and the Espresso Language. Because Application Builder tools are such an essential part of any project based on Espresso components, the paper also gives highlights of the visual programming tool we are building.

1 Introduction

Component-based software today is largely based on models that require code-intensive work to produce programs. Tool-based development using software assistants has long been recognized [1] as a means to produce higher quality systems, with fewer defects and in less time. To maximize their effectiveness, tools must aid in the visual composition of components, without the need to generate code. Tool-based interconnectivity requires a component model that is able to expose interface features and capabilities, preferably at run-time and without support from external "description" files. Component models in use today, such as COM, OpenDoc and CORBA don't support run-time interconnectivity. To connect two components together requires multiple steps, most of which require human intervention, because the interconnection requirements of components is not advertised in a way that facilitates tool-based approaches.

Espresso is a new component model based on the JavaBeans specification. The model was designed specifically to make a tool-based approach to software development as powerful as possible. Espresso components support a visual composition development model: components are connected together in a graph using mouse gestures, without code generation. Espresso components can be nested, therefore a graph of components can be enclosed in another Espresso component. By supporting nesting to arbitrary levels, low level components can be combined into larger and larger components until an entire application component is developed.

Software development using Espresso emphasizes rapid visual programming using pre-built components obtained from a user repository. While the Espresso model allows developers to write their own code using traditional edit-compile-link techniques, rapid development is predicated on the availability of a library of pre-built components, much like the TTL libraries used by hardware designers. Visual programming is a powerful notion that has been studied for some time. Chang describes some of the early systems exploring visual programming based on objects [2]. Other component-based environments, such as Audition [3], require the developer to interact with a language-based class system to produce components. Espresso component graphs are built interactively using *live* binary components retrieved from a repository, supporting an incremental development model where everything is live during the design phase.

* This research is supported by the US Air Force, Rome Laboratory, under Contract Number F30602-96-C-0205.

Espresso components are also designed to be distributable over the World Wide Web and to be loadable using browsers in use today. Espresso supports run-time functional querying, allowing designers to search the Web for components supporting a given set of features. Having located useful components, users can download them and install them into their local repository with a single mouse click. Espresso components are very lightweight and are generally quite small, making Web transfers short. Downloaded components are also self-installing, so users aren't faced with a complicated and machine-dependent installation process.

2 Espresso Goals

The Espresso model has one fundamental goal: to simplify software development through a model that supports a tool-intensive approach using visual component composition. Espresso components need to be easily distributable, allowing developers to use the Web to locate components and build non trivial local repositories of components. To minimize transmission time, Espresso components must be lightweight. To this end the *Espresso overhead*, defined as the extra baggage attached to a software product to turn it into an Espresso component, is designed to be very small. The exact amount of overhead depends on the number of Espresso features embedded in a component, but for typical deployed component the overhead is smaller than 1 KB.

Because the Web is the primary distribution medium, it is important for Espresso components to be platform independent, so one component fits all machines, or at least a wide variety of machines. This requirement is satisfied by using Java as the sole development language. To support Application Builder tools to the greatest extent, Espresso components are entirely self-contained units. All meta-data regarding internal composition, interfaces to the outside world, internal graphical layout, and optionally even source code and debugging support information, is stored inside each Espresso component.

This meta-data is made available both at compile time and run time. The requirement to keep overhead as low as possible requires executable code, necessary for deployment, to be "detachable" from development and test information, such as symbol tables and source code. Meta-data includes support for versioning, manufacturer information, legal copyrights and trademarks, component composition, component interconnections and component inheritance. Regarding this last feature, Espresso supports *dynamic* inheritance, which allows the *ancestor* of a component to be changed at run-time. All constituents of an Espresso component can be modified at run-time, allowing developers to work with live components during all phases of design, testing and deployment.

3 The Espresso Component Model

The Espresso model was designed to facilitate a tool-based development model, in which developers create software systems using gestures such as mouse clicking, dragging and dropping. The model is not designed for a specific application domain. Espresso-based developments environments are quite different from others, such as Janus [4], Argo [5] or ADM [6]. The Espresso model is not a design notation, nor a software development assistant, rather an implementation specification. Espresso components are implementations of a particular design and Espresso-based environments are tools designed to build components or complete applications as graphs of components. Espresso systems are displayed as visual graphs, and programmers reason about systems in terms of the relationships between components. The emphasis is shifted from code production to component composition. Users have the option of injecting their own source code into a system by packaging it as a special Espresso User Component.

Espresso components may either be visible or not at run-time. Visible objects include user interface controls, which receive input from the user, and graphical objects, which display other pictorial information that doesn't require user interaction. Non-graphical components are useful for incorporating business logic

into an application, and include objects like component collections (e.g. hash tables and vectors), event timers, data filters and adaptors. A Software Component Engineering Development Environment (SCEDE) tool using Espresso could both create a GUI application, such as the client-side of a multi-tiered database information system, or a non-GUI one, such as a matrix inversion algorithm.

Current commercial systems, like Borland Delphi [7] or Microsoft Visual Basic [8], are geared specifically toward GUI applications, with support for database programming. Although the component models they support (Borland VCL and Microsoft COM, respectively) are generic, they were designed primarily to support user-interface components. To create non-GUI components, developers must resort to low-level traditional code development.

Espresso components are not a solution for all programming situations. Being built in Java, Espresso components can only go where Java goes. Because Espresso components are (currently) run in the interpreted environment of the Java Virtual Machine (JVM), they are not suited for real-time applications. Java microprocessors are under development now and will be available shortly. Machines using such processors will be able to run Java code natively, alleviating the real-time restriction for Java.

Espresso components are also not currently adequate for life-critical systems, the limitation stemming from the relative immaturity of the Java language, and not from the Espresso model itself. Because JVMs are still in evolution, as is the Java language itself, there is the risk that undiscovered defects in the runtime environment may be a source of instability.

Espresso components are designed to be composed together to create more powerful components. The model itself doesn't impose a development methodology, which can be top-down, bottom-up, by stepwise refinement, etc. Two main types of developers are envisioned to be Espresso users: component producers and consumers. Producers develop components for distribution or resale. Consumers are typically application developers who use pre-packaged components. Component development can use both a top-down or bottom-up approach. Large components can be broken down into collection of smaller interconnected ones recursively down the smallest component. Small components can be interconnected to yield larger ones recursively up to a full application component. Note that composition is entirely a run-time activity, which means Espresso projects can scale up indefinitely. Component interconnection requires no source code production, editing, compiling or linking. Connections are entirely dynamic and persistent.

Being Java incarnations, Espresso components can be designed as stand-alone applications or embedded in Web pages. When created as Web applets, Espresso components deliver active content to end-users, with no programming overhead: applet support is provided by the Java run-time environment. Security is controlled by the Java Security Manager. Espresso components can also be packaged in special *envelopes*, for delivery over the Web and installation into user repositories. HTML-document-embedding and Web delivery are essential properties of Espresso components, because they enable components to do away with the need for a proprietary distributed object database, in favor of standard technologies such as http, wide area search engines and Web browsers.

4 Espresso Diagrams

The structure of an Espresso component, as well as the structure of a graph of components, can be expressed both graphically and textually. Graphical presentations use the Espresso Notation, in which parts and relationships between parts use a set of standard symbols. Textual presentations use the Espresso Language to describe components and graphs of components.

4.1 The Espresso Notation

Espresso components are denoted by boxes. Connection points with the outside world are called *ports*, and denoted by an arrow. The notion of port as a component connection point is not new [9], but Espresso ports

possess qualities that differentiate them from older types of ports. Espresso ports are intelligent, in the sense they contain methods a tool can call to get a description of every port feature at runtime. They also act as contract advertisements, governing the behavior not of an entire component, but of collections of logically related methods. Contracts as software interface specifications are well defined in the literature [10] [11]. All Espresso components possess one fundamental port: the `Descriptor Port`, which provides Application Builder tools with connectivity data, versioning info and meta-data regarding internal structure and internal layout. Program control flows in and out of a component through ports. The direction of flow is denoted by an arrow. Program control flow shows the direction of method calls. Data may or may not flow in the same direction as program flow. Figure 1 shows a simple Espresso component.

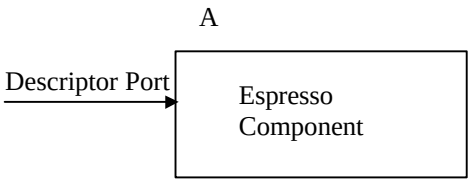

Figure 1 - A simple non-graphical Espresso component.

Components are named with a label. When the label is outside the box, it appears usually above the upper left-hand corner. What appears inside the box depends on the Espresso component. Graphical components, like buttons and spreadsheets, can show their runtime presentation. Figure 2 shows a simple user interface component.

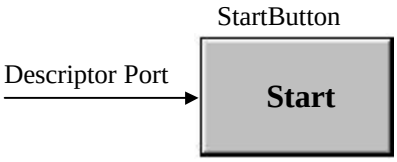

Figure 2 - A graphical component at design-time.

Non graphical components can display a graph of the components aggregated inside them. Figure 3 shows a non-graphical component containing 3 components.

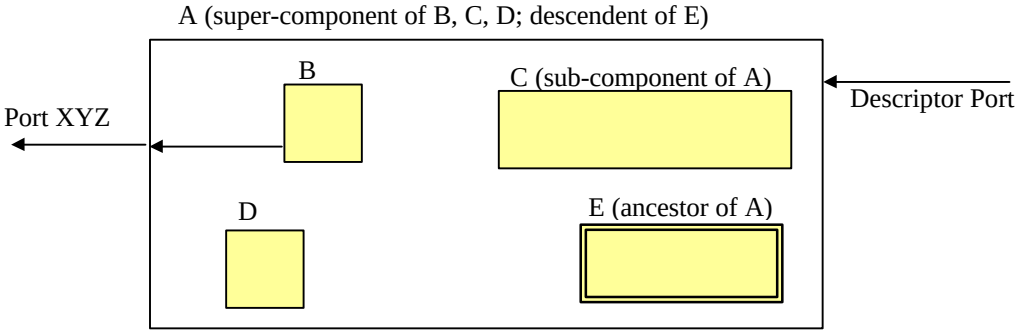

Figure 3 – A composite Espresso component.

Components placed inside another component are called *sub-components*. The outer component is called the *super-component*. When one component is derived from another, the former is called the *descendent*, the latter the *ancestor*. Ancestors are drawn inside their descendent with a double border. Espresso does not support multiple inheritance at the component level. Components are not required to have an ancestor, and may have no more than one.

The display of an Espresso component is the responsibility of SCEDE tools, so the Espresso Notation must be known by tool developers. Espresso components can give information to tools regarding the layout and interconnection of their sub-components and ancestor. Layout information is provided in the form of a graphic representation, including the size and position of each component, the paths of their connecting lines, the position of their ports, plus embellishments such as textual annotations and embedded pictures. Figure 4 shows an example of an Espresso component displayed with this rich formatting information.

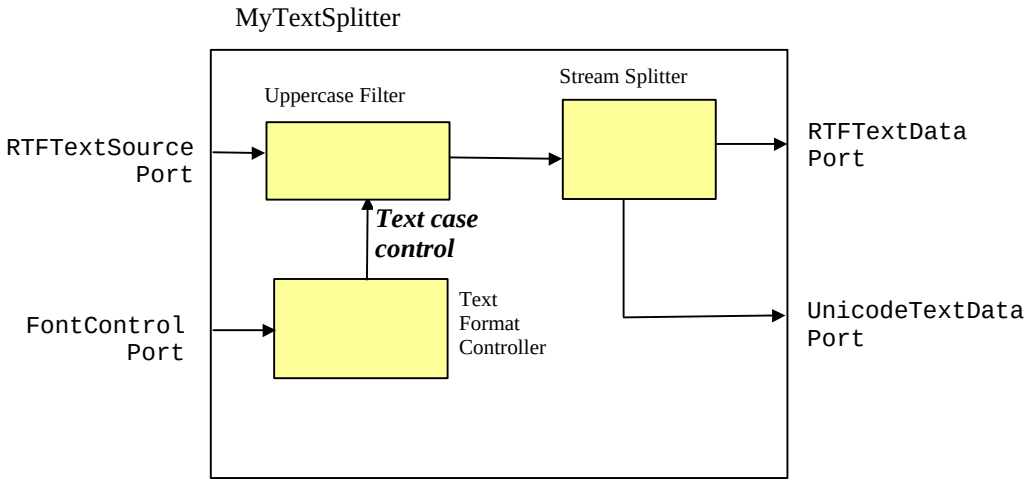

Figure 4 – A richly formatted Espresso presentation.

Text and pictures may be embedded arbitrarily in an Espresso component. Component boxes may be filled with a solid color or a background image. It is important to emphasize that all the information required to display the component in Figure 4 is extracted from the component itself, with no intervening help from external descriptor files. The Espresso model doesn't prescribe a particular structure for SCEDEs. It only guarantees that information required to reconstruct the visual representation of a component will be available from the component at run-time in a standard format.

4.2 The Espresso Language

Graphical Espresso representations are designed for human consumption. For complex graphs of components, it is sometimes more convenient to have a word description. We are developing the Espresso language to be a simple a textual representation of the structure of Espresso components and the structure of Espresso graphs. The language is meant to convey the structure – not the visual layout – of Espresso systems. In this sense, the Espresso Language conveys only a subset of the information displayed by the Espresso Notation, which conveys rich layout information. The Language is designed as a sub-product of the Espresso Notation, primarily for the benefit of Application Builder tools like verifiers and testers, for which textual data is much easier to process than graphics. Although it is possible to translate an Espresso Language script into a complete functional component graph, the lack of layout information would make such a system difficult to use in a visual programming sense. We anticipate the use of Espresso Language as one-way only: language scripts will be generated from an Espresso component, and not vice-versa.

The Espresso Language conveys two basic types of information: the internal structure of a component, and the structure of a graph of components. A number of keywords are defined, such as `Component`, `Port` and

`Line` to denote standard items in an Espresso graph. The following is a fragment of the Espresso Language script that defines the component shown in Figure 4.

```
Component MyTextSplitter: TextSplitter
  Port RTFTextSource: RTFInputPort
    Line void setRTFText(String theText): Input
      Interlocutor: void
    …
    Line Integer getBufferSize(): Input
      Interlocutor: void
  End
  Port FontControl: FontControlPort
    …
  End
  …
End
```

Figure 5 – A sample Espresso Language fragment.

A complete description of the Espresso language is beyond the scope of this paper, however the language uses a simple syntax, is easy to read and is entirely descriptive in nature. The keyword `Interlocutor` used in Figure 5 denotes the component, port and line to which a line is connected. Espresso Language scripts describe structure and relationships – not function. The language is not a full-fledged programming language: there is no support for conditional control, variables, loops or other features available in traditional programming languages. The Espresso Language describes a *component map*, showing how components are interconnected, not what they do or how they work.

5 Connecting Components Together

Espresso components use the concept of port to represent connection pathways with the outside world. Ports are groupings of items called *Lines*. Lines can be used for input or output and encapsulate individual method calls into or out of a component. An Input port contains only Input Lines, an Output Port only Output Lines. A port containing both Input and Output Lines is an I/O port. Ports contain groupings of Lines associated with logically similar methods, i.e. methods that cooperate to support a non-trivial feature.

All Espresso components contain a basic port called the `Descriptor Port`, an Input Port containing methods that provide versioning, connectivity and manufacturer information. Ports are the sole connection point between Espresso components. Ports link components dynamically, providing at runtime the same service as a traditional linkage editor. Using a tool, components can be connected together by their ports at run-time, with no need to modify, compile or link any code.

Output ports possess an attribute called *cardinality*, which describes the number of inputs it can control. Ordinarily, outputs have a cardinality of 1 or greater-than-1. The former ports are called *unicast*, the latter *multicast*. When a multicast port is connected to multiple Input Lines, control is transferred sequentially from the Output Lines to each of the attached Inputs. Ports are described using a formal notation [12], that characterizes the sequence of control flow over its Input and Output lines, timing constraints, pre/postconditions, and other information that allows a tool to verify the correctness of a port interconnection.

Developers can use an Application Builder tool to connect ports together with a drag-and-drop mouse gesture. To make the connection, the tool can interrogate each port participating in the connection for its layout, and the precise description of each of its lines. The tool can then *tell* the ports to connect themselves. Connecting two ports entails the interconnection of all compatible lines on each side of the connection.

Input Lines are implemented as references to Java Method objects. Output Lines are implemented as references to Input Lines, as shown in Figure 6.

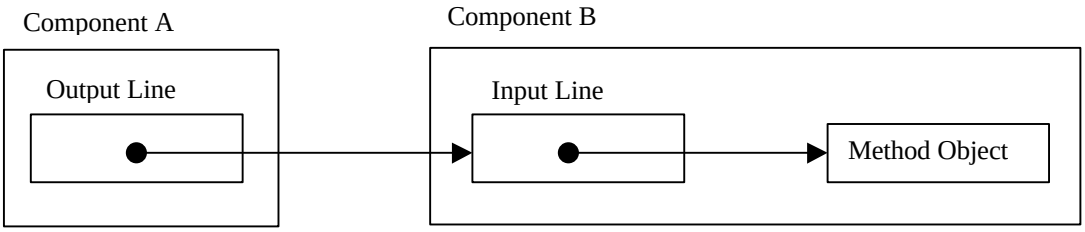

Figure 6 - The interconnection of an Output to an Input Line.

Outputs can be connected to nothing. Using a port with unconnected outputs may or not be allowable, as determined by the owner component. When not allowed, the component may throw an exception or generate an error when the line is accessed. The notation used to indicate two connected ports is a simple line, as shown in Figure 7.

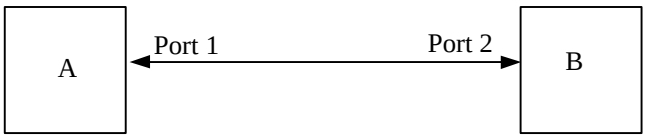

Figure 7 - Two interconnected ports.

Arrows indicate the nature of control flow over the interconnection. The line indicates two things: that the two ports are completely compatible with each other, and that all the lines of one are connected to lines of the other. Partially compatible ports will have a subset of their lines connected. To indicate which lines participate in the connection, a *Port Breakout* notation is used, as shown in Figure 8.

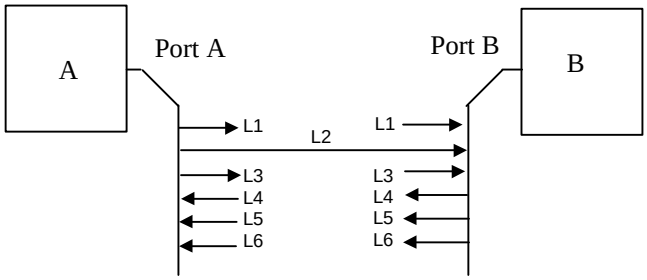

Figure 8 - The Port Breakout notation.

Figure 8 shows two ports with only a single line (L2) connected. Arrows indicate the direction of each line. The port breakout diagram for two compatible ports interconnected would show all the lines of one port connected to lines of the other.

An important property of ports is polymorphism. If an output is connected to an input, the output doesn't *know* which actual `Method` will be called. The Input Line can map the call to any `Method` object, and if the internal structure of a component is changed, mappings between Input Lines and Method Objects may change, but the process is entirely transparent to any connected Output Lines. Polymorphism is an important property, because it allows components to be insensitive to the internal structure of other components. If the internal structure of a component is changed to the extent that it no longer supports a given Input Line, then any connections to that Line are automatically removed.

Because ports represent contracts, they can also encode specific *contract patterns*. For example, many transactions between components are asynchronous by nature: One component `A` issues a command `C1` to another component `B`, then continues with its internal activity. When `B` has completed the command, it notifies `A` by sending the command `C2`. This arrangement encodes the pattern of a *feedback loop*, where the commands `C1` and `C2` are inseparable. Espresso encodes such a pattern in an I/O port that appears as in Figure 9, using port breakout notation:

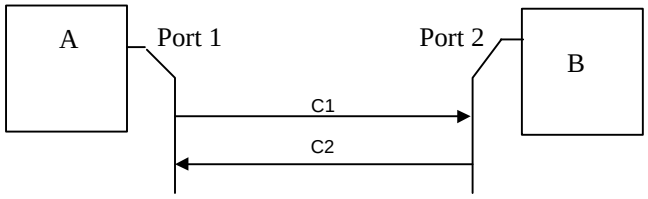

Figure 9 - An I/O port used to encode a feedback loop pattern.

Other lines may also participate in the feedback loop connection. Multiple feedback loops might also be encapsulated in a single pair of ports.

The ability of ports to contain self-descriptive information allows an Application Builder tool to ensure *Connectivity Integrity*, i.e. the verification and validation of the connectivity of a graph of Espresso components. Compatibility between connected lines guarantees that parameters are passed and returned correctly both in type and number. For testing purposes, debuggers also have the ability to place special Watch objects on a connection, to monitor the data flowing over it.

6 Component packaging

How a component is packaged conditions the way it can be distributed. A *transparent* package, which exposes all the internal features, is not acceptable because it provides no protection for intellectual property associated with the component. A successful packaging scheme must allow component developers to expose only as much information from a component as they wish.

Espresso allows developers to create multiple versions of a component, each exposing a different amount of information. For example, a manufacturer might distribute a free version of a component openly on the internet, while a second version might be available for a fee. The first version would expose little or no information about the component's internal structure, the second might provide a full graphical representation of sub-components used, and also provide source code and other debug support information to aid in the reuse of the component.

Espresso components generally contain graphs of sub-components, which in turn contain more sub-components. Because there is a certain overhead in each method call made over a port connection, a complex component may incur a non-negligible amount of overhead. To alleviate this problem, the component's structure can be *flattened*, meaning a designated set of its sub-components, including all of them, can be recompiled into a single component. Method calls across port connections are replaced with direct Java method calls, removing all port overhead.

Flattening has the side effect of removing Espresso structure, so it is advantageous for deployment, but not for development. Flattening is generally a one-way technique used to pass from the development to the deployment phase. Flattening requires that all the source code for the sub-component in the flattening set is available, because it must be extracted and rearranged. Except for their external ports, flattened components contain only source code, so they become *opaque* in Espresso terms. Opaqueness can be a desirable feature, when a manufacturer wishes to provide a component version that has full debugging support without giving

away the details of its internal sub-component structure. Flattening is also useful for components for which performance and/or size is a premium.

7 Packaging Existing Code

It is unrealistic to believe a complete software system can be built exclusively with pre-built components from a library. Digital designers use libraries of integrated circuits, but most circuits contain additional logic – often called *glue* -- implemented using lower level components like resistors, transistors and capacitors. Espresso supports the same concept, by allowing developers to write their own source code and package it as a component, called a *User Component*. These components are indistinguishable on the outside from ordinary Espresso components. The difference is on the inside: rather than contain a graph of sub-components, they simply contain compiled code. User components are the escape mechanism that frees developers from total dependency on pre-built components. No matter how rich a library is, there will always be the possibility that a development scenario will need logic that no component supports.

To develop a User Component, the developer derives a class from the Espresso superclass `EspressoBean`, provided in the Espresso toolkit. This class provides support for the basic Espresso features, such as ports, sub-components and ancestors. Using a Java compiler, the code is compiled into an Espresso component, which can then be installed into the user's component repository and used like any other component, including being redistributed over the internet and debugged using Espresso tools.

8 Debug support

The Espresso model allows code debugging using traditional techniques. Espresso supports a special *Debug* component that contains all the information required by debuggers, including a component's source code and symbol table. The component to which a Debug component is attached is called the *owner*. Because Debug components are useful only before deployment, we have designed them to be optional: they can be attached or detached from their owner. Being able to detach them before deployment is important because Debug components tend to be large, often larger than their owner. They also carry information that the manufacturer may not wish to distribute.

9 Espresso Development Environments

We are designing a development environment for Espresso components, called Software Component Engineering Development Environment (SCEDE, pronounced *seed*). It is a graphical environment that promotes a visual programming style. Developers select components from a tool palette and drop them in a Layout Editor. Components are connected together by using a tool to draw a line between their ports. Existing visual environments, such as the IBM VisualAge Parts system [13], rely on code generation to build and interconnect components. The Espresso model supports incremental component composition without code generation, eliminating the need to compile and link code. Figure 10 shows what an early prototype of the Espresso SCEDE looks like.

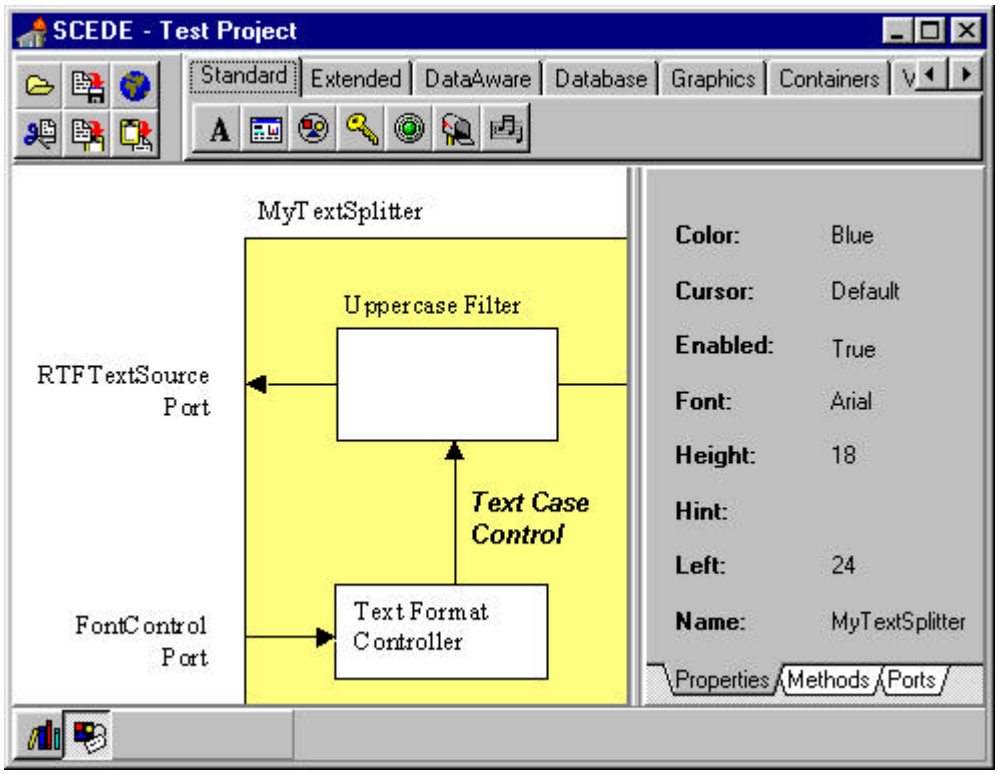

Figure 10 - An early prototype of the Espresso SCEDE.

When components are placed in the Layout Editor, they become live: they process data passed to them and support connections with their neighboring components. Graphical components also maintain a screen appearance that is similar to their final appearance at runtime.

The early SCEDE does not directly support the generation of User Components. Those must be created using Java code, then compiled with an off-the-shelf compiler. Once compiled, User Components can be used as regular components, including installation in the Component Repository and on the Component Tool Palette.

10 Espresso Phases

Projects developed using Espresso have 3 phases: development, testing and deployment. Components have internal states to facilitate software production. During development, components assume the *design-time* state. In this state, visual components appear as they will at run-time. Non-visual components appear as boxes whose interior is filled with the graphical layout of their internal components – when this information exists. At design-time, components are live: they process data fed to them and produce output on the screen where necessary. In the design-time state, the developer can visually interact with components to reposition and resize them. Components can also be customized through their properties and port connections.

During testing, components are switched to their *run-time* state. In this mode visual components assume their run-time appearance and non-visual components disappear from the screen. Components are obviously live in their run-time states. Components are switched from one state to the other by the SCEDE. In the run-time state, components are not allowed to be repositioned or resized by the SCEDE. Properties and port connections are frozen in the run-time state: they can be changed programmatically by the running components themselves, but not interactively using the SCEDE tools.

11 The Espresso Repository

If the goal of Espresso is to build the infrastructure for component-based development, a prerequisite to Espresso-based software development is the availability of many types of components, numbering in the hundreds. We use a Component Repository to manage these components. The repository is a hierarchical structure of components, whose structure is user-defined. Users can classify components anyway they find convenient. A Button component might be classified as a User Interface component. If a user has many dozens of user interface components, it might be advantageous to create different categories under the User Interface category, such as Standard, Extended, Database Controls, etc. Figure 11 shows the hierarchy for a simple repository.

```
Component Repository
        User Interface
                Standard
                        Button
                        BlueButton
                        Listbox
                        Icon Listbox
                        Grid
                Database Controls
                        Edit
                        Label
                        Listbox
        Security
                User Authentication
                Transaction Control
                Encryption
```

Figure 11 - The hierarchy of a small Espresso repository.

The SCEDE we are developing possesses a simple Component Repository. It allows components to be installed using drag-and-drop gestures. Components in the repository can be browsed, and their properties can be adjusted. Components stored in the repository are serialized components. Loading a component from the repository entails deserializing it and instantiating it in the SCEDE Layout Editor workspace.

An important activity in component-based development is searching and finding components that satisfy certain user criteria. We support the distribution of Espresso components over the internet. By packaging them in special constructs called *Envelopes*, Espresso components can be embedded on Web pages called *Spec Sheets*. The role of Spec Sheets is to contain searchable keywords that allow users to locate components of interest. Figure 12 shows the basic layout of a Spec Sheet.

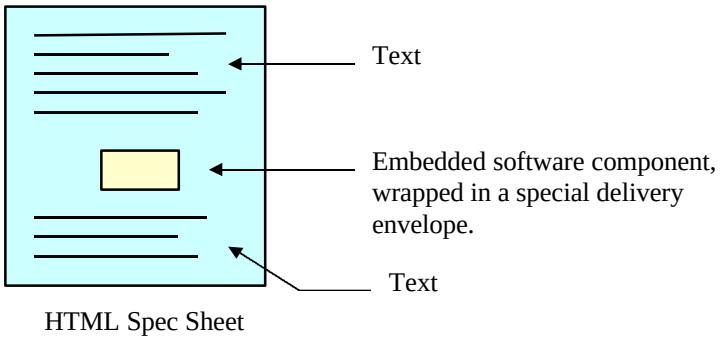

Figure 12 - A spec sheet with an embedded software component.

Search results returned by a standard search engine are normally ranked according to the match score of the text query. By searching for combinations of software component ports or other salient features, components can be located rapidly that support the required functionality.

Developers download components by clicking on their envelope. The envelope contains the component and a description of the envelope contents. Multiple components can be downloaded in a single envelope with a single http transaction. The SCEDE opens the envelope, extracts the components from it and installs those components into the user's Component Repository.

Developers can also create their own Spec Sheets to make their components available on the Web. An Envelope generator in the SCEDE packages components together and creates an envelope for them. The Spec Sheet is developed using traditional HTML document editors, and a hyperlink is embedded to the Espresso envelope. Using the Web as the distribution medium for Espresso ensures the widest availability possible today.

12 Conclusion

Espresso builds on many of the new strengths of the Java language, such as serialization, properties, and run-time meta-data. Java not only makes the Espresso implementation simpler, but also provides a measure of platform independence. The Espresso model provides a means to accelerate and simplify the software development process. Success of the component-based development model is predicated on the availability of a substantial number of pre-built components. Our current effort has the goal of creating a minimal set of components, to demonstrate the overall capabilities of the model, but we intend to subsequently embark on the development of a much larger set.

References

1 Kaiser, G., Feiler, P., Popovich, S. Intelligent Assistance for Software Development and Maintenance, IEEE Software, May, 1988, 40-49.

2 Chang, S. et al. Principles of Visual Programming, Prentice Hall, 1990.

3 Drinnan, A., Morton, D. Advantages of a Component-Based Approach to Defining Complicated Objects, OOPS Messenger, Jan 1993, 36-45.

4 Fischer, G. Domain-Oriented Design Environments. Proceedings of the 7th Knowledge-Based Software Engineering Conference, 204-213.

5 Robbins, J., Hilbert, D., Redmiles, D. Extending Design Environments to Software Architecture Design, Proceedings of the 11th Knowledge-Based Software Engineering Conference, 63-72.

6 Benner, K. Addressing Complexity, Coordination and Automation in Software Development with the KBSA/ADM, Proceedings of the 11th Knowledge-Based Software Engineering Conference, 73-83.

7 Delphi Technical White Papers, Borland on-line document, http://www.borland.com/delphi/papers/.

8 Visual Basic 5.0 Evaluators Guide, Microsoft on-line document, http://www.microsoft.com/visualtools/egs/visualbasic.htm.

9 Nierstrasz, O., Gibbs, S., Tsichritzis, D. Component-Oriented Software Development, Communications of the ACM., Sept 1992, 160-165.

10 Helm, R., Holland, I., Gangopadhyay, D. Contracts: Specifying Behavioral Compositions in Object-Oriented Systems, Proceedings of OOPSLA ECOOP '90, 169-180.

11 Meyer, B. Object-Oriented Software Construction, Prentice Hall, 1988.

12 Faison, T. Formal Component Interface Specifications in Espresso, Sept 1997, on-line document, http://www.faisoncomputing.com

13 Visual Builder Parts Reference, IBM on-line document, http://as400bks.rochester.ibm.com/cgi-bin/bookmgr/bookmgr.cmd/books/cppvbr00/contents.