Interactive Component-Based Software Development with Espresso*

Ted Faison
tedfaison@msn.com
Faison Computing Inc.
Irvine, CA 92606-8896

Abstract

Most component models in use today are language-independent, but also platform-dependent and not designed specifically to support a tool-based visual development paradigm. Espresso is a new component model that was designed with the goal of supporting software development through tool-based visual component composition. Being implemented in Java, Espresso components can run on any Java-enabled platform.

1 Introduction

Component-based software today is largely based on language-independent models that require code-intensive work to produce programs. Espresso is a new component model based on the Java Beans specification, designed specifically to support a tool-based approach to software development, where the building blocks are binary (Java bytecode) components. Java was chosen because it supports at the language level several important features, such as serialization and meta-data. Espresso components support a visual composition development model: components are connected together in a graph using mouse gestures, without code generation. Espresso components can also be nested, allowing a graph of components to 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. Espresso component graphs are built interactively using *live* binary components, supporting an incremental development model where everything is live during the design phase.

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 and applications as graphs of components. The emphasis is shifted from code production to component composition.

The Espresso model wasn't developed for a specific domain. Espresso is not the component equivalent of a domain-specific framework, but a generic model. Domain-specific issues can be handled using specialized Espresso component libraries.

2 Interactive Components

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 without user interaction. Non-graphical components are useful for incorporating business logic into an application, and include objects like component collections (e.g. vectors), event timers, application-specific algorithms and adapters.

An Espresso Application Builder tool can create both a GUI application, such as the client-side front end of a multi-tiered database information system, or a non-GUI one, such as a matrix inversion algorithm.

Two types of developers are envisioned to be typical Espresso users: component producers and consumers. Producers develop components for distribution or resale. Consumers are typically application developers who use interconnect pre-packaged or user-developed components to implement more powerful components or complete applications.

Component interconnection requires no source code production, editing, compiling or linking. Connections are made through persistent references. The time required to

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

generate a connection between components is not project-size-dependent.

Being Java incarnations, Espresso components can be designed as stand-alone applications or embedded in Web pages. Espresso components can also be packaged in special *envelopes*, for delivery over the Web using standard browsers. Envelopes can be used for commercial delivery of components, and support security through code signing and optional encryption. Developers can also package their own algorithms and source code into special Espresso components, called *User* components. These components contain compiled Java code, and are the escape mechanism that frees developers from total dependency on pre-built components. From the point of view of an Application Builder, a User component looks and acts like any other Espresso component.

Espresso components are denoted by boxes. Connection points with the outside world are called *ports*. The notion of port as a component connection point is not new [1], 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, describing the behavior not of an entire component, but of collections of logically related methods. Ports are different from traditional interfaces [2][3] in that they can contain not only inputs, but also outputs, which can be hooked up to inputs of other components. An arrow denotes whether a port is an input, an output or is bidirectional. Espresso ports are described using formal specifications, allowing a tool to verify the validity of an interconnection. The specification describes the sequence of control flow in the port, pre and post conditions, timing constraints and the parameter types related to each input and output.

Components can also contain sub-components. Figure 1 depicts a component with 3 sub-components.

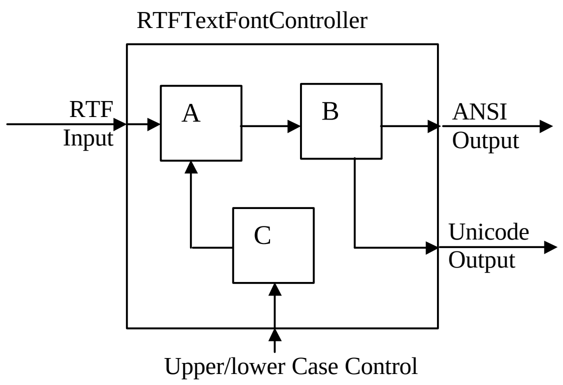

Figure 1 - A simple Espresso component.

Two ports are said to be *compatible* if all of the inputs of one are compatible with the outputs of the other. Compatibility is determined by an analysis of the formal port specifications. Connecting compatible ports together connects all the inputs of one with the corresponding outputs of the other.

Labels, notes, comments, annotations and pictures may be embedded arbitrarily in an Espresso component. All the information required to display the component in Figure 1 is extracted from the component itself, with no intervening help from external descriptor files. 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. Espresso components can optionally be distributed with an embedded *Debug* component, which holds the full source code, symbol tables and other information required to debug components at the source code level.

The Espresso model doesn't prescribe a particular structure for Application Builders. 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.

Experience results are not available yet for complete Espresso-based systems. We are currently in the process of creating a small library of components, to test crucial aspects of compositional component-based development. Important issues still being studied are port-interconnection runtime overhead, formal port specifications, tool-based automatic validation of component interconnections and component-based debug support.

3 Conclusion

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, they are not suited for real-time applications.

Space doesn't allow us here to describe in more detail many features of Espresso, such as Application Builder support, the Espresso repository and component distribution over the Web. An on-line paper [4] is available that gives an overview of the current state of Espresso.

[1] Nierstrasz, O. et al. Component-Oriented Software Development, Communications of the ACM, Sept 1992, 160-165.
[2] Meyer, B. Object-Oriented Software Construction, Prentice Hall, 1988, 183-215.
[3] Rogerson, D. Inside COM, Microsoft Press, 1997, 15-34.
[4] Interactive Component-Based Software Development with Espresso, on-line document, http://www.FaisonComputing.com/Espresso.html.