Software Farm, Inc.
211 HighView Dr.
Boulder, Colorado 80304
303 546-6503 FAX: 303 546-6708

A Framework (FRED) and Framework Language (ETHYL)

Michael L. Davis

Software Farm, Inc. (mdavis@csn.net)

September 22, 1995

Note: This is a system that is very much *in-progress*. As such this paper is being distributed to a very small number of people in order to gather feedback and to provide a 'heads-up' about what the FRED&ETHYL product from Software Farm is all about. This product was named VisualADE 3.0 but sounded so much like IBMs visualAge that we are looking at using other names.

Abstract

As the industry matures the feature set of the average software application becomes more well-defined. This allows frameworks to be written that supply these features. Usually these frameworks are accessed by using 3GL languages and are therefore difficult to use. This paper decribes a scalable architecture, framework and 'framework language' that 'configures' or 'programs' a framework at a very high-level of abstraction. This 'framework language' is what is then used to do the work to connect the features together to create an application.

As usual with high-levels of abstraction, productivity increases come at the expense of generality. For the (very) expert programmer customization is possible by modifying (deriving specialized classes from) the framework source code. However, as the quality of the framework improves over time this will rarely be necessary.

Keywords - Frameworks, User Interface, User Interface Management System, Framework Patterns, Framework Configuration Languages

1.0 Introduction

The holy grail of software programmers is to create a software development environment so advanced and so easy to use that many of the tedious and/or repetitive details are avoided and the process of developing applications becomes exclusively one of creative customization.

This complements the popular notion of the *software IC*. A set of *Software ICs* is like an automobile parts catalog. A customizable feature-full framework is like a factory-built automobile. Many people find that cars are easier to build by purchasing one that is already built and swapping out parts as opposed to building one from parts out of the parts catalog (which is admittedly better than starting with a forge and hammer and file).

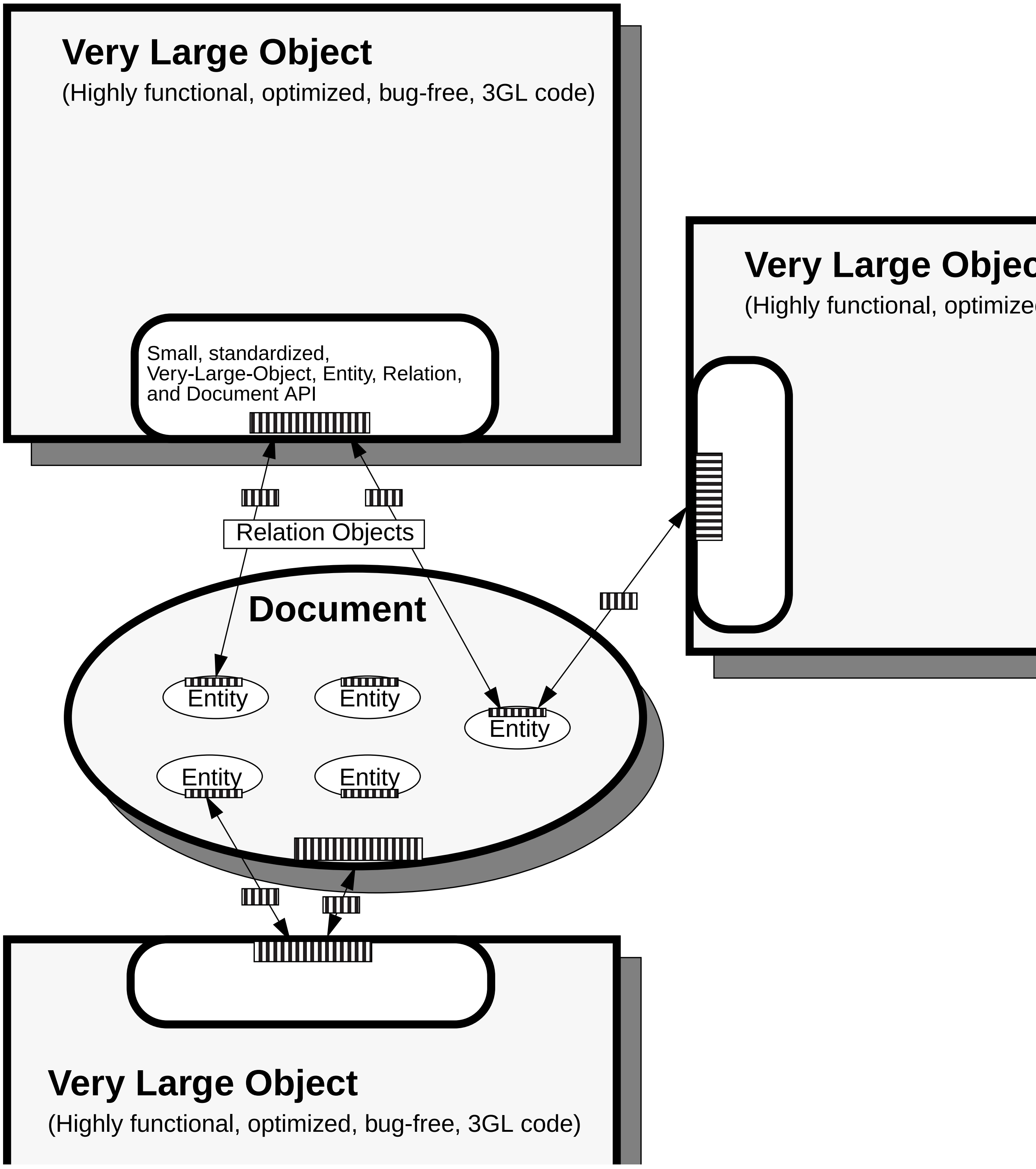
Very Large Object
(Highly functional, optimized, bug-free, 3GL code)
Small, standardized,
Very-Large-Object, Entity, Relation,
and Document API
Relation Objects
Document
Entity
Entity
Entity
Entity
Entity
Very Large Objec
(Highly functional, optimize
Very Large Object
(Highly functional, optimized, bug-free, 3GL code)

FIGURE 1. The Architecture: Overall design

Similarly, many engineers assemble PCs by combining motherboards and power supplies and pre-built containers. They do not usually start with chips and solder and a piece of sheet-metal. For example contrast a graphics accelerator chip with a graphics accelerator card.

There are two general methods used to build things:

- by building something the ground up using small parts, usually used to go beyond the current state of the art and derive 'value' by virtue of the technological advance,
- by connecting together currently available technology to build something useful and which derives its value from the way in which the parts are combined.

The goal of this system is to assist both methods of development by 1) providing the ability to add new technology by extending the system using low-level languages like C++ (assuming that there are *some* parts of the system that are in common with other applications) and 2) providing a small number of (very) large objects that can be customized and tied together in ways that generate different applications. The focus of this system is always on goal #2.

This paper covers the architecture of the system, the language interface to the system, the built-in functionality provided with the system, examples of the use of the system, and finally the conclusion and the description of future work.

1.1 Nomenclature

The terminology of software system architectures is still non-standardized and so the nomenclature used in this paper will be briefly defined. An *architecture* is the high-level design of an application. A *framework* is the implementation of an architecture. The purpose of an architecture and framework is to organize software into functional groups and to provide functionality that is 1) common a all groups with in the application and/or 2) common to a group of applications. An *application* is a group of software components collected together into a unified whole. A *component* is a subset of an application that supplies a single service. The *semantic* component is the part of an application that contains the software related to the purpose of the application. The *presentation* component is that part of the application that supports the user in viewing and interacting with the application. *A functional* component is a semantic or presentation component. An *entity* is a data object. *Relations* are objects that reference a source and destination component. The *system* is the totality of architecture, framework, language and supporting functionality.

2.0 The Architectural Design

This is a data-centric or document/view architecture. It consists of a number of independent *components* and *entities* that are connected together with *relations*. Some of the components are prebuilt and come with the system while others must be written to provide needed functionality unique to a particular application.

It is the supposition of this architecture that many (if not all) applications can be represented, if necessary at a macro level, by this limited set of constructs.

2.1 Components

Components are usually used for viewing and/or editing entities. Components may have Fields exactly like Entities (see below).

2.2 Entities

Entities are the semantic data objects of the application. They contain a list of *Fields*, each of which has a name, type, value, range constraints, and other information. The field type can be a built-in type (e.g.: text, int, float) or a derived application type (called a *FieldDefintion*).

2.3 Relations

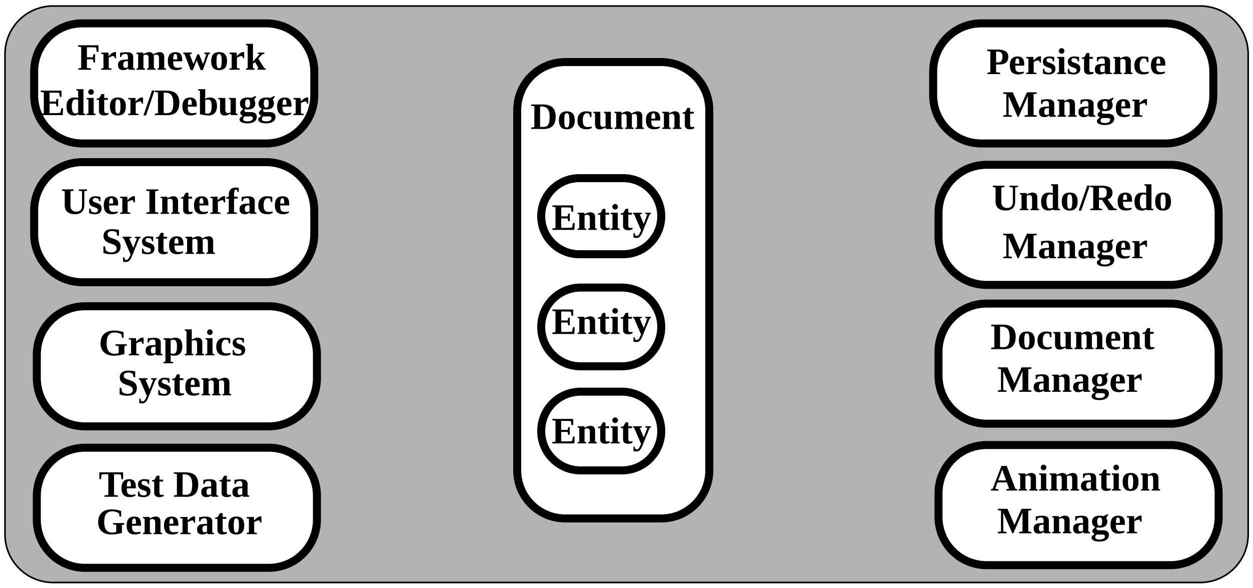

FIGURE 1. The Support System of the Framework

Relations provide the only means of communication and data transfer between components. Relations may have Fields exactly like Entities.

2.3.1 Control Flow

Control flow is the means by which an event occurring in one component causes an event to occur in another component.

The control flow for an application is specified entirely by the MessageFlow relations. (see Figure 1 on page 3). MessageFlow relations specify which messages to send to which components, entities or relations in response to a particular message received. Messages are generated in response to events which are usually, bot not necessarily, generated by the user. Messages are objects that have a name and a list of (untyped) arguments.

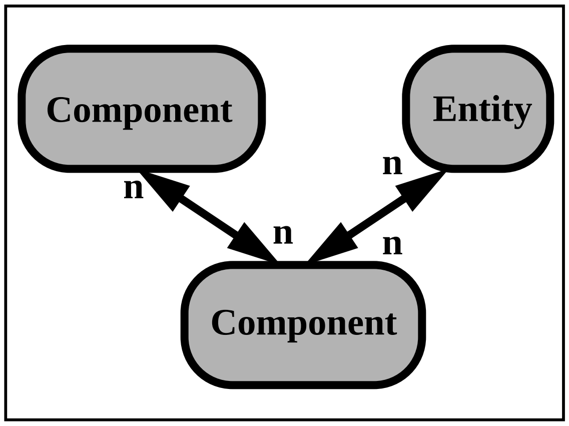

FIGURE 1. MessageFlow relations in the Architecture.

2.3.2 Information Flow

Information flow is the means by which data is transferred between components and is specified by the DataFlow and View relations.

- The Dataflow Relation

The Dataflow relation specifies an bi-directional constraint on the values of two data items. These data items are often Fields of an Entity. However they can also be aggregate collections of data from many entities (for example the names of all students of Profes-

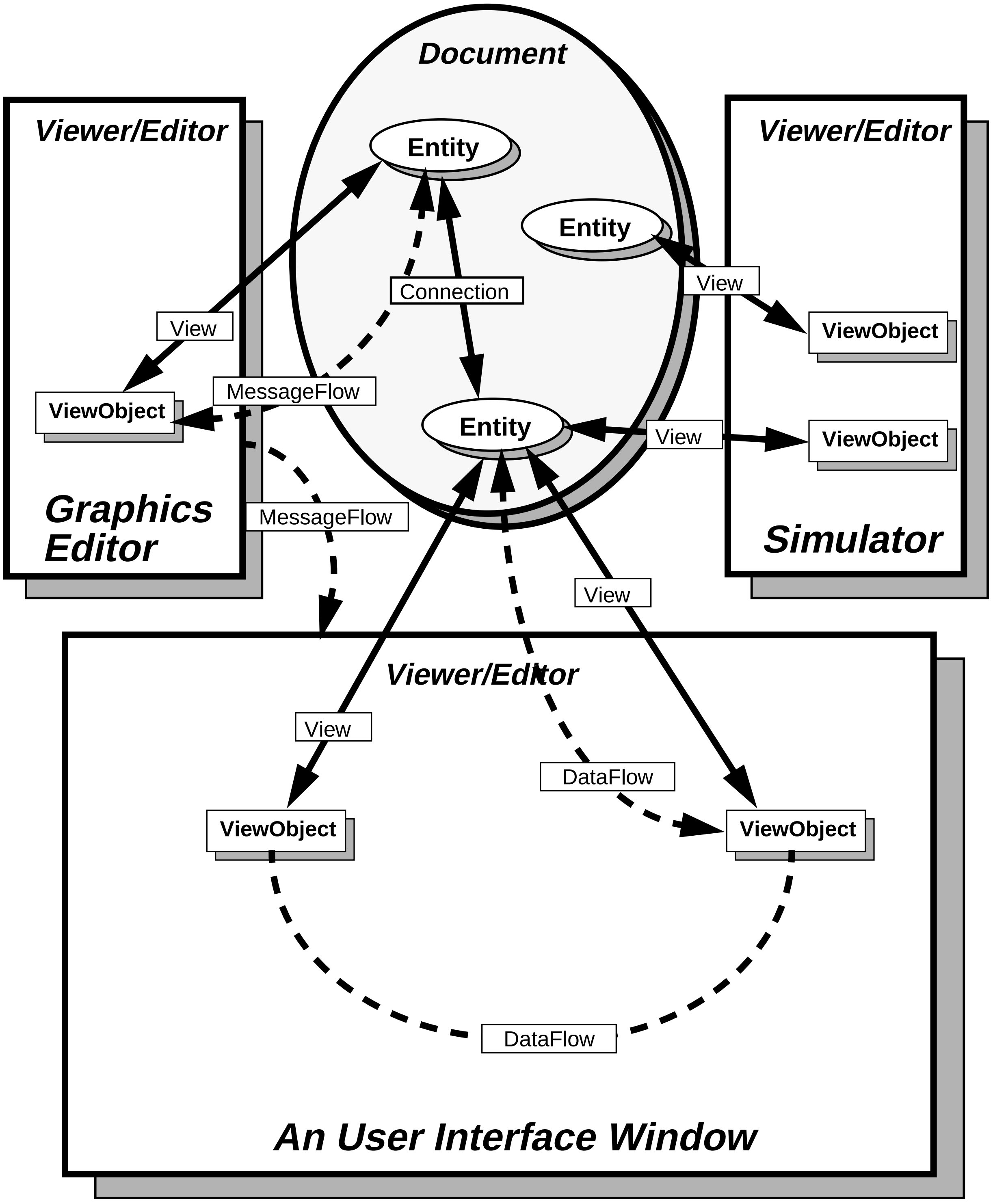

FIGURE 1. The Architecture [illegible]

sor X can have a DataFlow relation with the contents of a GUI scrolledList widget).

If the value of one end of a DataFlow constraint changes then the other is automatically updated by the system. This relation can be constrained to be uni-directional.

A Filter object may be attached to the DataFlow relation. The Filter object performs a test on a set of components at one end of the DataFlow relation to determine which components are to be included as part of the DataFlow relation.

Similarly a Format object may be attached to the DataFlow relation. The Format object extracts one or more values from each component at one end of a DataFlow relation and formats the values into text strings using a ‘C’ language sprintf-like specification.

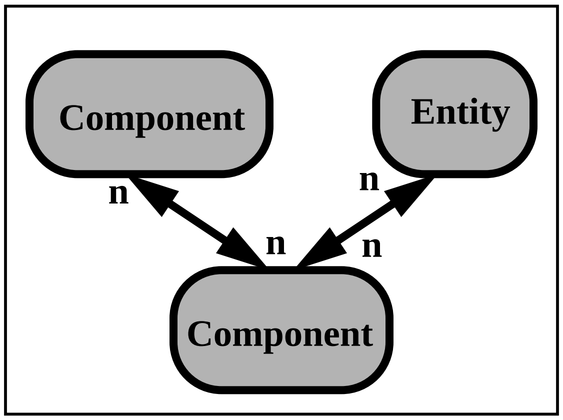

FIGURE 2. DataFlow Relations in the Architecture.

- The View Relation

The View relation assigns a component, Relation or Entity (the subject) to a component. This component then can view and/or modify the subject. Optionally the subject is represented in the component by a different object (called a viewObject). For example an Entity (the subject) may be represented by an icon (the viewObject) in a graphical editor (the component).

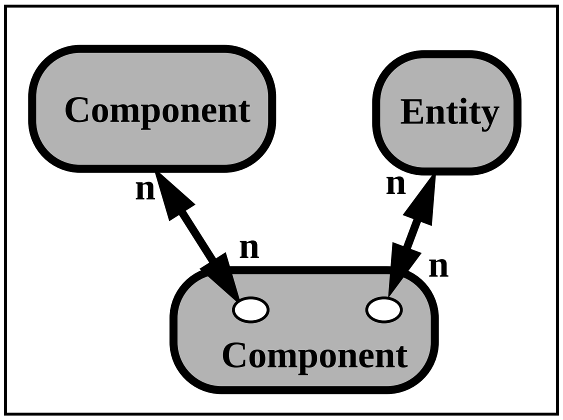

FIGURE 3. View Relations in the Architecture.

2.3.3 Connections

Connections are relations that have a name and a source and destination component. Each connection type also has a list of valid source and destination components. Only components from these lists are able to have this type of connection between them. In this way design rules about connectivity are established and enforced.

3.0 The Support System

A number of pre-built components are supplied/built-in to the framework. Some of these must be explicitly declared by the programmer and some are implicitly always available (these will be indicated by a *).

The built-in system components are:

(see the Appendix for a more detailed description of this support system)

3.1 General

- Application
- Help Manager
- Internationalization Manager
- Document

3.2 The User Interface System

- Window
- AutomaticWindow
- Menubar
- Panel
- Layout

3.3 Graphics

- Editor
- Graphics
- IconWell
- Locator
- Magnifier
- AnotherView

3.4 Animation

- Engine

3.5 Debugging Assistance

- TestDataGenerator
- Debugger

3.6 Programmer Assistance

- Framework Editor

3.7 The Persistence Manager*

3.8 The Undo/Redo Manager*

4.0 The Language

The goal of the language is to be a terse but very readable, high-level but complete, specification for the generation of modern GUI-rich applications. By terse it is meant that the language resembles a programming language more than the English language. By readable it is meant that one does not have to refer to a manual to understand someone elses source code. By high-level it is meant that, as much as possible, only information that is absolutely necessary to create the application is required, not gratuitous information that helps the author of the framework or is required by the underlying toolkits. And by complete it is meant that the language should support all reasonable feature requests. This last is accomplished by making the components large and highly customizable and by allowing for language extensions using C++.

The language is used to specify class definitions, relations and declarative constraints. Instances and instance operations are not referenced directly. The language is compiled at runtime into memory-resident class descriptions, relations and constraints. The file containing source code in this language is called a *description* file.

The language has some basic object-oriented capabilities. Entities support multiple inheritance (virtual base classes, changing of inheritance trees at runtime, and static class data are not supported at this time).

Attribute values are specified by using the keyword-value syntax (i.e. color = yellow). Any name in a keyword-value pair can be an environment variable by prefacing it with a '$' (i.e. color = $BackgroundColor). Unless otherwise indicated, all keywords are case-insensitive.

Lines of text can optionally end with a semi-colon. If a pound sign '#' character or double slash '//' appear anywhere in a line of text, the text to the right of the character(s) is ignored. Quotes '"' may be used for any value and must be used if a value contains a special character (i.e. one of , ,, #, //,), ().

4.1 Classes

Classes are created by specifying the type and name of the class and then the class body:

```
ClassType ClassName
    {
    }
```

Each class type has type-specific attributes, type-specific subComponents and type-specific messages that it responds to.

The available classes are:

- Application
- Editor
- IconWell
- Graphics
- Window
- AutomaticWindow
- Panel
- Menubar
- Engine
- TestDataGenerator
- Debugger
- Connection
- Entity
- Document

4.2 Attributes

Each class can have type-specific attributes assigned values by specifying the *Attributes* of the class:

```
ClassType ClassName
   {
   Attributes(AttributeName = AttributeValue, ...)
   }
```

Each class can have an type-specific subComponent specified in the following manner:

```
ClassType ClassName
   {
   SubComponentType(AttributeName =
        AttributeValue, ...)
   }
```

4.3 Fields

Each class may have fields added by the following method:

```
ClassType ClassName
   {
   FieldType(name = FieldName,
        default = DefaultValue,
        minimum = MinumumValue,
        maximum = MaximumValue,
        step = ValueIncrement)
   }
```

Available FieldTypes are:

- Int
- Float/Real
- Text
- Enum

Available Field attributes are:

- name
- default
- minimum
- maximum
- step
- minimumLength
- maximumLength
- readOnly
- ReadWritePermissions

Any field may be used as a reusable FieldDefintion. For example:

```
ClassType ClassName
   {
   Int(name = MyInteger, default = 0)
   }
ClassType2 ClassName2
   {
   MyInteger(name = MyInteger2);
   MyInteger(name = MyInteger3);
   }
```

Or FieldDefinitions may be defined by the following method:

```
ClassType ClassName
   {
   FieldDefintion(name = MyInteger, type = int,
        default = 0)
}
```

4.4 Messages

Messages are named objects that are sent to components. Messages may contain a list of named arguments and their values. Messages supported by all components are:

- save
- saveAll
- Load
- Refresh
- Delete
- Create
- Copy
- setValue(keyword = value)
- addView(name = viewName)
- removeView(name = viewName)

An example of the *setValue* message is:

```
DestinationComponent.setValue(
        attributeName = attributeValue)
```

4.5 MessageFlows

Each class can have message handling extended by the following method:

```
ClassType ClassName
    {
    MessageFlow(
        message = IncomingMessageName,
        action = DestinationClassType.outgoing-
        MessageName)
    }
```

For example:

```
ClassType ClassName
    {
    MessageFlow(message = Selected, action =
        myWindow.popup)
    }
```

4.6 DataFlows

Each class can have data transfer and data constraints added by the following methods:

```
ClassType ClassName
    {
    DataFlow(sourceClassType.field
        = destinationClassType.field)
    }
```

For example:

```
ClassType ClassName
    {
    DataFlow(.myScrolledList.contents
        = myApplication.contents,
        direction = DestinationToSource,
        filter = myFilter)
        )
    }
```

Where myFilter is defined as:

```
Filter myFilter
    {
    Attributes(
        format =
        %-10.10name    %-40.40description)
    }
```

Conversely, the DataFlow relation may be created by the following method:

```
    scrolledList(name = myScrolledList,
        contents = @(r)myApplication.contents
```

Note that this syntax (the ‘@’) does not provide a way to reference a Filter object. The ‘@’ indicates that the following text is to be interpreted as an address from which to get a value. The optional ‘(r)’ indicates that the constraint is directional and that therefore the referenced address is only to be read, not written.

4.7 Views

Each class can have a view added by the following method:

```
ClassType ClassName
    {
```

```
   View(name = myView, viewer = myView,
        viewObject = myViewObject)
   }
```

For example:

```
Graphics myIcon
   {
   Icon(filename = myIconPixmap.xpm)
   }
Editor myGraphicsEditor
   {
   }
ClassType ClassName
   {
   View(name = myView, viewer = myGraphicsEd-
        itor, viewObject = myIcon)
   }
```

4.8 Connections

Each class can have data transfer and constraints added by the following methods:

```
Connection connectionClassName
   {
   Attributes(
                source = sourceClassName1, des-
        tination = destinationClassName1,
                source = sourceClassName2, des-
        tination = destinationClassName2)
   }
```

5.0 Language examples

The following program does:

6.0 Discussion of the strengths and weaknesses of the Language

6.1 Strengths

Declarative data constraints are easy to use.

Large amounts of graphics user-interface and graphics editor features easily available.

Automatic design rule checking on connections.

Automatic value checking on Entity fields.

6.2 Weaknesses

Uses C-like language structures that some non-programmers may find difficult to learn and use.

Entity-Entity relations are vague. If such relations are specified then the relations are attached to particular entities based on other relations that may already exist between the entities.

7.0 Discussion of the strengths and weaknesses of the Framework

7.1 Strengths

Automatic session management, persistence, and undo/redo functionality are provided ‘free of charge’.

Automatic generation of window contents and layouts for fast prototyping.

Integrated system enables easy generation of test data.

7.2 Weaknesses

Currently the persistent store data format is very dependant on the components and their functionality (i.e. a component removes an attribute or changes an attributes name) as is the Undo/Redo file format (i.e. a component removes a message capability or changes it’s name).

7.3 Evolvability

Any framework implemented today encounters the onslought of continuously changing and evolving standards. Which I18N scheme to support, which distributed object methodology, which version of which toolkits, the list goes on and on. To address this issue the expected evolvablity of the system will be examined.

The evolvability of the system is enhanced if the following can be changed without affecting the other major parts of the system:

- Components and Component functionality
- Language definition
- Framework Editor
- Underlying toolkits

These areas are indeed independent of each other.

7.4 Size and speed of the resulting application

Both speed and size of an application using this framework are necessarily larger than if the application were ‘hand-coded’. However the code size will be smaller if there are a large number of windows which are automatically generated as opposed to being created by a code-generating GUI builder. And the speed will be greater if a highly optimized component is used instead of one hastily thrown together to make a ship date. And finally, size and speed are becoming less and less of an issue at the same time programmer productivity and application robustness are becoming leading concerns of software development.

8.0 Experience

The following sections describe our experience with developing applications using the FRED system.

8.1 Implementation

The core of the framework is implemented in about 5000 lines of C++. The run-time class system and language parser in less than 3000 lines of code. The relatively small amount of code is considered an asset when it comes to understanding and modifying the system.

Subclassing components to add functionality seems to be relatively easy and this will become increasingly more natural as the components mature.

Adding new components is straight forward, though there are quite a number, ten or so, of public functions that must be supported by each component. Some automatic registration functionality would reduce the burden of informing the language parser about the new component. Obviously adding new components is an important feature for healthy programming systems (i.e. Visual Basic VBXs, 3D Studio plug-ins, etc.).

8.2 Application Examples

This section describes the experience of building application using the system.

8.2.1 Ad-Hoc prototypes

Quick prototypes have been created with the system with various levels of success. The quality of the prototype implementation seems to be judged by the quality of a number of specific features:

- GUI appearance
- Graphics appearance

and not by other seemingly unrelated features:

- Context sensitive help
- Syntax checking and validation
- Robustness

Of course this is true of all prototypes and can be addressed by providing a number of examples of attractive GUIs and graphics objects.

8.2.2 A Direct-Manipulation Editor for Displaying

and Editing Diagrams

This application is a composite of the currently popular diagram editors as popularized by Visio and 20 or 30 other similar applications. As such the feature set is pre-determined and the development process becomes one of incrementally adding all of the features commonly associated with these class of applications. The addition of some of these features requires the use of C++ to expand the functionality of one or more components. It is expected that these new features:

- will be valuable to other applications in the future, and
- will reach a critical mass such that new requirements can be satisfied by functionality already present in the system.

This application is currently under development and is being written in the ETHYL language (eventually this may become a component itself).

8.2.3 An animation tool

This application modified the system using C++ to create a subclass of the Engine component (to add animation functionality) and of the Editor component (to add IC-like graphics primitives that represent this application's Entities).

9.0 Related Work

- Script languages (such as TkL and the CDE KornShell) are similar in that there is a language which has to access to large featurefull libraries. However these are largely procedural and lack support for describing the semantic data model. They are also traditionally not very readable.
- User interface languages (such as Motif UIL) specify UI layout and widgets in a declarative manner. However, these are usually quite weak when it comes to specifying what to do when the user interacts with the UI. Also, information about how the widgets are related to each other and how they are tied to the semantic data of the application is absent.
- The OpenDOC architecture, which is a document/view architecture, is similar to this architecture and has much to recommend it. However, it hasn't a high-level configuration language and must be programmed in low-level languages like C++.
- The ODMG specification language of the proposed standard for OODBMSs has many declarative commands that describe how data is related to other data. And similar to ETHYL, it maintains relations between objects automatically. However, it limits itself to this semantic data object domain and the language has a number of awkward peculiarities.
- The SmallTalk language is similar to FRED in that it has a large number of support classes that are configured and customized to develop applications. But the classes are small and there isn't a standard API for all components (beyond the generalized message handling capability). It also lacks a general glue class (i.e. for relations).
- The Abstraction-Link-View architecture [Hill, 1992] uses a sophisticated constraint manager to manage the links between Abstraction (semantics) and View (presentation). Unfortunately control of the application remains in the semantic component.

10.0Conclusion and Future Work

The FRED architecture provides a useful implementation paradigm for almost all programs including those containing graphics, databases, multimedia and external processes.

The FRED framework provides an advanced, easy to access, highly functional feature set available to the programmer using the ETHYL language.

Below is a list of items that are on the to-do list. This is by no means complete but communicates fairly well the directions FRED and ETHYL are headed in.

10.1 Language

10.1.1 Built-in and custom type-convertors.

Type conversion occurs when a DataFlow relation connects Fields of different types. At the present time both types at the end of a DataFlow relation are converted to text strings when assigned to one another. Type convertors would also be used to specify a range-map feature. This feature maps ranges in an input value to output values (for example 0.0-0.2 maps to blue, 0.20-0.40 maps to green, ...)

10.1.2 Views for individual Fields.

The usefulness of this feature becomes more clear when one considers views being assigned to Field-Defintions. For example one could specify that all integers will be mapped to slider widgets in a particular type of Panel and mapped to graphical dials in a particular graphics Editor.

10.1.3 Add more data-manipulation features to the TestDataGenerator

Add genetic algorithm capabilities to the TestDataGenerator so that the user or programmer can interactively evolve test data sets. The generation of directed-acyclic-graphs, of sample data-sets, of window layouts using genetic techniques are all useful pursuits in and of themselves. Giving the programmer access to these features for testing and for inclusion in the resultant application is expected to be quite useful.

10.1.4 Add general cross-platform Drag-N-Drop feature

The IconWell currently uses some Drag-N-Drop terminology and the goal would be to expand this so that every object in the system is able to be a Drag-N-Drop source and/or target.

10.1.5 Support for editing multiple documents simultaneously.

There is not now support for this feature.

10.1.6 Resolve Undo/Redo effects in different windows.

Undo/Redo is an application wide feature. If a user makes a change to window A, then a change to window B, then presses the undo button in window A, the undo will occur in window B, as it was the last operation executed.

10.2 Standards

10.2.1 Support for distributed objects

Integration with distributed object standards (Corba/OLE) using relations that span processes. The fact that only relations link components together and that components support a standard API will make this relatively straight forward and invisible to the programmer/user of the system.

10.2.2 Support for commercial databases

Support for commercial OODBMS and RDBMS products, perhaps by subclassing off of Persistence Manager.

10.2.3 Support for commercial GUI builders.

This is an enhancement to the system as well as the language so that code and/or UIL generated by commercial GUI builders can be referenced in the language just like widgets generated inside the language.

10.2.4 Port to MS Windows 95 and NT.

This will increase the commercial viability of the system.

10.2.5 Support for Object Embedding

Support OpenDOC/OLE-like capability that allows components to be associated with other applications which are able to take over parts of a window in a FRED application.

10.3 Graphics

10.3.1Support for functionality implemented but not accessible by the ETHYL language.

Adding commands to the language to provide an interface to instance calls and the graphical layout primitives, as well as the magnifier and otherView editors.

10.3.2Adding 3D (probably OpenGL) support to the graphics system.

10.3.3Add animation commands to the language.

10.3.4Add Video and sound capabilites to the system and language.

10.4 User Interface

10.4.1Support for more layouts in the AutomaticWindow component.

Only layout type #1 is currently supported.

10.4.2Add support for option menus and popup menus to the systems widgets.

10.5 Data Management

10.5.1Support caching of persistent data.

Currently all data is read into memory upon application initialization. This feature would have data be memory resident only if it is absolutely necessary.

10.5.2Support for importer and exporter components

10.6 Programmer Support

10.6.1Develop visual framework editor.

Next paper.

10.6.2Integrate framework editor and debugger.

10.7 Intelligence

10.7.1Heuristic rules which *suggest* operations.

10.7.2Using anthropomorphic agents to assist the user in these tasks.

10.7.3Automatically generated to-do lists for various task flows.

11.0References

Hill, Ralph D. (1992). The Abstraction-Link-View Paradigm: Using Constraints to Connect User Interfaces to Applications, *ACM CHI Conference Proceedings*, pp 335-342.

12.0Appendix 1: The Support System Reference

12.1 General

12.1.1Application

The Application component contains information about the application as a whole.

12.1.2The Help Manager SubComponent

The Help Manager provides support for context-sensitive help-key driven and/or balloon help for all widgets in the application.

12.1.3The Internationalization Manager SubComponent

The Internationalization (I18N) Manager provides support for I18N for the application.

12.1.4Document

The document is the container of a group of Entities and their persistent store and undo/redo files.

12.2 The User Interface System

12.2.1Window

The Window component provides the functionality of a User Interface Window. An example of a Window component is:

```
Window myWindow
   {
   Attributes(Title = my Window Border Title,
        menubar = myMenubar)
   }
```

12.2.2Menubar

The Menubar component provides the functionality of a window's menubar.

An example of a Menubar component is:

```
MenuBar myMenuBar
   {
   Attributes(backgroundcolor = gray60)
   menu(name = File)
   pushbutton(
        name = "Close",
        Accelerator = Ctrl<Key>c,
        AcceleratorText = ^c,
        action = close,
        ParentMenu = File);
   menu(name = Attributes)
   pushbutton(
        name = "Fonts...",
        Accelerator = Ctrl<Key>f,
        AcceleratorText = ^f,
        action = fontChooser.popup
        ParentMenu = Attributes)
   }
```

The widgets that may be added to Menubars are:

- PushButton
- ToggleButton
- Separator
- Menu (also used to create (shudder) cascading menus).

12.2.3Panel

The Panel component contains UI widgets and Layout subComponents. It is essentially a rectangular area in a window that contains UI widgets and/or Editors and that can be scrollable, have a rectangular frame, and can act like a radioBox.

Examples of UI Widgets are:

- PushButton
- ArrowButton
- Label
- TextField
- ToggleButton
- Frame
- Image
- Separator
- Slider
- Gauge
- ScrolledList
- ...

12.2.4AutomaticWindow

The AutomaticWindow component automatically creates a window and the user interface widgets that correspond to the Fields in the represented *Subject* component (usually an Entity). It does this with little or no explicit instructions by the programmer.

For example:

```
AutomaticWindow autoWindow
   {
   Attributes(title = Inspector, maxTextField-
        Length = 24)
   }
```

In this example the title of the automatically generated window will be 'Inspector' and the maximum length of a TextField widget will be 24 characters.

The AutomaticWindow component examines each field of the represented *Subject*. For each field a widget type is chosen based on the field's type. This choice is based on heuristics and the 'style.spec' file. This file specifies the widget type to use for various configurations of range, editability and field type.

A number of styles are available for AutomaticWindows. They are as follows:

1. A scrolledList of a column of ‘Label: TextField’ widget pairs. The labels display the Field name, the textfields display the editable Field value.
2. Same as #1 but with the addition of a scrolledList on the left side of the window that lists the names of the siblings of the *Subject*.
3. Same as #1 but with a scrolledList across of the bottom of the window that lists the names and major attributes of the children of the *Subject*.
4. Same as #2 combined with #3.

12.2.5SubComponent: Layout

This Subcomponent is used to organize it’s contents in rows and columns. It can be included in any Window and Panel. Its children can be any of the following:

- Editor
- Iconwell
- Panel
- Layout
- UI Widgets

An example is:

```
Window myWindow
   {
   Layout(row = (myIconwell,
        column(
                myToolbarPanel,
                myEditor
                )))
   }
```

This example creates a window with myIconwell on the left side, and on the right side myToolbarPanel is placed above myEditor.

12.3 Graphics

12.3.1Editor

The graphics Editor component creates an area in which graphics primitives (i.e. lines, rectangles, circles,...) are drawn. A large amount of automation and functionality is available with the Editor component.

Much of the interactive functionality of Editors is available through a large number (about 60) of *EventHandlers.* These eventHandlers perform actions on graphics primitives, or the editor itself, in response to user actions within the editor area. Examples of eventHandlers are:

- smoothPan
- zoomAroundCursor
- animatedZoomAroundCursor
- treeNodeDrag
- alternatingSelect
- iCreateSimpleConn
- jumpPan
- delete
- sendMsgToObjectUnderCursor
- ...

A number of graphics primitives are also available. Examples are:

- Text
- Circle
- Line
- Icon
- Rectangle
- ObjInBox
- ...

An example of the Editor component is:

```
Editor myEditor
   {
   Attributes(BackgroundColor = gray70,
        Width = 600, height = 100,
        scrollBars = False,
        autoPlace = True,
        doubleBuffered = True)
   animatedZoomAroundCursor()
   smoothPan()
   }
```

12.3.2IconWell

The IconWell component is a scrolledList specifically designed to hold UI widgets that represent Entities that can be Dragged-And-Dropped into the Editor.

An example is:

```
IconWell myIconWell
   {
   Attributes(dragAndDropTarget = chipEditor,
        scrollbarsPolicy = asNeeded)
   Layout(column = (
        Togglebutton(icon = myObject1.xpm,
        dragAndDropObjectRepresented =
        myObject1),
        Togglebutton(icon = myObject2.xpm,
        dragAndDropObjectRepresented =
        myObject2)))
   }
```

The IconWell will most likely be replaced when generic drag-n-drop functionality is available for the Panel component.

12.3.3Locator

The Locator is a read-only Editor window that provides a 'Birds-Eye View' (sometimes called a *panner*) of the graphics in another Editor. There are numerous attributes associated with the Locator that customize its appearance and functionality.

12.3.4Magnifier

The Magnifier is a read-only Editor window that provides a magnified view of the graphics in another Editor. There are numerous attributes associated with the Magnifier that customize its appearance and functionality

12.3.5AnotherView

The AnotherView is an Editor window that provides a different (editable) view of the graphics in another Editor. All attributes associated with Editors are available to this component.

12.4 Animation

12.4.1Engine

The Engine provides elementary support for animation. Frame rate, start and end times, fast forward and rewind, are capabilities provided by the Engine.

12.5 Debugging Assistance

12.5.1TestDataGenerator

The TestDataGenerator, when enabled, automatically generates data for the application. In this circumstance data is not loaded by the Persistence Manager. Attributes of the TestDataGenerator specify constraints on what and what not to create and on the values of the attributes of the created data.

12.5.2Debugger

The Debugger presents a static display of the component classes and their relations to each other in a graphics editor. Components are represented as rectangles and relations are represented as lines connecting the rectangles to each other. The lines are color-coded depending on their type. See the description of the Framework Editor for more details.

12.6 Programmer Assistance

12.6.1Framework Editor

Next paper.

12.6.2Trace Output File

The Trace Output File contains the nested output of all debug trace output of the system. Optionally subsets of this output can be sent to STDOUT.

12.6.3Log File

The Log File contains a list of all operations initiated by the user.

12.7 The Persistence Manager*

The Persistence Manager provides automatic persistence for all data in the application. Data associated with a particular document is saved to a disk file previously associated with the document. *Session management* (which windows are open and where they are located on the screen) is also handled by the Persistence Manager. This means that the user can exit the application at any time. Further, when returning to the application, all windows and data will return to their state they were in when the user last used the application.

12.8 The Undo/Redo Manager*

The Undo/Redo Manager provides automatic infinite undo/redo for all user operations in the application. Operations associated with a particular document is saved to a disk file previously associated with the document.

13.0 Appendix 2: Programmers Reference

There was not enough time to completely write this section. The examples are meant to be fully functioning demo applications (that will hopefully be useful in their own right).

13.1 General

- Application
- Help Manager
- Internationalization Manager
- Document

13.2 The User Interface System

- Window
- AutomaticWindow
- Menubar
- Panel
- Layout

13.3 Graphics

- Editor
- Graphics
- IconWell
- Locator
- Magnifier
- AnotherView

13.4 Animation

- Engine

13.5 Debugging Assistance

- TestDataGenerator
- Debugger

13.6 Programmer Assistance

- Framework Editor

13.7 The Persistence Manager*

13.8 The Undo/Redo Manager*

Window

Description:

The user interface window component.

Attributes

Name	Default Value	Possible Values
DialogBoxType	DialogBox	**DialogBox**, ModalDialogBox
Title		<any text string>
Menubar		<any defined Menubar>

Messages Handled

Name Arguments	Functionality Default Value	 Possible Values
popup	Make window visible and bring to the front of all other windows.	
Close	Close window after asking user to verify the loss of any changes.	
Cancel	Close window without asking user to verify the loss of any changes.	
Save	Save the contents of the window.	
OK	Save and close the window.	
PostMessage	Popup a small dialog with an information icon and a text message.	
DialogType	Ordinary	Ordinary, Error, Info, Query, Warning Working, Help
Message	<None>	<any text string>

Messages Generated

Name Arguments	Functionality Default Value	 Possible Values
Close	Close is generated when the user closes window by using the window manager.	

Available SubComponents

Name	Functionality
Layout	Arrange the contents of the window into rows and columns.

Available Containers

Name	Functionality
<None>	

Exceptions Generated

Name	Functionality
Layout	Window has Layout that has unknown keyword
	Unknown menubar specified
	%Location - unknown value: %value for keyword: %keyword.
	%Location - Unknown widget name: %widgetName.
	%Location - unknown command: %command.
	%Location: Window %windowName has dialogBoxType that has unknown value: %value.
	%Location: Window %windowName: Unknown widget type: %widget-Type.
	%Location: Window %windowName has ScrollBars that has unknown value: %value
	%Location: Window %windowName has ScrollBarsPolicy that has unknown value: %value
	%Location - Attribute: %keyword value retrival not supported.
	%Location - %windowName: Asked to obtain value of unknown attribute: %keyword.

Notes:

Inherited Features: Panel Attributes*.*

See Also: MenuBar, Panel, Layout, AutomaticWindow

Example:

```
Application example
   {
   Attributes(DefaultFirstWindow = myWindow)
   }

Menubar myMenubar
   {
   Attributes(backgroundcolor = gray60) menu(name = File)
   pushbutton(name = “Close”, Accelerator = Ctrl<Key>c, AcceleratorText = ^c, ParentMenu = File);
   }

Window myWindow
   {
   Attributes(Title = My Example Window, menubar = myMenubar)
   Layout(row = (myIconWell,
                column(
                        myToolbar,
                        myEditor,
                        myPlayerPanel,
                        Slider(name =mySlider, height = 40, value = @engine.currentTime),
                        myControlPanel)))
   }
```

AutomaticWindow

Description:

The user interface window component that automatically generates the widgets that will be used to display and modify the values of any assigned Entities..

Attributes

Name	Default Value	Possible Values
DialogBoxType	DialogBox	**DialogBox**, ModalDialogBox
Title		<any text string>
Menubar		<any defined Menubar>

Messages Handled

Name Arguments	Functionality Default Value	 Possible Values
popup	Make window visible and bring to the front of all other windows.	
Close	Close window after asking user to verify the loss of any changes.	
Cancel	Close window without asking user to verify the loss of any changes.	
Save	Save the contents of the window.	
OK	Save and close the window.	
PostMessage	Popup a small dialog with an information icon and a text message.	
DialogType	Ordinary	Ordinary, Error, Info, Query, Warning Working, Help
Message	<None>	<any text string>

Messages Generated

Name Arguments	Functionality Default Value	 Possible Values
Close	Close is generated when the user closes window by using the window manager.	

Available SubComponents

Name	Functionality
Layout	Arrange the contents of the window into rows and columns.

Available Containers

Name	Functionality
<None>	

Exceptions Generated

Name	Functionality
Layout	Window has Layout that has unknown keyword
	Unknown menubar specified
	%Location - unknown value: %value for keyword: %keyword.
	%Location - Unknown widget name: %widgetName.
	%Location - unknown command: %command.
	%Location: Window %windowName has dialogBoxType that has unknown value: %value.
	%Location: Window %windowName: Unknown widget type: %widget-Type.
	%Location: Window %windowName has ScrollBars that has unknown value: %value
	%Location: Window %windowName has ScrollBarsPolicy that has unknown value: %value
	%Location - Attribute: %keyword value retrival not supported.
	%Location - %windowName: Asked to obtain value of unknown attribute: %keyword.

Notes:

Inherited Features: All Window features.

See Also: Panel, Window

Example:

```
AutomaticWindow AutoMaticAttributesWindow
    {
    Attributes(Title = “Attributes”)
    Attributes(maxTextFieldLength = 24)
    }

Entity
    {
    View(name = attributes, Viewer = AutoMaticAttributesWindow, viewObject = this, required = False)
    MessageFlow(message = popupAttributes,
            action = AutoMaticAttributesWindow.popup,
            action = addView(name = attributes))
    }
```

This example creates an Entity that popups an automatically generated attributes window whenever it receives a *popupAttributes* message.

Panel

Description:

The user interface window component.

Panel Attributes

Name	Default	Possible
BackgroundColor	Undefined	
ScrollBars	True	True, HorizontalOnly, VerticalOnly
ScrollBarsPolicy	<None>	AsNeeded, AlwaysVisible
Frame	False	False, True
RadioBox	False	False, True

Messages Handled

Name Arguments	Functionality Default Value Possible Values

Messages Generated

Name Arguments	Functionality Default Value Possible Values

Available SubComponents

Name	Functionality
Layout	Arrange the contents of the window into rows and columns.

Available Containers

Name	Functionality
Window	
Panel	

Exceptions Generated

Name	Functionality

Notes:

Inherited Features: None.

See Also: Window, AutomaticWindow, Layout

Example:

```
Application exampleApplication
    {
    Attributes(DefaultFirstWindow = myWindow)
    }

Window myWindow
    {
    Layout(row = documentManager)
    }

Panel documentManager
    {
    DataFlow(.docNames.contents = sampleApplication.contents, direction = DestinationToSource, fil-
          ter = docFiter)
    Layout(column = (
          scrolledList(name = docNames),
          row(
                    pushbutton(name = open, sensitivity = @(r).docNames.anItemIsSelected,
                              acttion = Lue.openDocument(name = %docNames.selectedItem)),
                    pushbutton(name = copy, sensitivity = @(r).docNames.anItemIsSelected,
                              action = Lue.copyDocument(name = %docNames.selectedItem)),
                    pushbutton(name = create,
                              action = Lue.createDocument(name = newSampleDocument)),
                    pushbutton(name = delete, sensitivity = @(r).docNames.anItemIsSelected,
                              action = Lue.deleteDocument(name = %docNames.selectedItem)
          )))
      }
Filter docFilter
      {
      Attributes(format = %-10.10name     %-40.40description)
      }
```

MenuBar

Description:

The user interface window component.

Attributes

Name	**Default Value**	**Possible Values**
DialogBoxType	DialogBox	**DialogBox**, ModalDialogBox

Messages Handled

Name **Arguments**	**Functionality** **Default Value** **Possible Values**

Messages Generated

Name **Arguments**	**Functionality** **Default Value** **Possible Values**

Available SubComponents

Name	**Functionality**
<None>	

Available Containers

Name	**Functionality**
Window	

Exceptions Generated

Name	Functionality

Notes:

Inherited Features: None.

See Also: Window

Example:

```
Menubar myMenubar
   {
   Attributes(backgroundcolor = gray60)
   menu(name = File)
        pushbutton(name = “Close”, Accelerator = Ctrl<Key>c, AcceleratorText = ^c, ParentMenu =
        File);
   menu(name = Edit)
        pushbutton(name = “Group”, Accelerator = Ctrl<Key>g, AcceleratorText = ^g, ParentMenu =
        Edit, action = myEditor.group);
        pushbutton(name = “UnGroup”, Accelerator = Ctrl<Key>u, AcceleratorText = ^u, ParentMenu =
        Edit, action = myEditor.ungroup);
        pushbutton(name = “Bring to front”, Accelerator = Ctrl<Key>f, AcceleratorText = ^f, ParentM
enu = Edit, action = myEditor.bringToFront);
        pushbutton(name = “Send to back”, Accelerator = Ctrl<Key>b, AcceleratorText = ^b, Parent-
        Menu = Edit, action = myEditor.sendToBack);
   menu(name = Help)
        pushbutton(name = “Help on the Graphics Editor”, ParentMenu = Help);
   }

Window myWindow
   {
   Attributes(Title = My Example Window, menubar = myMenubar)
   Layout(row = myEditor),
   }
Editor myEditor
   {
   }
```

Layout

Description:

The user interface window component

Attributes

Name	Default Possible
Row	
Column	

Messages Handled

Name Arguments	Functionality Default Value Possible Values

Messages Generated

Name Arguments	Functionality Default Value Possible Values

Available SubComponents

Name	Functionality
<None>	

Available Containers

Name	Functionality
Window	
Panel	

Exceptions Generated

Name	Functionality

Notes:

Inherited Features: Panel Attributes*.*

See Also: Panel, AutomaticWindow

Example:

```
Menubar myMenubar
    {
    Attributes(backgroundcolor = gray60) menu(name = File)
    pushbutton(name = “Close”, Accelerator = Ctrl<Key>c, AcceleratorText = ^c, ParentMenu = File);
    }

Window myWindow
    {
    Attributes(Title = My Example Window, menubar = myMenubar)
    Layout(row = (myIconWell,
                    column(
                            myToolbar,
                            myEditor,
                            myPlayerPanel,
                            Slider(name =mySlider, height = 40, value = @engine.currentTime),
                            myControlPanel)))
    }
```

14.0Questions For You Reviewers

14.1 Do you understand the system?

14.2 Do you understand it's purpose?

14.3 Which sections helped you understand the system?

14.4 Which were the least helpful in understanding the system?

14.5 Which sections were exciting?

14.6 Which sections were boring?

14.7 Do you think that this architecture will be useful to you when you design applications in the future?

14.8 Do you think that this system would be useful to you in implementing applications in the future?

14.9 Do you think that the built-in components are about right in terms of their:

14.9.1purpose?

14.9.2functionality?

14.9.3customizability?

14.10What components should be added?

14.11What kind of information would help you better understand the purpose and structure of the system?

14.12What kind of information would help you better understand the usage of the system?

14.13What is your opinion of the Programmer’s Reference? Was the table layout structure helpful?

14.14Comments:

Addendum 7/6/98

Fred and Ethel evolved from VisualADE 2.0 and have since evolved into Cadabra. Cadabra provides a very general and simple methodology for modeling meta-data, meta-rules, meta-relations and meta-behaviors. It is expected that the Cadabra GUI sub-system will look a lot like the FRED and ETHYL GUI sub-systems.