C E C I

AthenaMuse® 2.2

Documentation

December, 1996

MIT AthenaMuse Software Consortium
Massachusetts Institute of Technology

Center for Educational Computing Initiatives 1 Amherst Street, Building E40-300 Phone 1 617 253 0173
Massachusetts Institute of Technology Cambridge, Massachusetts 02139-4307 Fax 1 617 258 8736

Written by

Wissam Ali-Ahmad
Philip Bailey
Issam Bazzi
Ana Beatriz Chiquito
Katherine Curtis
Robert Deroy
William Euerle
Adam Feder
Judson Harward
Mary Hopper
Cesar Hurtado
Masanori Kajiura
Justin Lapierre
Li Wei Lehman
Steven Lerman
Margaret Meehan
Cyril Morcrette
Kimberly Ringer
Yonah Schmeidler
Hiroshi Tominaga
Sigmund Tveit
Juan David Velasquez

of the

MIT AthenaMuse Software Consortium

Table of Contents

Chapter 1 Overview

AthenaMuse 2 (*AM2*) is a multimedia authoring tool designed for authoring by multiple users in a heterogeneous, networked environment. The following if an outline of this chapter, which provides a brief introduction to *AM2* and an overview of the document:

At the heart of *AM2* is a scripting language called the Application Description Language *(ADL)*. We conceived the *ADL* as the platform-independent storage format for *AM2* application descriptions. Our original intention was to build a series of direct manipulation editors to assist users of all levels of programming experience in developing AthenaMuse applications. Only one such editor, a prototype layout editor, has been developed. Consequently, the *ADL* has also become the primary authoring medium for *AM2* applications.

One reason for the *ADL*'s effectiveness as a scripting language is that we designed it in concert with the internal C++ classes that regulate the *AM2* environment. An *AM2* application at run-time may be thought of as a collection of class instances that responds to user input like an automaton. The *ADL* is a very economical description of the classes that are used to build that automaton. The goal of this documentation is to explain how application developers can use the *ADL* to build the classes necessary to implement a particular multimedia application. Our intended audience are application developers. We assume some programming experience and familiarity with the general principles of object-oriented programming, but knowledge of C++ is not necessary to use the *ADL*. At times, however, in what follows, we will compare or contrast the *ADL* to C++ and Smalltalk to clarify its use to those familiar with these other object-oriented languages.

The AthenaMuse environment currently runs on three flavors of UNIX (SunOS 4.2.n, Solaris 2.5, and HPP-UX 9) as well as on Win95 and Windows/NT 3.5.1. A preliminary version of AthenaMuse runs on Macintosh System 7, but as of the date of this document, this version is not supported.

1.1 Organization of This Document

To start you on the road to building that application we give you several types of information:

- **Overview**: a brief description of the structure of an application and the building blocks available to you in the *ADL*, plus a discussion of the *ADL*'s philosophy
- **Hello World**: provides a glimpse of *AthenaMuse*'s powerfully simple application description language (*ADL*) before the more formal descriptions that will follow in later chapters
- **Description of the *ADL*:** the simpler, basic constructs of the language and the more complex units made from these simpler ones
- **Using activities in *ADL***: a description of the mechanism by which *AM2* objects request notification of and respond to user actions and other external events, such as reaching the end of a video segment in an application
- **Annotated samples of *ADL* programs**: for those who like to learn by association and example, this section provides a series of programs of increasing complexity
- **Wrapped class reference**: description of the system-defined wrapped classes, including accessible members and methods, supported activities and sample programs illustrating common uses of wrapped classes
- **Creating wrapped classes**: explanation of the use of the wrap script, a tool for system developers who want to make their own C++ classes available to the *ADL*

The AthenaMuse environment attempts to offer true application portability across the UNIX and Windows platforms. This document will footnote platform dependencies where they exist.

Certain features were conceived as part of the original AthenaMuse design but have never been implemented. While there is no guarantee that these features will be implemented in the future, we have retained discussion of them in this document because the design of *AM2* is usually more comprehensible when they are included. All such features are clearly marked **unimplemented**.

1.2 The Structure of an Application

AM2 has two design goals that have affected the design of the application building environment:

1. The description of an application's interface should be separate from the content presented. For example, in a multimedia application that contains a video viewer you may want to use the viewer many times, but each video clip viewed with it is tied to the particular context.
2. An application should be as portable as possible across platforms and environments so that you can customize an application to suit a user's background and preferences, and so that you can take advantage of special features of a particular hardware configuration or operating system. For example, an application may use the English language on interface controls as a default, but it should also allow customization of the control labels in other languages. And the application should request general services, such as a video source, and determine how to access that source from a description of the system configuration.

Satisfying these two goals suggests that an application consist of three distinct parts:

1. **Application description** specifies the application's interface and functionality in as pure and platform-independent a form as possible
2. **Application content** is stored separately from the application description
3. **Customizations** of the application, are stored separately so that the same application description can run with different sets of customizations (known in *AM2* as *assets)*

In *AM2,* classes describe the application's interfaces and functionality. At run-time, instances of these classes are populated with content drawn from databases, files, network services, or the application itself. The use of classes to specify interfaces encourages users to think in terms of and to build with nested interface templates. The *ADL*'s rich set of features for initializing the instances of these templates marries them to the content. The joining of the two forms the screens, images, text fields, and buttons of the application. The initialization also contains a step that allows the user to customize each instance (**see Section 3.25, "Object Initialization" page 49**).

1.3 Road Map to the *ADL*

The elements of the *ADL* are defined briefly here. Italicized terms refer to other *ADL* terms also defined in this list.

- **Assets** allow a user to customize an application description using a special form of *initialization.*
- **Assignment** is a very simple type of statement that calculates the value of an *expression* and assigns it to a variable.
- **Base types** are the simplest system-defined variable types used in the *ADL*, e.g., `integer` or `string`.
- **Base type constants** are actual values of base typs that can be expressed in the *ADL*, e.g., `42` or `"Hello, world!"`.
- **Built-in function calls** are invocations of system-defined functions that can take arguments and return a value. They help the user manipulate base and compound values in ways that would be difficult if not impossible with *expressions*.
- **Class definitions** specify the members and *methods* of a class.
- **Complex types** are indexed collections of data of a specified type. The *ADL* supports both indexed and associative arrays.
- **Compound types** are *ADL* data types built from base types. They include lists, times and intervals.
- **Control structures** are built from statements and conditional *expressions* to form complex statements that can change the course of an application's execution. The `if` and the `while` statements are examples of control structures.

- **Dynamic object creation** provides a mechanism for the application developer to create objects as needed at run-time.
- **Expressions** are built from constants, variables, and operators. They are used at run-time to calculate new values.
- **Identifiers** name variables and classes in the *ADL*.
- **Inheritance** describes how one class builds on the members and *methods* of other classes.
- **Initialization** describes the various mechanisms for initializing class objects with their members and *inherited* bases.
- **Lexical conventions** determine how to format the *ADL* in a file or on the screen, including the mechanisms for embedding explanatory comments in a script.
- **Libraries** are collections of *classes* used as a starting point in building new classes, and the application as a whole.
- **Messages** are sent to *objects*, which must possess the appropriate *method* to handle the message. A message can be part of an *expression* if it returns a value or it can stand by itself as a statement.
- **Metaclass operations** describe special operators and *messages* used to treat a *class* as a special kind of *object*.
- **Method definitions** specify how *messages* with a particular selector, or name, are handled.
- **Object definitions** are similar to *variable definitions*. They specify class members or temporary objects in methods. An object definition can specify how the object is to be *initialized* at run-time.
- **Object destruction** describes what happens when a defined object is destroyed automatically, such as at the end of a *method*, or when a *dynamic object* is destroyed explicitly.
- **Object member reference** describes how to use parts of an object in *expressions*, *assignment* statements, and *messages*.
- **Program structure** describes how this whole hierarchy of components can be used to generate complete applications.
- **Scope** governs the visibility of variables and objects, that is, the portion of the program where their names are known.
- **Type conversion** specifies the rules the *ADL* uses to change the type of a value when it encounters one type but needs another, or when the result of an expression could be of several types. For instance, is the value of the expression `3.14159/2` an `integer` or a `real`?
- **Variable definitions** declare the developer's intention to use a named variable within a given context, or to *initialize* the variable to a particular value.

1.4 *AM2* Philosophy

AM2 uses a completely object oriented approach to provide a flexible and extensible multimedia environment. Early design discussions focused on describing an object oriented paradigm and deciding whether to use an existing language for implementation or to invent one. The design team decided to create a new language called Application Description Language, based on C++.

The *ADL* is a means of specifying an entire *AM2* application, with particular efficiency in describing user interface templates and their associated functionality. It also provides easy access to and manipulation of the underlying system objects provided by the *AM2* run-time environment.

The design team selected C++ as the *AM2* implementation language both for its portability and relative efficiency. While C++ is an extremely rich and complex programming language, the *ADL* requires only a small subset of that functionality. The language features supported by the *ADL* are both necessary to its task and sufficient to accomplish it. That is, the *ADL* supports the minimal set of language constructs necessary to specify the general set of multimedia applications.

The *ADL* never allows a C++ usage to have a different meaning in the *ADL* than it would have in C++. Nor does the *ADL* arbitrarily express C++ concepts using non-C++ usage without very strong reasons for all such variations. For instance, the *ADL* uses the keyword `common` to designate what in C++ would be a class `static` data member in order to reduce the ambiguity of the much overused static declaration. Any *ADL* concepts or mechanisms that C++ does not support are directly required by the *ADL*'s particular multimedia mission. For example, the list compound data type is highly desirable to support the construction of messages at run-time and to facilitate communication with underlying databases.

It is worth acknowledging the two single most important differences between the *ADL* and C++. First, the *ADL* does not associate pointers with specific data types as C++ does. And second, the *ADL* allows the entire contents of a message to be determined at run-time including the message selector and the number of arguments.

AthenaMuse was originally intended to offer transparent application portability across Windows, UNIX, and the Macintosh. Work on the Macintosh platform has been discontinued as of the summer of 1996, but the goal of application portability has been met on the other platforms. It is worth noting certain principles of *AM2*'s approach to platform portability:

- Since application portability is a goal, the use of platform specific features is discouraged, but not forbidden. That is, *AM2* should not deny the developer access to platform-specific features, but neither should it encourage their use.
- By default, an application running on a UNIX version of *AM2* should obey the Motif look and feel, while a version running under Windows should obey the Windows user interface guidelines. The preliminary Macintosh version followed the same principles.

The later feature makes the former less bothersome. AthenaMuse classes attempt to embody semantic functionality rather than low-level feature sets. For instance, the *AM2* approach to menus is platform independent. It is the registration of the menu with a particular application on a particular operating system that determines the visual style of the menu (e.g., pull-down or pop-up, tear-off, etc.).

Chapter 2 *Hello, World*

Ever since Brian Kernighan and Dennis Ritchie introduced us to *The C Programming Language*[1], it has become traditional to introduce a new programming language to its audience via the simplest of programs, one that produces the message, “`Hello, World`”. And that is exactly what we shall do below. Before the more formal descriptions that will follow in later chapters, we want to give a glimpse of *AthenaMuse*’s powerfully simple application description language (*ADL*).

2.1 Your First AM2 Program

So follow along now and type in the following lines using your favorite text editor.

```
1    anonymous: XFtop
2    {
3      XFbutton button {
4                            label = “Hello, world”;
5                            Pressed = {'Exit, theApp};
6                         };
7    } top {
8             height = button.height;
9             width  = button.width;
10   };
```

HelloWorld.adl

If you have installed AthenaMuse correctly, you should now be able to type

```
% am2Program hello.adl
```

To run this sample program, where ***am2Program*** is the name of the AthenaMuse executable on your system. The result should be a small frame containing a single button labelled “`Hello, World`”.[2] If you click on the button, the program exits and the frame disappears. All right. It may not be the most exciting program, but in a very few lines of code you have implemented a well-behaved application that creates a window button with which you can interact. Let’s look at the code line by line to see how it works.

1 First edition, Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

2 On Windows, this frame appears in the upper left corner of the screen. On UNIX, its location depends on your window manager. Many window managers allow the application to “float” as an outline box until you click to position it.

2.2 Explanation

Line 1: The building blocks of *AM2* applications are *classes* and *objects*. A *class* defines a set of structured data and a set of methods that operate on that data. An *object* is an instance of a particular class.

Our application starts off with the definition of a class that will describe the frame holding the single button of the application. We are only going to need one instance of this class so we don't need to name it. We declare the class with the keyword `anonymous`. If we were going to create another instance of this class later, we would need to name the class so we could refer to it again. In this case, we would declare it with the keyword `class` and supply a name for the class.

`XFtop` is the name of a system supplied *wrapped class*. `XFtop` provides a top-level shell or frame that can contain other components and that will interact with your system's window manager. By itself, instances of `XFtop` are fairly boring, but they perform a very real function. By dealing with the window manager and the other top level windows on your display, instances of `XFtop` handle requests to minimize the application and manage stacking order.

The colon between the keyword `anonymous` and the class name `XFtop` means that the nameless class we are describing *inherits* from `XFtop`. That is, it has all the characteristics of the system-supplied `XFtop`, plus a few extras that we will define in the following lines.

Line 2: The curly brace (`{`) marks the start of our special additions, which will distinguish our anonymous class from its `XFtop` base. It is matched by the closed curly brace (`}`) on line 7 that marks the end of our special additions.

Line 3: As we said, an `XFtop` that doesn't contain anything is boring. Our frame does contain something, a single button. And this is the line that inserts the button in the frame.

In *AM2*, the standard way to make one window system component (we call them *widgets*) contain another is to make the contained widget a member of the container parent class. Well, almost. Since `XFtop` is a built-in wrapped class, we can't alter it, say by adding a new member. But we can create a new class that inherits from the wrapped class. And that *derived* class can contain additional members. So wrapped classes implementing container widgets are used as bases for user defined classes that contain other widgets as members.

And we are defining just such a member here. The member is an instance of the wrapped class XFbutton, and we refer to it within this anonymous class by the name we assign it here, `button`. This is a little subtle. The class we are defining here has no name, but it has named members. All members must have names, but classes do not need to be named unless we are going to refer to them later.

The open curly bracket that follows the button declaration marks the beginning of a block of code that allows us to particularize this instance of the `XFbutton` class.

Line 4: The `XFbutton` wrapped class contains a member called `label`. You don't have to declare this member. It's built in. You don't even have to initialize it, although you almost certainly want to. If you don't, your button won't have a text label. In this line, we set `button`'s label to be the string "`Hello, World`". Notice that this initialization is actually an executable statement, and so must be followed by a semicolon.

Line 5: This line describes what we want to happen when the button is clicked by the mouse. *AM2* provides the programmer a very flexible way to monitor whatever the user is doing on the screen using *activities*. This functionality is described in Chapter 3, Using Activities (*provide cross reference*).

But we can do something much simpler here. `XFbuttons` possess a member called `Pressed` that specifies what the user wants to happen when the button gets pressed. The value of `Pressed` must be a *list* delimited by curly braces and containing two elements. The first element is a *message* and the second is a handle to the object to which the message should be sent. So, the Pressed member should contain values that look like `{` *message*, *target* `}`.

In the *ADL*, a message can also be a *list*, whose first element is a string called the *selector*. The selector determines which method will be invoked in the target of the message when the message is sent. The other elements of a message list are arguments to the method. But the message we want to send is so simple that it doesn't possess arguments, just the selector that all messages must have. And in the case of a message without arguments, the message list can be represented simply by the selector string, which is what has happened here. The message is a string, `'Exit`. Note that if a string consists of a single word, it can be preceded by a single quote (`'`) rather than enclosed in double quotes (`"`).

The second part of the value of `Pressed` must be a *handle* to the target of the message. A *handle* is the *ADL*'s version of a pointer or reference. A variable or member declared as a handle can contain references to other objects or variables. In this case, we are using the keyword `theApp` to specify a very special reference to the *application object*.

Everything in an *ADL* program defines a class including the application itself. The application, although it is not preceded by the keyword `anonymous` and curly braces, defines an anonymous class that includes all global variables and all other class definitions. When the application starts up, a single instance of this *application class* is created and initialized. This object can be accessed through the keyword `theApp` which always contains a handle to this *application object*. So, by making `theApp` the target of the `'Exit` message, we are assuming the application has a method called `Exit` that doesn't take any arguments. It does. As a matter of fact, it inherits that method from a class called `MCapplication` that serves as a silent base class to all applications. You can read about `MCapplication` in the wrapped class documentation in **Section 6.1, "Activities and Application Services" 113**.

Line 6: The closing brace on this line ends the block of initializations for the member `button`. It can be thought of as ending the declaration of the member, and all declarations must end with a semicolon.

Line 7: The first curly brace on this line, the closed brace, ends the definition of our anonymous class derived from `XFtop`. The next word, `top`, is the name of the instance of this anonymous class. Why do we need to name this instance? Well, this instance is a global variable in the application, and global variables are considered to be members of this application class. And, as we said above, all members must have names. Trust us on this one for now. It makes things come together much more nicely if we have to supply a name for this variable. The second curly brace, the open one, starts the initialization block for this single instance of our anonymous class derived from `XFtop`.

Line 8: Here we initialize the height of the frame to be the same as the height of the button that it contains. The units are pixels. It turns out that `XFtop` has a member called, you guessed it, `height`. We set the frame's height to be the same as the height of the frame's member called `button`. Note that both the frame and the button have a member, `height`. We use the `.` notation to access the member `height` of the frame's member `button`. The variable `height` to the left of the `=` refers to the frame's member of the same name because this initialization block belongs to the frame, not the button (or any other object).

Now an interesting question. Why do we have to specify the height of the frame but not the height of the button? By default, buttons with text labels are sized to be just big enough to contain the label. You can override that by setting the width and height of a button, but we have not done that here. Which is why the `"Hello World"` application ends up being so small on the screen. But `XFtops` have a default size that is very, almost vanishingly small. Why didn't we make them large enough to contain their contents by default? Well, we thought about it and tried it out, but application developers decided that it hardly ever turned out to be what they wanted. So we decided that you had to size the top-level frame. Children of this top-level frame (that is, the widgets contained by it) would have default position in the upper left hand corner. If you have more than one child widget, that is unlikely to be what you want for the second and later widgets. But in this case, we don't have to worry about positioning the button, just sizing the frame around it.

Line 9: This line is like the previous except it sets the width of the frame, not the height.

Line 10: We’re done. The curly brace closes the initialization block for the frame `top`, and because the initialization completes the definition of the anonymous class and its single instance, we must follow it by a semicolon.

- Try changing a few values here and there, or try to comment out a line by prepending two slashes (`//`) and then rerun the program. We have glimpsed only the very surface of *AM2*’s capabilities in this example, but it should give you some sense already of what it feels like to create an application in the *ADL*.

Chapter 3 The Application Description Language

The purpose of this chapter is to provide a detailed description of the *ADL,* which is the core of *AM2*. Note that this covers both the simpler, basic constructs of the language and the more complex units made from these simpler ones. The following outline describes the sections of this chapter:

3.1 Lexical Conventions

Lexical conventions determine how to format *ADL* code in a file or on the screen, including the mechanisms for embedding explanatory comments in a script.

Formatting

ADL code is not sensitive to the presence or absence of white space between language elements provided that all keywords and identifiers are distinguished. Thus the programmer has the same degree of freedom that C and C++ allow in formatting code. White space includes these characters: space (' '), newline ('\n'), carriage return ('\r'), form feed ('\f'), and vertical tab ('\v').

Commenting

Comments are indicated as they are in C++. Two contiguous slashes (//) indicate the start of a comment that continues to the end of the current line. This form is the usual method for annotating individual, single-line statements. A slash immediately followed by an asterisk (/*) indicates a comment that continues until the reverse sequence (*/) is encountered. This form is the usual method for inserting multi-line comments.

3.2 Identifiers

An identifier is the name of a variable, class, method, or built-in function. It must begin with a letter and consist of a sequence containing only letters, digits, or the underscore character (_). Case is significant. Identifiers can possess an arbitrary number of characters, and at least the first 32 characters are distinguishable. Some *ADL* implementations may consider more than 32 characters significant. If a particular implementation considers the first n($n \geq 32$) characters significant, and two identifiers differ only after the nth character, the implementation considers them identical.

3.3 Base Types

The system recognizes the base, or primitive, types `boolean`, `integer`, `real`, `string`, `vtype`, and `handle`.

Boolean

A boolean must have the value TRUE or FALSE.

Integer

An `integer` corresponds to a C/C++ long and is represented with at least 32 bits.

Real

A `real` corresponds to a C/C++ `double`. The minimum and maximum positive values of a `real` are platform dependent, but should accommodate a range of at least 10.0^{-38} to 10.0^{38}.

String

An *ADL* `string` represents an ASCII string implemented using an underlying C++ class, and, thus, does not correspond exactly to a C/C++ `char*`. The only restrictions on maximum string length are implementation dependent, but in all *AM2* implementations strings are guaranteed to have a maximum length of at least 65535 bytes. If the integer constant MAX_STRING_LENGTH has a value greater than 0, that value specifies the maximum allowed string length.

Vtype

A `vtype` allows storage of types of variables, and is used for type-checking.

Handle

A `handle` to a `lvalue` of a base type or a compound type (**see Section 3.5, “Compound Types” page 16**), or to an *ADL* object represents a reference to the underlying variable or object. It is a more general case of the C/C++ pointer, in that *AM2* will eventually support handles to objects in other *AM2* process spaces. *AM2* handles do not support the full semantics of C/C++ pointers. In particular, there is no relation between handles and arrays in the *ADL*, and handle arithmetic is not allowed.

3.4 Base Type Constants

The *ADL* supports constants in each of the primitive system types: `boolean`, `integer`, `real`, `string`, `vtype` and `handle`.

Boolean Constants

Boolean constants must be one of the keywords TRUE or FALSE.

Integer Constants

Integer constants are always decimal and can be signed, e.g., *255, -1, +100000*. The named integer constants MAX_INTEGER and MIN_INTEGER represent the values of the largest positive integer and the smallest negative integer on the host system.

Real Constants

Real constants follow the standard C language form of an integer part, a decimal point, an e or E, and an optionally signed integer exponent. At least one of the integer and fraction parts must be present, as well as either the decimal point or the exponent. Examples: `3.1416, 1E+9, -0.21e-3`.

The named floating point constants MAX_REAL and MIN_REAL represent the values of the largest and smallest positive real numbers that can be represented on the host system.

The `REAL_EPSILON` constant represents the smallest positive real value such that

```
0.0 + REAL_EPSILON != 0.0
```

String Constants

A string constant that contains only letters (A-Z, a-z), digits (0-9), or the underscore character (_) can be so designated by preceding it with the single quote character ('). This feature simplifies the specification of messages (**see Section 3.13, "Messages" page 30**). You can always enclose a string constant within double quotes ("), and you can embed the standard C escaped character constants[1] in strings.

```
"Hello, world\n"

'GetWidth
```

Figure 3.1 String Examples

Type Constants

Type constants can be one of the keywords `booleanType, integerType, realType, stringType, vtypeType, handleType, listType, timeType, or intervalType`.

Handle Constants

The only permissible handle constant is expressed by the keyword NULL. It represents a value that cannot be a valid handle to any variable or object. Thus, NULL indicates that a handle variable does not point to any valid target.

3.5 Compound Types

The `ADL` supports three compound data types: `list, time, and interval`. It also supports both multidimensional indexed arrays and one dimensional associative arrays. Arrays are *complex* data types, however, and are discussed further in a later section (**see Section 3.9, "Complex Data Types" page 25**).

List

A `list` is a compound type built up out of expressions of base and compound types. Lists can, therefore, nest. There is no syntactical requirement that all list elements be of the same type. You create lists using the list delimiters ({}), and separate list elements with commas.

A list element in a ({}) expression can be an expression itself. In this case, the expression is evaluated and the list element is initialized with the result. The list contains the result of evaluating the expression, not the expression.

Note that lists are not objects, and objects cannot be members of lists. Handles to objects, however, are a base type and so can be list members. The *empty list* denoted by ({}) can be used in comparisons and to initialize lists.

[1] Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language (1988), 193-194.*

```
{'Monday, 'Tuesday, 'Wednesday, 'Thursday, 'Friday }

{'ClipA, {0, max+1} }

{ dayOfMonth, month, year }
```

Figure 3.2 List examples

Time

A `time` is an ordered 4-tuple of integers representing a period of time, and is not meant to be used for absolute (real) time. A `time` consists of numbers of hours, minutes, seconds, and milliseconds, and can be either positive or negative.

Time constants consist of from one to four integers separated by colons. If only one number is given, a colon must precede it. The four numbers correspond to hours, minutes, seconds, and milliseconds. If fewer than four numbers are given, then they are assumed to be the less significant (and more precise) components of a time (i.e., 3:4 is taken to mean 3 seconds and 4 milliseconds). White space between the parts of a time constant is not allowed.

All times are kept in standard form: milliseconds are between 0 and 999, minutes and seconds are between 0 and 59. This means that constants such as 90:0 are automatically converted to 1:30:0 internally. If any integer in a `time` constant is negative, all integers forming the constant must be negative or 0, so that -1:-30:0 is legal but -1:30:0 is not legal.

Interval

An `interval` is an ordered pair of `integers` or `real` numbers with an associated *open* or *closed* condition for each half of the pair. You can convert appropriately formatted lists consisting of

`{ {`*boolean, integer*|*real*`}, {`*boolean, integer*|*real*`} }`
or
`{ {`*boolean, time*`}, {`*boolean, time*`} }`

to intervals, and vice versa. Note that this alternate representation of an `interval` consists of a two part list of lists. Each sublist contains a `boolean` and either a `real`, an `integer`, or a time. A TRUE value corresponds to the *closed* condition, and a FALSE corresponds to the *open* condition. You can represent the `interval` constant as follows:

- an opening left parenthesis or square bracket
- an `integer, real, or time` constant
- a comma
- a second `integer, real, or time` constant
- a closing right parenthesis or square bracket

A parenthesis indicates an open condition, and a square bracket indicates a closed condition. The interval variable is assigned only as a unit. You can access and alter the interval endpoints and the associated open and closed conditions only by using the corresponding `list` form of the interval, or a built-in function. An interval appears in expressions only if it uses the special relational operator (:),pronounced *in*.

```
integer|real|time : interval
```

Such a (sub)expression evaluates to TRUE if and only if the left argument falls within the interval defined by the right argument. Interval expressions are intended primarily to test whether the current value of a state object falls within a particular range. See the example of interval usage in Figure 3.3.

```
4 : [0,9]   // returns TRUE

interval range;
integer frame;
range = (0 , 54000];
frame = 999;
if ( frame : range )
{ /* Do this */ }
```

Figure 3.3 Sample Interval Usage

3.6 Variable Definitions

You can define base and compound variables at the beginning of any scope (**see Section 3.26, “Scope” page 52**) using the keywords `boolean, integer, real, string, time, vtype, handle, list,` and *any* followed by a non-null list of identifiers (**see Section 3.2, “Identifiers” page 14**). A variable defined with the keyword any can contain a value of any base or compound type, while variables defined with the other types can only contain values of their defined type. It is an error to assign a value of one type to a variable of another type if the value type cannot be implicitly converted to the variable type (**see Section 3.11, “Type Conversion” page 28**).

Exiting from an enclosing scope destroys variables, and their values are not preserved across separate entries to that scope. The *ADL* does not support the C and C++ concept of `static` variables. The same functionality can usually be provided by a class common data member, although it is currently unimplemented.

```
integer anInt, myInt, yourInt;

list A, B;

string daysOfWeek;
```

Figure 3.4 Sample Variable Definitions

3.7 Expressions

You construct expressions from variables (**see Section 3.6, “Variable Definitions” page 18**), constants (**see Section 3.4, “Base Type Constants” page 15**), operators, and delimiters. The *ADL* operator set is a near subset of the C++ operator set with a few additions and is described in the figures below.

3.7.1 Delimiters

Delimiters in the *ADL* primarily group together related tokens. For example, you use parentheses to change precedence within an expression. *ADL* delimiters have a higher precedence than operators, thereby allowing them to control the order of evaluation.

DelimiterPair	Purpose	Sample Usage
`( )`	To change order of computation in expressions	`(1+2) * 3`
`{}`	To form a list from expressions	`{'A, 'World, 2*3}`
`[], [), (], ()`	To form an interval from pairs of numbers	`5 : (4,6)`

Figure 3.5 ADL Delimiters

In some cases delimiters create a new semantic item out of the elements they enclose. The interval delimiters serve this purpose by making an interval from two numbers. However, intervals are more than two numbers paired together. They possess “open” or “closed” attributes at each endpoint and support a test for membership in the interval.

3.7.2 Operators

When the same operator occurs in both the *ADL* and in C++, it has the same semantics, precedence, and associativity. The following points deserve special notice:

- The *ADL* normally evaluates both subexpressions of a binary operator before calculating the result of the binary operator. However, the boolean operators (&&) and (||) present a special case. The *ADL* evaluates the right subexpression only if the left subexpression is TRUE in the case of (&&) and FALSE in the case of (||).
- The *send message* operators `(=>)`,`(|>)`,`(?=>)`,and `(?|>)` have been overloaded to accept both *objects* and *object handles* as a second operand.

Operator	Operation	Precedence	Type of Operands	Type of Result
`+`	unary plus	6	*integer, real, or time*	*integer, real, or time*
`-`	unary minus	6	*integer, real, or time*	*integer, real, or time*
`*`	multiplication	7	*integer or real*	*integer or real*
`*`	multiplication	7	*(integer or real)* `*` *time*	*time*
`/`	division	7	*integer or real*	*integer or real*
`/`	division	7	*time* `/` *time*	*integer*
`%`	remainder	7	*integer*	*integer or real*
`+`	addition	8	*integer, real, or time*	*integer, real, or time*
`-`	subtraction	8	*integer, real, or time*	*integer, real, or time*

Figure 3.6 Arithmetic Operators[a]

a. Note the absence of the pre- and postfix operators ++ and --.

Operator	Operation	Precedence	Type of Operands	Type of Result
`==`	equal	11	*any*[a]	*boolean*
`!=`	not equal	11	*any*	*boolean*
`<`	less than	11	*integer, real, or time*	*boolean*
`<=`	less than or equal to	11	*integer, real, or time*	*boolean*
`>`	greater than	11	*integer, real, or time*	*boolean*
`>=`	greater than or equal to	11	*integer, real, or time*	*boolean*
`:`	is a member of interval	11	*integer, real, or time : interval*	*boolean*
`!:`	is not a member of interval	11	*integer, real, or time !: interval*	boolean

Figure 3.7 Relational Operators

a. *Any* implies any base or compound data type. The equal and not equal operators can also be applied to arrays (**Section 3.9, “Complex Data Types” page 25**), but not to objects.

Operator	**Operation**	**Precedence**	**Type of Operands**	**Type of Result**
!	not	6	boolean	*boolean*
&&	and	12	boolean	*boolean*
||	or	13	boolean	*boolean*

Figure 3.8 Boolean Operators

Operator	**Operation**	**Precedence**	**Type of Operands**	**Type of Result**
`( )`	built-in function call	1	*any*	*any*

Figure 3.9 Function Operators

Operator	**Operation**	**Precedence**	**Type of Operands**	**Type of Result**
`::`	scope resolution	1	*classname*`::`*member*	*any*
`.`	member selection	2	*object*`.`*member*	*any*
`->`	member selection	2	*handle*`->`*member*	*any*

Figure 3.10 Member Operators

Operator	**Operation**	**Precedence**	**Type of Operands**	**Type of Result**
&	address of	3	*variable or object*	*handle*
*	dereference	3	*handle*	*variable or object*

Figure 3.11 Handle Operators

Operator	Operation	Precedence	Type of Operands	Type of Result
`new`	object creation	14	`new` *class*	*handle*
`delete`	object destruction	15	`delete` *handle*	*none*
`clone`	clone object	3	clone *object or object handle*	*handle*

Figure 3.12 Object Operators

Operator	Operation	Precedence	Type of Operands	Type of Result
`classOf`	get handle to metaclass object	3	`classOf` *object or object handle*	*handle*
`the-Class`	get handle to metaclass object	3	`theClass` *classname*	*handle*

Figure 3.13 Metaclass Operators

Operator	Operation	Precedence	Type of Operands	Type of Result
`+`	concatenate	8	*string*	*string*
`&`	concatenate with space	9	*string*	*string*

Figure 3.14 String Operators

Operator	Operation	Precedence	Type of Operands	Type of Result
`<<`	append	10	*list* `<<` *any*	*list*
`+`	concatenate	8	*list*	*list*

Figure 3.15 List Operators[a]

a. See also the built-in list functions of **Section 3.12, “Built-in Function Calls” page 29**, and the `for i in list { . . . }` construction of **Section 3.15, “Control Flow” page 33**.

Operator	Operation	Precedence	Type of Operands	Type of Result
`?`	is element	6	*array element*	*boolean*
`remove`	remove	15	*array element*	*none*

Figure 3.16 Array Operators

Operator	Operation	Precedence	Type of Operands	Type of Result
`?`	is value set	6	*any*	*boolean*

Figure 3.17 Is Value Set Operator

Operator	Operation	Precedence	Type of Operands	Type of Results
`<<`	put	10	*list* << *any* *object* << *any* *object* << *object*	*list* *object* *object*
`>>`	get	10	*list* >> *any* *object* >> *any* *object* >> *object*	*list* *object* *object*

Figure 3.18 Stream Operators[a]

a. Stream operators are further discussed in **Section 3.14, "Stream Operators" page 31**.

Operator	Operation	Precedence	Type of Operands	Type of Result
`=>`	send synchronous message	4	*string or list* `=>` *object or object handle*	*any*
`|>`	send asynchronous message	4	*string or list* `|>` *object or object handle*	*any*
`?=>`	send optional synchronous message	4	*string or list* `?=>` *object or object handle*	*any*
`?|>`	send optional asynchronous message	4	*string or list* `?|>` *object or object handle*	*any*

Figure 3.19 Message Operators

Operator	Operation	Precedence	Type Of Operands	Type of Result
`@`	resource resolution	5	*string*	*string*

Figure 3.20 Resource Operators

```
3.14159 * (radius * radius)
length("Dogs and cats") <= 255
userMessage + "\n"
{ 'ResizeButton } + Args |> myButton
(oldWidth == ( 'GetWidth => myButton ))
```

Figure 3.21 Sample Expressions

3.8 Assignment

Assignment statements allow you to assign new values to declared variables of base (**Section 3.3, "Base Types" page 14**) and compound types (**Section 3.5, "Compound Types" page 16**). You can use any such declared variable as an lvalue in an assignment. When you copy a string or list in an assignment, you perform a deep copy. When you make an assignment to a previously initialized string or list, you destroy the previous contents.

```
buttonWidth = 'GetWidth => myButton;

buttonWidth = 2 * buttonWidth;
```

Figure 3.22 Sample Assignments

Declared objects are neither base nor compound system types, and hence an identifier declared to name an object cannot receive an assignment. Handles, however, are a base system type and a variable declared as a handle can receive a handle to an object in assignment statements.

Note that unlike C and C++ there is no assignment operator, and assignment cannot be a subexpression. There are no compound assignment operators like(+=). The lvalue determining the target of the assignment is guaranteed to be evaluated after the expression on the right side of the assignment statement (**see Section 3.11, "Type Conversion" page 28** for a discussion of type conversion in assignment).

3.9 Complex Data Types

Indexed Arrays

An indexed array is a complex data type derived from one of the base or compound types. All elements of an indexed array must be of the same type, but since lists can be array elements this restriction is not severe. Indexed arrays cannot be members of lists, but a handle to an indexed array can be such a member.

The type and dimensionality of an indexed array must be declared before its use, but indexed array bounds are not fixed. Indexed array indices must be integers, but need not be positive or even non-negative. An indexed array expands as it receives values assigned to new elements beyond the previous bounds. It is a run-time error to use the value of an indexed array element before that element receives an assignment. An indexed array element can be the target of an assignment and can appear in any expression where a constant or variable of its type is appropriate.

You can assign indexed arrays provided that the *rvalue* for the assignment is an array of the same dimensionality and type. A deep copy occurs in such a case, and all the previous data from the target array is lost. Indexed arrays can also appear with the relational operator `(==)` provided that both operands are indexed arrays. Two indexed arrays are considered equal if they possess the same dimensionality, the same type, the same bounds, the same assigned elements, and the corresponding elements are equal in each array.

Declaration

```
integer anInt;
integer twoDimArray<2>, anotherArray<2>;
list listArrayA<1>;
```

Element Assignment, Expression Use and Initialization Checking

```
twoDimArray[0,1] = 5;
anInt = twoDimArray[0,1] + 1;
anInt = twoDimArray[0,0];  // error, used before set
```

Array Assignment and Operations

```
anotherArray = twoDimArray;
if ( anotherArray == twoDimArray )
{ /* Do This */ }
anotherArray[0,0] = 1;
if ( anotherArray == twoDimArray )
{ /* Don't Do This */ }
```

Figure 3.23 Indexed Array Usage

Use arrays with caution because they require considerable storage over and above that required for their elements. The relation between indexed arrays and pointers in C and C++ does not hold between *ADL*'s arrays and handles. In the *ADL*, you can pass the handle of an array as an argument to a built-in function or a message. Note that in this case the handle being passed points to the original array, not a copy of it

Associative Arrays

An associative array is a complex data type derived from one or more of the base or compound types. An associative array is similar to an indexed array except that the indices, or *keys*, need not be an integer but can be of any base or compound type, even a `list`. Certain index types, however, are probably not useful. For instance, a `real` index could be misleading.

All elements of a single associative array must possess keys of the same type. You declare the key and value types in the array declaration. As with indexed arrays, associative arrays are dynamic, growing as elements receive assignment. They can receive assignment as a unit, and their equality tested as such. Associative arrays are equal if and only if the values and keys of both arrays are of the same types, they possess the same keys, and elements with the same keys possess the same value.

Declaration

```
string captionArray<integer>;
list authorArray<string>;
string holiday<string, integer>;
string key;
```

Element Assignment, Expression Use and Initialization Checking

```
captionArray[19774] = "The old covered bridge";
key = "Joseph Conrad";
authorArray[key] = { 'Nostromo, 'Victory, "Lord Jim" };
holiday[ 'July, 4 ] = "Independence Day";
```

Figure 3.24 Associative Array Usage

Array Element Operators

There are two special unary operators that you can use with array elements or expressions that evaluate to array element *lvalues*.

- The `(?)` operator tests whether the following array element specification refers to an element that exists. Since arrays are dynamic, a given key may or may not exist within a given array. The (?) operator allows the user to test for the existence of an array element without generating a `used before set` error.
- The `remove` operator removes the following array element from the corresponding array.

```
if ( ?authorArray["Jane Austen"] )
{
    remove authorArray["Jane Austen"];
}
```

Figure 3.25 Array Element Operations

3.10 Unset Values

Database tables regularly contain optional fields. If you retrieve a record from a table with an optional field, and the field does not contain a value, the field is said to contain a null value. The semantics of null values and a null handle are similar but subtly different. A null handle has a value; it simply doesn't point to any object. Programmers often use a null handle to mark an end of a list or an inactive option rather than missing data. Null values, however, have no value at all. They are unset. The *ADL* supports the concept of unset values primarily to provide a more uniform interface to databases.

The constant `UNSET` is typeless and represents an unset value. It can be assigned to a variable of any type, used in **(==)** and `(!=)` comparisons and put on lists. The examples shown in Figure 3.26 are legal.

```
integer answer;
string question;
list record = { "When?" };
any  elt;

if ( answer == UNSET )
{
  question = "Why";
}
elt = at( 1, record );
if ( elt != UNSET && isString( elt ) )
{
  question = elt;
}
record << UNSET;
```

Figure 3.26 Operations with the Constant UNSET

All uninitialized variables have an unset value. Unset values can appear wherever the constant `UNSET` can with one exception: the comparison of two values is a run-time error if one or both are unset. You can, however, compare an unset value to the constant `UNSET`. It is important to distinguish an unset string or list from an empty one. You can append to an empty string or list, but not to an unset one. Unset values can appear in lists, however, and the list append operator (<<) can append an unset value to a list.

You can use the operator `(?var)` to test for the existence of an array element and to test if a variable is set`.`The expression `var == UNSET` is syntactic sugar for it. Similarly, you can compare array elements with `UNSET` to determine if they exist. Finally, assigning an array element the constant `UNSET` is equivalent to calling the operator `remove` except that it is an error to `remove` a non-existent element, but you can assign `UNSET` to one.

3.11 Type Conversion

Type conversion happens in the following instances:

- Assignment
- Arguments to built-in function calls
- Arguments to messages
- Promotion in arithmetic and numerical relational expressions

In the first three cases `integers` convert to `reals`, and `reals` truncate to `integers`, as indicated by the associated type declarations. `Strings` are not automatically converted to numeric values. You can use the *ADL* built-in functions (**see Section 3.12, "Built-in Function Calls" page 29**) `toInteger()`and `toReal()` to perform explicit conversions. In the case of promotion in expressions, any binary operator with one `integer` and one `real` operand promotes the `integer` operand to `real` before performing the operation. Certain compound types are also implicitly converted.

- A `time` converts to an `integer`, a `real`, or a `list` as required, and vice versa. The `integer` or `real` version of a `time` contains the corresponding number of milliseconds. The `list` version contains four elements: hours, minutes, seconds, and milliseconds.
- An `interval` implicitly converts to a `list`, and an appropriately formatted `list` to an `interval` (**see Section 3.5, “Compound Types” page 16**). An error results if you attempt to convert an inappropriate `list` to an `interval` or `time.`

The only other case of promotion occurs when a string on the left hand side of a send message operator is promoted to a one member list. (**see Section 3.13, “Messages” page 30**).

3.12 Built-in Function Calls

The *ADL* provides a set of built-in function calls to access functions from the C standard library and to operate on expressions of base type, compound type, and complex type. **Appendix A, “Built-In Functions for ADL” page 261** documents these functions, which currently fall into the following categories:

- **“Input/Output” page 261**
- **“Time and Date” page 262**
- **“Conversion” page 262**
- **“Type Query” page 263**
- **“Sequences (lists and strings)” page 264**
- **“Mathematical” page 266**
- **“Handles” page 269**
- **“Classes and Inheritance” page 269**
- **“Networking” page 269**

User-defined functions are not permitted. Messages to global objects currently provide equivalent functionality. All arguments to built-in functions are passed by value, although such an argument can be a handle to an underlying base, compound, complex, or object type.

```
random( [1,5) )

length( name )
```

Figure 3.27 Sample Function Calls

3.13 Messages

A message is an operation that one object performs upon another. You can send a message to any object. It consists of three parts:

1. **Selector (message name)**: The string identifier for the message must have a match in the receiving object’s method dictionary.
2. **Message arguments**: Arguments are optional. The message selector and any arguments form the message list.
3. **Message target**: A reference to an object, or to the handle of an object, that is to receive the message list.

```
'Construct => Viewer;

'Init?=> Viewer;

{mName} + {"width", 40} |> myTarget;

dayMonth = 'GetDayOfMonth => Calendar
```

Figure 3.28 Sample Messages

The target object method has the option to return a value. If a method returns a value in some cases, it must return a value in all cases. The returned value may be *unset*. Messages that return a value can appear in expressions. Messages that do not return a value must appear in standalone statements.

If there are no message arguments, the message list can consist of a `string`. Otherwise, the message list must be a true list whose first element is a `string` specifying the message selector. The message list can be a `list` variable or a `list` valued expression. It is evaluated at run-time just before the message is sent so that all parts of the message, including the selector, are dynamic. A message argument may be *unset*.

One of the following message operators separates the message list and the message target:(=>),(|>),(?=>),and (?|>). The operators (=>) and (?=>) specify that the message is to be sent synchronously — that is, the message is sent immediately and the invoked method is executed before the next statement in the calling method. A synchronous message thus effectively creates a new stack frame. The operators (|>) and (?|>) conversely specify that the message is to be sent asynchronously —that is, the message is queued for later delivery. The system guarantees that an asynchronous message will be delivered and executed before the return to the overall event loop if an appropriate method exists in the destination object. No guarantees are made, however, about the relative execution of multiple asynchronous messages.

The operators (?=>) and (?|>) send optional messages, while the operators (=>) and (|>) send required messages. It is an error to send a required message to an object that does not possess a method for the message selector. If an object receives an optional message with a selector that it does not understand, however, it simply ignores the message. Optional messages are particularly useful for broadcasting system messages to all objects.

Since message arguments are passed as members of a `list`, they must be passed by value. No object can be a member of a `list`, and thus, no object can be an argument to a message. A handle to an object, however, can be passed as an argument to a message. The return value, if present, must also be one of the base or compound system types and is also passed by value. A message cannot return an object or a complex data type (i.e., indexed or associative arrays).

3.14 Stream Operators

To simplify the use of streams, the ADL provides two stream operators: put (<<) and get (>>).[2]

The operators act differently depending on their arguments. In all cases, the intent is that the << operator sends data into a stream and the >> operator receives data from the other end of the stream. In the next few paragraphs, we describe how the operators work for each set of argument types. Any other argument combinations are type errors. In all cases, the result of the expression is the left operand. When the first argument is an *lvalue* whose type is list, as in Figure 3.29, these operators make the list act as a stream.

```
list_lvalue << any_value;

list_lvalue >> simple_variable;
```

Figure 3.29

The << operator appends `any_value` onto the list specified by `list_lvalue`. The >> operator reverses this operation, removing the first value from the `list_lvalue` and storing it in the `simple_variable` on its right. Using the >> operator on an empty list is an error. It is also an error if the value to be stored by the >> operator does not match the type of `simple_variable`. Figure 3.30 shows the object on the left as a stream into which we put (<<) or from which we take (>>) simple types. The send operator (<<) sends the object a message with one argument, the value on the right hand side.

```
object << any_value;

object >> simple_variable;
```

Figure 3.30

[2] These operators differ from their C++ counterparts in that the meaning of the "send" and "receive" operators are essentially defined by the ADL. Class authors who define methods to handle the messages sent by the operators may modify their effects on objects.

The selector of this message is based on the type of the value. The selector is `"Send`” concatenated with the name of the type of the value. For instance, the following expression causes the message `{"SendBoolean", TRUE}` to be sent to `myStream`:

```
myStream << TRUE
```

The receive operator (>>) sends the object a message with no arguments whose selector is based on the type of the `simple_variable`. The selector is "`Receive`" concatenated with the name of the type of the variable. This message returns a value of the requested type. The operator then assigns this value to the `simple_variable`. For instance, in the following:

```
integer i;
myStream >> i;
```

`myStream` is sent the message `ReceiveInteger` and the value returned is assigned to `i`. When both operands are objects, as in Figure 3.31, the operators help the object on their right send and receive itself from the object on the left (presumably a stream).

```
object << object;

object >> object;
```

Figure 3.31

To accomplish this, the object on the right is sent a message with a handle to the object on the left. The selector of this message is either `SendTo` (<<) or `ReceiveFrom` (>>).

3.15 Control Flow

The *ADL* provides a standard and predictable set of control structures, as shown in Figure 3.32:

```
if ( BoolExpression ) { . . . }
[ else if ( BoolExpression ) { . . . } ]
. . .
[ else { . . . } ]
```
```
for i in list { . . . }
```
```
for i in array { . . . }
```
```
while ( BoolExpression ) { . . . }
```
```
do { . . . } while ( BoolExpression );
```
```
cond {
   ( BoolExpression1 ) { ... }
[  ( BoolExpression2 ) { ... }]
[ ... ]
}
```
```
break;
```
```
continue;
```
```
return [ value ];
```
```
forward => [ selector @] object/handle;
```

Figure 3.32 Control Structures

There are three non-C++ control structures:

1. `for i in list { . . . }` iterates over the elements of the list and executes the code in the block repeatedly with the variable i assigned to each member of the list in turn. This usage requires that all list elements are of the same type (the declared type of the variable `i`) or that `i` be of type any.
2. `for i in array { . . . }` iterates over the state of the array when the `for` statement is first entered. For each such assigned element, `i` is assigned a list whose first member is the value of the array element, and whose second through $n + 1^{st}$ members are the values of the element's indices, where `n` is the number of array indices. The order in which the elements are visited is unpredictable.
3. `cond`, borrowed from LISP, guarantees that at most one of the code blocks within the cond block is executed, the first that is preceded by a TRUE boolean expression. A default can be specified using TRUE.

We discuss the `forward` statement in **Section 3.21, "Method Definition" page 41**.

```
list alist;
any value

for value in alist {
    if (isInteger (value)) {
        echo(“Value is an integer\n”);
    {else   {
        echo(“Value is not an integer\n”);
        }
}
integer anArray<string>;
list alist;

anArray[“Steve”} = 43;
anArray[“Lori”] = 42;

for alist in anArray{
    echo(“Age of”&at(2,alist)&“is”&at(1,alist)+“\n”);
}

integer ival = 7;

cond{
     (ival >= 2) {
           echo(“Integer greater than 2\n”);
    }
     (ival >= 0 {
            echo(“Integer greater than or equal to 0
                  but less than 2\n”);
             }
      (TRUE) {
             echo(“Integer is negative\n”);
             }
}
```

Figure 3.33 Non-C++ Control Structures

3.16 Object Definition

Defining an instance of an object is similar to defining a variable, but the initialization of objects is more complex. We deal with this topic more fully in **Section 3.25, “Object Initialization” page 49**, where we discuss the process of initializing objects. In this section we are concerned only with the syntax for defining objects.

The default object declaration takes the form of a *class name* followed by a non-null list of object identifiers. More generally, an *object declarator* substitutes for each identifier. An object declarator is an identifier combined with two optional constructions that specify how the object is to be initialized.

Simple Object Declaration

```
MyClass myObj1, myObj2;
```

Object Declaration with Custom Constructor

```
MyClass { 'CustomConstruct, arg1, ..., argn} => myObj;
```

Object Declaration with Initializor Block

```
MyClass myObj1 {
   height=100;width=100;
};
```

Object Declaration with Special Constructor and Initializor Block

```
MyClass { 'CustomConstruct, arg1, ..., argn} => myObj1 {
   height=100;width=100;
};
```

Figure 3.34 Object Declaration

The simple form of object declaration shown in the first example of Figure 3.33 would create two instances of the `MyClass` named `myObj1` and `myObj2` respectively. When these two instances are created, the standard message `Construct` is, by default, sent to each instance. The more complicated example shown in that same figure illustrates how the default `Construct` message can be over-ridden. In this example, the message

```
{‘CustomConstruct, arg1, ... argn}
```

is sent when the instance `myObj` is created instead of the simple `Construct` message. This requires, of course, that the class named `MyClass` have a constructor method named `Custom-Construct` and that the member and type of the arguments in the message agree with those specified for the method. A class can have any number of constructor methods, but only one of these can be used to create any given instance of the class. As discussed further in **Section 3.25, “Object Initialization” page 49**, constructor methods are distinguished fom ordinary methods in *ADL* by beginning them with the keyword upon rather than the keyword on.

The third example in Figure 3.34 illustrates the use of *initializor blocks*. These blocks consist of a series of *ADL* statements enclosed in curly brace delimiters. Initializor blocks allow an *ADL* programmer to set values for the attributes of an instance of a class being constructed. In the example in the figure, the attributes `height` and `width` are each initialized to 100. Initializor blocks (or *izor blocks* for short) are used very often by *ADL* programmers becasue they provide a very convenient way to set attribute values for newly-created objects. However, it is important to stress that the *ADL* statements in an izor block are executed as though they were part of the object being created, not as part of the object in which they actually appear. This is termed a *foreign scope* in *ADL*. This means that the right hand side of the assignment:

```
height 100
```

refers to the variable named `height` in the new instance of `MyClass` being created. The consequences of an *izor* blocks being executed in a foreign scope are often not obvious to beginning *ADL* programmers. For example, consider a fragment of *ADL* code as follows:

```
integer xvalue=50;    //declare an integer initialized to 50
MyClass myObj {height=zvalue;}; //incorrect use of izor block
```

One migh incorrectly expect that the value of the attribute named `height` in `MyClass` would be set to 50 by this code. Instead, this code will produce an error because the *izor* block that attempts to set `height` is executed in the scope of the class `MyClass`, and the variable `xvalue` on the right hand side of the assignment is not defined in this scope. Thecorrect way to achieve the desired result in *ADL* would be using the following cde fragment:

```
integer xvalue=50;  //declares an integer initialized to 50
MyClass my Ob;      //create instance of MyClass named myObj
myObj.height=value; //set the height attribute of myObj
```

The fourth example in Figure 3.34 combines the use of a custom constructor with the useo f an izor block. While the detals of the order in which things occur when a new object is constructed are covered in much more detail in **Section 3.25, “Object Initialization” page 49**, it is worth noting that the execution f the custom constructor occurs before the execution of the *izor* block. Thus, any statements in the constructor method that set values of attributes that are also set in the *izor* block will be over-ridden.

An object declaration can contain multiple object declarators, each with its own identifier, just as it can contain multiple simple identifiers. Object instances are not variables and cannot receive assignments.[3] Objects declared within a scope are destroyed upon exit from that scope. Never destroy declared objects using the `delete` operator. In defining an object, you create a new instance of the wrapped C++ class, initialize it, and make it visible to the *ADL* in the current scope if the object is an instance of a system-defined wrapped class. If the object is an instance of a user-defined *ADL* class, then you create a new instance of that class along with all its members and bases (**see Section 3.23, “Wrapped Classes” page 44** for more information on wrapped, or system defined, classes and user-defined classes).

3.17 Dynamic Objects and Storage Management

You can create objects dynamically by using the `new` operator with the name of, or a handle to, a defined class. Such an object does not possess a name, and the user refers to it solely through the handle returned by the `new` operator. The object continues to exist even after exit from the enclosing scope, and if you are not careful, it is easy to destroy the handle accessing the object. Free the object using the `delete` operator before destroying its handle.

3 This avoids forcing the user to confront the complexity of the C++ copy constructor. Default cloning is provided (**see Section 3.17, “Dynamic Objects and Storage Management” page 36**).

You can use the new operator with both special constructors and initializor blocks. The created object is initialized using the default `Construct` method unless a special constructor message intervenes between the `new` operator and the class name. The optional initializor block follows the class name.

You can also use the `new` operator to create new instances of base and compound types, including arrays. As with objects, the new operator returns a handle to the `new` instance and not the instance itself. Again, free the instance using the `delete` operator.

```
class MyClass
{
   string name;
   upon Construct { ... }
   upon CustomConstruct : list date { ... }
};

class Foo
{
   on DoIt
   {
      handle aClassPtr, bClassPtr, aListPtr;
      aClassPtr = new MyClass;
      date = { 19, 'May, 1990 };
      bClassPtr = new {'CustomConstruct, date}
                        => MyClass { name = “M.I.T.”; };
      aListPtr = new list;
   }
};
```

Figure 3.35 Dynamic Instancing Example

The `clone` operator (**unimplemented**) can be applied to an object or object handle to produce a copy of an object. The default method for doing this is to first simulate a use of the new operator on the object’s class, and then do a recursive `clone` on member objects, a bitwise copy on members of base data types, and a deep copy on compound data members.

A non-default `clone` procedure is specified by defining a `Clone` method in the class to be cloned. This method is called with the handle of the cloned object as an argument. The method gets called after the simulated `new`, and substitutes for the remainder of the default `clone` procedure. Note that `Clone` is called as a method on the freshly cloned object, rather than on the object being cloned. Thus a `Clone` method should have no return value. Destroy cloned objects using the `delete` operator.

3.18 Object Destruction

AM2 destroys an object declared within a scope on exit from that scope. The `delete` operator also destroys an object. Apply it only to the handle of an object returned by the `new` or `clone` operators.

The default destruction of an object involves:

- The recursive destruction of all member objects and variables in reverse declaration order
- The recursive destruction of all bases in reverse declaration order
- The destruction of the deleted object

This may be insufficient if the object contains handles to other objects or data allocated using the new operator. So that you can specify custom actions to accompany an object's destruction, *AM2* checks for a `Destroy` method whenever you delete an instance of a user-defined class. *AM2* calls this method, if it exists in the deleted object's class, before the destruction of member objects and bases, and the freeing of object storage.

3.19 Object Member Reference

You can access the data members of an object using the (`.`) operator. If `myObject` is the name of an object instance generated by a declaration, then `myObject`.*memberName* refers to the appropriate object member. Access control is discussed in **Section 3.20, "Class Definition" page 40**. You can chain object references using the (`.`) operator to refer to members of objects that are themselves the members of an enclosing object. For instance, imagine a class named Viewer that describes a standard video viewer, and that possesses a button named Forward that starts video play. Then you might have the following statements:

```
{
   Viewer myViewer;

   myViewer.Forward.width = 40;
}
```

Figure 3.36 Object Member Reference

If an object member reference occurs in an expression, then the value is simply accessed. If an object reference occurs as an lvalue, that is as the target of an assignment, and the assignment appears in a method applied to the object itself, then a simple assignment is performed. Such assignments are called *native*. If, however, the object member reference occurs in a method applied to some other object, then the run-time system checks for a standard `Set_`*name* method in the class of the object whose member is the target of the assignment, with the member name as part of the selector and the value of the right side of the assignment as the argument.

Assignments that invoke `Set_` methods are called foreign. If such a `Set_` method is not found for a foreign assignment, then a standard assignment is applied.

For an example of native assignment, consider a class named `Book` that has an integer member called `CurrentPage`. If you call a `NextPage` method on an instance of the `Book` class, and the method contains the statement,

```
CurrentPage = CurrentPage + 1;
```

then, in effect, you increment `CurrentPage` without calling the `Set_CurrentPage` method. In contrast you use foreign assignment when assigning to members defined in bases. Thus, if class `Dictionary` inherits from `Book`, and a method of `Dictionary` contains an assignment to `CurrentPage`, this assignment invokes the `Set_CurrentPage` method, if it exists.

All classes possess a default `SetAttributes` method. This method takes one argument, a specially formatted list of lists, each of whose sublists consists of a pair of elements, a string member name, and a corresponding value. The `SetAttributes` message is handled by iterating over the argument list, and performing the appropriate *foreign* assignment for each member/value pair. You can use a `SetAttributes` message to set members in bases. You can override the `SetAttributes` method, but it is probably dangerous to do so. This feature is similar in intent to the X toolkit *varargs* style interface, and it simplifies the writing of special constructors (**see Section 3.25, “Object Initialization” page 49**).

```
XFbutton myButton;
list buttonList = {
                    { 'x, 50 },
                    { 'y, 50 },
                    { 'width, 200 },
                    { 'height, 100 }
                  };

{ 'SetAttributes, buttonList } => myButton;
```

Figure 3.37 Sample SetAttributes Call

You can create a `handle` to an object or a variable using the `(&)` operator. You cannot, however, take the `handle` of an object member that possesses a `Set_` method. You can access object members through a `handle` using the `(->)` operator. The `(*)` operator dereferences a `handle`, but it cannot be applied in such a way that it returns an inappropriate target for an assignment. For example, you cannot use it to return an object on the left hand side of an assignment.

```
{
   Viewer myViewer;
   handle vwrPointer;

   vwrPointer = &myViewer;
   vwrPointer->Forward.width = 40;
}
```

Figure 3.38 Object Handles

Note that unlike C++ pointers, *ADL* handles are dynamically typed. The built-in function `isKindOf()` provides run-time type checking of handles (**see Section 3.22, “Metaclass Operations” page 43**).

The keyword `parent` is always a handle to the object that has the current object as a direct member. In a method, the current object is always the object on which the method is being invoked. In an initializor block, the current object is the one being initialized. If the current object was dynamically created using the `new` operator, then there is no `parent`. This case can be tested using another keyword, `theHeap`. For dynamically created objects, `parent == theHeap`.

3.20 Class Definition

In the *ADL*, a user can define a new class from scratch (user-defined), create a subclass of a user-defined class, or create a subclass of a wrapped (system-defined) class. Class definitions can nest and you can define a class within another class, but not within its methods.

There are two forms for class definitions. The more common form defines a named class from which the user will later declare several instances. Such a class definition introduces a new type name into the *ADL* just as it does in C++. It consists of the following:

- the keyword `class`
- the name of the new class
- a colon if the new class is an explicit subclass followed by the names of parent classes (separated by commas)
- an open curly bracket ({)
- the class member declarations, if any
- the class method definitions, if any
- a close curly bracket
- an optional comma-separated list of instance names
- a trailing semicolon

```
class MyClass [: Base1[, Base2[, ... ]]] {
   Member Declarations
   Method Definitions
} [instance1[, instance2[, ... ]]];
```

```
anonymous [: Base1[, Base2[, ... ]]] {
   Member Declarations
   Method Definitions
} [instance1[, instance2[, ... ]]];
```

Figure 3.39 Class Definitions

You can declare instance variables of the defined class immediately by appending their names after the closing curly bracket of the class definition.

The second form defines an unnamed class, and is used when all the instances of the unnamed class can be declared at the time the class itself is defined. This form of the class definition substitutes the key word `anonymous` for `class` and omits the class name. By default, all instances of a class contain their own copies of class data members. If the intention of the application developer is that all class instances should share *one* copy of a data member, then that data member's declaration should be prefaced with the `common` keyword (**unimplemented**). `Common` members correspond to the `static` class members of C++. The ADL has adopted a different nomenclature because the word `static` is overused in C++.[4]

A subclass can redefine a member or method of a parent class. Such a redefinition is said to *hide* the original member or method definition in the parent class. That is, the subclass cannot access the parent member or method without using a scoping operator (`::`) (**see Section 3.24, "Inheritance" page 47**).

AM2 currently provides only primitive access control to limit the developer's ability to get and set the values of a class instances members. The ADL does not possess a scheme similar to the C++ categories of `public`, `protected`, and `private` members. In C++ all access control is checked at compile-time, which is possible because of the language's strict type checking. The *ADL* is less strictly typed (`handles` form a single type), which forces any access control to be implemented using run-time mechanisms. We believe that it would be too computationally-intensive to implement the three C++ access categories. One possibility under consideration is to dispense with the `protected` category and to modify the `private` category so that `private` members and methods of an object could only be accessed from within that object itself. In C++, `private` methods can be called on an object from any object of the same class. The developer can use `Set_` methods (**see Section 3.19, "Object Member Reference" page 38**) to make a member read-only in methods outside the immediate class and or in class methods applied to other instances of the same class.

Since wrapped classes are system-defined you cannot modify them, but you can create subclasses. One of the great strengths of the *AM2* environment is that wrapped classes are almost indistinguishable from user-defined classes, yet they are implemented in C++ and may interface to third-party libraries. The following section discusses wrapped classes and the relationship to user-defined classes in more detail.

3.21 Method Definition

Method definitions are contained in blocks introduced by the keyword `on` followed by the *message prototype*. The prototype must begin with an identifier called the *selector* that is unique within the method's class. The selector appears as an unquoted character string as the first component of the *message list* (**see Section 3.13, "Messages" page 30**). The *selector* can be the same as the *selector* of a method in the parent class or one of its ancestors. In this case, the new method overrides the ancestor's method.

[4] See B. Stroustrup, *The C++ Programming Language*2 (1991) 166.

If the method receives arguments, you must declare these arguments in the message prototype after the selector and a colon separator. The arguments must be of base or compound type. Neither arrays nor objects can be message arguments, but handles to them are allowed. The type of the arguments, if they are present, are checked each time a method is invoked. Message arguments may be unset.

The *ADL* does not allow the overloading of selectors. That is, a method is always called with the same type of arguments in the same order. The same method selector cannot be specified for two separate methods that are distinguished by their argument lists, as in C++. The method can return a value, once again of base or compound type. If it does so, you must declare the type of the return value as the last part of the prototype following all the argument declarations. This declaration takes the form of the keyword `return` followed by the type of the return value.

The block that follows the message prototype contains the executable *ADL* code that defines the method. If a method declares a return value, this code must indicate the return value using the keyword `return` followed by an expression that evaluates to the return value. The returned value may be *unset*. The method terminates upon execution of a `return` statement, or if the method does not return a value it terminates after execution of the last statement of the block. In methods with no return value, do not follow a `return` statement indicating premature termination by an expression.

In an object method, the keyword `self` is always a handle referring to the object and the keyword sender is always a handle to the object that sent the message currently being handled. If the message was sent by the system, e.g., a default or special constructor or an `Init` message, then `sender` is `NULL`. An object can directly set the value of an object member without using a `Set_` message to self.

Methods, like object members, can be `common`, that is, they can apply to the class as a whole rather than a particular instance of the class (**unimplemented**). You can declare a common method by inserting the keyword `common` between `on` and the *selector* in the method definition. Common methods have access only to the common members of a class, not to the regular members, because you cannot refer to a regular member without referring to an instance.

You may declare a method to be `local` if you do not wish the method to be inherited by a subclass. That is, a local method may not be invoked by a message directed to an instance of a derived class.

See Figure 3.40 for the general form of a method definition. The colon is required if either an argument list or return type is present.

```
on [common]  [local]  selector [:[type1 arg1, ... , typen argn]
[return  rettype]]
{
   Method Declarations
   Method Statements
}
```

Figure 3.40 Method Definitions

The `forward` statement (**unimplemented**) indicates that the current message is to be forwarded to another object. The statement has two forms illustrated in Figure 3.41.

```
forward => object/handle;

forward stringSelector => object/handle;
```

Figure 3.41 Two Forms of the Forward Statement

In both cases, execution transfers to a method in the new target specified by the object or handle appearing after the `(=>)` operator. The first case of this example issues an identical message to the new target. The second case uses the same arguments, but specifies a new *selector* by a string. The new target appears to return directly to the message sender in the case of a synchronous two-way message, not to the forwarding object. An object can forward a message to a different method within itself by using the construction

```
forward 'newSelector => self;
```

The forward statement is largely syntactic sugar except that it guarantees that the message is forwarded synchronously, even if the original message is asynchronous. That is, once a method has started to handle a message, the `forward` statement is seen as an extension of the original handler, and not the transmission of a new message.

3.22 Metaclass Operations

The future development of editors in the *ADL* requires the capability to create *ADL* subclasses dynamically. Such an editor may need to access the class (metaclass object) of a wrapped or *ADL* class. The `classOf` operator, when applied to an object identifier or an object handle, returns a handle to the corresponding metaclass object. It may also be necessary to send a message to the metaclass object of a named class. The operator `theClass`, when prefixed to a class identifier, likewise returns a handle to the specified class.

```
class MyClass : MyBase { ... } myObject;
class YourClass { ... };

on Foo
{
  if ( isKindOf( &myObject, theClass MyBase ))
  {  /* Do This */ }
  if ( isKindOf( &myObject, theClass YourClass ))
  {  /* Don't do this */ }
}
```

Figure 3.42 Run-time Type Checking Using isKindOf()

The `isKindOf(hObject,hClass)` built-in function provides run-time type checking of *ADL* handles. You can call it with two arguments: the first is a handle to an object, the second is a handle to the metaclass object for a class. The second argument is usually derived from one of the expressions shown in Figure 3.42.

The `isKindOf()` function returns TRUE if and only if the class pointed to by the second argument is the class or a base of the class of the object pointed to by the first argument. You can create an instance of a class from a handle to the metaclass object using the `new` operator by enclosing the handle in parentheses, as shown in Figure 3.43.

```
handle hButton = new ( theClass XFbutton );
handle hButton = new {‘Create, self} => ( theClass XFbutton );
```

Figure 3.43 Creating a Class Instance from the Handle to a Metaclass Object

3.23 Wrapped Classes

Wrapped classes are C++ classes that are “wrapped” with the necessary information to make them usable in the *ADL*. Wrapped classes are also called system-defined classes. They cannot be modified although they can be adapted through subclassing. *AM2* comes with a set of wrapped classes, which are described in **Chapter 6, “Wrapped Class Reference” page 111**.

By convention, the names of wrapped classes begin with two capital letters that define the module the class belongs to, followed by a lower case word specifying the class within the module. If more than one word is necessary, the names are concatenated with second and later words starting with a capital letter. The following samples are typical wrapped class names:

```
XFtop        // a top level frame or shell window
XFbutton
XFcheckBox   // note the capitalization
MMimage
```

The exceptions to this convention are the set of wrapped notification classes (**see Section 4.2, “Using Notification Request Objects” page 62**) and certain abstract wrapped classes, e.g., `ActivityManager`, that never appear in *ADL* programs. Most programmers try to distinguish their ADL from wrapped classes by starting the names of *ADL* classes with only a single capital letter, e.g., `ExitButton`. Note that `XFtop` and `XFbutton` will frequently be used as example wrapped classes in this document.

System developers can also create wrapped classes using the wrap script (**see “Creating Wrapped Classes” on page 271**). We use wrapped classes and user-defined classes (those written in the *ADL* by an application developer) in almost exactly the same way, with some restrictions on member access for wrapped classes. This section discusses how wrapped classes differ from user-defined classes.

There are five things that you can do with any class: refer to it by name, create a subclass of it, create an instance of it, access members of its instances, and invoke methods of its instances. Let’s look at each activity in turn.

3.23.1 Scope of a Class Name

An *ADL* programmer specifies a class by its name when creating a subclass or an instance of the class, and when getting a handle to the class (using `theClass` operator). The scope of a class name represents the part of the program in which the name refers to that class (**see Section 3.26, "Scope" page 52**). A wrapped class' name has the same scope as that of a non-nested, user-defined class. This means that a wrapped class can be used anywhere in an *ADL* application.

One difference between wrapped classes and user-defined classes is that the latter, when defined with the same name as a wrapped class silently hides the corresponding wrapped class. For instance, suppose there is a wrapped class named `XFselectList`. A user-defined class named `XFselectList` would hide the wrapped class of the same name so that any objects of type `XFselectList` would be instances of the user-defined class, not instances of the wrapped class. Two user-defined classes with the same name generate a semantic error.

3.23.2 Creating an Instance of a Class

You can instantiate, that is represent by a concrete instance, all wrapped classes with the exception of an abstract wrapped class. An abstract wrapped class is one that exists only to provide organization in the hierarchy of classes. It is not a complete class in itself and therefore instances of it are not allowed.

3.23.3 Creating a Subclass

You can create a subclass, which is a class that inherits from a superclass, of most wrapped classes. For example, an *ADL* program might create a subclass of the wrapped class `XFtop` to add `XFbutton` member instances as part of creating separate user interfaces.

Some wrapped classes, however, do not allow for the creation of subclasses. For example, the wrapped class `XFwidget` is an incomplete class that exists to provide organization in the hierarchy of classes. Most abstract wrapped classes are in this category. One notable exception is the `ActivityManager` class, an abstract class that you can use to create subclasses (**see Section 4.5, "Creating ADL Classes That Manage Activities" page 70**). It is abstract because without the information stored in its subclasses an instance of it has little use. It is the subclass that provides the information that makes it useful.

3.23.4 Member Access

Wrapped classes can have simple and compound members, namely booleans, integers, reals, strings, lists, intervals, times, and handles. They cannot have arrays or instances of other objects as members. There are three operations that you can attempt on a simple or compound member of an object: taking a handle to the member, getting the member's value, and setting the member's value. The following sections describes the operations available for using those members and the restrictions on the way they are referred to by name.

Operating on Members

We call instances of wrapped classes *wrapper objects*. Taking a handle to a member of a wrapper object is not allowed because the member does not necessarily exist. Some members of a wrapper object are actually implemented by making method calls on the C++ object that implements the wrapper instance. Other members, known as *attributes,* are actually contained and managed by the underlying windowing or operating system. For instance, think of all the possible attributes of a button, width, height, borderWidth, foreground, background,etc. The XFbutton passes management of these attributes down to the underlying windowing system on each platform.

You cannot assign unset values to members of wrapped classes. For the most part, however, you can get and set the members of a wrapper object just like the members of an instance of a user-defined class. Just as a user-defined class can prevent the setting of one of its members by providing a `Set_memberName` method, and either not setting the member or calling the `die` built-in function, a wrapped class can prevent the getting or setting of its members in a similar way. For example, the `width` member of the `MMimage` wrapped class is read-only and attempting to set it is a fatal error. See the documentation for wrapped classes in **Chapter 6, "Wrapped Class Reference" page 111** to determine which members are accessible.

Naming of Members

As with members of user-defined objects, you can refer to each member of a wrapper object by its name. For example, the class `XFlabel` has a member called `width`. In subclasses of `XFlabel`, you simply refer to `width` if the subclass does not have another member with the same name that hides it. From outside the label class, you can refer to `width` as a member of `XFlabel` by a construct similar to `myLabel.width` or `handleToMyLabel->width`.

Unlike the members of a user-defined object, in a wrapper object you cannot qualify the members of a wrapped class by the name of the base that provides it. For instance, `XFfontable` is a superclass that provides the font member for other classes, such as `XFtext`. You cannot refer to this font member as `XFfontable::font`. You can, however, refer to `XFtext::font`.

Qualifying `self`

Unlike the base objects of a user-defined object, you cannot access the base of a wrapped object from the *ADL*. For instance, suppose that `MySelectList` is a user-defined subclass of `XFselectList`. It is an error to access the `XFwidget` base of `XFselectList` from within `MySelectList` using the conventional `XFwidget::self` because there is not necessarily an object that properly represents this base.

3.23.5 Method Invocation

The invocation of methods works just as it does for user-defined objects, except that a method cannot be sent to a base class of a wrapped class. In the *ADL,* a message goes to the appropriate base object to invoke an overridden method of that class. Since it is not possible to access any base object of a wrapped object, as discussed in the previous section, it is not possible to access overridden methods.

3.24 Inheritance

All variable or object members defined in a parent class are accessible in a subclass unless they are *hidden* by a member of the same name. In such a case, the inherited members are still accessible, provided that you identify them using the scoping operator `ParentClassName::memberName`. Such an expression is known as a *scope pair*. If the parent class name is not known, you can use the keyword `inherited` in place of a class name before the scoping operator to access the nearest occurrence of an otherwise *hidden* member in the chain of inheritance.

```
class Dad
{
   XFbutton button;
   on DoIt { ... }
};

class Child : Dad
{
   XFbutton button;
   on Foo
   {
      string message;
      button.width = 40;  // OK. Member hides parents
      'DoIt=>self; // Dad::DoIt()
      inherited::button.width = 100; //Dad::button
   }
};
```

Figure 3.44 Example of Inheritance and Member Concealment

You can invoke a method defined in a parent class by a message to a derived class unless the method has been declared `local` in the parent class. The exceptions are the system messages `Init` and `Destroy`, the system default constructor message `Construct`, and any other method declared as a constructor by the developer. These methods are local to the class in which you declare them by default, and you may not invoke them by a message to a derived class. The rationale for this is that these messages are sent to objects and bases during initialization and destruction. If these messages could be inherited and were not redefined in each class, they might execute multiple times during the initialization or destruction of an object.

You can redefine or override methods in a derived class. If you want to access the version of a method defined in a base, you can direct the message using the `self` keyword qualified with the scope operator and the name of the base, provided the message executes in one of the methods of the derived class. There is currently no way to send a message to a base of an arbitrary object although there is a proposal to allow *scope pairs* with member `self` in member selection expressions Figure 3.45.

```
class A
{
  on DoIt { ... }
};

class B : A
{
  on DoIt
  {
    ...
    'DoIt => A::self;
  }
};

B b;

on Init
{
  'DoIt => b.A::self;  // proposal; not implemented
}
```

Figure 3.45 Example of Inheritance and Method Concealment

Since a user-defined class can be a subclass of multiple classes, the *ADL* supports multiple inheritance. Multiple inheritance creates ambiguity when there are methods or members defined with the same name in two separate parent classes of the same subclass. Such ambiguous references in a method of a subclass are an error in C++, but the methods used to detect such conflicts are compute intensive and more appropriate for the compile-time type checking of C++ than the run-time lookup of the *ADL*.

AM2 searches superclasses in a depth-first manner during inheritance lookup. If this order is not appropriate, the author may use `inherited` declaration (**unimplemented**) to indicate in which subclass a method or member should be sought. The keyword `inherited` introduces such a declaration, followed by a base class name, the scoping operator, the type of the inherited member in the base, and the member name. After such a declaration, the member name refers to the similarly named member in the specified base, even if that member is not the first of that name in inheritance order. Figure 3.46 provides an example.

The choice of depth-first search is arbitrary but simple, and has the virtue that it establishes an unambiguous priority among parent classes. In the examples in Figure 3.46, `Child` is more closely related to `Dad` and `GrandDad` than it is to `Mom`.

The usage of the keyword `derived` in a *scope pair* with a member name or with the `self` keyword parallels the usage of the `inherited` keyword. The expression `derived::self`, when it appears in a method of a base of a derived object, refers not to the base but to the whole derived object. Likewise, `derived::` prefixed to a member name refers to the member not (necessarily) in the base but rather the first occurrence of the member in inheritance order in the whole derived object. The rationale for the `derived` keyword is to provide part of the functionality of virtual functions in C++. You can divide this functionality in two parts:

```
class GrandDad
{
   on Go { ... }
}

class Dad : GrandDad
{
   XFbutton button;
   on DoIt { ... }
};

class Mom
{
   XFbutton button;
   on DoIt { ... }};
   on Go { ... }

class Child : Dad, Mom
{
   inherited Mom::XFbutton button;  // unimplemented
   on Foo
   {
      string message;
      button.width = 40;  // Uses Mom::button
      'DoIt => self;      // Uses Dad::DoIt
      'Go => self;        // Uses GrandDad::Go
   }
};
```

Figure 3.46 Multiple Inheritance and Scope Example

- To implement a regulated polymorphism where C++ pointers to base combined with virtual functions allow the developer to treat instances of related classes as part of the same collection and still to have the instances retain the specific behavior of their classes
- To allow communication from a base to the full derived object

ADL handles provide a freer, though less safe, version of C++'s polymorphism. Indeed, since handles are untyped, they provide total polymorphism. There is no check that an object will understand a message until the message is received at run-time. The polymorphism provided by C++ virtual functions is therefore unnecessary. But communication from base to derived object can be very important, especially in the case of mix-in classes. The `derived` keyword provides this communication. (For an implementation of the `derived` keyword, refer to the example program in **Figure 4.11, "A Class Inheriting from the ActivityManager Class" page 76**.)

3.25 Object Initialization

An `ADL` description of an *AM2* application consists of class definitions and initialized instances of those classes. Most class definitions include object members that are instances of other classes.

The initialization of these instances is what gives an *AM2* application its particularity. It is what-makes one interface screen different from another and what distinguishes a particular interface button from the next.

The *ADL* provides several mechanisms for initializing object instances. Each of these is optional, and each is applied successively. The initial state of an object is the result of these cumulative and possibly overlapping initializations. The complete initialization sequence for an *ADL* object is as follows.

1. The *ADL* creates the object. That is, the *ADL* allocates storage for the object so it has an address.
2. The *ADL* creates and initializes all bases and members recursively, each set in declaration order. (Creation means allocation, as discussed in the first step. Initialization refers to all of the steps discussed in this list.)
3. A constructor message is sent to the object and handled if the corresponding method exists. If a special constructor is specified in the object definition or `new` statement, that is the constructor message sent, it must be handled or the system generates an error. If the object definition or a `new` statement does not specify a special constructor, then the default `Construct` message is sent without arguments. This message is optional, so the system does not generate an error if there is no corresponding `Construct` handler.
4. The optional initializor block, if it exists, executes as if it were a method of the object being initialized. All assignments in the initializor block are treated as *foreign*.
5. The *ADL* queries the asset manager about the object, and applies any assets that pertain to the object.
6. The *ADL* sends an optional `Init` message to the object. If there is an appropriate handler, it executes. Otherwise it is ignored.

Let us illustrate this sequence with a few examples:

```
class MyClass
{
    Member Declarations;
 upon Construct { ... }
  upon Create : handle h { ... }
  on Init { ... }
};

MyClass aInstance { Initializor Block};
MyClass {'Create, &aInstance} => bInstance;
```

Figure 3.47 Simple Initialization Example

In this case, `aInstance` and `bInstance` are instances of the class `MyClass`. We initialize aInstance in the following steps:

1. Create the class instance `aInstance`.
2. Create and initialize all members of `aInstance` using this procedure starting at the first step.
3. Call the Construct method on the object `aInstance` since there is no special constructor specified.
4. Execute the initializor block that follows the declaration of `aInstance` as a scope (**see Section 3.26, "Scope" page 52**). Member reference follows the pattern of a method inside `MyClass`. That is, the user may refer to members by simple member name rather than the combination `aInstance.`*memberName*. Member protection, however, follows the rules for an external reference. Assignment to class members from within an initializor block invokes the appropriate `Set_` method.
5. Consult the asset database to locate any resources that apply to the class MyClass and the instance `aInstance`, and apply them to the instance.
6. Call the `Init` method.

The initialization of `bInstance` is similar except that it calls the special constructor `Create` instead of the default `Construct`, and it does not execute an initializor block.

Constructor methods require comment since their definition uses a special syntax. In general, bases need not know anything about their derived instances. But there are circumstances where this is not the case. For instance, windowing systems generally refuse to create widgets without knowing the parent widget. In the *ADL*, widget containment is implemented as class membership. That is, a manager widget contains its child widgets as members. If those child widgets are subclasses of the base wrapped widgets, then the initialization of the subclassed child widgets must inform the wrapped bases of their parent during processing of the constructors.

As an example, consider a specialization of the base button class, `XFbutton`, called `ExitButton`. `ExitButton` has special behavior, background color and label. If we go to create a manager that contains an exit button, then the constructor for this `ExitButton` must somehow inform the base `XFbutton` of its manager parent. The *ADL* distinguishes constructor method definitions in order to add the mechanism to make this possible. Figure 3.48 illustrates this.

```
upon selector [: type1 arg1, ... , typen argn]
[init { CtorMessage1 => base1, ... , CtorMessagen => basen }]
{
    Method Declarations
    Method Statements
}
```

Figure 3.48 Constructor Method Syntax

Constructor definitions start with the keyword `upon` instead of `on`. The constructor body can be preceded by an optional `init` block that specifies constructor calls for direct bases. The constructor messages in this block must correspond to special constructors defined in the bases. The constructor messages in the `init` block are evaluated in the scope of the constructor execution so they have access to the constructor arguments.

The `SetAttributes` (**see Figure 3.37, "Sample SetAttributes Call" page 39**) method alleviates a scoping problem for constructors and initializor blocks. Initializor blocks provide the application developer far greater flexibility in initializing object members than constructors, but because they have object method scope, they cannot refer to variable values from the scope in which the object is being initialized. A special constructor that takes an attribute/value list as its sole argument can import an arbitrary set of values from the initializing scope and use them to initialize the object, thus circumventing the fixed argument list of the special constructor and the restricted scope of the initializor block.

3.26 Scope

A *scope* is a region of a program in which a variable or set of variables has definition. In *AM2*, each class (**see Section 3.20, "Class Definition" page 40**) and method definition (**see Section 3.21, "Method Definition" page 41**) as well as each initializor block forms a scope (**see Section 3.25, "Object Initialization" page 49**).

AM2 scopes are of two kinds, transparent and opaque. The variables defined in an enclosing scope are also visible in an enclosed transparent scope. In an enclosed opaque scope, they are not. Method definitions form transparent scopes and class definitions and initializor blocks opaque ones. Class definitions are visible in the current scope and in all enclosed transparent scopes as you would expect. Classes defined at the top level of an application are also visible everywhere.

You can access members and variables from the enclosed scope using the syntax for object member reference if an enclosed scope has been named (**see Section 3.19, "Object Member Reference" page 38**). Occasionally, you may need to refer to a global object, although avoid doing so wherever possible. Consider two application modules that are never simultaneously visible, but which each possess a button that puts its own module to sleep and calls up the other. These would normally be implemented as instances of module classes. The button actions for each must be able to send a wake up message to the other module, and must therefore be able to see the name of the other module in the enclosing application scope.

The *ADL*, therefore, allows you to specify that a symbol has one of two kinds of scoping:

- **Local scop**e is the default. The object or variable is destroyed upon exit from the scope in which it is defined. Separate declaration is not necessary.
- **Application scope** is specified by the keyword `global` and indicates that the symbol is visible in the scope in which the global declaration occurs, but it is actually defined in the top level application scope.

The keyword `global` must accompany and precede a type specifier (e.g., `string`) or class name in a declaration. If the declaration refers to a global instance of an anonymous class, there is no class name to use as the type specifier. In this case, you can use the key word anonymous as a type specifier. You can combine global scope with the `( . )` or `( -> )` operators. The global declaration merely designates where to look for the left-most member of a `(.)` or `(->)` chain.

Note that an enclosed scope cannot refer to an enclosing scope without a global declaration. Programming languages typically make the identifiers of an enclosing scope visible to an enclosed scope. Our aim here is to increase the modularity of the *ADL* by minimizing name clashes. This should encourage the reuse of interface and module templates and the development of template libraries. An enclosing scope can make one of its members visible inside an enclosed opaque scope by passing a handle to the member as an argument to a custom constructor.
(**see Section 3.17, "Dynamic Objects and Storage Management" page 36**).

```
class Aclass
{
   XFbutton aButton;
   . . .
};

class Bclass
{
   on Foo
   {
      global Aclass aInstance;
      aInstance.aButton.width = 40;
   }
};

Aclass aInstance;
Bclass bInstance;
```

Figure 3.49 Scope Example

3.27 Assets

Assets allow the customization of *AM2* applications on several levels. They also help to separate the implementation of an interface and its look and feel.

You can initialize any variable in an *ADL* program using assets, thereby allowing these items to be customized on a per platform, per installation, per user, and per application basis. Note that different platforms may support different degrees of customization. For example, Macintosh and Windows 3.1 systems do not have separate user accounts.

Suppose an author builds an application containing a button that causes the application to exit. In the U.S., you might use the label “Quit.” However, in Norway you would probably use the label “Avslutt.” You can create the two labels using assets without making modifications to the actual program code.

AM2 assets correspond roughly to X Window System resources, Microsoft Windows 3.1 and Windows NT .INI files, and Macintosh preferences. However, *AM2* uses its own asset mechanism rather than the native one for each platform in order to provide a portable, common interface. An *ADL* programmer or an application editor need only create one asset file, for use with the *ADL* code on all platforms.

3.27.1 Asset File Structure

Asset files are *ADL* code files containing application data initializationsthat the user can customize. There are three types of asset blocks: class, member and global. Class and member asset blocks are associated with an identifier and can contain statements, class asset blocks, or member asset blocks. When an asset block is applied to an object, any nested asset blocks are then applied when creating members of that object. Also, any statements it contains are evaluated in the context of the object being created, after evaluation of the izor block and before sending the `Init` message.

Class Asset Blocks

The assets in class asset blocks apply to all objects of the named class except those that are created dynamically (**see Section 3.17, “Dynamic Objects and Storage Management” page 36**). They typically appear at the top level, i.e., not embedded in any other asset block.   Top-level class asset blocks do not, however, affect , dynamically created objects (**see “Global Asset blocks” on page 54**).

Member Asset Blocks

The assets in a member asset block apply to the member with the same name in the class associated with the most closely enclosing class or member asset block.

Global Asset blocks

Global asset blocks are evaluated immediately after being parsed, before the application has been completely defined or instantiated. This is intended to be used to set paths for the library mechanism. Any statements are evaluated in the scope of the wrapped asset manager class.

```
// class asset block:
// all buttons under here will be red
class assets XFbutton
{
   background = ’red;
}

// member asset block:
// the member “theExitButton” of this class will be labelled “Quit”
member assets theExitButton
{
   label = ’Quit;
}

// global asset block:
// all objects of class ExitButton in the application (including
// dynamically created objects) will be labelled “Avslutt”
global assets
{
   class assets ExitButton
   {
      label = ’Avslutt;
   }
}
```

Figure 3.50 Asset Block Examples

Global asset blocks can contain class asset blocks but not member asset blocks or other global asset blocks. Such class asset blocks are then associated with both the heap and the application class, and are applied to all objects, including dynamically created objects.

Assets and the Library Mechanism

Libraries are an abstraction that allow collections of files, both for code and for data, to be grouped without worrying about portable pathnames. Library mappings, that is associations between library and directory names, can be made in platform, installation, and user dependant asset files. Files in these libraries can then be accessed via the statement `”file”@”libraryname”` both in `uses` statements and elsewhere in *ADL* code, such as in media element constructors. The `AppLib` library automatically maps to the directory containing the original *ADL* file given on the command line. The wrapped asset manager handles library mappings. To set a library path, use the `’SetLibrary` method, and to retrieve a mapping use the `’GetLibrary` method. Note that the path returned by `’GetLibrary` always ends in a directory separator that so you can concatenate it directly to a file or subdirectory name. For example, lines 3-5 of Figure 3.51 retrieve the path for the `AppLib` library, append the name of the `code` subdirectory, and then set the library named `MyCode` to this new path. The `uses` statement on line 7 then includes `mybutton.adl` from that library. So if the *ADL* program being run were `/mit/ceci/user/demo/buttons.adl`, it would be including `/mit/ceci/user/demo/code/mybutton.adl`. Similarly, the second example constructs an `MMimage` named `mBird` using the file `bird.gif` from the same directory as the main *ADL* program.

```
global assets
{
   { ’SetLibrary,
     ’MyCode,
     ( {’GetLibrary, ’AppLib} => self ) + ”code” } => self;
}
uses ”mybutton.adl”@”MyCode”;
```

```
MMimage {’MEimage, {’MAfile, ”bird.gif”@”AppLib”}} => mBird;
```

Figure 3.51 Library Mechanism Examples

3.27.2 Assets and Precedence

You can determine precedence in the asset mechanism using these simple rules to determine which assets will be applied to an object. Assets that have higher precedence are evaluated later, causing their values to override any assigned earlier.

- At any level, member asset blocks have higher precedence than class asset blocks.
- Assets specified closer to the current object have higher precedence. The distance is determined by the number of containers between declarations.
- Values set using assets are inherited unless the derived class possesses a member of the same name as the attribute being set by assets in the base class. For example, if the class `exitButton` inherits from `XFbutton,` any asset set for the class `XFbutton` will apply to the instances of `exitButton`.

The order of initialization is crucial in understanding the effect of asset specifications (**see Section 3.25, “Object Initialization” page 49**). For instance, in the example involving `exitButton` described above, if the `exitButton` constructor sets attribute `label` to `"Exit"`, but the user’s asset file sets all `XFbutton labels` to `"XFbutton"`, an `exitButton` will have label `"Exit"`. Why? Because the derived constructor is executed after the assets for the base class are consulted.

3.27.3 Example of Using Assets

Figure 3.52 and Figure 3.53 present `hello.adl`, rewritten to use assets. Notice that it is broken up into separate asset and code files. As a convention, the asset file has a `.am` extension. In this example the asset file is explicitly included via a `uses` statement, but the asset specifications could just as well have been placed in the platform specific asset files. On UNIX, for instance, the assets in the file `.am2rc` in the user’s home directory are read in during application startup.

```
uses ”hello.am”;

class exitButton : XFbutton
{
   upon Construct
   {
      Pressed = {’Exit, theApp};
   }
};

class Greetings : XFtop
{
   exitButton hello;
} myGreetings;
```

Figure 3.52 hello.adl Using Assets

```
class assets Greetings
{
   member assets hello
   {
      label = ”Hello, world!”;
      height = 40;
      width = 200;
   }
   height = 40;
   width = 200;
}

class assets exitButton
{
   label = ’Exit; // This gets overriden by the member assets above!
}
```

Figure 3.53 hello.am

3.28 Program Structure

An *ADL* program consists of a succession of the following elements in any order:

- class (including anonymous) definitions
- global variable definitions, including global object definitions with initializor blocks
- global method definitions
- `uses` statements

```
uses "SlideViewer.cl";   // defines class SlideViewer
class SlideShow
{
   SlideViewer myViewer;
   . . .
};
```

Figure 3.54 Uses Example

Global variables and global methods are implicit members of an anonymously declared instance or subclass of the system-defined application class, theAppClass.

You can use the `uses` statement to include the contents of a file or files that replace the statement at parse time. If multiple `uses` statements referring to the same file are parsed, the file is still included only once. A `uses` statement has two forms:

```
uses “fileName”@”libraryName”;

uses “fileName”;
```

Figure 3.55 Uses Statement Forms

In the first form, the statement asks for the file named “`fileName`” to be loaded from the library named “`libraryName`”. In the second form, it asks for the file named “`fileName`” to be loaded from the same library as the file in which the statement occurs. *AM2* designates the files in the application file’s directory to be `"AppLib"`.

Chapter 4 Using Activities in *ADL*

Newcomers to *AM2* often find the idea of activity management a difficult concept to master. Once you learn it, however, you will find it one of the most powerful features of the system. In this section we provide a detailed guide to activity management.

Activities in *AM2* provide the basic mechanism by which objects handle events generated by user actions in applications. Activities also handle events that occur in *AM2* applications due to a timer or arriving network messages. For example, a standard *AM2* button object provides a way to notify other objects when the application user presses it. In the terminology of activity management, the button *manages* the pressed activity. Similarly, events corresponding to an activity *trigger* that activity.

All objects that manage activities maintain a list of things to do when events trigger those activities. This is, in essence, a list of messages that *AM2* sends to objects when the triggering occurs. The *ADL* programmer can add and delete things from this list.

We discuss the use of activity management in five stages, each described in a separate subsection:

4.1 Using the Pressed Attribute

One of the most common uses of activities is the use of button objects. For example, one often places buttons in a multimedia application that, when pressed by the user, trigger some computation or presentation. This use of buttons is so common that *AM2* provides a shortcut to simplify the general activity mechanism in such cases.

The wrapped class `XFbutton` creates a simple button on the screen. (see **Section 6.2.9, “XFmessageDlg” page 136** for detailed documentation on the class.) The XFbutton class provides typical attributes such as width, height, (`x,y`) location on the screen, foreground and background colors, and a text label. For example, the following code is a very simple *ADL* program that puts a button at coordinates (50,50) inside an instance of an object that is a subclass of the `XFtop` shell class

```
1    anonymous:XFtop
2    {
3       XFbutton aButton {x=50; y=50; height=100; width=200;
4          label=”Push Me”;};
5    } myApplication {height=300; width=400;};
```

Figure 4.1 A Simple ADL Application with a Button

In this example, there is a single instance of a class that inherits from the `XFtop` wrapped class. The `anonymous` keyword in line 1 indicates that this class has no name; only this instance of the class is named. (In this case, this instance is named `myApplication`.) The anonymous class has a single member, a button object name `aButton` defined in lines 3 and 4.

This *ADL* program draws the button on the screen. Clearly, we need a way of assigning the trigger event when the user presses the button. The `XFbutton` class supports a special member called `Pressed.` You can assign this special member a list that instructs the program to take some action when the user clicks the mouse on that button. In its simplest form the `Pressed` attribute is a list that has two values: a `string` giving the name of the method invoked, and a `handle` to the object that receives a message when the user presses the button. For example, the assignment

```
aButton.Pressed = {‘Exit, theApp};
```

registers the fact that the message `Exit` goes to the built-in handle `theApp.` This is a predefined message that terminates the program when sent to the application.

The example in Figure 4.2 displays some text information. Suppose we want to have a help button in an application that displays some text information when pressed. The *ADL* program below shows this using an instance of the `XFtext` wrapped class which is initially not visible (done by setting the `visible` attribute to `FALSE`), and reversing that attribute when the user presses the button.

```
1    anonymous:XFtop
2    {
3      XFbutton helpButton {x=50; y=50; height=50; width=150;
4          recomputeSize=FALSE; label=”Help”;};
5      XFbutton exitButton {x=250; y=50; height=50; width=100;
6         recomputeSize=FALSE; label=”Exit”;};
7      XFtext helpText {x=50;y=100;height=100;width=200;wordWrap=TRUE;
8          visible=FALSE; editable=FALSE;
9          text=”This is an example of a help message.”;};
10
11      upon Construct
12      {
13         exitButton.Pressed = {‘Exit, theApp};
14         helpButton.Pressed = {‘BPress, self};
15      }
16
17      on BPress
18      {
19        if (helpText.visible) {
20           helpButton.label = “Help”;
21        }
22        else {
23           helpButton.label = “Remove Help”;
24        }
25        helpText.visible = ! helpText.visible;
26      }
27   } myApplication {height=300; width=400;};
```

Figure 4.2 An Example of a Help Button

In this example there are two buttons, one to exit the application and one to trigger the visibility of the help text. The special constructor method named `Construct` automatically receives a message when the application starts, and sets the `Pressed` attributes for these buttons. The assignment statement in line 14 sets the `Pressed` member so that the message `BPress` goes to the variable `self`. (The variable `self` is an automatically-generated handle that references the object itself.) Thus when the user presses the button, the `BPress` method that starts on line 17 executes. This method changes the text of the label on the help button and, on line 25, reverses the visibility of the text object.

You can also use the `Pressed` attribute to send a message to a method that requires one or more arguments. For example, suppose we wanted to create two buttons which, when pressed, move an instance of a label ten pixels to the right or left respectively. To do this, let’s create a method that changes the `x` attribute of the label by `n` pixels, where `n` is an argument to the method. For example, if the name of the label object is `myLabel`, then the following method would move it:

```
on MoveLabel: integer n
{
   myLabel.x = myLabel.x + n;
}
```

If we name the two buttons `leftButton` and `rightButton`, we set their `Pressed` members as follows:

```
leftButton.Pressed = {{‘MoveLabel, -10}, self};
rightButton.Pressed = {{‘MoveLabel, 10}, self};
```

4.2 Using Notification Request Objects

Using the `Pressed` attribute gives rise to some limitations. First, application developers frequently want a button to trigger a number of different actions, yet the `Pressed` attribute can send a message to a single method only. If you reset the `Pressed` member of a button, you lose the old setting. Second, the method that receives a message from the `Pressed` method cannot have arguments. This makes it impossible to provide the method receiving the message information such as the (`x,y`) coordinates of the mouse at the time the event occurs. In *AM2*, these more general uses of activities are supported through the use of Notification Request Objects, or NROs.

There are standard NRO classes provided in *AM2*.[1] Some are system-defined and some are user-defined (see **Section 3.23, “Wrapped Classes” page 44** for a discussion of classes.) With the exception of NROs used to handle timer events (described in **Section 4.3.2, “Timer NROs” page 67**) an NRO is an object that has the following four members:

- *Activity name:* a `string` naming the activity, to satisfy notification requests
- *Target object:* a `handle` to the object that receives notification when an event triggers the activity
- *Target method:* a `string` providing the name of the method to receive a message when an event triggers the activity
- *Client data:* an arbitrary piece of information that can by of any *AM2* data type.

Note that the second and third of the NRO members on the list above are identical to the members used in setting the `Pressed` attribute of button objects.

The base NRO class is the Nro wrapped class. It has a special constructor named `Create` that takes the four arguments listed above. The following example defines an instance of an NRO for an activity named `MouseDown`. This NRO requests that the message `ButtonDown` go to the target object `self` along with the string `“ClientData”` as an argument:

```
Nro {‘Create, ‘MouseDown, self, ‘ButtonDown, “Client data”} => downNro;
```

Each object that manages an activity must have at least two methods: `Subscribe` and `Unsubscribe`. These methods have as their sole argument a handle to an NRO. The `Subscribe` method registers the NRO for notification when an event triggers the activity named in that NRO. For example, suppose you created an NRO named `downNro` as shown above. The following statement would subscribe that NRO to the button named `myButton`:

[1] NRO wrapped classes are the only current exception to the conventionthat wrapped class names begin with two capital letters. The NRO classes could have been given the names NRgeneral, NRtimer, NRmouse -- but they weren’t. No one remembers why. Instead they bear the slightly more readable names Nro, TimerNro and MouseNro.

```
{'Subscribe, &downNro} => myButton;
```

After this statement executes, `downNro` is registered with the `MouseDown` activity of the button object. Whenever the user clicks the mouse on that button, the message `ButtonDown` goes to the object pointed to by `self`.

The `Subscribe` method actually returns a value that is often ignored by *ADL* programmers. This value is a handle that indicates whether or not the subscription successfully completed. The return value is `NULL` if the subscription is not performed and a handle to the subscribed NRO otherwise.

The Nro class assumes that the receiving method has three arguments in the following order:

- An argument of the type `any` that contains client data
- A `list` that contains strings with the names for the information sent by the activity, referred to as the *keys* for the activity
- A `list` that contains the values for the information sent by the activity

For example, the `MouseDown` activity (and all other standard activities that describe mouse events) provides six pieces of information when it sends its message to the target method. These six items in the lists of keys and values are as follows:

1. *x*: an `integer` giving the `x` location of the mouse when the user presses the mouse button
2. *y:* an `integer` giving the `y` location of the mouse when the user presses the mouse button
3. *button:* an `integer` with the number of the pressed button on the mouse (The interpretation of this number is platform-dependent.)
4. *shift:* a `boolean` that is `TRUE` if the shift key is down when the user presses the mouse button
5. *command:* a `boolean` that is `TRUE` if the key designated as the command key is down when the user presses the mouse button (The definition of the key that corresponds to the command key is platform-dependent.)
6. *modifier:* a `boolean` that is `TRUE` if a key designated as a modifier key is down when the user presses the mouse button (The interpretation of the key that corresponds to the modifier key is platform-dependent.)

It is important to note that the `(x,y)` coordinates returned by any mouse activity are given relative to the upper left corner of the widget where the mouse event occurred.

An example of an *ADL* program that includes a method with the appropriate arguments follows. In this example, the method named `ButtonDown` outputs the names and values of the arguments it receives.

In Figure 4.3, the NRO named `downNro` subscribes to the button `myButton`. When the user presses the mouse button, the method `ButtonDown` receives a message. This method outputs the client data (in this case, the string `"Client data")` from the NRO and then outputs the name and value pairs by going through the entries on the list one by one.

```
anonymous:XFtop
{
   XFbutton myButton {x=50; y=50; height=50; width=150;
      recomputeSize=FALSE; label=”Press Here”;};
   XFbutton exitButton {x=250; y=50; height=50; width=100;
     recomputeSize = FALSE; label=”Exit”;};
   Nro {‘Create, ‘MouseDown, self, ‘ButtonDown,
     “Client data”} => downNro;

   upon Construct
   {
      exitButton.Pressed = {‘Exit, theApp};
      {‘Subscribe, &downNro} => myButton;
   }

   on ButtonDown: any clientData, list keys, list values
   {
      any tempValue;
      integer count=1;
      echo(“Client data is” & toString(clientData)+”\n”);
      while (count <= length(keys)) {
         echo(“Name=”+toString(at(count, keys)) &
       “Value=” + toString(at(count, values))+”\n”);
         count = count+1;
      }
   }
} myApplication {height=300; width=400;};
```

Figure 4.3 An Example Using an NRO

A typical output from this application is as follows:

```
Client data is Client data
Name=x Value=27
Name=y Value=24
Name=button Value=1
Name=shift Value=TRUE
Name=command Value=TRUE
Name=modifier Value=FALSE
```

Note that in this example, the shift and control keys are held when the button is pushed down. Also, the (`x,y`) coordinates sent when an event triggers the activity are with respect to the upper left corner of the object managing the activity, which in this case is the button.

4.3 Using Other Types of System-defined NRO Classes

4.3.1 Mouse NROs

Objects can subscribe to any activity using the standard NRO class. There are some cases, however, where use of special classes makes programming simpler. These special NRO classes are tailored to a particular activity and usually provide simpler argument lists than the general NRO class, making it easier to write the target method.

For example, the use of NROs for mouse events is so common that *AM2* provides a special NRO for these events. This NRO returns the client data and the six relevant items about the event in separate variables. Thus, the MouseNro wrapped class sends a message that has seven arguments to the target method:

1. A value of type `any` containing the NRO's client data
2. An `integer` with the `x` coordinate where the mouse-related activity occurred
3. An `integer` with the `y` coordinate where the mouse-related activity occurred
4. An `integer` with the number of the button that was pressed
5. A `boolean` that is `TRUE` if the shift key was pressed when the event activity occurred
6. A `boolean` that is `TRUE` if the command key was pressed when the event activity occurred
7. A `boolean` that is `TRUE` if the modifier key was pressed when the event activity occurred

It is important to emphasize that the use of any of the special NRO forms is entirely optional. It is always possible to use the general NRO object described in **Section 4.2, "Using Notification Request Objects" page 62**.

The *ADL* program in Figure 4.4 illustrates the use of special NROs that track the press and release, and the movement of the mouse on a simple shell widget. To do this, we use three different activities: `MouseDown`, `MouseUp` and `MouseMove`. The example uses one instance of the `MouseNro` class for each activity. The code below shows the complete application.

```
anonymous:XFtop
{
   XFlabel reportLabel {x=10; y=10; height=50; width=250;
     recomputeSize=FALSE; label=””;};
   XFbutton exitButton {x=250; y=50; height=50; width=100;
     recomputeSize = FALSE; label=”Exit”;};
   MouseNro {‘Create, ‘MouseDown, self, ‘MouseTrack,
      “Down”} => downNro;
   MouseNro {‘Create, ‘MouseUp, self, ‘MouseTrack,
       “Up”} => upNro;
   MouseNro {‘Create, ‘MouseMove, self, ‘MouseTrack,
       “Move”} => moveNro;

upon Construct
   {
      exitButton.Pressed = {‘Exit, theApp};
      {‘Subscribe, &downNro} => self;
      {‘Subscribe, &upNro} => self;
      {‘Subscribe, &moveNro} => self;
   }

on MouseTrack:any clientData, integer xval, integer yval,
     integer button, boolean shift, boolean command,
     boolean modifier
   {

      reportLabel.label = “Mouse” & clientData + “:x=” +
          toString(xval) & “y=”+toString(yval);
   }
} myApplication {height=300; width=400;};
```

Figure 4.4 Using MouseNro Objects

4.3.2 Timer NROs

ADL provides a general purpose timer that you can use to trigger actions at pre-specified intervals. To use this timer, subscribe an instance of a class called `TimerNro` to the application using the special handle named `theApp`. In a perfect computational environment, the timer activity would always occur when scheduled. However the main event loop of *AM2* must handle many events, the duration of which may not be known beforehand. For this reason an actual timer event may be delayed or even missed. In order to help application developers cope with this possibility, `TimerNro` sends a message with two `integer` arguments to its clients:

1. A value called `late` that is the number of milliseconds between the exact time the activity is scheduled and the time it is actually triggered
2. A value called `missed` that is the number of successive triggering of timer events missed due to delays

For example, consider a simple stopwatch that counts seconds from the time you first press the button until you press it a second time. This clock needs to count intervals of 1000 milliseconds and trigger an activity after each interval. You can do it in the *ADL* by subscribing the NRO created by the following definition:

```
TimerNro {‘Create, 1000, self,‘ClockTick, NULL} => clockNro
```

The first argument of the Create message is the requested interval (in milliseconds) between triggering of the clock activity. The next three arguments are: the target of the activity’s message, the name of the method to receive the message, and a value that can be any *ADL* type used for conveying client information.

Figure 4.5 shows an *ADL* program that implements a stopwatch. The example creates three buttons: one for starting the watch, one for stopping the watch, and one for exiting the application. The variable named `clock` counts the number of seconds once the start button is pressed, and a label displays the elapsed seconds.

The method `ClockTick` handles the timer activity. The values provided as arguments by the activity manager give information about the time at which the activity is actually triggered. Line 28 in Figure 4.5 uses the value of the `missed` argument to correct the counter of seconds.

```
anonymous:XFtop
{
   XFbutton start {x=5; y=5; height=30; width=60;
      label='Start;};
   XFbutton stop {x=70; y=5; height=30; width=60;
      label='Stop;};
   XFbutton clear {x=135; y=5; height=30; width=60;
       label='Clear;};
   XFbutton exit {x=135; y=45; height=30; width=60;
      label='Exit;};
   XFlabel timeValue {x=5; y=45; height=30; width=125;
      borderColor='red;borderWidth=1;
      recomputeSize=FALSE;};

   TimerNro {‘Create, 1000, self, ‘ClockTick,
       NULL}=>clockNro; integer  clock;

   upon Construct
   {
      start.Pressed = {‘Start, self};
      stop.Pressed = {‘Stop, self};
      clear.Pressed = {‘Clear, self};
      exit.Pressed = {‘Exit, theApp};
      ‘Clear => self;
   }

   on ClockTick: any cd, integer late, integer missed
   {
      clock = clock + 1 + missed;
      timeValue.label = toString(clock);
   }

   on Start
   {
      {‘Subscribe, &clockNro} => theApp;
   }

   on Stop
   {
      {‘Unsubscribe, &clockNro} => theApp;
   }

   on Clear
   {
      clock = 0;
      timeValue.label = ‘0;
   }

} top { height = 80; width = 200; title="Timer Demo";};
```

Figure 4.5 Example Using A Timer

4.4 NROs Derived from System-defined NROs

AM2 also comes with a library of special-purpose NROs that you can use in *ADL* programs. Unlike the system-defined (or wrapped class) NROs described in **Section 4.2, "Using Notification Request Objects" page 62** and **Section 4.3, "Using Other Types of System-defined NRO Classes" page 65**, these library NROs are written in the *ADL*. They are classes derived from the Nro wrapped class. You can use them by including the file `nro.adl` in the standard *ADL* library.

One such NRO is the vanillaNro class. This NRO sends only the client data in its messages. For example, suppose we want to simplify our mouse tracking program shown above so that it only reports the type of the last mouse action, not the location. In this case, the (`x,y`) coordinates of the mouse activity are not needed. We could instead use three instances of the `vanillaNro` object defined as follows:

```
vanillaNro {‘Create, ‘MouseDown, self, ‘MouseTrack,
   “Down”} => downNro;
vanillaNro {‘Create, ‘MouseUp, self, ‘MouseTrack,
    “Up”} => upNro;
vanillaNro {‘Create, ‘MouseMove, self, ‘MouseTrack,
    “Move”} => moveNro;
```

We can then rewrite the `MouseTrack` method as show in Figure 4.6.

```
on MouseTrack: string clientData
{
   reportLabel.label = “Mouse” & clientData & “activity”;
}
```

Figure 4.6 New Version of MouseTrack Method Using vanillaNros

An even simpler NRO available in the standard library that sends no arguments when an event triggers the activity. This is the simpleNro class. Any method that handles an activity subscribed to using a simpleNro must have no arguments.

For example, we could rewrite the mouse tracking program in Figure 4.6 so that each of the three activities being subscribed to send a message to a different method. In this case, we could use the simpleNro class instead of the vanillaNro class, as follows:

```
simpleNro {‘Create, ‘MouseDown, self, ‘MousePush,
     ““} => downNro;
simpleNro {‘Create, ‘MouseUp, self, ‘MouseRelease,
    ““} => upNro;
simpleNro {‘Create, ‘MouseMove, self, ‘MouseChange,
    ““} => moveNro;
```

In this case, the method named `MouseTrack` shown in Figure 4.6 would be replaced by three simpler methods named `MousePush`, `MouseRelease` and `MouseChange,as shown in` Figure 4.7.

```
on MousePush
{
   reportLabel.label = “Mouse down activity”;
}
on MouseRelease
{
   reportLabel.label = “Mouse up activity”;
}
on MouseChange
{
   reportLabel.label = “Mouse move activity”;
}
```

Figure 4.7 Methods Using simpleNro Objects

4.5 Creating *ADL* Classes That Manage Activities

When application authors create their own classes, they often want these classes to manage activities. They can do it in two ways: design the new class so that it inherits from a class that already manages activities, or have the new class inherit from a wrapped class called `ActivityManager` that provides the general activity management mechanism to the new class. The first case applies when the base class you inherit from already manages the activity of interest or when you want to add a new activity to the base class. The second case gives you the flexibility to define an entirely new class that manages its own activities. We explore each of these cases below.

4.5.1 Inheriting An Existing Activity From a Class That Manages Activities

The first situation is the creation of a class that inherits from an existing class, such as `XFtop`, `XFbutton`, `XFlabel` or any of the other wrapped classes that have built-in activity management. In this case, the author does not need to do anything special. The inheritance mechanism in *AM2* automatically gives the new class all the activity management capabilities of the class it inherits from.

Consider creating a subclass of the `XFtop` shell widget, which has the property that it always displays the (`x,y`) coordinates of the mouse in its upper left corner. This type of widget might be useful as a building block in a still image editor that allows the user to crop and scale photographs. The *ADL* class named `LocationTop` shown in Figure 4.8 accomplishes this. This new *ADL* class inherits all the activity management capabilities of the `XFtop` wrapped class. Thus, it accepts `Subscribe` and `Unsubscribe` messages and sends messages to any subscribed target object when an event triggers an activity. All of these properties are the result of *AM2*’s inheritance capabilities.

The NROs in this class subscribe to both `self` (the handle to the object itself) and the instance of `XFlabel`. This is necessary because the `XFlabel` object can be thought of as though it were “on top” of the instance of `LocationTop,` to which it belongs. Any mouse-related events on the label trigger the activities of the label. They are not passed through to the underlying `XFtop`. If lines 17 through 19 were removed from the class declaration, the label would act as a “hole” in the widget; events may trigger the activities of the label, but there are no registered NROs and the activities of the label would therefore have no effect.

```
class LocationTop:XFtop
{
   XFlabel reportLabel {x=0; y=0; height=50; width=250;
     recomputeSize=FALSE; alignment=’left; label=””;};
   MouseNro {‘Create, ‘MouseDown, self, ‘MouseLocation,
      ““} => downNro;
   MouseNro {‘Create, ‘MouseUp, self, ‘MouseLocation,
       ““} => upNro;
   MouseNro {‘Create, ‘MouseMove, self, ‘MouseLocation,
       ““} => moveNro;

   upon Construct
   {
      {‘Subscribe, &downNro} => self;
      {‘Subscribe, &upNro} => self;
      {‘Subscribe, &moveNro} => self;
      {‘Subscribe, &downNro} => reportLabel;
      {‘Subscribe, &upNro} => reportLabel;
      {‘Subscribe, &moveNro} => reportLabel;
   }

on MouseLocation:any clientData,integer xval,integer yval
   {
          reportLabel.label = “x=”+toString(xval) &
            “y=”+toString(yval);
   }
};
```

Figure 4.8 A Class Inheriting from XFtop Class

4.5.2 Creating a New Activity

Note: Many AthenaMuse 2 users will not need to create classes that manage activities. This section can be skipped without any loss of continuity in the presentation.

In some situations, an application developer using *ADL* needs to add a new activity to a class that inherits activity management from a base class. Consider a class that displays a color palette. We would like such a class to have an activity called `ColorSelected` that provides the name of the chosen color in the message sent when an event triggers the activity. Since our color palette class inherits from the *AM2* wrapped class `XFlayout`, which already manages activities, our new class automatically has the capability to manage any activity. We need only add the new activity and define what should be done when an event triggers the activity.

Figure 4.9 shows an implementation of the `ColorPalette` class. This *ADL* code is a simplified version of a more general color palette provided in the *ADL* standard library. In the simplified version the color palette has exactly eight colors, the buttons for these colors are fixed in size, and the layout of the eight colored buttons in the palette is horizontal. All of these restrictions are relaxed in the more complete version.

The `ColorPalette` class inherits from the `XFlayout` class, thereby inheriting its members, methods and activity management capabilities of that class. The member `colorList` in the `ColorPalette` class (defined on lines 5 and 6) provides the names of the colors in the palette. The other class member, `buttonArray`, is an array of lists. After an instance of a `ColorPalette` is constructed, each element of this array is a list of two handles: a handle to a button and a handle to an NRO for that button. The index for this array is an `integer` between one and eight, and corresponds to the positions of the elements in the `colorList`.

The most significant component of this example is line 9, which causes the initialization of the member `ActivityInfo`. This list is inherited from the base class `XFlayout`. The sublist in `ActivityInfo` provides the information about managing additional activities; each entry in the list corresponds to a different new activity. The general form for the sublists is:

```
{<name of activity>, {<keyname1>,<keyname2>, ...}}
```

Thus, in Figure 4.9 the name of the new activity is `ColorSelected`, and the single key provided to any target of this activity has a single entry called `Color`.

The `Construct` method for the `ColorPalette` class simply sets the default height and width of the palette. The `Init` method for the `ColorPalette` class starting at line 19 does most of the work of setting up the buttons and handles in the class. It uses the `new` operator to create instances of buttons and NROs from the heap, and then stores the handles to these buttons and NROs as elements of the `buttonArray`. It also subscribes an NRO to each button, using the background color in each of the buttons as the client data in the `vanillaNro` instance. Thus, when a user presses any of the eight buttons, the `ColorChosen` method of the `ColorPalette` receives a message with the color of the button as the client data.

```
uses “nro.adl”@”StdLib”;
class ColorPalette: XFlayout
{
/* This list provides the colors in the palette. */
  list colorList = {‘white, ‘black,’red, ‘blue, ‘green,
                    ‘yellow, ‘tan, ‘gray};
  list buttonArray<integer>;
  list ActivityInfo = {{“ColorSelected”, {“Color”}}};

  upon Construct
  {
    height=30; width=240; borderWidth=1;
  }
 /* This method creates the buttons and the color patch */
on Init
  {
    integer count=0; handle hButton, hNro;
    while (count < 8) {
      hButton = new {‘Create, self} =>XFbutton {height=30;
         width=30; label=””; recomputeSize=FALSE;};
      hButton->x = count*30;
      count = count+1;
      hButton->background = at(count, colorList);
      hNro = new {‘Create, ‘Pressed, self, ‘ColorChosen,
         at (count, colorList)} => vanillaNro;
      {‘Subscribe, hNro} => hButton;
      buttonArray[count] = {hButton, hNro};
    }
  }

/* This method deletes all allocated instances */
  on Destroy
  {
    integer count=1;
    while (count <= 8) {
        delete at( 1, buttonArray[count]);
        delete at( 2, buttonArray[count]);
        count = count+1;
    }
  }
/*This method handles callback when  button is pressed */
 on ColorChosen: string cdata
 {
  {‘TriggerNotification, ‘ColorSelected, {cdata} }=>self;
 }
}; /* end of class ColorPalette */
```

Figure 4.9 Example of Class With Added Activity

The `Destroy` method starting on line 36 recovers the memory allocated for buttons and NROs when an instance of a `ColorPalette` is created. This method automatically receives a message whenever a `ColorPalette` is no longer valid or when the `delete` operator is applied to a handle to a `ColorPalette` object. The implementation of this method loops through the array of buttons and deletes the buttons and their corresponding NROs.

The method named `ColorChosen` is the target for the message sent when a user presses any of the eight buttons. This method simply calls the method `TriggerNotification`, which is inherited in any class derived from a class that manages activities. It is this method that then sends messages to every NRO subscribed to the activity. For example in line 49, the message takes the form:

```
{‘TriggerNotification, ‘ColorSelected, {cdata} } => self;
```

Note that the `TriggerNotification` method always takes two arguments: a string giving the name of the activity that was triggered and a list of values transmitted to the target. In this case, the activity name is `ColorSelected`, and the list of values has the name of the selected color.

Figure 4.10 illustrates how the `ColorPalette` class might be used. The most notable aspect of this program is that once implemented, the activitymanagement capabilities of the `ColorPalette` class are treated exactly the same as the corresponding capabilities of standard wrapped classes such as `XFbutton` and `XFtext`.

```
anonymous: XFtop
{
  ColorPalette myPalette{height=45;};
  Nro {‘Create, ‘ColorSelected, self, ‘EchoColor, ““} =>
               selectNro;
  XFlabel colorLabel {x=10; y=50; height=50; width=200;
   recomputeSize=FALSE; label=””;};
  XFbutton exitButton {y=110; x=10; height=50; width=100;
   label=”Exit”;};

  upon Construct
  {
    exitButton.Pressed = {‘Exit, theApp};
    {‘Subscribe, &selectNro} => myPalette;
  }

  on EchoColor: any cd, list names, list values
  {
    colorLabel.label = “Chosen color is” & at(1, values);
  }
} myApplicaton {height=200; width=350;};
```

Figure 4.10 Example Using the ColorPalette Class

4.5.3 Creating Classes That Inherit From the ActivityManager Class

Note: Many AthenaMuse 2 users will not need to create classes that manage activities. This section can be skipped without any loss of continuity in the presentation.

Another situation of interest to the *ADL* programmer is the need to create a class that manages activities but does not inherit from a standard wrapped class that manages activities. In this situation, the programmer inherits from an *ADL* class called `ActivityManager` which supplies the needed functionality.

The `ActivityManager` class is an example of a class designed specifically for inheritance purposes. It has the following functionality:

- It provides a way to specify a list of activity names to be managed. This list is called `ActivityInfo` and is an attribute of the ActivityManager class.[2]
- It handles `Subscribe` and `Unsubscribe` messages for the activities it manages.
- It provides a method called `TriggerNotification` that can receive a message whenever an event triggers an activity. This method sends all subscribed NROs the appropriate messages.

Consider a situation where we want to create a general class of movable objects. Any object inheriting from this class has the property of tracking the motion of the mouse when clicked. In addition, any object that inherits from the `Movable` class has an activity called `Update,` triggered when the user releases the mouse while moving the object. The values provided when the event triggers the `Update` activity are the new `(x,y)` coordinates of the object. This type of generic class can be combined with specific user interface classes to create specific types of movable classes, such as movable buttons, labels, or text.

Classes such as `Movable` are intended for use as a base class in combination with other classes. For example, a class might inherit from both the XFlabel wrapped class and the `Movable` class, thereby inheriting the activities, members and methods of both base classes. This type of class is often referred to as a mix-in class.

Figure 4.11 displays the *ADL* code that implements the `Movable` class. The member `ActivityInfo` declares the name of the new activity (`Update`) and the list of keys for that activity. The example uses three instances of the MouseNro class to subscribe to the `MouseUp`, `MouseDown` and `MouseMove` activities.

The Construct method subscribes the three NROs to the object itself. Note that the `Subscribe` messages in this method on lines 12 through 14 are sent to the handle `derived::self`. The `derived` keyword in *ADL* indicates that the message goes to the object that is at the end of the chain of inheritance (i.e. the member of the "most derived" class).

[2] In future releases of *AM2*, the ActivityInfo attribute may be implemented as a `common` attribute, i.e. an attribute that is shared by all members of the class. The value of this attribute should therefore not be changed.

This is necessary because the `MouseDown`, `MouseUp` and `MouseMove` activities are not activities managed by the `Movable` class. Rather, they are managed by whatever class we mix-in with the `Movable` class. Sending the Subscribe message to `derived::self` ensures that the NROs are subscribed to the correct activity manager.

Lines 17 through 22 implement the method named `Down` that receives a message when the `MouseDown` activity is triggered. This method stores the location of the mouse when the button pressed.

The method `Move` shown in lines 24 through 29 change the (`x,y`) coordinates of the object when the mouse has been moved. Note again that we must use the `derived` keyword before the attributes `x` and `y` because they do not belong to the `Movable` class; they must exist in the class derived from the `Movable` class.

```
1    /* generic class for movable objects */
2    class Movable: ActivityManager
3    {
4      list ActivityInfo = { {‘Update, {‘x, ‘y}} };
5      MouseNro {‘Create, ‘MouseDown, self, ‘Down, NULL} => downNro;
6      MouseNro {‘Create, ‘MouseUp, self, ‘Up, NULL} => upNro;
7      MouseNro {‘Create, ‘MouseDrag, self, ‘Move, NULL} => moveNro;
8      integer oldX, oldY;    /* used to store (x,y) of mouse press */
9
10     upon Construct
11     {
12       {‘Subscribe, &downNro} => derived::self;
13       {‘Subscribe, &moveNro} => derived::self;
14       {‘Subscribe, &upNro} => derived::self;
15     }
16   /* This method is messaged when the MouseDown activity is triggered */
17     on Down: any cd, integer xval, integer yval, integer button,
18        boolean shift, boolean command, boolean modifier
19     {
20       oldX = xval;   /* store (x,y) coordinates of mouse press */
21       oldY = yval;
22     }
23   /* This method is messaged when the MouseMove activity is triggered */
24     on Move: any cd, integer xval, integer yval, integer button,
25        boolean shift, boolean command, boolean modifier
26     {
27         derived::x=derived::x + xval-oldX; /* update x coordinate */
28         derived::y=derived::y + yval-oldY; /* update y coordinate */
29     }
30   /* This method is messaged when the MouseUp activity is triggered */
31     on Up: any cd, integer xval, integer yval, integer button,
32        boolean shift, boolean command, boolean modifier
33     {
34     {‘TriggerNotification, ‘Update,{derived::x,derived::y}}=>self;
35     }
36   }; /* end of class Movable */
```

Figure 4.11 A Class Inheriting from the ActivityManager Class

The last method, `Up`, triggers the `Update` activity notification by sending the `TriggerNotification` message. It provides the new `(x, y)` coordinates of the object as part of them message, and then disarms the object. Figure 4.12 shows a simple case using the class. Lines 2 through 4 declare a new class named `MovableLabel` that inherits from both the XFlabel

```
1    /* Create a class of Movable buttons */
2    class MovableLabel : XFlabel, Movable
3    {
4    }; /* end of class MovableLabel */
5
6    /* Create top window */
7    anonymous: XFtop
8    {
9      MovableLabel newLabel {x=20; y=20; height=40;
10      width=100; label='MoveMe; borderWidth=1;};
11     XFlabel location {x=0; y=200; height=40; width=100; };
12     Nro {'Create, 'Update, self, 'ChangeLocation, NULL} => updateNro;
13
14     upon Construct
15     {
16       {'Subscribe, &updateNro} => newLabel;
17       location.label = "x=" + toString(location.x) &
18           "y=" + toString(location.y);
19     }
20
21     on ChangeLocation: any cd, list keys, list vals
22     {
23       location.label = "x=" + toString(at(1, vals)) &
24            "y=" + toString(at(2,vals));
25     }
26   } myTop {width=300; height=250; title="Movable Label";};
```

Figure 4.12 An Example Using the Movable Class

wrapped class and the user-defined `Movable` class. The new class needs no additional methods or members since it inherits all its useful functions from it base classes.

Lines 7 through 26 declare an anonymous class derived from the XFtop wrapped class. This class has an instance of a `MovableLabel` and an instance of an XFlabel. The latter of these two is used to display the (`x,y`) coordinates of the former. The NRO named `updateNro` subscribes to the `Update` activity. This NRO causes the method `ChangeLocation` to receive a message when the `Update` activity is triggered. This method, defined in lines 21 through 25, changes the value of the label named `location` to indicate the new coordinates of `MovableLabel`.

4.6 Creating Customized NROs

Note: Many AthenaMuse 2 users will not need to create NROs. This section can be skipped without any loss of continuity in the presentation.

Some advanced users of *AM2* may find it useful to create their own, customized NRO classes. For example, in the user-defined `ColorPalette` class shown in Figure 4.9, we use the standard NRO that provides the usual three arguments (client data, the list of the names of the values returned and the corresponding list of values). Since all the target message method really needs is the name of the color selected, most of this information goes unused. You might develop a customized NRO that only provides a string containing the name of the selected color as its argument.

To create specialized NROs, it is necessary to understand how the standard Nro wrapped class works. This class has a special constructor named `Create` that stores its four arguments in members of the Nro class. These four members are declared as follows:

```
string mActivity; //  the activity name[3]
string mMethod;   //  the name of the method to receive a
                 //  message when the activity is triggered
handle mClient;   //  handle to the client object to receive
                 //  a message when the activity is triggered
any mClientData;  //  any client data
```

A new class that inherits from the Nro class automatically has these four members.

A second key aspect of the Nro class is the method `HandleActivity`. This is the method that receives a message whenever an event triggers an activity. It always receives the list of value names and the corresponding list of values sent by the activity as its arguments. In the standard NRO, this method is as follows:

```
    on HandleActivity: list keys, list values
    {
       {mMethod, mClientData, keys, values} => mClient;
    }
```

The key to writing a new type of NRO is to define a new class that inherits from the standard NRO but provides a new version of the `HandleActivity` method, which sends the arguments of interest to the target method. Figure 4.13 shows a new NRO class called `ColorNro`.

Note that the `ColorNro` class has a special constructor named `Create` that appears to do nothing even though its base class, Nro, has the same special constructor. *This is needed because special constructors are never inherited.* The useful work done by this special constructor is accomplished by initializing the base class using *ADL*'s base class initialization feature (the code block following the `init` keyword). In this case, the special constructor sends a base class initialization message to the Nro class. The `HandleActivity` method in the `ColorNro` class overrides the method with the same name in the Nro base class. In line 9, the client of the activity (`mClient`) receives a message invoking the target method (`mMethod`) with the only argument being the first element on the list of values. This element contains the name of the color.

[3] The member `mActivity` is a "read only" attribute. It should never be changed after it is initialized.

```
class ColorNro: Nro
{
   upon Create: string act, handle cli, string mtd, string cd
      init {{‘Create, act, cli, mtd, cd} => Nro}
   { }

  on HandleActivity: list keys, list values
  {
    {mMethod, at(1,values)} => mClient;
  }
};
```

Figure 4.13 Example of a Custom NRO

Figure 4.14 shows how the example program in Figure 4.10 can be modified to make use of the customized NRO.

```
anonymous: XFtop
{
  ColorPalette myPalette{height=45;};
  ColorNro {‘Create, ‘ColorSelected, self, ‘EchoColor, ““} =>
             selectNro;
  XFlabel colorLabel {x=10; y=50; height=50; width=200;
   recomputeSize=FALSE; label=””;};
  XFbutton exitButton {y=110; x=10; height=50; width=100;
   label=”Exit”;};

  upon Construct
  {
    exitButton.Pressed = {‘Exit, theApp};
    {‘Subscribe, &selectNro} => myPalette;
  }

  on EchoColor: string colorName
  {
    colorLabel.label = “Chosen color is” & colorName;
  }
} myApplicaton {height=200; width=350;};
```

Figure 4.14 Revised Example Using ColorPalette and Custom NRO

4.7 Using Activities for Notification of Subscriptions

Note: Many AthenaMuse 2 users will not need this feature. This section can be skipped without any loss of continuity in the presentation.

There are some situations when it is important to know if an object has any NROs subscribed to one or more of its activities. For example, suppose you wanted to develop a new class of buttons that have the property that the button's label is presented in different colors depending on whether or not any NROs are subscribed to its `Pressed` activity. In this situation, AM2 can provide notification whenever the number of NROs subscribed to an activity changes from zero to one or more, or vice versa. This capability is a standard feature of all activities.

In *AM2*, all activities have an associated second activity with the same name as the original followed by a question mark (?). You can use it for subscription notification. Thus, if an object has a `Pressed` activity then it automatically manages an activity named `Pressed?` that is triggered whenever the number of NROs subscribed to the Pressed activity changes from zero to one or from one to zero.

Chapter 5 Example *ADL* Programs

The following sections are designed to provide annotated examples of working *ADL* programs:

These programs should help those who like to learn by association and example.

5.1 Toggle Button Class

In this example, we develop an *ADL* class that provides a user interface button that changes its label when pressed. Each label corresponds to a different value for some variable, thereby allowing any *ADL* program using the button to set other variables or take some action that depends on the new value. Since the toggle button has most of the characteristics of a regular `XFbutton` object, it is natural for the `ToggleButton` class to inherit from `XFbutton`.

A typical situation where such a button is useful is a case where we need a button to set a value with two mutually exclusive options. For example, we want to have a button that when pressed sets a value that is used to determine if video will be played with or without subtitles. The button has two possible labels: “Show Subtitles” and “Hide Subtitles”. Each time the user presses the button the label changes from one value to the other, and the value stored by the button shifts from `TRUE` to `FALSE`.

You can configure the button so that you can set the number and contents of possible labels when the button is created. To do this, the `ToggleButton` class has two members that are lists. The first, called `labelList`, is a list of character strings corresponding to the set of labels the button may display. The second list, called `valueList`, holds the corresponding values.

The integer member `currentPosition` stores the number of the currently displayed list element. Each time the user presses the button, the value of `currentPosition` changes. In addition, a member called `buttonValue` holds the currently active value for the button. Because the values can be of any type (`list`, `integer`, `handle`, etc.), the member of `buttonValue` is of the *ADL* type `any`.

We also provide the toggle button with default labels “Yes” and “No”, with corresponding values of `TRUE` and `FALSE`. The defaults are set up so that the initial label is the string “Yes” and the initial value of the button is `TRUE`. You can override these defaults when using the `ToggleButton` class.

5.1.1 *ADL* Implementation of the ToggleButton Class

Here is the *ADL* code for the `ToggleButton` class, with detailed comments.

Line 1 declares the class named `ToggleButton` and indicates that this class derives from the `XFbutton` class.

Line 3 declares the member `labelList` to be of type `list` and initializes its value to a list of two `strings`, `“Yes”` and `“No”`. These strings are the default labels for the button unless the programmer using the class changes them.

Line 4 declares the `integer` member `currentPosition` and initializes its value to 1. This indicates that by default the first string in the list `labelList` is the initial label for an instance of a `ToggleButton`.

Line 5 declares an NRO (notification request object) named `toggleNro`. As with all NROs, the `Nro` class has a special constructor called `Create` that has four arguments:

`a string` giving the name of the activity (in this case `Pressed`)

`a handle` giving the target of the notification required (in this case `handle self` indicates that the message is to be sent to `ToggleButton`)

`a string` giving the name of the method to be messaged when the activity occurs (in this case a method named `Toggle`)

arbitrary client data (in this case the `handle` value `NULL`)

Line 6 declares the member `valueList` to be of type `list` and initializes its value to a list of two `booleans`, `TRUE` and `FALSE`. These are the default values associated with the labels `“Yes”` and `“No”` for the button, unless the programmer using the class changes them.

Line 7 declares the member `buttonValue` to be of type `any.` This member stores the current value associated with the `ToggleButton`.The values stored in buttonValue will be entries from the `list valueList`. The use of the *ADL* data type any allows users of the `ToggleButton` class to replace the default `valueList` with a `list` containing any valid data type.

```
1    class ToggleButton: XFbutton
2    {
3      list labelList = {‘Yes, ‘No};
4      integer currentPosition = 1;
5      Nro {‘Create, ‘Pressed, self, ‘Toggle, NULL} => toggleNro ;
6      list valueList = {TRUE,FALSE};
7      any buttonValue;
8
9    upon Create: handle hparent
10       init {{‘Create, hparent} => XFbutton}
11     { }
12
13   on Init
14     {
15       label = at(currentPosition, labelList);
16       buttonValue = at(currentPosition, valueList);
17       {‘Subscribe, &toggleNro} => self;
18   }
19
20   on ChangeValue: any newValue
21     {
22       any temp;
23       integer counter=1;
24       for temp in valueList {
25         if(getType(newValue) == getType(temp) && newValue == temp) {
26           buttonValue = temp;
27           label = at(counter, labelList);
28           currentPosition = counter;
29           return;
30         }
31         counter = counter+1;
32       }
33     }
34
35   on Toggle: any cd, list keys, list vals
36     {
37       currentPosition = currentPosition+1;
38       if(currentPosition > length(labelList)) {
39         currentPosition = 1;
40       }
41       label = at(currentPosition, labelList);
42       buttonValue = at(currentPosition, valueList);
43     }
44    }; /* end of class ToggleButton */
```

Figure 5.1 ToggleButton Class

Line 9 begins the special constructor method `named Create.` This method is provided so that a user of the `ToggleButton` class can use the `new` operator to create instances of the class `ToggleButton` on the heap. In *ADL*, any object that is drawn on the screen and is created dynamically at run time needs a special constructor so that it can pass a `handle` to the parent of the new widget. This is done through the method argument `hparent`. Note that we do not need a special constructor for the case where we declare a static instance of a `ToggleButton` because the standard, default constructor `Construct` provided by *AM2* is adequate for the task.

Line 10 is a base class initialization statement, indicated by the keyword `init`. This statement ensures that when we create an instance of a `ToggleButton` using the `new` operator, the instance of the class from which `ToggleButton` derives is correctly initialized. In this case, the *ADL* statement

```
{'Create, hparent} => XFbutton
```

ensures that the `Create` special constructor for the base `XFbutton` class is called rather than the default `Construct` method. Any user interface widget or subclass of such a widget created from the heap using the `new` operator must have its parent widget explicitly set. The default `Construct` method does this automatically for statically created widgets.

Line 11 is an empty body for the `Create` method. There are no additional initializations that we need to perform here.

Line 13 is the start of an `Init` method. This method automatically receives a message whenever an object of this class is constructed. This message is sent after all the other steps in the creation of an object, i.e. after the `Construct` message is sent (or when some other special constructor is used), after the initialization block, (also called an izor) is executed, and after any asset declarations are applied. The body of the `Init` method is used to set initial values and to subscribe to the `Pressed` activity for the button.

Line 15 sets the initial value of the `label` for the button to a value on the `labelList`. The initial label is set to the `currentPosition`-th element of the `labelList`. This allows the *ADL* programmer using the `ToggleButton` class to set the `currentPosition` member in an izor block, thereby changing the initial value of the `label`.

Line 16 sets the initial value of the member `buttonValue`. It uses the built-in function `at()` to extract a value from the list `valueList`.

Line 17 sends the `Subscribe` message to the button, registering the NRO named `toggleNro`. This results in the message `Toggle` being sent to any member of the `ToggleButton` class whenever the user presses the button.

Line 18 closes the `Init` method.

Lines 20-21 start a method called `ChangeValue`. We use this method to reset the value of the `ToggleButton` to some new value. It provides the *ADL* programmer with a way to alter the label and corresponding value of the `ToggleButton` after it is created. This feature gives the programmer a way of changing the label without requiring the application's user to actually press the button. The `ChangeValue` method uses a single argument of type `any` because the `ToggleButton` class allows the elements of the `valueList` to be of any valid *ADL* type, including compound types such as a `list`.

Line 22 declares `temp` to be a local variable of type `any`.

Line 23 declares an `integer` variable named `counter` and sets its initial value to 1.

Line 24 begins a loop over all the values in the `valueList`.

Line 25 tests if the value of the argument `newValue` equals one of the values in `valueList`. Note that this comparison is done by first testing if the value in temp is the same as the value being checked on the list, and then by checking if the two values are equal. This is necessary because the equality operator `(==)` cannot be used on values of different types. Even though both of the variables being compared are declared of type `any`, the values they hold have definite *ADL* types which may not be the same.

Lines 26-29 execute when the value requested is found in `valueList`. This group of statements sets the variable `buttonValue`, changes the `label` of the button to the corresponding value in `labelList`, sets the `currentPosition` member appropriately, and returns.

Line 29 updates the value of `counter` each time the loop over the elements in `valueList` is executed.

Lines 32-33 close the loop and the method definition respectively.

Lines 35 is the start of the declaration of the method `Toggle`. This method is called every time the `ToggleButton` is pressed.

Lines 37-40 begin by incrementing the value of the `currentPosition` by 1. If the revised value of `currentPosition` is greater than the number of elements in the `labelList` (computed by applying the built-in *ADL* function `length()` to `labelList`), then the `currentPosition` is reset to 1. This code has the effect of moving through the `labelList` and wrapping around to the start of the list when the last value on the list is reached.

Line 41 updates the value of the button label.

Line 42 updates the value of the variable `buttonValue` to correspond to the label now being displayed.

Line 43 closes the `Toggle` method.

Line 44 ends the declaration of the `ToggleButton` class.

5.1.2 An Example of Using the ToggleButton Class

Given the example shown above, we now turn to a simple *ADL* program that actually uses the `ToggleButton` class. Suppose that we want to have a button on an application that sets the background color for the main application shell widget. We might have a button that toggles among several different colors. Each time the user presses the button, the background color of the `XFtop` in which the application is running changes. In our example, we allow the user to set the possible background colors to white, black, red, blue and green.

The *ADL* program shown here implements this idea.

```
1    anonymous : XFtop {
2      ToggleButton myButton {x=50; y=50; height=50; width=150;
3            recomputeSize=FALSE;
4            labelList={‘White,‘Black,‘Red,‘Blue,‘Green};
5            valueList={‘white, ‘black,‘red, ‘blue, ‘green};};
6
7      upon Construct
8        {
9          background = myButton.buttonValue;
10         myButton.Pressed = {‘ShiftBackgroundColor, self};
11       }
12
13     on ShiftBackgroundColor
14     {
15         background = myButton.buttonValue;
16     }
17
18   } myTop {height=400; width=400;};
```

Figure 5.2 ToggleButton Example

Line 1	starts the definition of an anonymous instance (used when one or a very few instances of the class are needed) of an object that inherits from the *ADL* top level shell class, `XFtop`.
Lines 2-5	declare a member of this anonymous class that is an instance of a `ToggleButton` named `myButton`. The button has a list of labels and values set in an izor block that correspond to the colors we want to use as options.
Lines 7 - 8	start the `Construct` method for the anonymous class. Since this is a constructor, it begins with the `upon` keyword. This method is automatically invoked when the object is created.
Line 9	sets the initial background color of the widget to the value stored in the `buttonValue` member of the `ToggleButton`.

Line 10	sets the value of the `Pressed` attribute of `myButton` (corresponding to the `Pressed` activity of the button) to a `list` that gives the name of the method to be invoked when `myButton` is pressed, and the target of the method. In this case, the method `ShiftBackgroundColor` of the anonymous instance that is the shell widget (as indicated by the special handle `self`, automatically defined for every *ADL* instance) is messaged when the button is pressed. Note that this could also be done by declaring an appropriate constructed notification request object and then subscribing that object to `myButton`. We discuss this further in the next subsection below.
Lines 13-16	define the method `ShiftBackgroundColor`. This method resets the background color of the shell to the current value stored in `myButton`.
Line 18	closes the definition of the anonymous subclass of the standard shell widget. It gives the widget the name `myTop` and uses an izor block to set the widget's `width` and `height`.

5.1.3 Implementation Options

One of the interesting aspects of button widgets that the toggle button example illustrates is the use of notification request objects (NROs) and the `Pressed` attribute of a button. In the *ADL* code for the class `ToggleButton`, we intentionally use an NRO to register for the `Pressed` activity, while in the *ADL* program that uses the `ToggleButton` class we set the `Pressed` attribute. It is reasonable to ask why we did not use the `Pressed` attribute in defining the `ToggleButton` class.

The answer to this question requires a clear understanding of the `Pressed` attribute. Basically, setting the `Pressed` attribute is a shortcut that has the effect of subscribing an implicit NRO to a button's `Pressed` activity. However, this way of setting an activity handler has two limitations: only one method can be registered in this way, and the method must not require any arguments be passed when the activity is triggered. If the `Pressed` attribute of a button is set more than once, only the last setting takes effect. All earlier settings are lost.

Thus, if we set the `Pressed` attribute in the class definition for `ToggleButton` rather than subscribing to the activity using an explicit NRO, we force users of the `ToggleButton` class to not use the `Pressed` attribute, but rather subscribe an NRO instead. If the users of the `ToggleButton` class erroneously reset the Pressed attribute, the `ToggleButton` will no longer work correctly. This makes the `ToggleButton` class far less useful and leads to hard-to-diagnose errors when the class is used incorrectly.

The implementation of `ToggleButton` could be improved in several ways. First, we have not included any error checking to make sure that the number of items in `valueList` is equal to the number of items in `labelList`. We could rewrite the `Construct` method so that if the lists provided by the user are of different lengths, the longer of the lists is truncated so that they the lists actually used by the `ToggleButton` have the same length.

Another improvement would be to make some of the values in the `ToggleButton` “read only” from any *ADL* statement outside the scope of the object class. In *ADL*, this is done by defining a “`Set_`” method for such variables. For example, it makes sense that the member `buttonValue` should never be set outside of the class; only the class methods `Init`, `ChangeValue` and `Toggle` should be able to alter this member. To do this, we could add the following method to the `ToggleButton` class:

```
on Set_buttonValue: any val
{ }
```

Figure 5.3 ToggleButton Set_buttonValue Example

This method turns any assignment statement outside the `ToggleButton` class with the member `valueButton` on the left hand side into a null operation.

5.2 A Simple Image Viewer Class

In the second example we explore the creation of a simple image viewer. This class has the ability to display a still image from a file. It is “reloadable” in the sense that the same viewer can be used to display different images, one at a time. We implement the viewer so that the construction of the viewer is separate from the loading of an image into the viewer, allowing us to create viewers at the start of an application and load images when needed.

5.2.1 *ADL* Implementation of the Viewer Class

Line 1	defines the beginning of the `Viewer` class. This class inherits from the `XFtop` class.
Line 3	defines the `handle pimage`. This stores the `handle` to the image being displayed.
Line 4	defines an instance of the `XFvisual` class named `screen`. This is the display surface for the image.
Line 5	defines a `string` named `currentImage`. This holds the name of the file containing the image to be displayed.
Lines 7 - 10	define the `Construct` method. This method is the default constructor for the class. It simply sets the visibility of the object to `FALSE` so that it does not appear on the user’s display until it is loaded with an image.
Line 12 - 16	provide an alternative special constructor called `ConstructAndLoad`. This is used when the image is to be loaded at the same time a viewer is constructed. The constructor sets the visible attribute to `FALSE` and invokes the method `LoadImage`.
Line 18	begins the `LoadImage` method. This has a single argument containing a string with the name of the file where the image to be loaded is stored.

```
class Viewer : XFtop
{
  handle pimage;
  XFvisual screen {x=0; y=0;borderWidth=0; height=640; width=480;};
  string currentImage = "";
/* standard constructor */
  upon Construct
  {
    visible = FALSE;
  }
/* special constructor that also loads image */
  upon ConstructAndLoad: string iname
  {
    visible = FALSE;
    {'LoadImage, iname} => self;
  }

/* load a new image */
  on LoadImage: string imageName
  {
    IOfile imageFile;
    XFmessageDlg {'Create, self} => openFailDialog
     {title="File Open Failed";
       message="Attempt to open file failed:"; dialogIcon="warning";
       buttonSet="ok";};
 /* check if image is already displayed */
    if(currentImage == imageName) {
       visible = TRUE;
       return;
    }
   /* delete old image */
    if(?pimage) {
      delete pimage;
    }
/* check if new image file can be opened */
    {'OpenNative, imageName, 'ReadOnly} => imageFile;
    if('Fail=>imageFile) {
       openFailDialog.message = openFailDialog.message & imageName;
       'PostModal => openFailDialog;
        return;
    }
    'Close => imageFile;
```

Figure 5.4 Simple Image Viewer Class

```
42   /* create new image object */
43        pimage = new {'Construct, {'MEimage, {'MAfile, imageName}}}
44            =>MMimage;
45        screen.height = pimage->height;
46        screen.width = pimage->width;
47   /* display the image */
48        {'PresentOn, &screen} => pimage;
49         height = pimage->height;
50         width = pimage->width;
51         currentImage = imageName;
52         title = imageName;
53         visible = TRUE;
54     }

55     on Destroy
56     {
57       if(pimage != UNSET) {
58         delete pimage;
59       }
60     }
61   };  /* end of class Viewer */
```

Figure 5.4 Simple Image Viewer Class

Line 20 defines an instance of the IOfile wrapped class. This is used to open and read the file where the image is stored.

Lines 21-24 defines an instance of the `XFmessageDlg` wrapped class. This is used for simple dialog boxes. In this case, we use it to display warning and error messages associated with loading the image file.

Lines 26-29 test if the file is already loaded into the viewer. If so, then the viewer is made visible and the method returns.

Lines 31-33 test whether the value of `pimage` (the `handle` to the image being displayed) is set. If it is already set, then the image the `handle` points to is deleted.

Line 35 attempts to open the image file by sending the `OpenNative` message to `imageFile`.

Lines 36-40 test whether the image file was successfully opened. If it was not opened, then the message in the dialog box is set and the dialog box is posted by sending it the `PostModal` message. The method then returns if the image file could not be opened.

Line 41 closes the image file.

Lines 36-40 test whether the image file was successfully opened. If it was not opened, then the message in the dialog box is set and the dialog box is posted by sending it the `PostModal` message. The method then returns if the image file could not be opened.

Lines 43-44 use the `new` operator to construct an image object on the heap by sending the `Construct` message to the `MMimage` class.The type of file and the name of the file are sent as a list argument to the `MMimage` class.

Lines 45-46 set the width and height of the XFvisual on which the image is displayed.

Line 48 uses the `PresentOn` method to cause the image to be displayed on the visual.

Lines 49-50 set the height and width of the shell widget containing the viewer.

Lines 51-53 set the value of the `currentImage`, the `title` of the shell widget and set the visibility of the entire viewer to TRUE, making it appear on the display.

Lines 51-53 set the value of the `currentImage`, the `title` of the shell widget and set the visibility of the entire viewer to TRUE, making it appear on the display.

Lines 55-60 implement a destructor method named `Destroy` for the class. This method deletes the image pointed to by `pimage` if that value is set.

Line 61 ends the definition of the `Viewer` class.

5.2.2 Example Use of the Viewer Class

The table below shows a simple *ADL* application that makes use of the `Viewer` class. This application provides a text field for the user to input the name of the file to be displayed, a button labelled “Load” to load that file into a viewer, and an exit button to end the application.

```
anonymous: XFtop
{
  XFtextField fileName {x=5; y=5; height=50; width=300;};
  XFbutton loadButton {x=5; y=60; height=50; width=100;
      label="Load";};
  XFbutton exitButton {x=115; y=60; height=50; width=100;
      label="Exit";};
  Viewer myViewer;

  upon Construct
  {
    loadButton.Pressed = {'LoadViewer, self};
    exitButton.Pressed = {'Exit, theApp};
  }

  on LoadViewer
  {
     {'LoadImage, fileName.text} => myViewer;
  }
} myTop {height=200; width=400;};
```

Figure 5.5 Image Viewer Class Example

Line 1	declares this to be an anonymous instance inheriting from the `XFtop` class.
Lines 2-7	declare an instance of an `XFtextField` and two buttons, one labelled `Load` and the other labelled `Exit`.
Line 8	declares the variable `myViewer` to be an instance of the `Viewer` class. This is used to display images.
Lines 10 -14	provide the default constructor for the application. The `Construct` method sets the `Pressed` attributes of the load and exit buttons. When the load button is pressed, the method `LoadViewer` is messaged.
Lines 16-19	define the method named `LoadViewer`. This method messages the `LoadImage` method of the `Viewer` and provides the name of the file given in the text widget as an argument.
Line 20	closes the anonymous class's definition and uses an initialization (izor) block to set the height and width of the shell that displays when the application starts.

5.3 A Picture Button Class

In this example we create a class, called `PictureButton`, that behaves like a standard *AM2* button but has an image displayed within its borders rather than a text label. This class manages an activity called `Pressed` that accepts subscriptions from standard *ADL* Notification Request Objects (NROs).

As with a standard *ADL* button, the `PictureButton` class shows a highlighted border when the mouse is pressed down on it. The button triggers the activity when the mouse is released, but only if the release occurs while the mouse is still positioned on top of the image. This allows the user of the `PictureButton` to change his or her mind after the mouse is depressed by rolling the mouse cursor outside of the image.

By default, a `PictureButton` object is sized automatically to the match the size of the image inside of it. The programmer using the `PictureButton` class can change the position in the button where the image is displayed and can override the default size or the image by cropping or zooming. This is done by resetting any of the following six members of the class:

1. `xloc` - An integer indicating the starting *x* position where the upper left corner of the image is displayed. This defaults to 0.
2. `yloc` - An integer indicating the starting *y* position where the upper left corner of the image is displayed. This defaults to 0.
3. `clipW` - An integer indicating the clipped width of the image that is displayed. The default value of -1 is used to indicate that the width of the image is not to be clipped.
4. `clipH` - An integer indicating the clipped height of the image that is displayed. The default value of -1 is used to indicate that the height of the image is not to be clipped.

5. `offsetX` - An integer indicating where along the *x* dimension in the source image the displayed image is to start. This value is used to clip the left side of the image. The default value is 0, indicating no clipping.
6. `offsetY` - An integer indicating where along the *y* dimension in the source image the displayed image is to start. This value is used to clip the top of the image. The default value is 0, indicating no clipping.

5.3.1 *ADL* Implementation of the PictureButton Class

Here is the *ADL* code for the `PictureButton` class, with detailed comments on specific lines.

```
1    class PictureButton: XFvisual
2    {
3      string fname;                /* name of file where image is stored */
4      handle pimage;               /* handle to image object */
5      boolean armed = FALSE;       /* flag for whether button is armed */
6      boolean hilighted = FALSE;   /* flag for whether button is hilighted */
7      string hilightColor = "LightYellow"; /* highlight color for border */
8      string nohilightColor = "Black";     /* regular color for border */
9      Nro {'Create, 'MouseDown, self, 'ShowDown, {}} => nroDown;
10     Nro {'Create, 'MouseUp, self, 'ShowUp, {}} => nroUp;
11     MouseNro {'Create, 'MouseMove, self, 'ShowMove, {} } => nroMove;

12   /* parameters for image presentation */
13     boolean defaultSize=TRUE; /* use width and height of image */
14     integer xloc=0;           /* location where image is presented */
15     integer yloc = 0;         /*     ""  */
16     integer clipW= -1;        /* the clipping width--default no clip */
17     integer clipH = -1;       /* the clipping height--default no clip */
18     integer offsetX = 0;      /* the x offset of source image */
19     integer offsetY = 0;      /* the y offset of source image */
20     list ActivityInfo = {{'Pressed, {}}};
21     list Pressed={};
22     Nro {'Create, 'Pressed, NULL, "",TRUE} => nroPress;
```

Figure 5.6 PictureButton Class

```
/*  Construct method for class */
  upon Construct: string f
  {
    {'Startup,f} => self;
  }
/*  Create method for class */
  upon Create: string f, handle hparent
     init {{'Create, hparent} => XFvisual}
  {
    {'Startup,f} => self;
  }
  on Startup: string f
  {
    fname = f;
    borderColor = nohilightColor;
    borderWidth = 1;
  }

/* Init method to load image,present, scale image and register NROs*/
   on Init
   {
      pimage = new {'Construct, {'MEimage, {'MAfile, fname}}} =>MMimage;
      if(defaultSize) {
        height = pimage->height - offsetY;
        width = pimage->width - offsetX;
        if(clipW > 0) {
          width = clipW-offsetX;
         }
        if(clipH > 0) {
          height = clipH-offsetY;
        }
      }
      else {
        {'Zoom, toReal(width)/toReal(pimage->width),
            toReal(height)/toReal(pimage->height)} => pimage;
      }
      {'PresentClipped, self, xloc, yloc,clipW, clipH,
            offsetX, offsetY} => pimage;
      {'Subscribe, &nroDown} => self;
      {'Subscribe, &nroUp} => self;
      {'Subscribe, &nroMove} => self;
    }
```

Figure 5.6 PictureButton Class

```
/* Method called when mouse is moved */
  on ShowMove: any whatever, integer xval, integer yval, integer nbut,
     boolean shift, boolean command, boolean modifier
  {
    if (armed && xval>=0 && xval < width && yval>=0 && yval<height) {
      borderColor = hilightColor;
      hilighted = TRUE;
    }
    else if(armed &&(xval<0 || xval>=width || yval<0 || yval>=height)) {
      borderColor = nohilightColor;
      hilighted=FALSE;
    }
  }

/* Method called when mouse is pressed */
   on ShowDown: any whatever, list keys, list values
   {
      armed = TRUE;
      hilighted = TRUE;
      borderColor = hilightColor;
   }

/* Method called when mouse is released */
   on ShowUp: any whatever, list keys, list values
   {
     if (armed) {
      armed = FALSE;
      borderColor = nohilightColor;
        if (hilighted){
          {'TriggerNotification, 'Pressed, {}} => self;
          if (Pressed != {}) {
            at(1, Pressed) => at(2, Pressed);
          hilighted = FALSE;
          }
       }
   }

/* destructor method for the class */
  on Destroy
  {
     delete pimage;
  }
}; /* end of class PictureButton */
```

Figure 5.6 PictureButton Class

Line 1	declares the beginning of the definition of the `PictureButton` class. This class inherits from the `XFvisual` class. The inheritance from the `XFvisual` class provides the ability to display an image and manage standard mouse activities such as `MouseDown`, `MouseUp` and `MouseMove`.
Line 3	declares the member `fname`. This is a string that is used to hold the name of the file that stores the image for the button.
Line 4	declares a `handle` member named `pimage`. This stores the handle to the `MMimage` object that holds the actual image that appears on the `PictureButton` object.
Lines 5-6	declare two `boolean` members. The first of these, called `hilighted`, is a flag that indicates whether the border around the button is highlighted. This value is set to `TRUE` when the mouse is depressed on the `PictureButton`. The second `boolean`, `armed`, is set to `true` when the button is armed (i.e. when a release of the mouse on the image results in the `Pressed` activity being triggered.)
Lines 7- 8	declare the members `hilightColor` and `nohilightColor`. These strings contain the names of the colors used for the border of the `PictureButton`. They are initialized to their default values of `LightYellow` and `Black` respectively. The programmer using the class can override these defaults.
Lines 9-11	declare three NRO members of the class. These three NROs are used to deal with `MouseDown`, `MouseUp` and `MouseMove` activity. The first two are instances of the `Nro` class. The NRO for the `MouseMove` activity is an object of the `MouseNro` class because we will use the *(x,y)* coordinates of `MouseMove` events in the activity handler method, `ShowMove`.
Line 13	declares the boolean member `defaultSize`. This is set `TRUE` when if the button is to be sized to match the image displayed within it.
Lines 14-19	declare the six `integer` values that describe the clipping and scaling of the image to be displayed. They are all initialized to their default values.
Line 20	declares a `list` named `ActivityInfo`. This list is used by the *ADL* activity manager to determine what additional activities (beyond those in the base classes) are to be managed. In this case, the `Pressed` activity is added.
Line 21	declares the member `Pressed`. This value is set by the user of the `PictureButton` class to set the target and message to be sent when the button is pressed.
Line 22	creates an NRO that will be used by the `Pressed` activity. It is initialized with its target set to `NULL`.
Lines 24-27	define the special constructor `Construct`. This method just invokes the method named `Startup`.

Lines 29-33	define the special constructor `Create`. This method is used when an instance of the PictureButton class is created from the heap. It send the base class initialization message to the XFvisual base class, informing that class about the handle to the parent of the widget. It then invokes the method named `Startup`.
Line 34	defines the method called `Startup`. This method is invoked by both the `Create` and `Construct` methods. Its sole argument is the name of the file containing the image to display.
Line 36	assigns the argument `f` to the member `fname`. This member contains the name of the file that stores the image.
Line 37	assigns the default color `nohighlightColor` to the widget's `border-Color`.
Line 38	sets the value of the widget's `borderWidth` to 1.
Line 40	begin the `Init` method for the class. This method is automatically invoked as the last step in the creation of any object in *AM2*.
Line 43	uses the `new` operator to create an instance of the `MMimage` object. `Con-struct`, the special constructor for this object, is messaged with a `list` argument that has a string with the type of the element (`MEimage`), and a `list` containing a pair of values, specifically the type of image storage (`MAfile`), and the name of the file that stores the image (`fname`). The `handle pimage` is assigned the handle returned by the special constructor.
Lines 45-46	set the `height` and `width` of the `PictureButton` to the `height` and `width` of the image if the button is supposed to be sized automatically.
Lines 47-51	determine the clipped `width` and `height` of the button.
Lines 55 -56	are executed if the image is to be zoomed to a preset size. They invoke the `Zoom` method on the image.
Lines 58-59	causes the image to be presented in the button.
Lines 60 -62	subscribe the NROs to the object.
Lines 65 -76	implement the `ShowMove` method that is messaged whenever a `Mouse-Move` activity is triggered. This method checks if the button is armed from an earlier `MouseDown` event. If it is, the method turns the highlighting of the image on and off depending on whether the mouse is located inside or outside the button. This gives the user visual feedback as to whether releasing the mouse will cause the button to be pressed.
Lines 78-83	implement the `ShowDown` method. This method is messaged when the NRO `downNro` is triggered. The method arms and highlights the button.
Lines 85- 98	implement the `ShowUp` method. This method is messaged when the NRO `upNro` is triggered. The method checks if the button is armed. If it is, the value of `armed` and `borderColor` are reset, and the method checks

whether the button is highlighted. If it is highlighted, then it calls the `TriggerNotification` method. This method is part of the activity management mechanism. In this case, the implementation of `TriggerNotification` is inherited from the `XFvisual` base class. The `TriggerNotification` method takes two arguments: the name of the activity and the `list` of values to be sent to the target of the activity. In this case, the `Pressed` activity is triggered and the argument list is empty.

The `ShowUp` method also checks to see if the `Pressed` member of the `PictureButton` class has been set. This member is a `list` that can be set by the user of the class as a shortcut for setting a single action to be taken when the button is pressed. This mechanism exactly parallels the use of the `Pressed` member in standard ADL button widgets.

Lines 100-103 define the `Destroy` method for the class. This method is automatically messaged whenever an instance of the class is deleted. This occurs when an automatic instance of that type goes out of scope or when the `delete` operator is called on an instance created using the `new` operator. The `Destroy` method deletes the memory allocated for the `MMimage` instance where the image was stored.

Line 104 is the end of the definition of the class `PictureButton`.

5.3.2 An Example Using the PictureButton Class

Given the code shown above, we now turn to a simple *ADL* program that uses the `PictureButton` class. The example puts an instance of a `PictureButton` on the screen along with a text label for that button. Pressing the `PictureButton` reverses the visibility of the label. Here is the *ADL* code.

Line 1 starts the definition of an anonymous instance of an object that inherits from the *ADL* top level shell class, `XFtop.` (We use `anonymous` to create a single instance or very few instances of a class.)

Lines 3-4 declare a member of this `anonymous` class that is an instance of a `PictureButton` named `button1`. The image that is displayed on the button is in the file `dragon.gif`. The button has a border that is two pixels wide and is at *(`x,y`)* coordinates (`50,75`).

Line 5 declares an `XFlabel` at coordinates (`120,300`) that has the text label `Picture of Dragon`.

Lines 6-7 declare an `XFbutton` at coordinates (`50,350`) that is 45 pixels high and 100 pixels wide. This button has no border and its size is not adjusted depending on the space needed for the button’s label. The button contains the text `Exit`.

Line 9 declares an instance of an `Nro`. This NRO’s activity is `Pressed` and it sends a message to the `BPress` method of `self` when triggered.

```
1    anonymous: XFtop
2    {
3      PictureButton {‘Construct, “dragon.gif”} => button1
4            {borderWidth = 2; x = 50; y = 75;};
5      XFlabel mylabel {x=120;y=300;label=”Picture of Dragon”;};
6      XFbutton exitButton {x=50; y=350; height=45; width=100;
7            borderWidth=0; recomputeSize=FALSE; label=”Exit”;};
8      Nro {‘Create, ‘Pressed, self, ‘BPress, {}} => nroPress;
9
10     upon Construct
11     {
12       {‘Subscribe, &nroPress} => button1;
13       exitButton.Pressed = {‘Exit, theApp};
14     }
15
16     on BPress: any cdata, list keys, list values
17     {
18       mylabel.visible = !mylabel.visible;
19     }
20   } myTop {height=400; width=500; title=”Picture Button Demo”};
```

Figure 5.7 PictureButton Class Example

Lines 10-14 define the `Construct` method for the shell object. The body of this method subscribes `nroPressed` to the `PictureButton` and sets the `Pressed` attribute of the `XFbutton` named `exitButton` so that the `Exit` message is sent to the application when that button is pressed.

Lines 16-20 define the method `BPress`. This is the method invoked when the `Pressed` activity of `PictureButton` is triggered. The body of this method reverses the `visible` attribute of the label.

Line 21 closes the declaration of the shell widget. The `height` and `width` of the shell widget are set in an izor block.

5.4 A Video Viewer Class

In this example, we create the `VCR` class using the wrapped class `MMmovie`.

5.4.1 *ADL* Implementation of the VCR class

The *ADL* code for the `VCR` class is given below. Detailed comments on specific lines are below.

```
1    class VCR : XFlayout
2    {
3
4    handle Screen = NULL;
5    handle Movie = NULL;
6    handle PlayButton = NULL;
7    handle RewindButton = NULL;
8    handle PauseButton = NULL;
9    handle FastForwardButton = NULL;
```

```
handle StopButton = NULL;

string MovieTitle = "";
boolean Paused = FALSE;

upon Create : handle VisualParent, list SizePosition, string InitialMovie
init { {'Create, VisualParent} => XFlayout }
{  integer Xpos;
   integer Ypos;
   integer Width;
   integer Height;

   Xpos = first(SizePosition);
   Ypos = at(2, SizePosition);
   Width = at(3, SizePosition);
   Height = at(4, SizePosition);

   x = Xpos;
   y = Ypos;
   width = Width;
   height = height;

   MovieTitle = InitialMovie;

} // End of Create Method

upon Construct
{
   MovieTitle = "";
} // End of Construct Method

on Init
{
   integer ButtonWidth;

   ButtonWidth = width/5;

   Screen = new {'Create, self} => XFvisual {
               visible = TRUE;
               x = 0;
               y = 0;
               };
   Screen->width = width;
   Screen->height = height - 20;

   PlayButton = new {'Create, self} => XFbutton{
               visible = TRUE;
               label = ">";
               fontRequest = {'Helvetica, 14, {'bold}, 'roman};
               };
   PlayButton->x = 0;
   PlayButton->y = height - 20;
   PlayButton->width = ButtonWidth;
   PlayButton->height = 20;
   PlayButton->Pressed = {'Play, self};

   RewindButton = new {'Create, self} => XFbutton{
               visible = TRUE;
               label = "<<";
               fontRequest = {'Helvetica, 14, {'bold}, 'roman};
               };
   RewindButton->x = ButtonWidth;
   RewindButton->y = height - 20;
   RewindButton->width = ButtonWidth;
   RewindButton->height = 20;
   RewindButton->Pressed = {'Rewind, self};

   PauseButton = new {'Create, self} => XFbutton{
               visible = TRUE;
               label = "||";
               fontRequest = {'Helvetica, 14, {'bold}, 'roman};
               };
   PauseButton->x = 2*ButtonWidth;
   PauseButton->y = height - 20;
   PauseButton->width = ButtonWidth;
   PauseButton->height = 20;
   PauseButton->Pressed = {'Pause, self};

   FastForwardButton = new {'Create, self} => XFbutton{
               visible = TRUE;
               label = ">>";
               fontRequest = {'Helvetica, 14, {'bold}, 'roman};
               };
   FastForwardButton->x = 3*ButtonWidth;
   FastForwardButton->y = height - 20;
   FastForwardButton->width = ButtonWidth;
   FastForwardButton->height = 20;
   FastForwardButton->Pressed = {'FastForward, self};

   StopButton = new {'Create, self} => XFbutton{
               visible = TRUE;
               label = "[]";
               fontRequest = {'Helvetica, 14, {'bold}, 'roman};
               };
   StopButton->x = 4*ButtonWidth;
   StopButton->y = height - 20;
   StopButton->width = ButtonWidth;
   StopButton->height = 20;
   StopButton->Pressed = {'Stop, self};

   {'SetMovie, MovieTitle} => self;

} // End of Init Method

on SetMovie : string NewMovieTitle
{
   if (isValid(Movie)) {
   delete Movie;
   Movie = NULL;
   }

   if (NewMovieTitle != "") {
   Movie=new{'Construct,{'MEavi,{'MAfile,NewMovieTitle}}}=> MMmovie;
   {'RegisterOn, Screen} => Movie;
   }

   MovieTitle = NewMovieTitle;
   'UpdateButtons => self;

} // End of SetMovie Method

on UpdateButtons
{
   if (isValid(Movie)) {
   PlayButton->disabled = FALSE;
   RewindButton->disabled = FALSE;
   PauseButton->disabled = FALSE;
   FastForwardButton->disabled = FALSE;
   StopButton->disabled = FALSE;
   }
   else {
   PlayButton->disabled = TRUE;
   RewindButton->disabled = TRUE;
   PauseButton->disabled = TRUE;
   FastForwardButton->disabled = TRUE;
   StopButton->disabled = TRUE;
   }

} // End of UpdateButtons Method

on Show
{
   visible = TRUE;
} // End of Show Method

on Hide
{
   visible = FALSE;
} // End of Hide Method

on Play
{
   integer a, b;
   list l;
   interval i;
   if (isValid(Movie)) {
   if (! Paused ) {
   'Present => Movie;
   }
   else {
   'Resume => Movie;
   }
   Paused = FALSE;
   }
} // End of Play Method

on Rewind
{
   integer a, b;
   list l;
   interval i;

   if (isValid(Movie)) {

   if (((Movie->position) - 10) > Movie->startPosition) {
   a = (Movie->position) - 10;
   }
   else {
   a = Movie->startPosition;
   }

   if (! Paused) {
   b = Movie->endPosition;
   }
   else {
   b = a;
   }

   l = { {TRUE, a}, {TRUE, b}};
   i = toInterval(l);

   {'PlayInterval, i} => Movie;

   }

} // End of Rewind Method

on Pause
{
   integer a;
   list l;
   interval i;
   if (isValid(Movie)) {
   if (! Paused) {
   'Pause => Movie;
   Paused = TRUE;
   }
   else {
   'Resume => Movie;
   Paused = FALSE;
   }}
} // End of Pause Method

on FastForward
{
   integer a, b;
   list l;
   interval i;

   if (isValid(Movie)) {

   if (((Movie->position) + 10) < Movie->endPosition) {
   a = (Movie->position) + 10;
   }
   else {
   a = Movie->endPosition;
   }

   if (! Paused) {
   b = Movie->endPosition;
   }
   else {
   b = a;
   }

   l = { {TRUE, a}, {TRUE, b}};
   i = toInterval(l);

   {'PlayInterval, i} => Movie;

   }

} // End of FastForward Method

on Stop
{
   integer a;
   list l;
   interval i;

   if (isValid(Movie)) {

   'Stop => Movie;
   Paused = FALSE;

   a = Movie->startPosition;
   l = { {TRUE, a}, {TRUE, a}};
   i = toInterval(l);

   {'PlayInterval, i} => Movie;

   }

} // End of Stop Method

}; // End of VCR Class Definition


anonymous : XFtop

```

```
280  {
281     VCR myVcr{x = 10; y = 10; width = 400; height = 300;};
282
283   upon Construct
284   {
285      {'SetMovie, "NAME.avi"} => myVcr;
286      'Show => myVcr;
287
288   } // End of Construct Method
289   } mytop{width = 410; height = 310;};
290         */
```

Line 1	declares `VCR` class as a subclass of `XFlayout`. In other words, the `VCR` class inherits behavior from the `XFlayout` class.
Lines 4-13	declare internal variables used by the class such as button handles, the title of the current movie and a `boolean` value that tracks whether the current movie is paused or not.
Line 15	declares the "`Create`" method as a constructor using the "`upon`" key word. This method will takethree arguments including a visual parent, a list containing the size and position of the VCR and an initial movie title.
Line 16	initializes the base class of the VCR by sending the visual parent to the `XFlayout` inherited class.
Lines 17-20	declares internal variables to be used within the method. These variables do not exist outside of this method and are used for readability.
Lines 22-25	assigns each variable the appropriate value from the list which describes the coordinates and its width and height of the VCR.
Line 32	saves the filename for the initial movie to an internal class variable for later use.
Line 34	ends the declaration of the "`Create`" method.
Line 36	declares the "`Construct`" method as a constructor using the "`upon`" key word. This method takes no arguments and can only be used when the "visual parent" of the VCR can be found by asking the windowing system. Use this method when you statically create a VCR (see application example below).
Line 38	ensures that the "`MovieTitle`" variable is properly set to the default value.
Line 39	ends the declaration of the "`Construct`" method.
Line 41	begins the declaration of the "`Init`" method. This method is used to initialize internal variables, to create and position buttons and to initialize the movie to be played, if any.
Line 43	declares an internal method variable used to calculate and store the width of the buttons.

Line 45	calculates the width of all of the buttons based on the width of the VCR.
Line 47	creates a new instance of the "`XFvisual`" class and stores the new handle in the variable "`Screen`".
Lines 48-50	initialize some of the attributes of the newly created `XFvisual`, such as `coordinates` and `visibility`.
Line 52	sets the width of the "`Screen`" to be the same width of the VCR.
Line 53	sets the height of the "`Screen`" to be as large as possible but still leave room for the VCR buttons.
Line 55-59	creates a new instance of the "`XFbutton`" class and stores the new handle in a variable. Some of the attributes of the new button are set including the "`label`", `visibility` and `font` to be used.
Lines 60-63	these lines set the other attributes of the newly created button, namely the `coordinates` and `width` and `height` of the button.
Line 64	this sets the both the method name and object called when the button is pressed. In this case, the method "Play" will be sent to the "self" i.e. the current instance of the VCR class. This completes the declaration of the "Play" button for the VCR class.
Lines 66-108	these lines follow the same format and pattern as Lines 55-64, creating the "`Rewind`", "`Pause`", "`Fast Forward`" and "`Stop`" buttons of the VCR class. Pay particular attention to the "`label`" attribute of each button and Lines 75, 86, 97 and 108 which set the appearance of the button and the response of the button when pressed, respectively.
Line 110	calls the method "`SetMovie`" with the argument "`MovieTitle`" within the VCR class. This call activates the internal method responsible for loading the initial movie and preparing it for viewing.
Line 112	ends the declaration of the "`Init`" method.
Line 114	begins the declaration of the "`SetMovie`" method. The "`SetMovie`" method takes one argument in the form of a `string` which contains either a movie filename or an empty string. This method is responsible for preparing a movie to be viewed within the VCR class.
Line 116	this line checks to see if a current, valid movie exists.
Line 117	if a valid movie is currently loaded, delete the "`old`" movie.
Line 118	then, after deletion, reinitialize the "`Movie`" handle to `NULL`. This allows error checking should the new movie title be the empty string.
Line 121	this line checks to see if the "`NewMovieTitle`" is the empty string.
Line 122	if not, a new instance of `MMmovie` is created using the "`NewMovieTitle`" and the handle is stored in the internal variable "`Movie`".
Line 123	registers the new movie for display on the "`Screen`" of the VCR.

Line 126	stores the new movie title in the internal variable "`MovieTitle`".
Line 127	this line calls the method "`UpdataButtons`" within the VCR class. This method is responsible for enabling and disabling the VCR control buttons.
Line 129	ends the declaration of the "`SetMovie`" method.
Line 131	begins the declaration of the "`UpdateButtons`" method. This method enables or disables the VCR control buttons depending on whether a valid movie is loaded or not.
Line 133	checks to see of the handle stored in the internal variable "`Movie`" is valid.
Lines 134-138	if it is valid, all of the buttons are enabled so the user may use them to control viewing.
Lines 140-145	otherwise, the buttons are disabled so they may not be used.
Line 148	ends the declaration of the "`UpdateButtons`" method.
Line 150	begins the declaration of the "`Show`" method. This method displays the VCR (but does not start a movie playing).
Line 152	sets the VCR class visibility attribute to `TRUE`.
Line 153	ends the declaration of the "`Show`" method.
Line 155	begins the declaration of the "`Hide`" method. This method hides the VCR (but does not stop a movie from playing).
Line 157	sets the VCR class visibility attribute to `FALSE`.
Line 158	ends the declaration of the "`Hide`" method.
Line 160	begins the declaration of the "`Play`" method. This method is responsible for playing a valid movieor unpausing a valid, currently paused move.
Line 163	this line checks to ensure that a valid movie is loaded.
Line 164	if there is a valid movie loaded, this line checks to see if the movie is paused.
Line 165	if the movie is not paused, it is "`Present`"-ed which begins playing the movie on the "`Screen`".
Lines 167-168	if the movie is paused, it is "`Resume`"-ed which unpaused and continues the movie.
Line 171	sets the "`Paused`" state variable equal to `FALSE`.
Line 172	this line ends the "`if`" statement ensuring a valid movie is loaded.
Line 174	ends the declaration of the "`Play`" method.
Line 176	begins the declaration of the "`Rewind`" method.
Lines 178-180	declare variables to be used within the "`Rewind`" method. These variables will be used to declare segments of the movie to be displayed.
Line 182	checks to ensure the current movie is valid.

Line 184	if the current movie is valid, this line checks to see if the movie can be "`rewound`" 10 frames withoutmoving before the "`starting position`" of the movie.
Line 185	if the current position can be moved back 10 frames, the variable "`a`" is set to the current position minus 10.
Lines 187-188	if the current position can not be moved back 10 frames, the variable "`a`" is set to the starting position of the movie.
Line 191	this line checks to see if the movie is currently paused.
Line 192	if the movie is not paused, the variable "`b`" is set to the "`ending position`" of the current movie.
Lines 194-195	if the movie is currently paused, the variable "`b`" is set equal to the variable "`a`".
Line 198	this line sets the variable "l" equal to a list that is formatted to be converted into an interval of the movie that can be displayed. The format says that the variable "`a`" is the first value of the interval and will be included within the bounds of the interval and the variable "`b`" is the second value of the interval and will also be included within the bounds of the interval.
Line 199	converts the list variable "l" into an interval and stores the value in the variable "`i`".
Line 201	displays the interval of the movie defined by "`i`".
Line 203	ends the "`if`" statement ensuring a valid movie is loaded.
Line 205	ends the declaration of the "`Rewind`" method.
Line 207	begins the declaration of the "`Pause`" method.
Line 209	checks to make sure a valid movie is currently loaded.
Line 210	if there is a valid movie loaded, this line checks to see if the movie is currently paused.
Line 211	if the movie is not currently paused, pause the movie and set the variable "`Paused`" equal to `TRUE`.
Lines 214-216	if the movie is currently paused, unpause the movie and set the variable "`Paused`" equal to `FALSE`.
Line 219	this line ends the "`if`" statement ensuring a valid movie is loaded.
Line 221	ends the declaration of the "`Pause`" method.
Line 223	begins the declaration of the "`FastForward`" method.
Lines 225-227	declare variables to be used within the "`FastForward`" method. These variables will be used to declare segments of the movie to be displayed.
Line 229	checks to ensure the current movie is valid.

Line 231	if the current movie is valid, this line checks to see if the movie can be "`fast forwarded`" 10 frames without moving beyond the "`ending position`" of the movie.
Line 232	if the current position can be moved forward 10 frames, the variable "`a`" is set to the current position plus 10.
Lines 234-235	if the current position can not be moved forward 10 frames, the variable "`a`" is set to the endingposition of the movie.
Line 238	this line checks to see if the movie is currently paused.
Line 239	if the movie is not paused, the variable "`b`" is set to the "`ending position`" of the current movie.
Lines 241-242	if the movie is currently paused, the variable "`b`" is set equal to the variable "`a`".
Line 245	sets the variable "l" equal to a list that is formatted to be converted into an interval of the movie that can be displayed. The format says that the variable "a" is the first value of the interval and will be included within the bounds of the interval and the variable "`b`" is the second value of the interval and will also be included within the bounds of the interval.
Line 246	converts the list variable "l" into an interval and stores the value in the variable "`i`".
Line 248	displays the interval of the movie defined by "`i`".
Line 250	ends the "`if`" statement ensuring a valid movie is loaded.
Line 252	ends the declaration of the "`FastForward`" method.
Line 254	begins the declaration of the "`Stop`" method.
Lines 256-258	declare variables to be used within the "`Stop`" method. These variables will be used to move the movie back to the beginning.
Line 260	this line checks to see if the current movie is valid.
Line 262	if the movie is valid, stop the movie.
Line 263	resets the "`Paused`" variable to `FALSE`. A stopped movie is no longer paused.
Line 265	sets the variable "`a`" to the "`starting position`" of the movie.
Line 266	sets the variable "l" equal to a list that is formatted to be converted into an interval of the movie that can be displayed. The format says that the variable "`a`" is the first value of the interval and will be included within the bounds of the interval and the variable "`a`" is also the second value of the interval and will also be included within the bounds of the interval. This interval defines a point in the movie to be displayed, in this case, the beginning of the movie.

Line 267	converts the list variable "l" into an interval and stores the value in the variable "`i`".
Line 269	displays the interval of the movie defined by "`i`" effectively showing the first frame of the movie and stopping.
Line 271	ends the "`if`" statement ensuring a valid movie is loaded.
Line 273	ends the declaration of the "`Stop`" method.
Line 275	ends the declaration of the VCR class.
Line 277	begins a comment which includes a small, sample application which uses the VCR class. In order to use the sample application, the programmer will need to remove the "`comment`" markers.
Line 278	declares an "`anonymous`" subclass of the class `XFtop`. These are often used as the main windowfor applications.
Line 281	statically declares the creation of a `VCR object` and sets the values for `x`, `y`, `width` and `height`. In performing this type of object creation, the "`Construct`" method of the object is called. The name of the `VCR` object within the "`anonymous`" class is "`myVcr`".
Line 283	begins the declaration of the "`Construct`" method for the "`anonymous`" class.
Line 285	calls the "`SetMovie`" method of the VCR object "`myVcr`" with the argument "`NAME.avi`". The programmer should change "`NAME.avi`" to an appropriate, sample `.avi` file.
Line 286	calls the "`Show`" method of the VCR object "`myVcr`" effectively making the VCR object visible and usable by the user.
Line 288	ends the declaration of the "`Construct`" method for the "`anonymous`" class.
Line 289	completes the declaration of the "`anonymous`" class and creates an instance of the "`anonymous`" class named "`mytop`" along with setting the initial `width` and `height` of the "`anonymous`" class.
Line 290	this marks the end of the "`comment`" markers surrounding the sample application.

Chapter 6 Wrapped Class Reference

Wrapped classes are C++ classes in *AM2* that are visible in the *ADL* (**see Section 3.23, “Wrapped Classes” page 44**). They provide an interface to the *AM2* library in the *ADL*. In the sections that follow, we describe the following seven types of system-defined wrapped classes, as well as list their methods, members, and activities:

You can use wrapped classes as is, or create a subclass to add additional features. This chapter describes the system-defined wrapped classes that come with the *ADL*. To learn how to create your own wrapped classes **see Section Appendix B, “Creating Wrapped Classes” page 271**.

By convention, wrapped classes possess names that start with two capital letters specifying the module, followed by one or more concatenated words, all but the first of which are capitalized. For instance, `XFmenuCommand` is the wrapped class used to implement menu items that initiate commands. The initial two letters, “`XF`”, indicate that this wrapped class belongs to the user interface module along with other classes like `XFbutton` and `XFtextField`. The exception to this convention are the wrapped classes implementing notification request objects (`Nro`, `MouseNro` and `TimerNro`) and the abstract wrapped classes `ActivityManager` and `AttributeManager`.

Nothing prevents *ADL* programmers from following a similar convention in developing his or her own classes. In general, preexisting module prefixes should be avoided, except in instances where the developer is prototyping an *ADL* class that he or she intends to replace with a similarly named wrapped class at a later date.

The important features of a wrapped class are divided into four areas:

- **Superclasses** indicate inheritance, which is a key concept that is critical to fully understanding the behavior of each class. To help in understanding the inheritance relationships within and between classes, inheritance diagrams will be provided. As the legend for each class will show, black boxes indicate classes and gray boxes indicate abstract classes (**see Section 3.24, “Inheritance” page 47**).
- **Methods** are used to provide much of the advanced functionality of the wrapped classes. You can use methods of wrapped classes exactly like methods of user-defined *ADL* classes (**see Section 3.21, “Method Definition” page 41**).

  Constructors are special methods which are distinguished by starting with the keyword `upon` rather than the keyword `on`. Use constructors only when creating instances of objects; if no default constructor is listed for a class, then you must specify one of the constructors listed when creating the instance of the class. We do not list the `Init` method and the `Destroy` method because you should never call them directly.
- **Attributes** are similar to the members of user-defined *ADL* classes (**see Section 3.20, “Class Definition” page 40**). However, attributes of wrapped classes may be designated as “read-only,” or have side effects when changed. Also, some classes provide easy access to frequently used activities through special attributes.

  In the lists of members, the *Access* column indicates whether you can set the attribute at creation time (C), whether you can also set it after creation (S), and whether you can read it (G). In both the lists of attributes and activities, names in **gray** indicate features that are not yet implemented.
- **Activities** provide asynchronous notification of events. You can trigger activities by the mouse, keyboard, network, or other things outside the application. Although there is no direct equivalent in the *ADL*, you can write classes in the *ADL* to provide a similar notification mechanism. **See Chapter 4, “Using Activities in ADL”**for more information.

6.1 Activities and Application Services

The classes included in this section are used by almost every *AM2* application, but the *ADL* user may often be unaware of their existence. `MCapplication` is a base class to every application class, but the user should not (indeed, can not) declare it explicitly. The `ActivityManager` and

AttributeManager classes are likewise abstract base classes for any wrapped class that supports activities (**see "Using Activities in ADL" on page 59**) or attributes (**see "Member Access" on page 45**), and many applications can be written using the Pressed member rather than the underlying notification request classes `Nro`, `MouseNro`, and `TimerNro`. On the other hand, the services provided by these classes are fundamental. For example, every *ADL* program should use the `'Exit` message provided by the `MCapplication` class. Detailed documentation for the these classes are provided in the following sections:

The class inheritance tree for the Activities and Application Services (AAS) of *AM2* appears in Figure 6.1. Inheritance relationships are shown by the black lines that connect the boxes. Moving from the top to the bottom of the figure, classes inherit from those where a connection exists and they become more specific as you go down the tree. Classes with more that one connection to a superclass are said to have a multiple inheritance relationship

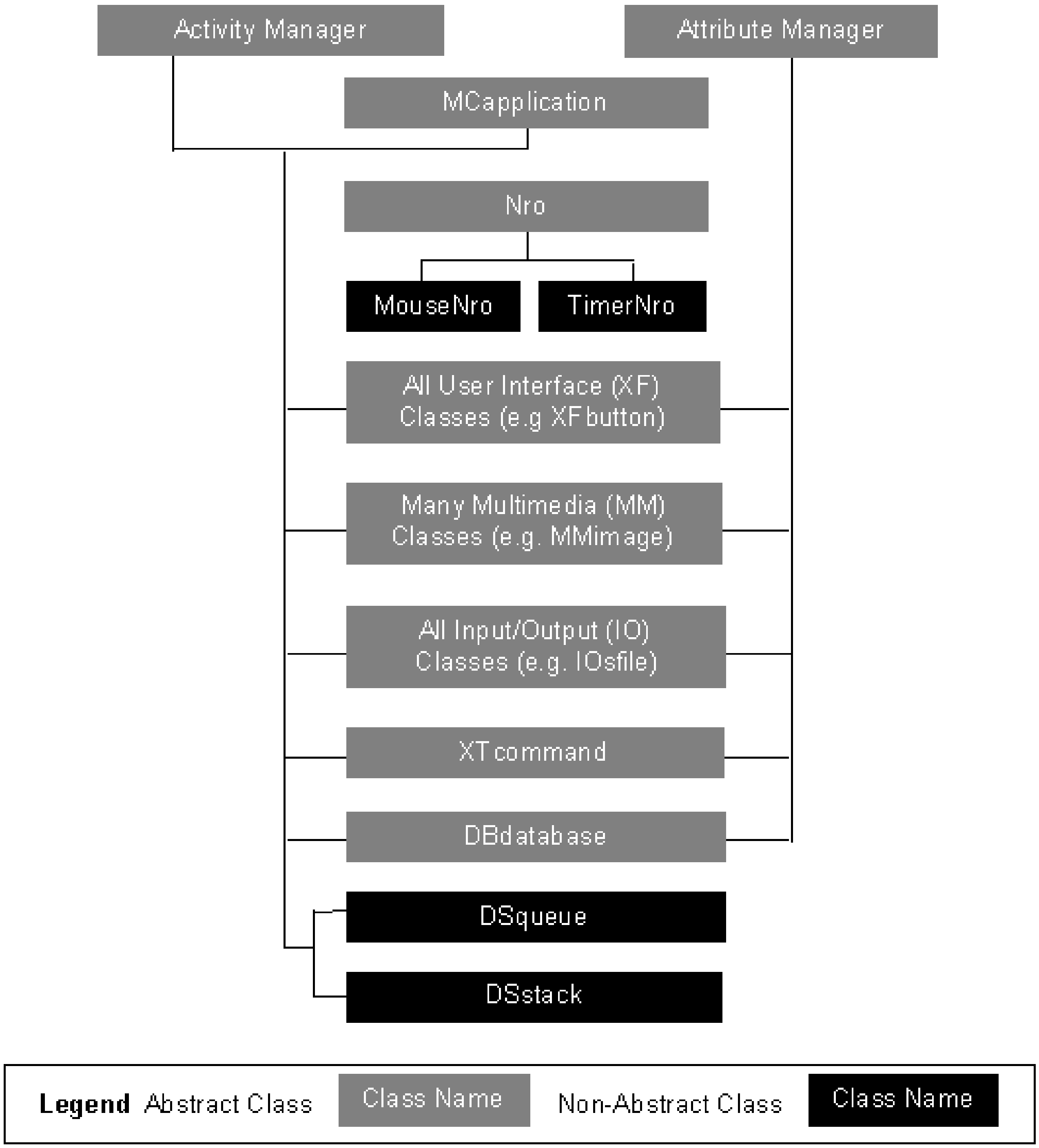

Figure 6.1 Activities and Applications Services Wrapped Classes Inheritance Tree

6.1.1 MCapplication - *Abstract*

This class serves as the implicit base of every application class, but should not be used explicitly, either as the class of any member or the base of a user defined class. That is, the global level of every *ADL* application (`theApp` Class) inherits from the `MCapplication` wrapped class although its name never appears in the program. This class provides certain application services that do not fit naturally into any other wrapped class. These include interaction with the event loop, registration of certain error handling methods for mathematical built-in functions, and subscription for timers and idle time work procedures. In order to invoke `MCapplication methods`, the user should send the appropriate message to `theApp`, an instance of theAppClass.

Superclasses

None

Methods

on Exit

Exit the event loop and terminate the application *after* returning to the event loop and executing all queued asynchronous messages and propagating all constraints. This means that any statement after the method invocation in the calling method executes: `'Exit => theApp`; Use the `Die()` built-in for emergency exits.

on Sync

Normally attribute updates take effect, asynchronous messages are delivered, and constraints propagate when an activity callback returns control to the event loop. This method forces these actions to be taken immediately.

on SetFatalErrors: list *errorList*

The mathematical built-ins recognize six named types of errors (see page 266). Two of these are fatal (`DOMAIN`, `SING`) and the other four (`OVERFLOW`, `UNDERFLOW`, `TLOSS`, `PLOSS`) are ignored by default. You can change this behavior by invoking this method on `theApp`. The single argument in the message should be a list of lists. Each sublist should be a pair consisting of a string specifying the name of an `error` type and a `boolean` constant (`TRUE`, `FALSE`) indicating whether or not the error is to be treated as fatal.

on SetBypass: integer *nEvents*

Normally the *AM2* event loop follows a strict priority model. High priority I/O events are always processed before timer events, which are processed before normal user interface events, which are processed before idle time events (work procedure callbacks). But this strict model can lock out the processing of user interface events indefinitely, say during a long stretch of software media decompression. This method inverts the event loop for the processing of *one* event every *nEvents* events. Normally, an inverted cycle of the event loop starts with checking for a user interface event, then a timer event, and finally a high priority I/O event. If the `WorkProcBypass` message has been sent to theApp with an argument of TRUE, however, then the inverted cycle will start with checking for an idle time notification request (work procedure) and will only go on to check for user interface events if none is found.

on WorkProcBypass: boolean *includeWorkProc*

If the `includeWorkProc` is TRUE, then an inverted event cycle (see the previous method) includes checking for idle time notification. Otherwise, it does not. `WorkProcBypass` can be called repeatedly to toggle the value `on` and `off`.

Attributes

None

Activities

Activity	Keys	Description
Tick	integer late, missed	periodic timer notification
Idle	*none*	application idle

Figure 6.2: MCapplication Activities

This class supports periodic timers and idle time work procedures via a mechanism that resembles activities, although it does not use the `ActivityManager` base class. From the point of view of the developer there is no difference. The developer should subscribe for the `Tick` activity used by the Nro or TimerNro classes, which contains an `Extra` member that specifies the timer interval in milliseconds.

Examples

"A Simple ADL Application with a Button"   page 60

6.1.2 Activity Manager - *Abstract*

This class is described for reference only. For further informaiton on how to use activites in ADL programs (**see Section 4.5.3, "Creating Classes That Inherit From the ActivityManager Class" page 75**).

Superclasses

None

Methods

on Subscribe: handle *hNro,* handle *hTarget* return handle

Register for the activity specified in the `Nro` pointed to by `hNro`, making the object pointed to by `hTarget` the recipient of the future notification. This method returns a NULL handle if the subscription is refused, otherwise the value of `hNro`.

on Unsubscribe: handle *hNro,* handle *hTarget* return handle

Unregister the subscription for the object pointed to by `hTarget` for the activity specified in the `Nro` pointed to by `hNro`. Return a NULL handle if the subscription could not be removed for any reason, otherwise the value of `hNro`.

on TriggerNotification : string *activityName*, list *valueList*, handle *hTarget*

Trigger a notification for the activity `activityName` sent to the *ADL* object pointed to by `hTarget`. The notification message will have the selector activityName and three arguments:

1) the client data supplied in the subscribed `Nro` of type any;

2) a list of strings that supplies the keys for the activity, drawn from `ActivityInfo` member;

3) a list of values, `valueList`, that match the keys in 2.

The message `Triggernotification` is usually only sent if the programmer has defined a class with it's own, non-standard activities. See the example in section 3.4.2.

on IsAnyoneSubscribed : string *activityName*, handle *hTarget* return boolean

Check if the object pointed to by `hTarget` has subscribed for activity `activityName`. Return TRUE or FALSE.

Attributes

Attribute	Type	Description	Access
ActivityInfo	List	This member is normally read-only, unless the user has subclassed ActivityManager to define his or her own activity. See the discussion in section 3.4.2. ActivityInfo is a specially formatted list of lists in which each sublist has the form { activityName, listOfActivityKeys }.	CG

Figure 6.3: Activity Manager Attributes

Activities

None

Example

Table 4.11, “A Class Inheriting from the ActivityManager Class,” on page 76

Table 4.12, “An Example Using the Movable Class,” on page 77

6.1.3 Attribute Manager - *Abstract*

This class is an abstract class which manages attributes in the underlying operating or windowing system so that they appear to the *ADL* programmer as if they are members of the wrapped classes. Most of the User Interface (XF) classes inherit from `AttributeManager` because attributes of the window system components like width or foreground (color) are really maintained by the underlying window system code and are not stored as actual members of the wrapped user interface classes (**see "Wrapped Classes" on page 44**).

Superclasses

None

Methods

on Commit

Normally attributes are not updated until after an activity callback returns just before the next activity is dispatched for processing. The user can force updating of an object's attributes by sending a '`Commit` to the object.

Attributes

None

Activities

None

Example

None

6.1.4 Nro

This class implements a general notification request object that is more efficient than a pure *ADL* version. For a general discussion of the use of this class, see **Chapter 3 "Using Activities in ADL" page 59**, and for a specific discussion of using a generic NRO, **see Section 4.2, "Using Notification Request Objects" page 62** .

Superclasses

None

Methods

upon Create: string *activity,* handle *client,* string *method,* any *clientData*

Create a notification request object for the specified *activity* that invokes *method* on the *client* with the argument *clientData* as the first argument of the callback message.

upon CreateExtra: string *activity*, handle *client*, string *method*, any *clientData*, any *extra*

Same as the previous with the exception that it initializes the member Extra to the value *extra*.

on HandleActivity: list *keys*, list *values*

Should not be called from the *ADL*, although it may be overwritten in a derived NRO class. This method is called as part of the notification sequence. *keys* contains the string names of the attributes pertaining to this notification, and *values*, contains the corresponding data in the same order.

on Lookup: string *key*, list *keys*, list *values* return any

Lookup up *key* in the notification attribute-value pairs specified by *keys* and *values* and return the appropriate value. This method should become class `common`, when `common` is implemented.

Attributes

Attribute	Type	Description	Access
mActivity	string	name of activity for which notification is requested	CG
mExtra	any	extra notification request data	CG
mClient	handle	notification target object	CSG
mMethod	string	notification target method	CSG
mClientData	any	user-supplied notification argument	CSG

Figure 6.4: Nro Attributes[a]

a. For further explanation, **see Section 4.6, “Creating Customized NROs” page 78.**

Activities

None except for the *actvityName?* activities (**see “Using Activities for Notification of Subscriptions” on page 80**).

Example

Table 4.3, “An Example Using an NRO,” on page 64

6.1.5 MouseNro

This class implements a special notification request object for mouse activities that is more efficient than a pure *ADL* version. This NRO is used with all of the supported mouse activities in widgets. For a further discussion of `MouseNro`, **see Section 4.3.1, “Mouse NROs” page 65.**

Superclasses

Section 6.1.4, “Nro” page 118

Methods

upon Create: string *activity*, handle *client*, string *method*, any *clientData*

Create a mouse notification request object for the specified *activity* that invokes *method* on the *client* with the argument *clientData* as the first argument of the callback message.

on HandleActivity: list *keys*, list *values*

Should not be called from the *ADL*, although it may be overwritten in a derived NRO class. This method is called as part of the notification sequence. *keys* contains the string names of the attributes pertaining to this notification, and *values*, contains the corresponding data in the same order. For the list of attributes corresponding to mouse activities **see “Basic Widget Activities” on page 127**.

Attributes

None

Activities

None

Example

Table 4.4, “Using MouseNro Objects,” on page 66

6.1.6 TimerNro

This class implements a special notification request object for timers that is more efficient than a pure *ADL* version. This NRO is used with the `Tick` activity of `MCapplication` base to the application class (**see "MCapplication - Abstract" on page 115**). For further information about the TimerNro, **see Section 4.3.2, "Timer NROs" page 67**.

Superclasses

Section 6.1.4, "Nro" page 118

Methods

upon Create: integer *ival*, handle *client*, string *method*, any *clientData*

Create a notification request object for the `Tick` activity that invokes *method* on the *client* every *ival* milliseconds with the argument *clientData* as the first argument of the callback message.

on HandleActivity: list *keys*, list *values*

Should not be called from the *ADL*, although it may be overwritten in a derived NRO class. This method is called as part of the notification sequence. *keys* contains the string names of the attributes pertaining to this notification, and *values*, contains the corresponding data in the same order.

Attributes

None

Activities

None

Example

"Example Using A Timer"   page 68

6.2 User Interface

The class inheritance tree for the User Interface (XF) wrapped classes appears in Figure 6.5. As the legend shows, black boxes indicate classes and gray boxes indicate abstract classes. Documentation for all classes in this tree appears in this section.

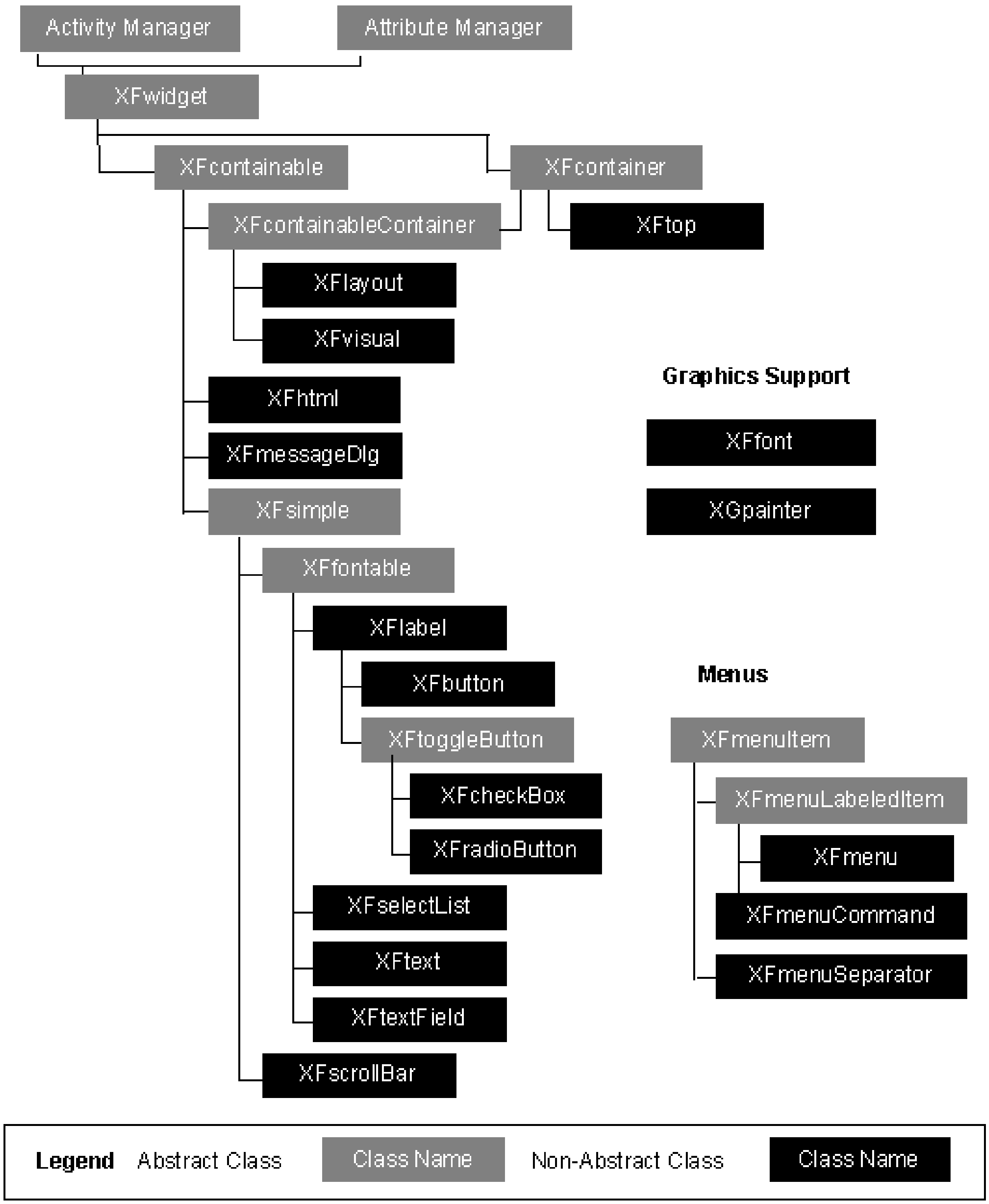

Figure 6.5: User Interface Wrapped Classes Inheritance Tree

6.2.1 XFwidget - *Abstract*

XFwidget, the base class for all windowed widgets, defines all common attributes and activities.

Superclasses

Section 6.1.2, “Activity Manager - Abstract” page 116

Section 6.1.3, “Attribute Manager - Abstract” page 118

Methods

None

Attributes

Attribute	Type	Description	Default	Access
x	integer	x-coord upper left outside corner of widget	0	CGS
y	integer	y-coord upper left outside corner of widget	0	CGS
width	integer	widget width (in pixels), including border	100	CGS
height	integer	widget height (in pixels), including border	100	CGS
borderWidth (only on UNIX)	integer	width of the border surrounding the widget	0	CGS
foreground (only on UNIX)	string	foreground drawing color for the widget	platform	CGS
background (limited on NT)	string	background drawing color for the widget (unavailable on NT for XFmessageDlg, XFhtml, XFcheckBox, XFradioButton, XFtext, XFselectList, XFtextField, XFfont nor any classes derived from XFmenuItem)	platform	CGS
borderColor	string	color of the border surrounding the widget	platform	CGS
visible	boolean	Status of the widget relative to the screen; if set to TRUE, the widget is displayed.	TRUE	CGS
disabled	boolean	Determines if widget receives user input. When set to TRUE, widget is disabled and doesn't receive keyboard or mouse input.	FALSE	CGS
systemLook	boolean	Specifies if widget should maintain the look of the platform. Setting this to TRUE may override other attributes on the widget.	FALSE	CG

Figure 6.6: Basic Widget Attributes

Activities

Activity	Keys	Description
MouseUp	integer x, y, button boolean shift, control, modifier	mouse button released
MouseDown	*same*	mouse button pressed
MouseDblClick	*same*	mouse button double-clicked
MouseMove	*same*	mouse moved
MouseDrag	*same*	mouse moved while button held down
Destroyed	*none*	widget destroyed
Shown	*none*	widget displayed
Hidden	*none*	widget removed from display
Resized	integer width, height	widget resized
Moved	integer x, y	widget moved
FocusIn (only on UNIX)	none	widget gained input focus
FocusOut (only on UNIX)	*none*	widget lost input focus
Refresh	integer x, y, integer width, height	widget redrawn
Help	*none*	help requested for widget
KeyPressed	*string character, bool-ean shift, boolean command, boolean modifier*	key pressed in widget
KeyRepeat	*same*	key held down in widget

Figure 6.7: Basic Widget Activities

Example

None

6.2.2 XFcontainable - *Abstract*

The `XFcontainable` class is an abstract class that defines the interface and behavior of all containable widgets such as `XFbuttons`, `XFtextFields`, and so on. A widget is said to be "`containable`" if it can be placed as a child of another widget (the container).

Superclasses

Section 6.2.1, "XFwidget - Abstract" page 125

Methods

None

Attributes

None

Activities

None

Example

None

6.2.3 XFcontainer - *Abstract*

The `XFcontainer` class is an abstract class that defines the interface and behavior of all container widgets such as the `XFtop`. A widget is said to be a container if it can contain or hold other widgets as children.

Superclasses

Section 6.2.1, "XFwidget - Abstract" page 125

Methods

None

Attributes

None

Activities

None

Example

None

6.2.4 XFcontainableContainer - *Abstract*

The `XFcontainableContainer` class is an abstract class that defines the interface and behavior of all containable widgets which are also containers such as `XFlayout` and `XFvisual`.

Superclasses

Section 6.2.2, "XFcontainable - Abstract" page 128

Section 6.2.3, "XFcontainer - Abstract" page 128

Methods

None

Attributes

None

Activities

None

Example

None

6.2.5 XFtop

The `XFtop` class is a top-level widget, most commonly used as a container for other widgets. It defines the standard appearance for the primary windows of an application. `XFtop` defines two basic areas: a menu bar and a work area. The menu bar area is optional and is created only if there is a menu attached to the `XFtop`.

Superclasses

Section 6.2.3, "XFcontainer - Abstract" page 128

Methods

upon Construct

Default constructor.

Attributes

Attribute	Type	Description	Default	Access
title	string	caption to be displayed in the widget's title bar	"XFtop"	CGS
windowStyle	list	Specifies the style for the widget's decorations. This list should be composed of zero (0) or more of the following style names: "titleBar", "resizeControl", and "windowFrame".	{"titleBar", "resizeControl", "windowFrame"}	CG
menuBar	handle	defines a handle to the XFmenu object associated with this XFtop	NULL	CGS

Figure 6.8: XFtop Attributes

Activities

Activity	Keys	Description
Active	*none*	topShell has been activated
Deactive	*none*	topShell has been deactivated
MenuCommand	string command, list commandPath	a menu command was selected

Figure 6.9: XFtop Activities

Example

This code creates an instance of an anonymous class derived from `XFtop` and assigns appropriate values for some of the `XFtop`'s attributes.

```
// In order to use XFtop as a container we use inheritance

    anonymous : XFtop
    {
      // ... Some useful members here ...
      // ... We can put other widgets here ...

    } topShell { width = 200; height = 300; title = "ModuleName";};
```

6.2.6 XFlayout

The `XFlayout` class represents a widget that acts as a manager and container for other widgets. As a manager, it provides simple geometry management for children widgets and does not force positioning or sizes on them.

Superclasses

Section 6.2.4, "XFcontainableContainer - Abstract" page 129

Methods

upon Construct

Default constructor.

upon Create: handle *hParent*

Alternate constructor; `hParent` is a handle to its container widget.

Attributes

None

Activities

None

Example

```
// Create a class derived from XFlayout to contain to XFlabel objects
class Canvas : XFlayout
{
  XFlabel screen
    {
      x=35; y=20; width=135; height=100; background="steelBlue";
      label = ""; borderWidth = 5; borderColor = "darkOrchid";
    };

  XFlabel title
    {
      x = 50; y = 150; width = 100; height = 30; label = "Layout";
      fontRequest = {"Helvetica", 14, {"bold"}, "roman"};
    };
};

anonymous : XFtop
{
  // Create an instance of the XFlayout derived class
  Canvas layout {x=100; y=50; width=200; height=200; visible=FALSE;};

  XFbutton controller
    {
      x = 100; y = 270; width = 70; height = 40; label = "Show";
```

```
            recomputeSize = FALSE; background = “grey”;
            fontRequest = {“Helvetica”, 14, {“bold”}, “roman”};
          };

        XFbutton quitButton
          {
            x = 230; y = 270; width = 70; height = 40; label = “Quit”;
            background = “grey”;
            fontRequest = {“Helvetica”, 14, {“bold”}, “roman”};
          };

        // This method changes the visibility of the XFlayout when called
        // Note that because the XFlayout contains another two objects,
        // their visibility also changes
        on ChangeVisibility
          {
            if ( layout.visible ) { controller.label = “Show”; }
            else { controller.label = “Hide”; }
            layout.visible = !layout.visible;
          }

        on Exit
          {
            “Exit” => theApp;
          }

        upon Construct
          {
            controller.Pressed = {“ChangeVisibility”, self };
            quitButton.Pressed = {“Exit”, self };
          }

      } top {width = 400; height = 350; background = “grey”;};
```

6.2.7 XFvisual

The `XFvisual` class provides a display surface for media objects. Since this class inherits from `XFcontainableContainer`, it can provide very simple geometry management of multiple widget children.

Superclasses

Section 6.2.4, “XFcontainableContainer - Abstract” page 129

Methods

upon Construct

Default constructor.

upon Create: handle *hParent*

Alternate constructor; `hParent` is a handle to its container widget.

Attributes

None

Activities

None

Example

```
anonymous : XFtop
{
  // Create an XFvisual to display an image
  XFvisual screen
    {
      x = 40; y = 20; width = 320; height = 200;
    };

  handle hPicture;
  upon Construct
    {
      width = 300;
      height = 200;
      // Create the image to display
      hPicture = new {“Construct”, {“MEimage”,
         {“MAfile”,”dragon.gif”}}} => MMimage;

      // Add the XFvisual object as the sink for this image
      {“AddSink”, &screen} => hPicture;
      “Show” => hPicture;
      ExitButton.Pressed = {“Quit”, self };
    }

  XFbutton ExitButton
```

```
        {
          x = 150;
          y = 240;
          width = 100;
          height = 60;
          label = “Quit”;
          fontRequest = {“Helvetica”, 18, {“bold”}, “roman”};
        };

      on Quit
        {
          “Exit” => theApp;
        }

    } topShell { width = 400; height = 320; title = “XFvisual example”;};
```

6.2.8 XFhtml

This class provides a display surface for MMhtml objects.

Superclasses

Section 6.2.2, “XFcontainable - Abstract” page 128

Methods

upon Construct

Default constructor.

upon Create: handle *hParent*

Alternate constructor; `hParent` is a handle to its container widget.

on SetFont: string *targetElementType*, handle *anXFfont*

Sets the font for all elements of tag type `targetElementType` in this HTML display surface; currently, `targetElementType` can be “H1”, “H2”, “H3”, “H4”, “H5”, “H6”, “LISTING”, “PLAIN”, “ADDRESS”, “FIXED”, “ITALIC”, “BOLD”, and “NORMAL”.

Attributes

None

Activities

Activity	Keys	Description
AnchorPressed	integer element_id, string anchor_name, text, href	A hyperlink anchor has been pressed and released. The anchor pressed is contained in anchor_name.
HTMLSubmitForm	string href, method, integer attribute_count, list attribute_names, attribute_values	An HTML submit button (for forms) has been pressed and released
ImageMapPressed	string image_src, integer x, y, element_id, string anchor_name, href, text	An inlined HTML image map has been pressed and released. The key "image_src" is the URL of the inlined image. "x" and "y" are the coordinates of the user's mouse click within the image map. "element_id" is for future extension only. "anchor_name" is the name of the anchor for this image map. "href" is the URL of the object which contains the mapping information. "text" contains the hypertext associated with the anchor.

Figure 6.10: XFhtml Activities

Example

The methods `XFhtml` and `MMhtml` work together (**see Section 6.3.12, “MMhtml” page 195**).

6.2.9 XFmessageDlg

The `XFmessageDlg` class defines a simple message dialog box, which is normally used to present transient messages. An `XFmessageDlg` consists of a message symbol, a message, and a number of push-buttons that are used to respond to the message and dismiss the dialog box.

Superclasses

Section 6.2.2, "XFcontainable - Abstract" page 128

Methods

upon Construct

Default constructor.

upon Create: handle *hParent*

Alternate constructor; `hParent` is a handle to its container widget.

on PostModal: return *string*

Posts dialog and forces the user to respond or dismiss the dialog.

Attributes

Attribute	Type	Description	Default	Access
title	string	text to be displayed in the widget's title bar	"XFmessageDlg"	CGS
message	string	text to be displayed in the dialog.	" "	CGS
dialogIcon	string	Symbol to display along with the message: one of "error", "warning", "information", or "question"	" "	CGS
buttonSet	string	Set of buttons to be displayed in the dialog box: one of: "ok", "okCancel", "yesNo", or "yesNoCancel"	"okCancel"	CGS
defaultButton	string	The default button for the dialog box: must be one of: "ok", "cancel", "yes", or "no". If the specified button is not part of the button set, this attribute is ignored.	"ok"	CGS

Figure 6.11: XFmessageDlg Attributes

Activities

None

Example

The following example illustrates the use of `XFmessageDlg` objects

```
anonymous : XFtop
{

  // Create a message dialog to prevent the user from quitting
  // without confirming
  // Note that creating the QuitDialog object does NOT pop up the
  // dialog box.
  XFmessageDlg QuitDialog
    {
      title=“Quit”;
      message=“Do you really want to exit\nthis great application?”;
      dialogIcon=“question”;
      buttonSet=“yesNo”;
      defaultButton=“no”;
    };

  XFbutton ExitButton
    {
      x = 50;
      y = 45;
      width = 100;
      height = 60;
      label = “Quit”;
      font=new {“Create”, “Helvetica”, 18,{“bold”},“roman”}=> XFfont;
    };

  upon Construct
    {
      ExitButton.Pressed = {“Quit”, self };
    }

  on Quit
    {
     // Post the dialog in modal mode. After returning from this call
     // the dialog object still exists and can be posted again
      string answer = {“PostModal”} => QuitDialog;
      if ( answer == “yes” )
   {
     echo(“Quitting...\n”);
     “Exit” => theApp;
   }
    }

} topShell { width = 200; height = 150; title = “Modal Messages”;};
```

6.2.10 XFsimple - *Abstract*

The class `XFsimple` is an abstract class that defines the interface and behavior of simple widgets that cannot contain any other widgets.

Superclasses

Section 6.2.2, "XFcontainable - Abstract" page 128

Methods

None

Attributes

None

Activities

None

Example

None

6.2.11 XFfontable - *Abstract*

The `XFfontable` class is an abstract class that defines the interface and behavior of all other classes for which a font can be set. Some examples of such classes are the `XFselectList`, `XFlabel`, `XFbutton`, and `XFtext`.

Superclasses

Section 6.2.10, "XFsimple - Abstract" page 138

Attributes

Attribute	Type	Description	Default	Access
fontRequest	list	specifies a request to use a certain font to display the text in the widget	{}	CGS
font	handle	real font used to display the text in the widget	platform dependent	CGS

Figure 6.12: XFfontable Attributes

Activities

None

6.2.12 XFlabel

The `XFlabel` class represents a static text label. This, along with the `XFbutton` is one of the most widely used widgets in GUI-based applications. The `XFlabel` class defines the basic behavior and interface to render and manage static text by controlling its color, font, alignment, and other visual attributes.

Superclasses

Section 6.2.11, "XFfontable - Abstract" page 138

Methods

upon Construct

Default constructor.

upon Create: handle *hParent*

Alternate constructor; `hParent` is a handle to its container widget.

Attributes

Attribute	Type	Description	Default	Access
label	string	text to be displayed in widget	"XFlabel"	CGS
alignment	string	label alignment (language, left, center, or right)	"center"	CGS
marginTop (only on UNIX)	integer	amount of space between top of label text and top margin	0	CGS
marginBottom (only on UNIX)	integer	amount of space between bottom of label text and bottom margin	0	CGS
marginLeft (only on UNIX)	integer	amount of space between left of label text and left margin	0	CGS
marginRight (only on UNIX)	integer	amount of space between right of label text and right margin	0	CGS
recomputeSize	boolean	determines whether the widget resizes itself to accommodate its text	TRUE	CGS

Figure 6.13: XFlabel Attributes

Activities

None

Example

The following example illustrates the use of `XFlabel` objects.

```
anonymous : XFtop
{
  XFbutton ExitButton
    {
      x = 150;
      y = 120;
      width = 100; height = 60; label = “Quit”;
      fontRequest = {“Helvetica”, 18, {“bold”}, “roman”};
    };

  // Create a label that will have the current time as its text
  XFlabel timeLabel
    {
      width = 400;
      height = 100;
      background = “darkOrchid”;
      fontRequest = {“Helvetica”, 48, {“bold”}, “roman”};
      // Assign the currentTime to its label attribute
      label = toString(localTime());
      // Always be center aligned
      alignment = “center”;
      // Do not change size after the label is set time by time
      recomputeSize = FALSE;
    };

  // Create timer
  PMclock timer; any timerKey;

  upon Construct
    {
      ExitButton.Pressed = {“Quit”, self };
      // Subscribe for time notification every second
      timerKey = {“Subscribe”, 999, “Tick”, {}, self} => timer;
    }

  on Tick : list cd, list ad return boolean
    {
      // Assign the current time to the label
      timeLabel.label = toString(localTime());
      return TRUE;
    }

  on Quit
    {
      // Remove timer notification
      {“Unsubscribe”, timerKey} => timer;
      “Exit” => theApp;
    }

} topShell { width = 400; height = 200; title = “TimeLabel”;};
```

6.2.13 XFbutton

The `XFbutton` class represents a basic interface push-button. Push-buttons are one of the most widely-used widgets in GUI-based applications. `XFbutton` supports activities through which an application can perform an action in response to some user interaction. The appearance of an `XFbutton` changes to make it look either pressed in when selected or raised when unselected.

Superclasses

Section 6.2.12, “XFlabel” page 139

Methods

upon Construct

Default constructor.

upon Create: handle *hParent*

Alternate constructor; `hParent` is a handle to its container widget.

Attributes

Attribute	Type	Description	Default	Access
showAsDefault	boolean	specifies whether the button should be marked as the default button	FALSE	CGS

Figure 6.14: XFbutton Attributes

Activities

Activity	Keys	Description
Pressed	*none*	button pressed and released in widget

Figure 6.15: XFbutton Activities

Example

The following code creates an instance of a class derived from `XFtop` which contains an `XFbutton` member. When the button is pressed, the application exits.

```
// Create an XFtop to be used as a module top-level widget
anonymous : XFtop
{
     // Lets create a button member
     XFbutton ExitButton
       {
         x = 50;
         y = 20;
         width = 100;
         height = 60;
         label = “Quit”;
    };
     // ... Some other useful members here ...
  // Constructor for the XFtop
    upon Construct
       {
         // Use special XFbutton member Pressed to subscribe
      // for the Pressed activity When the Pressed actity occurs
         // send the message Quit to this XFtop (self)
         ExitButton.Pressed = {“Quit”, self };
       }
  // Quit Method
     on Quit
       {
         echo(“Quitting...\n”);
         “Exit” => theApp;
       }

} topShell { width = 200; height = 300; title = “ModuleName”;};
```

6.2.14 XFtoggleButton - *Abstract*

The XFtoggleButton class represents a button widget that is either set or unset. XFtoggleButtons are most commonly used in groups with either a one-of-many behavior, which means that only one button in the group can be set at a time, or an n-of-many behavior, which means that any number of buttons in the group can be set at one time. The `XFtoggleButton` class defines the common members, methods, and activities for two more specific classes: **see “XFcheckBox” on page 143**, and **“XFradioButton” page 144**.

Superclasses

Section 6.2.13, “XFbutton” page 141

Methods

upon Construct

Default constructor.

upon Create: handle *hParent*

Alternate constructor; `hParent` is a handle to its container widget.

on Toggle

Toggles the state of the `XFtoggleButton` object.

Attributes

Attribute	Type	Description	Default	Access
set	boolean	determines whether the XFtoggleButton object is set or not	FALSE	CGS

Figure 6.16: XFtoggleButton Attributes

Activities

Activity	Keys	Description
StateChange	none	state of the XFtoggleButton changed

Figure 6.17: XFtoggleButton Activities

Example

For a sample program using this class, see specific examples of `ToggleButtons` "`XFcheckBox` and `XFradioButton`" **see "XFradioButton" on page 144**.

6.2.15 XFcheckBox

The `XFcheckBox` class represents a kind of toggle button, a button that is either set or unset, to be used in groups with an n-of-many behavior (that is, any number of `XFcheckBoxes` in the same group may be set at one time.) It is important to note, however, that this behavior is not enforced by the widget itself, and the application should follow this interface guideline. An `XFcheckBox` consists of an indicator (a square on all platforms) and a label area. As its name suggests, the indicator tells whether a particular `XFcheckBox` is set or unset.

Superclasses

Section 6.2.14, "XFtoggleButton - Abstract" page 142

Methods

upon Construct

Default constructor.

upon Create: handle *hParent*

Alternate constructor; `hParent` is a handle to its container widget.

on Toggle

Toggles the state of the `XFcheckBox` object.

Attributes

None

Activities

None

Example

For a sample program using this class, see "`XFcheckBox` and `XFradioButton`" **see "XFradioButton" on page 144**.

6.2.16 XFradioButton

The XFradioButton class represents a kind of toggle button, a button that is either set or unset, to be used in groups with a one-of-many behavior (that is, only one `XFradioButton` in the same group can be set at a time.) It is important to note that this behavior is not enforced by the widget itself, and that the application should follow this interface guideline. An `XFradioButton` consists of an indicator (diamond or circle depending on the platform) and a label area. As its name suggests, the indicator tells whether a particular `XFradioButton` is set or unset.

Superclasses

Section 6.2.14, "XFtoggleButton - Abstract" page 142

Methods

upon Construct

Default constructor.

upon Create: handle *hParent*

Alternate constructor; `hParent` is a handle to its container widget.

on Toggle

Toggles the state of the `XFradioButton` object.

Attributes

None

Activities

None

Example

This example of the Toggle Buttons, `XFcheckbox` and `XFradiobutton`, also shows several uses of the `XFfont` class.

```
// This class defines a NRO for XFradioGroup activities
class RadioNro : Nro
{
  upon Create: string act, handle cli, string mtd, any cd init
    {
      {“Create”, act, cli, mtd, cd} => Nro
    }
  {
  }
  on HandleActivity: list keys, list values
    {
     {mMethod,mClientData,{“Lookup”,“Selection”, keys, values}
             =>self}
                  => mClient;
    }
};
// This class implements the one-of-many behavior that groups of
// XFradioButtons should follow
class XFradioGroup : ActivityManager
{
  handle mCurrent = NULL;
  // Define the activity StateChange
  list ActivityInfo = { {“StateChange”,{“Selection”}} };

  on AddRadioButton : handle hRadio
    {
    if (hRadio->set)
      {
    if (mCurrent == NULL)
         {
           mCurrent = hRadio;
         }
    else
         {
     echo(“\n***ERROR:***\n”);
     echo(“The” & mCurrent->label &
               “XFradioButton is already set\n”);
     echo(“Can’t have two set XFradioButtons in the same group\n”);
          die(“Aborting..”);
         }
      }
```

```
        // Subscribe with the new added XFradioButton to be
        // notified when its state changes
        {“Subscribe”, new {“Create”,“StateChange”, self,“Selection”,
            hRadio} => Nro } => hRadio;
      }
    on Selection : handle hRadio, list keys, list values
      {
        if (hRadio->set)
        {
          if (mCurrent != NULL)
           {
            if (mCurrent != hRadio)
              {
               mCurrent->set = FALSE;
              }
           }
          mCurrent = hRadio;
          // Notify all dependents that the group state has changed
         {“TriggerNotification”, “StateChange”, {hRadio}} => self;
        }
      }
  };
  anonymous : XFtop
  {
    anonymous : XFlayout {
   // Create group of XFradioButtons to handle a one-of-many behavior

      XFradioGroup mGroupBox;

      XFlabel BaudRateLabel
        {
        x=5; y=10; recomputeSize=FALSE; alignment=“left”; width=160;
  label=“Baud Rate”; fontRequest={“Helvtica”,14,{“bold”},“roman”};
        };
      // Create different XFradioButtons
      XFradioButton baudRate1
        {
     recomputeSize = FALSE; x = 10; y = 30; width = 100; height = 30;
     label=“1200”; fontRequest={“Helvetica”, 12, {“bold”}, “roman”};
        };
     XFradioButton baudRate2
        {
     recomputeSize = FALSE; x = 10; y = 60; width = 100; height = 30;
     label=“2400”; fontRequest={“Helvetica”, 12, {“bold”}, “roman”};
        };
      XFradioButton baudRate3
        {
     recomputeSize = FALSE; x = 10; y = 90; width = 100; height = 30;
        label=“4800”; fontRequest={“Helvetica”, 12,{“bold”},“roman”};
        };
         XFradioButton baudRate4
           {
     recomputeSize = FALSE; x = 10; y = 120; width = 100; height = 30;
        label=“9600”; fontRequest={“Helvetica”,12,{“bold”},“roman”};
```

```
         };

          // Create two XFcheckBoxes
          XFcheckBox parity
            {
            recomputeSize=FALSE; set = !set; x = 160; y=30; width=150;
            height = 30; label = "Check Parity";
            fontRequest = {"Helvetica", 12, {"bold"}, "roman"};
            };

          XFcheckBox carrier
            {
            recomputeSize = FALSE; x = 160; y = 60; width = 150;
            height = 30; label = "Detect Carrier";
            fontRequest = {"Helvetica", 12, {"bold"}, "roman"};
            };
         RadioNro {"Create", "StateChange", self, "GroupChange", {}}=>
               groupNro;
          upon Construct
            {
              // Add XFradioButtons to the group
              {"AddRadioButton", &baudRate1} => mGroupBox;
              {"AddRadioButton", &baudRate2} => mGroupBox;
              {"AddRadioButton", &baudRate3} => mGroupBox;
              {"AddRadioButton", &baudRate4} => mGroupBox;
              // Subscribe for any changes in the group's state
              {"Subscribe", &groupNro} => mGroupBox;
            }
          on Init
            {
              if (mGroupBox.mCurrent != NULL)
               {
                 echo(mGroupBox.mCurrent->label + "\n");
               }
            }
          on GroupChange : any cdata, handle hRadio
            {
              // Print out the label of the selected XFradioButton
              echo(hRadio->label + "\n");
            }
        } layout { x = 10; y = 10; width = 300; height = 155; };

    XFbutton ExitButton
         {
          x = 110;
        y = 175;
        width = 100;
        height = 60;
        label = "Quit";
        font=new{"Create","Helvetica", 18, {"bold"},"roman"}=> XFfont;
         Pressed = {"Exit", theApp};
          };
    } top {height=250; width=320; title="Communications Settings";};
```

6.2.17 XFselectList

The `XFselectList` class represents a widget that allows selection from a list of different choices. `XFselectList` displays a single column of text items or choices that can be selected in a variety of ways, using both the mouse and the keyboard.

Superclasses

Section 6.2.11, "XFfontable - Abstract" page 138

Methods

upon Construct

Default constructor.

upon Create: handle *hParent*

Alternate constructor; `hParent` is a handle to its container widget.

on AddItem: string *anItem*

Appends an item to the end of the list.

on InsertItem: string *anItem*, integer *aPosition*

Inserts an item at the specified position.

on GetItemPos: string *anItem* return integer

Returns the position of the first ocurrence of the specified item in the list. If the item is not in the list, the method returns 0 (zero).

on GetItemCount: return integer

Returns the number of items in the list.

on GetSelectedCount: return integer

Returns the number of selected items in the list.

on GetSelectedItems: return list

Returns a list of strings containing all selected items in the list. It returns an empty list if no items are selected.

on GetSelectedPos: return list

Returns a list of integers containing the positions of all selected items in the list. It returns an empty list if no items are selected.

on IsPosSelected: integer *aPosition* return boolean

Determines if the item at the specified position is selected. Returns TRUE if the item is selected, FALSE otherwise.

on SelectAtPos: list *thePositions*

Selects and highlights the items at the specified positions in the list.

on DeselectAtPos: integer *aPosition*

Deselects and unhighlights the item at the specified position in the list.

on RemoveItem: string *anItem*

Removes the first ocurrence of the specified item from the list.

on RemoveAllItems

Removes all items from the list.

on RemoveAtPos: integer *aPosition*

Removes the item at the specified position from the list.

Attributes

Attribute	Type	Description	Default	Access
selectionMode	string	The mode in which the list should support selections. Possible values are: “single” and “multiple”.	“single”	CGS
items	list	the list of choices	{}	CGS

Figure 6.18: XFselectList members

Activities

Activity	Keys	Description
Selection	*none*	Item selected. This activity is also reported when an item is deselected in a multiple selection list.
ListAction	string item, integer itemPosition	Action initiated on an item. This usually happens when the user double-clicks on an item or presses return or enter when an item is selected.

Figure 6.19: XFselectList activities

Example

The following code illustrates the use of `XFselectList` objects

```
anonymous : XFtop
{
  XFbutton deleteButton
    {
      x = 285; y = 75; width = 100; height = 60;
      label = “Delete”; background = “steelBlue”;
      fontRequest = {“Helvetica”, 18, {“bold”}, “roman”};

  XFbutton clearButton
```

```
        {
          x = 285; y = 195; width = 100; height = 60;
          label = “Clear”; background = “steelBlue”;
          fontRequest = {“Helvetica”, 18, {“bold”}, “roman”};
        };

      XFbutton exitButton
        {
          x = 285; y = 315; width = 100; height = 60;
          label = “Quit”; background = “steelBlue”;
          fontRequest = {“Helvetica”, 18, {“bold”}, “roman”};
        };

      XFlabel listTitle
        {
          x = 10; width = 250; height = 30;
          label = “List of Items”; background = “steelBlue”;
          fontRequest = {“Helvetica”, 14, {“bold”, “italic”}, “roman”};
        };

      // Create a selectable list
      XFselectList itemList
        {
          x = 10; y = 30; width = 250; height = 335; background = “grey”;
          fontRequest = {“Times”, 12, {“bold”}, “roman”};
          // Allow multiple selections
          selectionMode = “multiple”;
        };

      XFlabel textTitle
        {
          x = 10; y = 370; width = 250; height = 30;
          label = “Add Item”; background = “steelBlue”;
          fontRequest = {“Helvetica”, 14, {“bold”}, “roman”};
        };

      XFtextField addField
        {
          x = 10; y = 400; width = 250; height = 40; background = “grey”;
          fontRequest = {“Times”, 14, {“bold”}, “roman”};
        };

      // Create a NRO  for XFtextField’s TextAccept activity
      Nro {“Create”, “TextAccepted”, self, “Accept”, {}} => AddNro;

      upon Construct
        {
          exitButton.Pressed = {“Quit”, self };
          deleteButton.Pressed = {“Delete”, self };
          clearButton.Pressed = {“Clear”, self };
      // Subscribe for notification on XFtextField’s TextAccept activity
          {“Subscribe”, &AddNro } => addField;
        }

```

```
  on Clear
    {
      // Clear the list
      “RemoveAllItems” => itemList;
    }

  on Delete
    {
      // If there are any items selected, delete them
      list selection = “GetSelectedItems” => itemList;
      string item;
      for item in selection
   {
     {“RemoveItem”, item } => itemList;
   }
    }

  on Accept : list clientData, list keys, list values
    {
 // If a text was entered in the textField, add it to the list of items
      if (addField.text != ““)
   {
     {“AddItem”, addField.text } => itemList;
     addField.text = ““;
   }
    }

  on Quit
    {
      “Exit” => theApp;
    }

} topShell { width = 400; height = 450;
        background = “steelBlue”; title = “Add List”;};
```

6.2.18 XFtext

The XFtext class represents a multiple line text widget that allows text to be inserted, deleted, modified, and selected.

Superclasses

Section 6.2.11, “XFfontable - Abstract” page 138

Methods

upon Construct

Default constructor.

upon Create: handle *hParent*

Alternate constructor; `hParent` is a handle to its container widget.

on ScrollText: integer *lines*

If lines is positive it scrolls the text upward. If it is negative it scrolls the text downward.

Attributes

Attribute	Type	Description	Default	Access
text	string	text to be displayed	“ ”	CGS
cursorPosition	integer	position of cursor in text string	0	CGS
horizontalScrollBar (only on UNIX)	boolean	determines whether the widget has a horizontal scrollBar or not	TRUE	CGS
verticalScrollBar (only on UNIX)	boolean	determines whether the widget has a vertical scrollBar or not	TRUE	CGS
editable	boolean	determines whether the text is read-only or not	TRUE	CGS
wordWrap	boolean	determines whether the widget should break lines automatically between words.	FALSE	CGS

Figure 6.20: XFtext Attributes

Activities

Activity	Keys	Description
TextChange	*none*	text in widget has been changed

Figure 6.21: XFtext Activities

Example

The following code illustrates the use of `XFtext` objects:

```
anonymous : XFtop
{
  XFbutton ExitButton
    {
      x = 150;
      y = 255;
      width = 100;
      height = 60;
      label = “Quit”;
      fontRequest = {“Helvetica”, 18, {“bold”}, “roman”}};
    };

  // Create an editable text widget
  XFtext editor
    {
      y = 30;
      width = 400;
      height = 220;
      background = “steelBlue”;
      // Break lines automatically
      wordWrap = TRUE;
      fontRequest = {“Times”, 12, {“bold”}, “roman”};
    };

  XFlabel editorTitle
    {
      width = 400;
      height = 30;
      label = “Simple Text Editor”;
      background = “darkOrchid”;
      fontRequest = {“Helvetica”, 14, {“bold”, “italic”}, “roman”};
    };

  upon Construct
    {
      ExitButton.Pressed = {“Quit”, self };
    }

  on Quit
    {
      “Exit” => theApp;
    }

} topShell { width = 400; height = 320; title = “Text Editor”;};
```

6.2.19 XFtextField

The `XFtextField` class represents a single-line text widget that allows text to be inserted, deleted, modified, and selected. As a single-line text editor, the `XFtextField` has a subset of the functionality of the XFtext `widget`.

Superclasses

Section 6.2.11, “XFfontable - Abstract” page 138

Methods

upon Construct

Default constructor.

upon Create: handle *hParent*

Alternate constructor; `hParent` is a handle to its container widget.

Attributes

Attribute	Type	Description	Default	Access
text	string	text to be displayed	“ ”	CGS
cursorPosition	integer	position of cursor in text string	0	CGS
editable	boolean	determines whether the text is read-only or not	TRUE	CGS

Figure 6.22: XFtextField Attributes

Activities

Activity	Keys	Description
TextChange	*none*	text in widget has been changed
TextAccepted	none	text in widget has been accepted

Figure 6.23: XFtextField Activities

Example

The following code illustrates the use of `XFtextField` objects

```
anonymous : XFtop
{
  XFbutton ExitButton
    {
      x = 150;
      y = 255;
      width = 100;
      height = 60;
      label = “Quit”;
      font=new{“Create”,“Helvetica”,18,{“bold”},“roman”}=> XFfont;
    };

  // Create the form with XFtextField and XFlabel objects

  XFlabel nameLabel
    {
      x = 10; y = 10; width = 80; height = 30;
      label = “Name :”; background = “steelBlue”;
      alignment = “language”;
      fontRequest = {“Helvetica”, 14, {“bold”}, “roman”};
    };

  XFtextField name
    {
      x = 100; y = 10; width = 280; height = 40;
      background = “steelBlue”;
      fontRequest = {“Times”, 12, {“bold”}, “roman”};
    };
  XFlabel ssLabel
    {
      x = 10; y = 55; width = 200; height = 30;
      label = “Social Security :”;
      alignment = “left”;
      background = “steelBlue”;
      fontRequest = {“Helvetica”, 14, {“bold”}, “roman”};
    };
  XFtextField social
    {
      x = 220; y = 55; width = 160; height = 40;
      background = “steelBlue”;
      fontRequest = {“Times”, 12, {“bold”}, “roman”};
    };
  XFlabel emplName
    {
      x = 10; y = 100; width = 100; height = 30;
      label = “Employer :”; background = “steelBlue”;
      alignment = “left”;
      fontRequest = {“Helvetica”, 14, {“bold”}, “roman”};
    };
  XFtextField employer
    {
      x = 120; y = 100; width = 260; height = 40;
      background = “steelBlue”;
      fontRequest = {“Times”, 12, {“bold”}, “roman”};
    };
  XFlabel addName
    {
      x = 10; y = 145; width = 200; height = 30;
      label = “Address :”;
      background = “steelBlue”;
      alignment = “left”;
      fontRequest = {“Helvetica”, 14, {“bold”}, “roman”};
    };
  XFtext address
    {
      x = 120; y = 145; width = 260; height = 80;
      background = “steelBlue”;
      fontRequest = {“Times”, 12, {“bold”}, “roman”};
    };
  upon Construct
    {
      ExitButton.Pressed = {“Quit”, self };
    }
  on Quit
    {
      “Exit” => theApp;
    }
} topShell { width = 400; height = 320; background = “steelBlue”;
        title = “Form”;};
```

6.2.20 XFscrollBar

The class `XFscrollBar` provides a widget to control the scrolling of the viewing area in other widgets. It allows users to view data that are too large to be displayed all at once, and is usually adjacent to the widget that contains the data for viewing.

Superclasses

Section 6.2.10, "XFsimple - Abstract" page 138

Methods

upon Construct

Default constructor.

upon Create: handle *hParent*

Alternate constructor; `hParent` is a handle to its container widget.

Attributes

Attribute	Type	Description	Default	Access
minimum	integer	the minimum value of the slider	0	CGS
maximum	integer	the maximum value of the slider	100	CGS
position	integer	the slider’s position	0	CGS
increment	integer	amount the position changes due to the user’s moving the thumb1 increment	1	CGS
pageIncrement	integer	amount the position changes due to user’s moving thumb 1 page increment.	10	CGS
orientation	string	direction the scrollbar is displayed. Values are “vertical” and “horizontal”.	“vertical”	CG

Figure 6.24: XFscrollBar Attributes

Activities

Activity	Keys	Description
Increment	*integer position*	Thumb’s position incremented by one.
Decrement	same	Thumb’s position decremented by one.
PageIncrement	same	Thumb’s position incremented by one page.
PageDecrement	same	Thumb’s position decremented by one page.
ThumbTrack	same	The position of the thumb changes while being dragged.
ThumbPosition	same	Thumb has changed position.

Figure 6.25: XFscrollBar Activities

Example

```
uses “nro.adl”@”StdLib”;
class colorLabel: XFlabel
{
  upon Create: string bg
    {background=bg; label=bg; x=0; y=0 width=80; height=50;}
};
class Sequence : XFlayout
{
  colorLabel {“Create”, “darkOrchid”} => purple    { x = 0;  };
  colorLabel {“Create”, “lightBlue”}  => lightBlue { x = 80; };
  colorLabel {“Create”, “plum”}       => plum      { x = 160;};
  colorLabel {“Create”, “orange”}     => orange    { x = 240;};
  colorLabel {“Create”, “red”}        => red       { x = 320;};
  colorLabel {“Create”, “yellow”}     => yellow    { x = 400;};
  colorLabel {“Create”, “green”}      => green     { x = 480;};
  colorLabel {“Create”, “royalBlue”}  => royalBlue { x = 560;};
  list colors={“darkOrchid”,”lightBlue”,”plum”,”orange”,
                ”red”,”yellow”,“green”,”royalBlue”};
};
```

```
   anonymous: XFtop
   {
    Sequence samples { x = 30; y = 195; width = 640; height = 50; };
     // Create a horizontal scrollbar that acts as a slider
     XFscrollBar slider
       {
         orientation = “horizontal”;
         x = 30; y = 245; width = 640; height = 30;
         //Set the scroll range
         minimum = 1;  maximum = 8;
       };
     // Create a notification request object to subscribe for
     // the XFscrollBar’s ThumbPosition activity
     vanillaNro {“Create”,“ThumbPosition”,self,“ChangeColor”, {}}
                   => colorNro;
     XFlabel colorSample
       {
         label = ““; x = 60; y = 30; width = 580; height = 120;
         borderWidth = 3;
       };
     XFbutton ExitButton
       {
         x = 300; y = 290; width = 100; height = 60;
        label=“Quit”;
            fontRequest={“Helvetica”, 18, {“bold”}, “roman”};
       };
     on ChangeColor : list unused
       {
 // Whenever slider’s thumb changes because of user input change
 // labels background color to the appropriate color in sequence
      colorSample.background = at(slider.position, samples.colors);
       }
     on Exit
       {
         “Exit” => theApp;
       }
     upon Construct
       {
         {“Subscribe”, &colorNro } => slider;
         // Initialize the color sample to the first in the sequence
         {“ChangeColor”, {}} => self;
         ExitButton.Pressed = {“Exit”, self };
       }
} colors { width = 700; height = 360; title = “Color Sequence”;};
```

6.2.21 XFmenuItem - *Abstract*

The class `XFmenuItem` is an abstract class that represents all the different kinds of items that can be placed in a menu such as `XFmenuSeparators`, `XFmenuCommands`, and `XFmenus`.

Superclasses

Section 6.1.2, "Activity Manager - Abstract" page 116

Section 6.1.3, "Attribute Manager - Abstract" page 118

Methods

None

Attributes

None

Activities

None

Example

For a sample program, see "`XFmenu`, `XFmenuCommand`, and `XFmenuSeparator`" (**"XFmenuSeparator"   page 162**).

6.2.22 XFmenuLabeledItem - *Abstract*

The class `XFmenuLabeledItem` is an abstract class that represents labeled items that can be placed in a menu such as `XFmenuCommands` and `XFmenus`.

Superclasses

Section 6.2.21, "XFmenuItem - Abstract" page 159

Attributes

Attribute	Type	Description	Default	Access
enabled	boolean	Defines whether the menu item is enabled or not	TRUE	CGS
label	string	text to be displayed in the menu item label.	Defined by sub-class	CGS
mnemonic (only on UNIX)	string	Defines the key that can be used in conjunction with a modifier key to post a PullDownMenu. Platforms that support this attribute underline the character in the label string that matches the mnemonic. This attribute is ignored on the Mac platform	“ ”	CGS

Figure 6.26: XFmenuLabeledItem Attributes

6.2.23 XFmenu

The class `XFmenu` represents a logical menu that can be instantiated into a physical menu such as a menu bar, a pull-down menu or a pop-up menu, on demand. Applications typically create different menu systems by creating an instance (or instances) of this class and attaching them to other objects. For example, to create a menu bar, an application can create an `XFmenu` object (or objects) and attach it to an existing `XFtop` object.

Superclasses

Section 6.2.22, “XFmenuLabeledItem - Abstract” page 159

Methods

upon Construct

Default constructor.

upon ConstructTop

This constructor should be used when the `XFmenu` object to be created is the top-most menu.

upon Create: handle *hContainer*

Alternate constructor; `hContainer` is a handle to its container menu.

Attributes

Attribute	Type	Description	Default	Access
tearOff (only on UNIX)	boolean	Defines whether the menu is a tear-off menu or not. This attribute works as a hint to the platforms that support this kind of menus	FALSE	CGS

Figure 6.27: XFmenu Attributes

Activities

None

Example

For a sample program, see "`XFmenu`, `XFmenuCommand`, and `XFmenuSeparator`" (**"XFmenuSeparator" page 162**).

6.2.24 XFmenuCommand

The class `XFmenuCommand` represents menu items that initiate some command as they are selected.

Superclasses

Section 6.2.22, "XFmenuLabeledItem - Abstract" page 159

Methods

upon Construct

Default constructor.

upon Create: handle *hContainer*

Alternate constructor; `hContainer` is a handle to its container menu.

Attributes

Attribute	Type	Description	Default	Access
accelerator (onlyon UNIX)	string	Defines the key that can be used in conjunction with the command key (Command or Ctrl) as a direct shortcut to invoke a menu item without popping up its menu pane. Platforms display a readable representation of this key sequence to the right of the item in the menu.	“ ”	CGS

Figure 6.28: XFmenuCommand Attributes

Activities

None

Example

For a sample program, see “`XFmenu`, `XFmenuCommand`, and `XFmenuSeparator`” (**“XFmenuSeparator” page 162**).

6.2.25 XFmenuSeparator

The class `XFmenuSeparator` represents a menu item that separates other items in a menu by drawing a horizontal line between them.

Superclasses

Section 6.2.21, “XFmenuItem - Abstract” page 159

Methods

None

Attributes

None

Activities

None

Example

```
// Create a subclass of XFmenu to include several menuItems
class XFtopMenu: XFmenu
{
  upon Construct
    init {  “ConstructTop” => XFmenu }
  {
  }
};

anonymous: XFtop
{
 // Create the logical menu structure
 anonymous: XFtopMenu
  {
   anonymous: XFmenu
    {
      XFmenuCommand quit       { label = “Quit”;  accelerator = “Q”; };
    } fileMenu { label = “File”; mnemonic = “F”; };
   anonymous: XFmenu
    {
     XFmenuCommand add       { label = “Add”; accelerator = “A”; };
     XFmenuCommand reset     { label = “Reset”; accelerator = “R”; };
     XFmenuSeparator separator;
     XFmenuCommand removeCmd { label=“Remove”; accelerator=“v”; };
    } testMenu { label = “Test”; mnemonic = “T”; tearOff = TRUE;};
   } mainMenu;

  Nro {“Create”,“MenuCommand”,self,“MenuCommand”,{}}=> menuCmdNro;

  on Init
   {
     // Attach the logical menu to an XFtop menuBar
     menuBar = &mainMenu;
     // Subscribe for any menuCommand in the physical XFtop’s menu
     {“Subscribe”, &menuCmdNro} => self;
   }

  // This method gets called when a command is selected
  on MenuCommand: any clientData, list keys, list values
   {
     string command = at(1,values);
     // Call the command method
     command ?=> self;
   }

  on Quit
   {
     “Exit” => theApp;
   }

  on Add
   {
```

```
            // Add was selected do something
            echo(“Add selected \n”);
          }

        on Reset
          {
            // Reset was selected do something
            echo(“Reset selected \n”);
          }

        on Remove
          {
            // Remove was selected do something
            echo(“Remove selected \n”);
          }

      } top { width = 100; height = 100; };
```

6.2.26 XFfont

The `XFfont` class represent font objects that are commonly used with other `XF` classes such as `XFlabel`, `XFtext` and so on. This class provides constructors to create fonts from explicit information including the font name, size, style, and encoding.

Superclasses

None

Methods

upon Create: string *fontName*, integer *size*, list *style*, string *encoding*

FontName: the name of the font, such as ‘helvetica, ‘times, ‘fixed, ‘symbol, ‘courier. Other generic font names are defined on the Macintosh platform and will soon be defined on the other platforms. These generic names are: “applicationFont”, “textFont”, “typewriterFont”, and “systemFont”. To ensure better font handling across the different platforms, system configurations, and languages, use of these generic names is recommended.

Size: size of the font.

Style: a list of types, which may contain any of: ‘bold, ‘italic, ‘underline, ‘underlineDouble, ‘underlineDotted, ‘superscript, ‘subscript, ‘outline, ‘smallCaps.

Encoding: one of ‘roman, ‘japanese, ‘arabic, ‘hebrew, ‘greek, ‘symbol.

This special constructor creates a font object from the given information. AthenaMuse 2 attempts to find the best match for the font requested; the members described below can be examined to see exactly what font is being used.

Attributes

Attribute	Type	Description	Default	Access
fontName	string	current font name	platform	G
size	integer	current font size	platform	G
style	list	list of styles for current font	platform	G
encoding	string	current font encoding	platform	G

Figure 6.29: XFfont Attributes

Activities

None

Example

The following example shows several uses of the `XFfont` class.

```
uses “nro.adl”@”StdLib”;

anonymous : XFtop
{
// Create different fonts to include in selection list
XFfont{“Create”,“Times”,12,{},“roman”} => Times12;
XFfont{“Create”,“Times”,14,{},“roman”} => Times14;
XFfont{“Create”,“Times”,24,{},“roman”} => Times24;
XFfont{“Create”,“Times”,14,{“bold”},“roman”} => TimesBold;
XFfont{“Create”,“Times”,14,{“italic”},“roman”} => TimesItalic;
XFfont{“Create”,“Times”,14,{“bold”,”italic”},“roman”}=>TimesBItalic;
XFfont{“Create”,“Helvetica”, 12, {}, “roman”} => Helv12;
XFfont{“Create”,“Helvetica”, 14, {}, “roman”} => Helv14;
XFfont{“Create”,“Helvetica”, 24, {}, “roman”} => Helv24;
XFfont{“Create”,“Helvetica”, 36, {}, “roman”} => Helv36;
XFfont{“Create”,“Helvetica”, 14, {“bold”},“roman”} =>HelvBold;
XFfont{“Create”,“Helvetica”, 14, {“italic”},“roman”}=>HelvItalic;
XFfont{“Create”,“Symbol”,14,{}, “symbol”} => Symbol;

 list fonts={&Times12, &Times14, &Times24, &TimesBold, &TimesItalic,
      &TimesBItalic, &Helv12, &Helv14, &Helv24, &Helv36, &HelvBold,
      &HelvItalic, &Symbol};

  XFlabel fontSample
    {
      y = 230; width = 450; height = 100;
      label = “The quick brown fox”;
      background = “steelBlue”;
      recomputeSize = FALSE;
    };

  XFselectList fontList
    {
      x = 75; y = 15;
      width = 300;
      height = 200;
      items = {“Times 12”, “Times 14”, “Times 24”, “Times Bold”,
         “Times Italic”, “Times Bold & Italic”,
         “Helvetica 12”,“Helvetica 14”,“Helvetica 24”,“Helvetica 36”,
         “Helvetica Bold”,”Helvetica Italic”, “Symbol font”};
      // Setting the fontRequest attribute creates an XFfont object
      // that is then assigned to the font attribute
      fontRequest = {“Helvetica”, 12, {“bold”}, “roman”};
    };

  XFbutton ExitButton
    {
      x = 175;
      y = 355;
      width = 100;
      height = 60;
      label = “Quit”;
      //Set the button’s font to be Helvetica, 18 points, and bold.
      //Create the font dynamically and then assign it to
      //the font attribute which expects a font handle.
      font=new {“Create”, “Helvetica”,18, {“bold”},“roman”}=>XFfont;
    };

  // Create a notification request object for the
  // XFselectList’s Selection activity
  vanillaNro {“Create”,“Selection”, self “ChangeFont”,{}}=>FontNro;

  upon Construct
    {
      ExitButton.Pressed = {“Quit”, self };
      {“Subscribe”, &FontNro } => fontList;
    }

  on ChangeFont : list clientData
    {
      string item;
      integer position = 1;
      list selection = “GetSelectedItems” => fontList;
      if (!isEmpty(selection))
   {
     position = {“GetItemPos”, at(1,selection) } => fontList;
     fontSample.font = at(position, fonts);
   }
    }

  on Quit
    {
      echo(“Quitting...\n”);
      “Exit” => theApp;
    }

} topShell { width = 450; height = 450;
        background = “steelBlue”; title = “Fonts”;};
```

6.2.27 XGPainter

The `XGpainter` interface provides a simple set of operations and represents the necessary state to perform 2D drawing. Many of the methods described use lists as the representation for points with the idea that it can be used for both 2D and 3D graphics. For instance if we were to draw a line from (1,2) to (10,10) we would make a call similar to:

```
{"DrawLine", {1,2}, {10,10} } => myPainter;
```

Superclasses

Section 6.2.11, "XFfontable - Abstract" page 138

Methods

upon Construct

Default constructor. This should set the members into a useful default.

upon Create: handle *hParent*

Alternate constructor; `hParent` is a handle to its container widget.

on ClearDrawingArea

Clears (erases) the entire drawing area.

on DrawPoint: list *point*

Draws the given point with the current `penWidth` and `color`.

on DrawPoints: list *points*

Draws the given points with the current `penWidth` and `color`.

on DrawLine: list *point1*, list *point2*

Draws a line from point1 to point2 with the current `penWidth` and `color`.

on DrawPolyLine: list *points*

Draws a line segment from the first point to the second point, then from the second point to the third, and so on.

on DrawRectangle: list *origin*, integer *width*, integer *height*

Draws the outline of a rectangle with the current `penWidth` and `color`.

on FillRectangle: list *origin*, integer *width*, integer *height*

Fills the specified rectangle with the current `color`.

on DrawEllipse: list *origin*, integer *width*, integer *height*

Draws the outline of an ellipse specified by its bounding rectangle with the current `penWidth` and `color`.

on FillEllipse: list *origin*, integer *width*, integer *height*

Fills the ellipse specified by its bounding rectangle with the current `color`.

on DrawPolygon: list *points*

Draw the outline of the polygon specified by the list of `points`, using the current `penWidth` and `color`.

on FillPolygon: list *points*

Fills the specified polygon with the current `color`.

on DrawCircle: list *center*, integer *radius*

Draws the outline of the specified circle using the current `penWidth` and `color`.

on Fill Circle: list *center*, integer *radius*

Fills the specified circle of the given `center` and `radius` with the current `color`.

on DrawText: list *startPos*, string *text*

Draws the specified `text` starting at `startPos` using the current `font`.

Attributes

Attribute	Type	Description	Access
drawingArea	handle	current drawing area. This is a handle to an XFvisual object.	CSG
penWidth	integer	current width of the pen used to draw	CSG
fontRequest	list	font requested to be used for text operations	CSG
font	handle	current font used for text operations	CSG
color	handle	This is the current color to be used in graphic operations. This is a handle to an MMcolor object. Specifies the raster operation or logical function to be used. Logical functions control how the source pixel values generated by a graphics request are combined with the destination pixel values already present on the screen. These operations are: copy, and, or, and xor.	CSG

Figure 6.30: XGpainter Attributes

Activities

None

Example

This example using the `XGpainter` creates a class called `Polygon`. The `anonymous` instance derived from `XFtop` after the `Polygon` class definition then uses the `Polygon` class to create a filled square (in black) and a unfilled triangle in yellow.

```
XGpainter painter;

// Class for a Polygon
class Polygon
{
  global XGpainter painter;

  list points = {};     //list of vertices in form { {x1,y1}, {x2,y2} ...}
  integer lineWidth=1;   // width of drawing line
  handle colorHandle;    // handle to MMcolor object
  string drawColor='black; // default color
  handle drawParent;     // parent widget to draw on
  boolean fill=FALSE;    // indicator for filled Polygon

// Constructor has parent widget for Polygon as argument
  upon Construct: handle p
  {
    drawParent = p;
    colorHandle= new {'CreateName, drawColor} => MMcolor;

  }

// Destructor deletes the allocated MMcolor object
  on Destroy
  {
    delete colorHandle;
  }

// Method to reset the drawing color
  on Set_drawColor: string color
  {
    delete colorHandle;
    drawColor=color;
    colorHandle = new {'CreateName, drawColor} => MMcolor;
    painter.color = colorHandle;
  }

// Method that draws the Polygon
  on Draw
  {

     list fullPoints = points;

     painter.drawingArea = drawParent;
     painter.penWidth = lineWidth;
     painter.color = colorHandle;

     if(fill)  {
        {'FillPolygon, points} => painter;
      }
      else {
        fullPoints << first(points);
        {'DrawPolygon, fullPoints} => painter;
      }
    }

};  /* end of class Polygon */

// Anonymous instance derived from XFtop that uses Polygon class
anonymous:XFtop {

        XFvisual vis {x=30; y=30; height=200; width=400;};
                     // drawing surface
        Polygon {'Construct, &vis} => triangle
                {points={{100,100}, {100,200}, {200,100}};
                drawColor='Green; lineWidth=4;};
        Polygon {'Construct, &vis} => filledSquare
                {points={{10,10}, {100,10}, {100,100}, {10,100}};
                drawColor='Black; fill=TRUE;};

        XFbutton quitButton {x=250; y=30; label="Quit";};
// The following NRO is used to Subscribe for refresh events on XFvisual
        Nro {'Create, 'Refresh, self, 'RedrawRefresh, NULL} => redrawNro;

        upon Construct
        {
                {'Subscribe, &redrawNro} => vis;
                quitButton.Pressed = {'Exit, theApp};
                'Draw => filledSquare;
                'Draw => triangle;
        }
// This method handles refresh on the XFvisual by redrawing Polygons
        on RedrawRefresh: any cd, list names, list vals
        {
                'Draw => filledSquare;
                'Draw => triangle;
        }

} myTop {x=100; y=100; height=300; width=500;};
```

6.3 Multimedia

Multimedia applications should be thought of as the juxtaposition and presentation of media elements to users through the user interface. The *ADL* contains two key features which enhance an application designer's ability to manage and control multimedia presentations more efficiently.

1. Multiple presentations of a single media element can be created and controlled independently. This is more efficient than duplicating a media element in order to make and manipulate multiple presentations of it. Each presentation request for a media element returns a special type of handle called a ***presentationID***, which can be passed as an argument to specify which presentation of a media element to control. Each presentation exists as long as the media element that created it exists, or until the particular presentation is removed, at which time the ***presentationID*** handle is made invalid. In the media classes, there are frequently two alternative forms of a method. The form without a handle will perform the specified action on all presentations of a media element, while the method (s) named "*methodID*" with a handle will perform the action on the specified ***presentationID***.
2. The reusability of application interfaces through the interchangeability of multimedia was a key design goal of *AM2*. The `MMbroker` adds a subtle capability which helps the *ADL* programmer to achieve this goal. It allows applications to access media elements without needing to be aware of their actual format. This makes it much easier to construct interchangeable sets of media. For further explanation of the use of `MMbroker`, **see Section 6.3.5, "MMbroker" page 182**.

The classes which are essential for supporting the above features are described in the first two sections of this section, followed by descriptions of how the specific media classes take advantage of these capabilities for different forms of media:

The class inheritance tree diagram for the Multimedia (MM) wrapped classes of *AM2* appears in Figure 6.31.

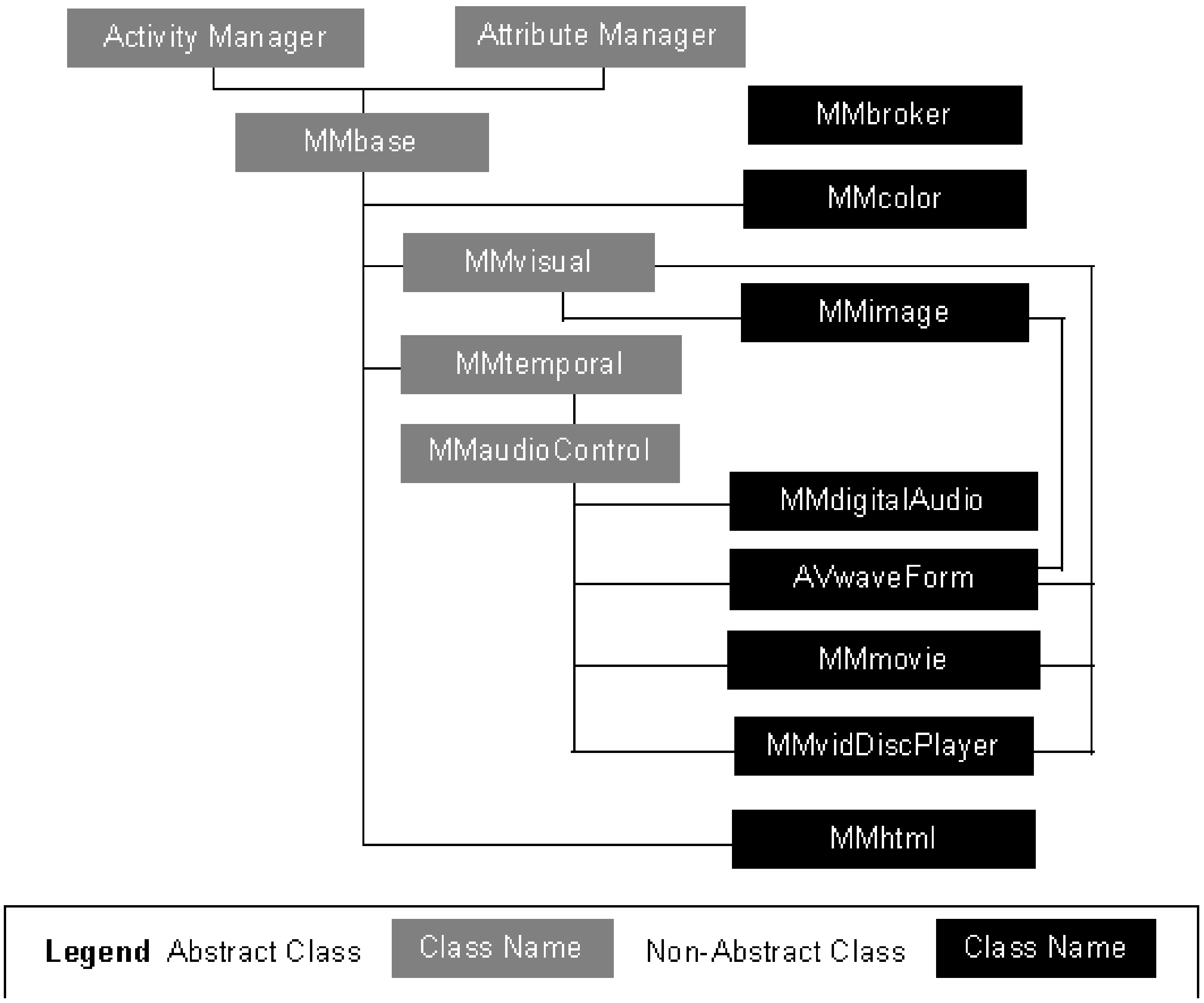

Figure 6.31: Multimedia Wrapped Classes Inheritance Tree

6.3.1 MMbase - *Abstract*

An abstract class can *not* be directly created within an *ADL* program or used as a base of a user-defined *ADL* class, so utilization of the `MMbase` methods, members and activities *must* be through the fully-derived wrapped media classes. However, all fully derived media classes inherit from `MMbase`, so they share the generic methods, members and activities listed below.

Superclasses

Section 6.1.2, “Activity Manager - Abstract” page 116

Section 6.1.3, “Attribute Manager - Abstract” page 118

Methods

Although the “`Present`” and “`Dismiss`” methods described below are available to all derived media classes, they exhibit different behavior for different media types. For this reason, these methods are repeated in the more media specific abstract classes (i.e. `MMvisual`, `MMtemporal` and `MMaudioControl`). The “`Remove`” methods exhibit similar behavior across all classes, so they are only described here and will not be repeated elsewhere.

on Present

Presents all existing presentations for a media element using the current settings for each individual presentation request of the element. The entire element is presented. Activities triggered: Depends on the actual element’s `Present` activities.

on PresentId: handle *presentationID*

Presents the element’s specified presentation request. Activities triggered: Depends on the actual element’s `Present` activities.

on Dismiss

Dismisses all presentation requests for an element. A dismissed presentation is not removed nor is the loaded data destroyed. An element that is dismissed may be thought of as being off-stage ready to be presented again. A visual element’s `Dismiss` method hides the element. Activities triggered: Depends on the actual element’s activities.

on DismissId: handle *presentationID*

Dismisses the element’s specified presentation request. Activities triggered: Depends on the actual element’s activities.

on Remove

Removes all presentations of an element, which can not be presented again without creating new presentations. `Dismiss` is called on the element’s presentations before they are removed. Activities triggered: Depends on the actual element’s activities.

on RemoveId: handle *presentationID*

Removes the element’s specified presentation request. Activities triggered: Depends on the actual element’s activities.

Attributes

Attribute	Type	Description	Access
name	string	a user or application defined element name	CSG

Figure 6.32: MMbase Attributes

Activities

Activity	Keys	Description
DataReady	none	media data available
Present	none	element is presented
Dismiss	none	element is dismissed
Destroyed	none	element is destroyed

Figure 6.33: MMbase Activities

Example

None

6.3.2 MMvisual - *Abstract*

Any visual element type is derived from the `MMvisual` class.

Superclasses

Section 6.1.2, "Activity Manager - Abstract" page 116

Section 6.1.3, "Attribute Manager - Abstract" page 118

Methods

The following apply to all of the following methods:[1]

- **handle hVisual** -- a handle to the `XFvisual`
- **integer x, y** -- the location on the `XFvisual` where the image is presented
- **integer clipW** -- the clipping width of the presented image, -1 - don't clip
- **integer clipH** -- the clipping height of the presented image, -1 - don't clip
- **integer offsetX** -- the x offset from the origin of the source image
- **integer offsetY** -- the y offset from the origin of the source image

[1] The newer PresentID, RegisterID and RemoveID methods should be used instead of "on AddSink: handle *hXFvisual* return handle *presentationID*" (where image is placed at 0.0, no clipping) and "on RemoveSink: handle *hXFvisual*" (stops the XFvisual from displaying all presentations of the image, if clearMode is TRUE the image is cleared). While these older methods remain available to older applications, it is highly recommended that the newer and more powerful methods be used whenever possible.

on Present

Registers and automatically shows all existing presentations of a media element.

on PresentID: handle *presentationID*

`Present` registers and automatically shows the specified presentation.

on PresentOn: handle *hVisual* return handle presentationID

on PresentAt: handle *hVisual*, integer *x*, integer *y* return handle presentationID

on PresentClipped: handle *hVisual*, integer *x*, integer *y*, integer *clipW*, integer *clipH*,

integer *offsetX*, integer *offsetY* return handle presentationID

on Register

Registers all existing presentations of the media element but does not show the presentations.

on RegisterID: handle *presentationID*

Registers the specified presentation but does not show the presentation of specified handle.

on RegisterOn: handle *hVisual* return handle presentationID

on RegisterAt: handle *hVisual*, integer *x*, integer *y* return handle presentationID

on RegisterClipped: handle *hVisual*, integer *x*, integer *y*, integer *clipW*, integer *clipH*,

integer *offsetX*, integer *offsetY* return handle presentationID

on Show

Maps all of the elements existing presentations onto their specified `XFvisuals`.

on ShowID: handle *presentationID*

Maps the presentation onto the specified `XFvisual` at the location specified.

on Hide

Hides all presentations of the image, if `clearMode` is TRUE, the areas are cleared.

on HideID: handle *presentationID*

Unmaps the specified presentation, the `presentationID` is not removed, and the media element is not unloaded. `Show` methods may be called after a `Hide` method.

on Dismiss

The element stops presenting all presentations of itself, any screen updates are stopped and, if the element is temporal, no new frames are presented. If clearMode is TRUE the images are cleared. The `presentationIDs` are still valid.

on DismissID: handle *presentationID*

Screen updates are stopped and, if the element is temporal, no new frames are presented. If `clearMode` is TRUE the image is cleared. The `presentationID` is still valid.

Attributes

Attribute	Type	Description	Access
width	integer	width of image	G
height	integer	height of image	G
clearMode	boolean	clears presentation from the screen automatically if presentation is hidden or removed, default TRUE	CSG

Figure 6.34: MMvisual Attributes

Activities

Activity	Keys	Description
DataReady	*none*	After a load is complete

Figure 6.35: MMvisual Activities

Example

None

6.3.3 MMtemporal - *Abstract*

`MMtemporal` is a base class for any element that has a temporal or sequential nature. In most cases, positions within the data stream are addressable and the media data is presented as a series of positions. All wrapped classes that are derived from `MMtemporal` share the same methods, members and activities. For some of the derived classes the methods may have different results or no action, for example currently you may not change the playback rate of digital audio.

Superclasses

Section 6.1.2, “Activity Manager - Abstract” page 116

Section 6.1.3, “Attribute Manager - Abstract” page 118

Methods

Note that some temporal methods behavior depends on the type of the specification provided:

- **time** - Play from current position for the amount of time.
- **interval** - Play the range specified within the interval (See PlayInterval).

  If either side of the interval is open, use the sequence’s start or end position.
- **integer** - Play at the specified Positions per second.
- **real** - Play at the requested rate.

on Present

“`Play`”s all current presentations from start to end using the current settings. Activities triggered: `Play` and `RateChange`.

on PresentId: handle *presentationID*

on Play: any *specification* (only on UNIX)

Behavior of this method is controlled by the type of specification provided (see note above). Activities triggered: `Play` and `RateChange`, `PositionChange` if seek to new location.

on PlayId: handle *presentationID,* any *specification*

on PlayAll

on PlayAllId: handle *presentationID* (only on UNIX)

Play the entire media element at the default or requested rate. Activities triggered: `Play` and `RateChange`, `PositionChange` if seek to new location.

on PlayInterval: interval *range*

on PlayIntervalId: handle *presentationID,* interval *range* (only on UNIX)

Play the range specified within the interval, if either side of the interval is open use the sequence’s start or end position. Interval values may be either timestamps or positions and may be mixed. Please note many media types do not support playing backwards. Activities triggered: `Play` and `RateChange`, `PositionChange` if seek to new location.

on PlaySeq: integer *start,* integer *end*

on PlaySeqId: handle *presentationID,* integer *start,* integer *end* (only on UNIX)

Play the sequence of positions starting at “start” and stopping at “end” at the default or requested rate. Please note may media types do not support playing backwards. Activities triggered: `Play` and `RateChange`, `PositionChange` if seek to new location.

on PlayUntil: value *val* (only on UNIX)

on PlayUntil: value *val* return handle presentationID (only on UNIX)

on Pause

on PauseId: handle *presentationID* (only on UNIX)

If the element is being presented, it pauses. A paused media element may be resumed. Activities triggered: `RateChange`.

on Resume

on ResumeId: handle *presentationID* (only on UNIX)

If the element is paused, presentation continues from where it was paused. Activities triggered: `RateChange`.

on Stop

on StopId: handle *presentationID*

If the element is being presented or paused it is stopped, a stopped element may not be resumed. The element is not unloaded and may receive new presentation or play commands, but by default it starts at the beginning again. Activities triggered: `Stop` and `RateChange`.

on Seek: any *location*, boolean *present*

on SeekId: handle *presentationID*, any *location*, boolean *present* (only on UNIX)

Seeks to location specified by the value location. Location is relative to the beginning of the sequence. Location may be either an integer, which is used as a media specific frame or position, or a time value, which is used as the number of milliseconds into the sequence. If present is TRUE and element is a visual type present the frame. Activities triggered: PositionChange

on GoStart

on GoStartId: handle *presentationID* (only on UNIX)

Seeks to start of sequence. Visual elements display first frame. Triggered: PositionChange

on GoEnd

on GoEndId: handle *presentationID* (only on UNIX)

Seeks to the end of sequence. Visual elements display last frame. Triggered: PositionChange

on Jump: any *location*, boolean *present*

on JumpId: handle *presentationID*, any *location*, boolean *present* (only on UNIX)

Jumps to location (an offset of the number of images from the current position). If `present` is TRUE and the element is a visual type, present the frame. Triggered: PositionChange

on GoPPS: integer *positionsPerSecond*

on GoPPSId: handle *presentationID*, integer *positionsPerSecond* (only on UNIX)

Presents the media at the specified positions per second. Activities triggered: `RateChange`

on GoRate: real *rate*

on GoRateId: handle *presentationID*, real *rate* (only on UNIX)

Attempts to present the media at the specified `rate` based on default `PPS = 1.0`.
Activities triggered: `RateChange`

on ToPosition: time *duration* returns integer

Does not affect the media element but converts a `time` value into an element specific native position, or number of positions. Positions are calculated using the default `PPS`.

on ToTime: integer *position* returns time

Does not affect the media element but converts an integer `position`, into an element specific timestamp or duration. Time is calculated using the default `PPS`.

Attributes

Attribute	Type	Description	Access
startPosition	integer	start position relative to start of media data	CSG
endPosition	integer	end position relative to start of media	CSG
position	integer	current position relative to start of sequence	SG
timestamp	time	current position from start of sequence calculated using default data rate	SG
PPS	integer	current Positions per Second, native media frame	SG
rate	real	current rate calculated using default data rate 1.0 is the default rate.	SG
length	integer	length of the sequence in positions	CSG
duration	time	length of the sequence in time, calculated based on default rate	CSG
requestPPS	integer	request presentation rate, does not start the media. May have no effect	CGS
requestRate	real	request presentation rate as multiplier of default Positions Per Second (PPS)	CGS

Figure 6.36: MMtemporal Attributes

Activities

Activity	Keys[a]	Description
Play	*see footnote*	playing started
Start	*see footnote*	element starts, also on resume
Stop	*see footnote*	stopped, also on pause
RateChange	*see footnote*	on any rate change
StartChange	*see footnote*	after creation if the beginning location is changed
EndChange	*see footnote*	after creation if the end of sequence is changed
PositionChange	*see footnote*	seek, not called during normal movement
PositionReached		must be subscribed for prior to start
Periodic		sets timer, notify every clock ‘tick’
NextSample		more data available
LateSample		sample expected, is not ready
EndOfSequence		must be subscribed for before sequence starts
DroppedSample		element has skipped a sample

Figure 6.37: MMtemporal Activities

a. Note: The activity data returned from many temporal activities is a MDtemporalState value. Returned are: MDstatus status, integer position, integer PPS

6.3.4 MMaudioControl - *Abstract*

`MMaudioControl` deals with audio gain, channel selection, and input/output selection. All wrapped classes that are derived from `MMaudioControl` share the same methods, members and activities. For some of the derived classes, the methods have different results or no action.

Superclasses

Section 6.1.2, "Activity Manager - Abstract" page 116

Section 6.1.3, "Attribute Manager - Abstract" page 118

Methods

on Present

on PresentId: handle *presentationId*

Presents the entire sound element using the current output gain and device. Activities triggered: `Play` and `RateChange`.

on PresentSequence: integer *start*, integer *end*, real *volume* return handle presentationID

Creates, registers ad presents an audio presentation request. The `start` position, `end` position and volume are set to the specified lvalues. The `presentationID` is returned.

on PresentVolume: real *volume* return handle presentationID (only on UNIX)

Creates, registers and presents an audio presentation request. The volume is set to the specified level, the `start` and `end` positions default to beginning and end of data, and the `presentationID` is returned.

on RegisterSequence: integer *start*, integer *end*, real *volume* return handle presentationID

Creates and registers an audio presentation request. The `start` position, `end` position and volume are set to the specified lvalues. The `presentationID` is returned.

on RegisterVolume: real *volume* return handle presentationID (only on UNIX)

Creates and registers an audio presentation request. The volume is set to the specified level, the `start` and `end` positions default to beginning and end of data, and the `presentationID` is returned.

on Seek: any *location*, boolean *present*

Seeks to the location specified by the value `location`, the specified position is relative to the beginning of the sequence. `Location` may be either a integer which is used as a media specific frame or position, or a `UTtime` which is used as the number of milliseconds into the sequence. Note: An audio position is not presented upon `Seek` since a single sample is not useful. Activities triggered: `PositionChange`.

Attributes

Attribute	Type	Description	Access
numChannels	integer	number of channel currently supported	CSG
gain	real	sets output gain for all channels	CSG
gainA	real	sets output gain for channel A (left)	CSG
gainB	real	sets output gain for channel B (right)	CSG
level	real	level is the current signal value for all channels	G
levelA	real	level is the current signal value for channel A (left)	G
levelB	real	level is the current signal value for channel B (right)	G
output	string	specifies the local output device	S
record	boolean	record mode	CG
input	string	selects input device	CGS

Figure 6.38: MMaudioControl Attributes

Activities

Activity	Keys[a]	Description
AudioChanged	*none*	gain for any channel changed
AudioAChanged	*none*	gain for channel A changed (left)
AudioBChanged	*none*	gain for channel B changed (right)

Figure 6.39: MMaudioControl Activities

a. Note: The activity data returned from many audio activities is a MDaudioState value.
Returned are: MDstatus status, integer channels, real channelA, real channelB

Example

None

6.3.5 MMbroker

The multimedia broker acts as a media object factory. It isolates the *ADL* from having to declare and call a specific constructor for the actual data format of the requested media element. The *ADL* deals with abstract element types (such as image, audio, and movie) but the actual object created by the wrapper must be the fully derived media element class that supports the specific data type (`MEgif`, `MEtiff`, `MEjpeg`, etc.). The `MMbroker`, if given a generic element type, tests the media data to determine the actual C++ class that must be constructed. Since the broker may construct any media type, specification of the media element type may be supplied by external or run-time data. The `MMbroker` parses media object descriptor lists. These lists define media elements and the element's internal objects. The format of an object descriptor list is as follows:

- The first item in the list must be a string that is the name of the requested C++ media class. A base class name may be used if it is possible to derive the actual class of the media element from the remaining arguments, or the media element's data.
- Additional items in the list may include object descriptor lists for objects which are used as members of the requested object, or an argument list.

An argument list has the following format:

- The first item must be the string "`arguments`"
- Additional items in the list are name/value lists with the first item an attribute name, and the second item the value. Currently the individual name/value lists only have two items.

File access may be specified through an `MAfile` descriptor list, `MAfile` descriptor lists follow the same format as any other object descriptor list, but in addition support a short list form:

`{"MAfile","pathname"}`	The short form must be a list of two strings.

Superclasses

Section 6.1.2, "Activity Manager - Abstract" page 116

Section 6.1.3, "Attribute Manager - Abstract" page 118

Methods

on MakeElement: list *mediaDescription* return handle

The "*mediaDescription*" is a list which matches the format specified above. A media element is created and a handle to it is returned.

on LoadColorNames: string *filePath*

Loads colors defined within the file as the color database colors. Deletes all colors currently in the default color database. See `/usr/lib/rgb.txt` for file format.

on AppendColorNames: string *filePath*

Appends colors defined in the file to the color database. All colors currently in the default color database remain. Multiple entries may have the same name and or color values, searches of the database usually return the first 'match' found. See /usr/lib/rgb.txt for file format.

Attributes

None

Activities

None

Example

None

6.3.6 MMcolor

MMcolor allows a user to define and control either `RGB` or `HVS` colors.

Superclasses

None

Methods

on construct

Default constructor. Name is ‘`none`’, `color` is all zeros (Black).

on createName: string *colorname*

Creates a color name. If the default color database contains the name of the `RGB,` values are set to match the name’s values, else they are set to zero (Black).

on createRGB: integer *green*, integer *red*, integer *blue*

Creates a `color` of the specified `RGB` settings. If the default color database contains a color with matching RGB values, `name` is set to match the found color name, else is set to ‘`none`’.

on createHVS: real *hue*, real *value*, real *saturation*

Creates a color of the specified `HVS` settings. If the default color database contains a color with matching `RGB` values `name` is set to match the found color `name`, else is set to ‘`none`’.

on createNamedRGB: string *colorname*, integer *green*, integer *red*, integer *blue*

Creates a color of the specified `name` with the specified `RGB` settings. Color database is not checked.

on createNamedHVS: string *colorname*, real *hue*, real *value*, real *saturation*

Creates a color of the specified name with the specified `HVS` settings.

Attributes

Attribute	Type	Description	Access
color	handle	A representation of the packed RGB values in a 32 bit integer	CSG
red	integer	This sets the level of red in a RGB color specification.	CSG
green	integer	This sets the level of green in a RGB color specification.	CSG
blue	integer	This sets the level of blue in a RGB color specification.	CSG
hue	real	This sets the hue in an HVS color specification. Range is 0.0 to 360.0	CSG
value	real	This sets the brightness in an HVS color specification. Range is 0.0 to 100.0	CSG
saturation	real	This sets the saturation in an HVS color specification. Range is 0.0 to 100.0	CSG
name	string	Specifies a user or applicaiton defined color.	CSG

Figure 6.40: MMcolor Attributes

Activities

None

Example

None

6.3.7 MMimage

Any type of image that is not a sequence of images that may be controlled, or indexed. Currently supported image classes; `MEpbm`, `MEgif`, `MEtiff`, `MEjpeg`, `MExbm` and `MEphotoCD` only on the Sun platform. In addition a special image class MEvideo supports a live analog video stream. The `MEvideo` class is supported by `MMimage` since the only controls for the video stream are its placement and visible state.

Superclasses

Section 6.3.1, "MMbase - Abstract" page 173

Section 6.3.2, "MMvisual - Abstract" page 174

Methods

upon Construct: list *description*

Default constructor. Requires access descriptor to image. Description should be of the form `{<type>, {'MAfile, <filename>}}`, where `<type>` is the type of the image (one of:

‘MEgif, ‘`MEjpeg`, ‘`MEpbm`, ‘`MEphotoCD`, ‘`MEtiff`, ‘`MExbm`, or ‘`MEimage` to auto-detect), and `<filename>` is the name of the image file. An example of a constructor follows:

```
'Construct, { ‘MEimage, { ‘MAfile, "bigcat.gif"}}
```

on List: returns list

Returns an object descriptor list, which may be used to re-create the object. The returned list is formatted to be used as the list argument to the `MMbroker` to `MakeElement` or to the class’s Construct method which has a list argument.

on Load: return integer

Loads the image into memory. Returns zero on success.

on Unload: return boolean

Unloads any media data, closes the elements data file, stops the refreshing of the image and returns TRUE when complete. The media element is not deleted or destroyed so it is possible to redisplay the image with another present command. Note: all existing presentations are removed and made invalid. If you plan on hiding and showing the image this is not the method to call since the media data will have to be reloaded and a new presentation requested.

on SetSize: integer *width*, integer *height*

Set the image to a new size.

on Zoom: real *scaleX*, real *scaleY*

`Zoom` changes the size of all presentations created from this element, currently the image must be Zoomed before the image is presented. To present the same image with different scale factors, another instance of the MMimage is required. This will change in the future. If a clip region is not specified, the presented image is the size of the image after the `Zoom`.

Attributes

Attribute	Type	Description	Access
width	integer	width of image	G
height	integer	height of image	G
clearMode	boolean	clear on remove sink automatically, default false	CSG

Figure 6.41: MMimage Attributes

Activities

Activity	Keys	Description
DataReady	*none*	after a load is complete

Figure 6.42: MMimage Activities

Example

```
global assets
{
  {'SetLibrary, "Examples", "/usr/lib/Examples"} => self;
}
anonymous: XFtop
{
 XFvisual visual { x=10; y=10;  width = 640; height = 480;
                   background="black";
                 };
 XFbutton bimg1  { x=200; y=500; width= 100; height = 30;
                  label="Image 1";
                  background="white"; foreground="black";
                 };
 XFbutton bimg2  { x=350; y=500; width= 100; height = 30;
                  label="Image 2";
                  background="white"; foreground="black";
                 };
 XFbutton bQuit { x=620; y=500; width= 50; height = 30;
                background="white"; foreground="black"; label="Quit";
                fontRequest={"Helvetica", 12,{"bold","italic"}, "roman"};
                };

  handle  hImage;
  upon Construct
    {
      bimg1.Pressed  = { 'Img1, self};
      bimg2.Pressed  = { 'Img2, self};
      bQuit.Pressed  = { 'Quit, self};
      hImage = NULL;
    }
   on Img1
    {
        if (hImage != NULL) { delete hImage; }
        hImage = new {'Construct, { "MEimage",
                        { "MAfile", "bigcat.gif"@"Examples" }}} =>
MMimage;
        {"PresentAt", &visual, 0, 0} => hImage;
    }
   on Img2
    {
        if (hImage != NULL) { delete hImage; }
        hImage = new {'Construct, { "MEimage",
                        { "MAfile", "dragon.gif"@"Examples" }}}
=> MMimage;
        {"PresentAt", &visual, 0, 0} => hImage;
    }
   on Quit
    {
        delete hImage;
        'Exit => theApp;
    }
}top {x=20; y=20; width=680; height=540; background='SlateBlue;};
```

6.3.8 MMdigitalAudio

Any type of digital audio, supported file formats include WAV, Sun au, and .SND. `MMdigitalAudio` is derived from the abstract classes `MMbase`, `MMtemporal`, and `MMaudioControl`. All base class methods and activities are supported.

Superclasses

Section 6.3.1, "MMbase - Abstract" page 173

Section 6.3.3, "MMtemporal - Abstract" page 176

Section 6.3.4, "MMaudioControl - Abstract" page 180

Methods

upon Construct: list *description*

Default constructor; Description list should follow this format where `<filename>` is the name of the data file:

```
'Construct {'MEdigitalAudio, {'MAfile, 'filename},
      {'arguments,{'gainA, 1.0}, {'gainB,0.0}}}
```

on List: returns list

Returns an object descriptor list, which may be used to re-create the object. The returned list is formatted to be used as the list argument to the `MMbroker` to `MakeElement` or to the class's `Construct` method which has a list argument.

on Load: return boolean

Loads the media data and confirms that audio services are connected. Activities triggered: `DataReady`.

Attributes

Attribute	Type	Description	Access
name	string	symbolic name associated with the element	CGS
source	string	access path, must be set prior to loading	CGS
masterGain	real	sets system output gain for all channels	GS
fileFormat (only on UNIX)	string	returns the current file format when recording is supported it may be set	CGS
output	string	specifies the local output device	S
record	boolean	record mode	CG
input	string	selects input device	CGS

Figure 6.43: MMdigitalAudio Attributes

Activities

None

Example

```
global assets
{
  {'SetLibrary, "Examples", "/usr/lib/Examples"} => self;
}

anonymous: XFtop
{
  XFbutton bimg1 { x=10; y=10;
                    width= 100; height = 30;
                    label="Sound 1";
                    background="black"; foreground="white";
                  };

  XFbutton bimg2 {  x=130; y=10;
                    width= 100; height = 30;
                    label="Sound 2";
                    background="black"; foreground="white";
                 };

  XFbutton bQuit { x=300; y=10;
                   width= 50; height = 30;
                   background="black"; foreground="white";
                   label="Quit";
                   fontRequest = {"Helvetica", 12, {"bold",
                     "italic"}, "roman"};
                 };

  handle  hAudio;

  upon Construct
    {
      bimg1.Pressed = { 'Sound1, self};
      bimg2.Pressed = { 'Sound2, self};
      bQuit.Pressed = { 'Quit, self};
      hAudio = NULL;
    }

    on Sound1
    {
        if (hAudio != NULL) { delete hAudio; }
        hAudio = new {'Construct, {'MEdigitalAudio,
                          { 'MAfile,
"Violin_Concerto_in_E_Major.au"@"audioDir"},
                          { 'arguments, {'gainA, 1.0}}
                     }} => MMdigitalAudio;
        {'PresentVolume, 1.0} => hAudio;
    }
    on Sound2
    {
        if (hAudio != NULL) { delete hAudio; }
        hAudio = new {'Construct, {'MEdigitalAudio,
                          { 'MAfile, "wolf-2.au"@"audioDir"},
                          { 'arguments, {'gainA, 1.0}}
                     }} => MMdigitalAudio ;
        {'PresentVolume, 1.0} => hAudio;
    }
   on Quit
    {
        delete hAudio;
        'Exit => theApp;
    }
}top {x=20; y=20; width=360; height=50; background='SlateBlue;};
```

6.3.9 AVwaveForm

`AVwaveForm` is a specialized class which visually represents a sound wave form, which may be presented as an image.

Superclasses

Section 6.3.1, “MMbase - Abstract” page 173

Section 6.3.2, “MMvisual - Abstract” page 174

Section 6.3.3, “MMtemporal - Abstract” page 176

Section 6.3.4, “MMaudioControl - Abstract” page 180

Section 6.3.7, “MMimage” page 184

Section 6.3.8, “MMdigitalAudio” page 187

Methods

on SetRecord: handle *hAudioPresID*, integer *rate*, integer *mseconds* return handle

Initializes the audio stream.The audio `PresentationId` defines the audio stream to be recorded. The frequency is set. Returns an audio `presentationID`.

on Process: handle *hAudioPresID*

Begins the recording process. Audio stream, duration and frequency are set within the `presentation ID`.

on MakeFreqImage: handle *hAudioPresID*, integer *width*, integer *height* return handle

If the `Process` method has not been called it is started, after `Process` complets the recorded audio is converted into an image.Sound represented as a frequency graphic of given width and height. Handle returned is a `MMimage`.

on MakeWaveImage: handle *hAudioPresID*, integer *width*, integer *height* return handle

If the `Process` method has not been called it is started, after Process complets the recorded audio is converted into an image. Sound represented as an amplitude wave of given `width` and `height`. Handle returned is an `MMimage`.

on BlankProcess: handle *hAudioPresID*

Calls `Process`, but no image is produced.

Attributes

Attribute	Type	Description	Access
numberToSum	integer	default value	CSG
xJump	integer	sets the height of wave to be represented visually	CSG
freqJump	integer	sets the width of wave to be represented visually	CSG
minConsecutive	integer	sets the minimum consecutive points to be represented as a smooth curve	CSG
tooClose	integer	x value of height too small to be registered visually	CSG
yTooClose	integer	y value of width too small to be registered visually	CSG
minVol	integer	the minimum volume of sound to be registered visually	CSG

Figure 6.44: AVwaveForm Attributes

Activities

None

Example

None

6.3.10 MMmovie

`MMmovie` provides the interface to all animated visual elements. If a movie element does not support audio, the audio methods perform no functions.

Superclasses

Section 6.3.1, "MMbase - Abstract" page 173

Section 6.3.2, "MMvisual - Abstract" page 174

Section 6.3.3, "MMtemporal - Abstract" page 176

Section 6.3.4, "MMaudioControl - Abstract" page 180

Section 6.3.7, "MMimage" page 184

Section 6.3.8, "MMdigitalAudio" page 187

Methods

upon Construct: list *description*

Default constructor. Requires access descriptor to image. Description should be of the form `{<type>, {‘MAfile, <filename>}}`, where `<type>` is the type of the movie (one of: ‘MEmpeg, ‘`MEavi`, ‘`MEjpeg`, ‘`MEfli`, ‘`MEqt` or ‘`MEmovie` to auto-detect), and `<filename>` is the name of the image file. The following is an example of a construction for an MMmovie:

```
'Construct, {‘MEanim, {‘arguments, {‘gain, 0.3}}, {‘MAfile, "claylzrd.avi"}}
```

on List: return list

Returns an object descriptor list, which may be used to re-create the object. The returned list is formatted to be used as the list argument to the `MMbroker` to `MakeElement` or to the class's `Construct` method which has a list argument.

on Load: return integer

Loads the image into memory; returns zero on success.

Attributes

None

Activities

None

Example

```
global assets
{
  {'SetLibrary, "Examples", "/usr/lib/Examples"} => self;
}
anonymous: XFtop
```

```

{
  XFvisual visual { x=10; y=10;
                    width = 640; height = 480;
                    background="black";
                  };

  XFbutton bStart { x=200; y=500;
                    width= 50; height = 30;
                    label="Start";
                    background="white"; foreground="black";
                  };

   XFbutton bQuit { x=620; y=500;
              width= 50; height = 30;
              background="white"; foreground="black";
              label="Quit";
             fontRequest = {"Helvetica", 12, {"bold","italic"}, "roman"};
                 };

  handle  hMovie;

  upon Construct
    {
      bStart.Pressed = { 'Start, self};
      bQuit.Pressed  = { 'Quit, self};
      hMovie = NULL;
    }

   on Init
    {
  hMovie = new {'Construct, { "MEanim", {"arguments", {"gain", 0.3}},
               { "MAfile", "claylzrd.avi"@"movieDir" }}} =>
MMmovie;
    }

   on Start
    {
       {"PresentAt", &visual, 0, 0} => hMovie;
    }

   on Quit
    {
        delete hMovie;
        'Exit => theApp;
    }
}top {x=20; y=20; width=680; height=540; background='SlateBlue;};
```

6.3.11 MMvidDiscPlayer (only on UNIX)

`MMvidDiscPlayer` is a class that controls a generic video disc device. While some of its methods are similar to those of `MMmovie`, the class is not a media element but a device controller.

The media class model expects that devices are managed by the media elements requesting their services. The player was made available to *ADL* mainly for testing and to allow run time specification of configuration information via the asset manager.

Note: The video disc player does not display the video, but only makes the analog signal available. The media element `MEvideo` provides an interface to show a 'live video stream' the `MEvideo` class is represented in the *ADL* as a type of `MMimage` since there is no control of the video stream but only its placement and mapping.

Superclasses

Section 6.3.1, "MMbase - Abstract" page 173

Section 6.3.2, "MMvisual - Abstract" page 174

Section 6.3.4, "MMaudioControl - Abstract" page 180

Methods

upon Construct: list *objectdescriptor*

Constructs the `MMvidDiscPlayer` object from an object descriptor list. For the video player, this requires device type information, serial line specification and configuration, and an optional video disc specification.

on List: returns list

Returns an video disc object descriptor list, which may be used to re-create the object. The returned list is formatted to be used as the list argument to the `MMbroker` to `MakeElement` or to the class's `Construct` method which has a list argument. The following is an example:

```
list objectList =
{‘MEvidDiscSeq,
  {‘arguments,{‘start, 12345},{‘end, 14500},{‘gainA,1.0},{‘gainB,0.0}},
  {‘MAvidDiscVol, {‘arguments,{‘volName,’Louvre_2}}}
};
```

on Load

If a disc is in the drive, it spins up the drive and positions the head to frame 1.

on Unload

Spins down the drive and, if supported, ejects the video disc.

on Mount

Unmounts any current disc, if it is not busy, loads the current disc, and updates the videoDiscAgents volume list.

on Unmount

Removes the disc from the videoDiscAgent’s volume table.

on PlaySeq: integer *position1*, integer *position2*, real *rate*

Plays the disc from position1 to position2, at a frame rate calculated based on the default rate (30 PPS). Depending on the state of video, audio1 & audio2 each analog signal may be output.

on Pause: time *duration*

Stops the disc and ignores duration.

on Scan: integer *speed*

on Seek: integer *offset*, integer *relativeTo*

on Step: integer *size*

on Stop

Attributes

Attribute	Type	Description	Access
position	any	Set: integer, real or UTtime Get: integer	SG
PPS	integer	current Positions per Second, native media frame	SG
rate	real		SG
video	boolean	video output enabled	CSG
audio1	boolean	audio 1 output on/off	CSG
audio2	boolean	audio 2 output on/off	CSG
index	boolean	frame index displayed on video	CSG

Figure 6.45: MMvidDiscPlayer Attributes

Activities

Note: These activities are normally only reported to the media element which may broadcast a notify event.

Activity	Keys	Description
DataReady		loaded
RateChange		disc current speedchanged
Start		special rate change
Stop		special rate change
AudioAChanged		channel’s state changed
AudioBChanged		channel’s state changed

Figure 6.46: MMvidDiscPlayer Activities

Example

None

6.3.12 MMhtml

An `HTML` media object.

Superclasses

Section 6.3.1, “MMbase - Abstract” page 173

Methods

upon Construct: list *description*

Default constructor. Description is of the form `{‘MEhtml, {‘MAfile, <filename>}}` or `{‘MEhtml, {‘URL, <url>}}`, where `<filename>` is the name of the `HTML` file and `<url>` is the `URL` address of the `HTML` document.

upon ConstructFromString: string *text* (only on UNIX)

Constructs an `HTML` object using the given text.

upon ConstructFromStream: handle *hStream* (only on UNIX)

Constructs an `HTML` object using the data extracted from `hStream`. As of the current release, `hStream` must be a handle to an instance of an `IOwebStream` subclass.

on Load: return integer

Loads the `HTML` document into memory; returns zero on success.

on Unload: return boolean

Unloads the `HTML` document from memory; returns `TRUE` on success.

on AddSink: handle *hSink*

Adds a sink on which the HTML document should be displayed; `hSink` should be a handle to an `XFhtml` or a subclass thereof.

on RemoveSink: handle *hSink*

Stops displaying the `HTML` document on a sink.

on Show

Displays the HTML document on whichever sinks have been added using `AddSink`.

on Hide

Stops displaying the HTML document on all sinks.

on GoToAnchor: string *anchor_name* return boolean

Displays the part of the HTML document referenced by `anchor_name` at the top of the HTML display surface; returns TRUE on success.

on GetHTMLSrc: return string

Returns the HTML document as an HTML formatted string. Please note that the HTML document must be loaded before this method is called, otherwise an empty string is returned.

on GetURL: return string

Returns the URL address of this HTML document. If this HTML document contains a BASE tag with an HREF attribute value, this method returns the URL contained in the BASE tag.

on GetTitle: return string

Returns the title for this HTML document; if there is no title, this method returns an empty string.

Attributes

None

Activities

None

Example

```
uses “nro.adl”@”StdLib”;
anonymous: XFtop
{
  handle hdoc = NULL;
  XFhtml  hyper
    {
       width = 700; height=650;
       borderColor = “SlateGray”;
       borderWidth = 2;
     }
  upon Construct
  {
     handle h = &hyper;
     width = 700;
     height = 650;
     background = “gray”;
     {‘Set_Current, “http:///test-map.html”} => self;
  }
  on Init
  {
     {‘Subscribe, new {‘Create, ‘AnchorPressed, self,
            ‘AnchorSelected, {}} => nro} => hyper;
     {‘Subscribe, new {‘Create, ‘ImageMapPressed, self,
            ‘MapSelected, {}} => nro} => hyper;
  }
  //-------------------------------------------------------------
  //Given a new href, get the corresponding html text from
  //the array, and display it in the XFhtml
  //-------------------------------------------------------------
  on Set_Current: string newCurrent
  {
    if (! (hdoc == NULL))
     {
       delete hdoc;
     }
    hdoc=new {‘Construct, {‘MEhtml, {‘URL, newCurrent}}} => MMhtml;
    {‘AddSink, &hyper} => hdoc;
    ‘Show => hdoc;
  }
  //------------------------------------------------------------
  //AnchorPressed  activity handler
  //-------------------------------------------------------------
  on AnchorSelected: list clientData, list keys, list values
  {
    nro mynro;
    string hRef = {‘Lookup, ‘href, keys, values} => mynro;
    string refText = {‘Lookup, ‘text, keys, values} => mynro;
    string anchor_name={‘Lookup,‘anchor_name,keys,values}=> mynro;

    echo (“\nAnchorPressed Activity invoked!!!\n”);
    echo (“anchor_name is “ + anchor_name + “\n”);
    echo (“href is “ + hRef + “\n”);
    echo (“reftext is “ + refText + “\n”);
    if (anchor_name == “quit”)
   { echo (“Good-Bye.\n”);
     ‘Exit => theApp;
        }
  }
  //-------------------------------------------------------------
  //ImageMapPressed activity handler
  //-------------------------------------------------------------
  on MapSelected: list clientData, list keys, list values
  {
     nro mynro;
     string image_src={‘Lookup, ‘image_src, keys, values }=> mynro;
     string anchor_name={‘Lookup,‘anchor_name,keys,values}=> mynro;
     string hRef = {‘Lookup, ‘href, keys, values} => mynro;
     string refText = {‘Lookup, ‘text, keys, values} => mynro;
     integer x =  {‘Lookup, ‘x, keys, values } => mynro;
     integer y =  {‘Lookup, ‘y, keys, values } => mynro;
     echo (“\nImageMapPressed activity invoked!!!\n”);
     echo (“image_src is “ + image_src + “\n”);
     echo (“x is “ + toString (x) + “\n”);
     echo (“y is “ + toString (y) + “\n”);
     echo (“anchor_name is “ + anchor_name + “\n”);
     echo (“href is “ + hRef + “\n”);
     echo (“reftext is “ + refText + “\n”);
  }
} top
```

6.4 Input/Output

The purpose of the `Input/Output (IO)` wrapped classes are to allow different operations that include file access, network access (`ftp` and `http`) and supports different notification of IOstreams. Documentation for the following classes appear in this section:

The class inheritance tree diagram for the `Input/Output (IO)` wrapped classes of *AM2* appears in Figure 6.47.

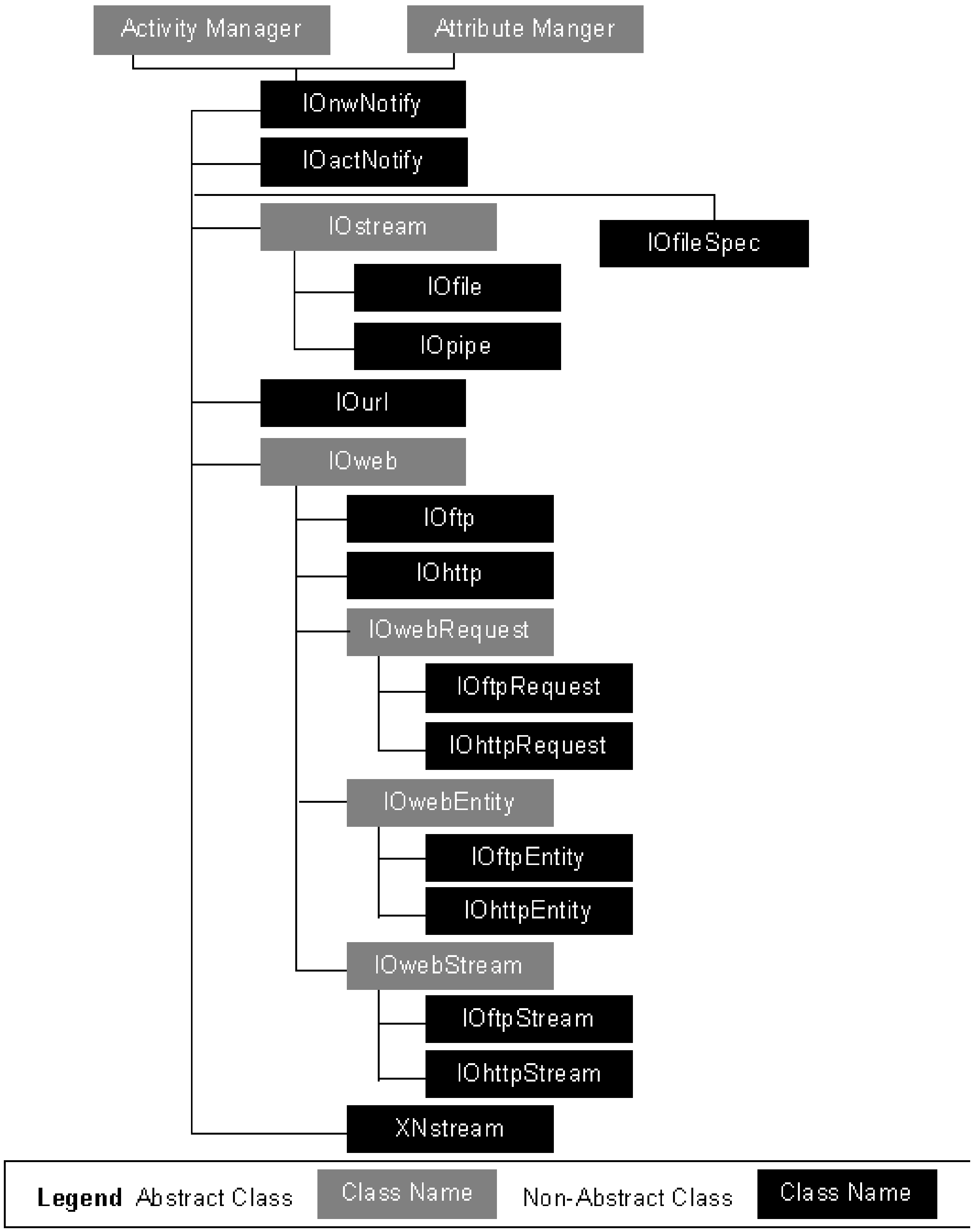

Figure 6.47: Input/Output Wrapped Classes Inheritance Tree

6.4.1 IOactNotify

This class allows notification on file stream events. It is used in combination with the `IOfile` object. There must be one instance of this class for each activity.

Superclasses

Section 6.1.2, “Activity Manager - Abstract” page 116

Section 6.1.3, “Attribute Manager - Abstract” page 118

Methods

upon *CreateFromStream*: handle *hStream*

Creates an instance to monitor the specified stream (`hStream` should be a handle to an instance of class `IOfile`).

on IsValidActivity: string *actName* return boolean

Returns TRUE if actName is a valid activity, FALSE otherwise.

Attributes

None

Activities

Activity	Keys	Description
ReadReady	*none*	data is ready for reading from port
WriteReady	*none*	OK to send data to port
ExceptionReady	*none*	exception encountered

Figure 6.48: IOactNotify Activities

Example

None

6.4.2 IOnwNotify

This class allows notification on network stream events. It is used with a network stream object, such as `XNstream`. There needs to be one instance of this class for each activity.

Superclasses

Section 6.1.2, “Activity Manager - Abstract” page 116

Section 6.1.3, “Attribute Manager - Abstract” page 118

Methods

upon CreateFromPort: integer *port*

Creates an instance to monitor the specified network port. This constructor is used when you first waits for connection request from other *AM2* application without blocking. This constructor is used in combination with `ConnectReady` activity. You cannot use this constructor for `ReadReady` activity.

upon CreateFromStream: handle *hStream*

Creates an instance to monitor the specified stream (`hStream` should be a handle to an instance of class `XNstream`). This constructor is used together with `ReadReady` activity.

on IsValidActivity: string *actName* return boolean

Returns TRUE if `actName` is a valid activity, FALSE otherwise.

on AcceptXN: return handle

Like Accept, but returns a handle to an instance of class `XNstream` instead.

Attributes

None

Activities

Activity	Keys	Description
ConnectReady	*none*	connection requested
ReadReady	*none*	data is ready for reading from port (after connection established)
WriteReady	*none*	OK to send data to port (after connection established)
ExceptionReady	*none*	exception encountered

Figure 6.49: IOnwNotify Activities

Example

None

6.4.3 IOstream - *Abstract*

This abstract class serves as a foundation class which contributes to the functionality of the subclasses `IOfile` and `IOpipe`.

Superclasses

Section 6.1.2, “Activity Manager - Abstract” page 116

Section 6.1.3, “Attribute Manager - Abstract” page 118

Methods

on Good: return boolean

Returns true if the file is open and ready to be accessed. This returns false if the file was not opened successfully, has been closed, or an error has occurred.

on Bad: return boolean

Returns true if the object does not contain a valid open file reference.

on Fail: return boolean

Returns true if the previous action was not successful. If this returns true then all operations fail until `Clear` is called.

on Ready: return boolean

Returns true if an input or output operation (as appropriate for the type of file initially opened) does not block.

on Eof: return boolean

Returns true if the file was opened in an input mode and the end of the file has been reached.

on SendBoolean: boolean *val*

on SendInteger: integer *val*

on SendReal: real *val*

on SendString: string *val*

on SendList: list *val*

on SendInterval: interval *val*

on SendTime: time *val*

on SendAny: any *val*

Writes specified value to the stream. These methods also accessable via *ADL* << operator.

on ReceiveBoolean: return boolean

on ReceiveInteger: return integer

on ReceiveReal: return real

on ReceiveString: return string

on ReceiveList: return list

on ReceiveInterval: return interval

on ReceiveTime: return time

on ReceiveAny: return any

Reads a value of specified type from file. These methods also accessable via *ADL*>>operator.

on Oct /on Dec / on Hex

Sets output mode for numbers to the specified radix (octal, decimal, or hexadecimal, respectively).

on Endl

Writes an end-of-line character to the file.

on Flush

Flushes the output stream. This has no effect on read only files.

on Text

Sets the file to text mode. In this mode, all data is converted to text before being written and converted from text when read. If conversion cannot be performed (i.e., the following data in the file is not of the correct format for the type being read), the operation fails.

on Binary

Sets the file to binary mode. In this mode, all data is output in binary form, and input is assumed to be in binary form. This format is not recommended, as it is not portable. Sending and receiving lists and intervals in binary mode fails, as does ReceiveAny.

on Tagged

Sets the file to tagged mode. In this mode, data is written as in binary mode, but preceded by a single byte tag which determines the format of the data. Strings are proceeded by an additional length tag. Input data is assumed to be in tagged format. Any discrepancy between expected and received data is reported as an error.

on Word

On subsequent string input, whitespace is used as delimiters, and non-whitespace character sequences returned.

on Line

String output is followed by a platform dependent end of line sequence, and string input returns all characters until the next end of line, and discard the end of line sequence.

on NoDelim

String input returns all available text as a single string.

on WordDelim: string *delim*

Sets the whitespace characters for Word mode to the characters in delim.

on LineDelim: list *delimseqs*

Sets the end of line sequences for Line mode to the strings in the list given. The first string in the list is used as the line terminator in Line mode. The list must contain only strings.

Attributes

None

Activities

None

Example

None

6.4.4 IOfile

This class provides file input and output. It may be in either native or portable mode, written as `text`, `binary`, or `tagged` data.

Superclasses

Section 6.4.3, "IOstream - Abstract" page 202

Methods

upon OpenConstruct: string *fName*, string *mode*

Opens the file `fName`. All text written out is converted to a portable format (Unicode); binary data is not changed. The mode argument must be one of '`ReadOnly`, '`ReadWrite`, '`WriteTrunc`, and '`WriteAppend`.

upon OpenNativeConstruct: string *fName*, string *mode*

Opens the file fName. All text written out is in the machine's native encoding format. The mode argument must be one of 'ReadOnly, 'ReadWrite, 'WriteTrunc, and 'WriteAppend.

on Open: string *fName*, string *mode*

on OpenNative: string *fName*, string *mode*

Same as OpenConstruct and OpenNativeConstruct, except not constructors. To be used to open a different file once the initial one has been closed.

on Close

Flushes and closes the file. Any further reads or writes fail until another file is opened with Open or OpenNative.

Attributes

None

Activities

None

Example

```
anonymous : XFtop
{
   IOfile file_stream;
   string line;
   upon Construct
   {
      {‘OpenNative, “temp.txt”, ‘ReadOnly } => file_stream;
         if ( ‘Good => file_stream)
            {
            ‘Line => file_stream;
            while (! ‘Eof => file_stream )
               {
               line = ‘ReceiveString => file_stream;
               echo(line + “\n\n”);
               }
            ‘Close => file_stream;
            }
      else {echo(“Invalid File \n”);
      }
   }
} top;
```

6.4.5 IOfileSpec

This is a wrapped class for file name specification. Each *AM2*-supported platform specifies files in a different way: the syntax for using pathnames is different, or discouraged as on the Macintosh platform. IOfileSpec isolates these problems in one class and, to some extent, provides automatic conversion between different ways of specifying a file. There is no requirement that the specified file actually exist. IOfileSpec also performs some operations on closed files, like DeleteFile.

Superclasses

None

Methods

upon CreateFileSpec: string *fName*

Constructor with a file name, or a full or partial path.

on DeleteFile: return integer

Deletes the specified file.

on FileExists: return boolean

Returns TRUE if a file of this name exists, FALSE otherwise.

on GetName: return string

Returns the file name without a path.

on GetNameLength: return integer

Returns the length of the name that is being used in the path specification. If a path is used, the length of the path is included in the returned result. The result is not only platform-dependent, but also depends on how the file was specified to the constructor.

on GetPath: return string

Returns the file name together with the path with which the file was specified. On platforms that do not use pathnames internally, a full path name is created for this function and returned.

Attributes

None

Activities

None

Example

```
anonymous : XFtop
{
   handle file_object;
   string filename;
   string path;

upon Construct
   {
      filename = “temp.txt”;

      file_object = new { ‘CreateFileSpec, filename } =>
                        IOfileSpec;

      if (‘FileExists => file_object )
         {path = ‘GetPath => file_object;
          echo(“This is the path: “ + path + “ \n\n”); }

      else
         {echo(“Filename: “ + filename + “ does not exist.\n\n”); }
   }
} top;
```

6.4.6 IOpipe

This class represents an input stream from and an output stream to an external process. The interface is the same as that of the IOfile class (described on page 205) except that there are no public constructors and the Close method is as noted here.

Superclasses

Section 6.4.3, “IOstream - Abstract” page 202

Methods

On Close: return integer

Closes the pipe and waits for the external process to finish.

Note that this call can block indefinitely. Returns 0 if the process terminated normally, and a machine dependent non-zero value otherwise.

Attributes

None

Activities

None

Example

None

6.4.7 IOurl

This class provides a Uniform Resource Locator (URL) object for parsing and constructing URL strings formatted for the World-Wide Web.

Superclasses

None

Methods

upon Construct

Default constructor.

on ExtractAccessMethod: string *url* return string

Returns the access method in the url string. Example

access methods are http, ftp, etc.

on ExtractAnchorName: string *url* return string

Returns the anchor name in the url string. According to the URL format, the anchor name is marked by "#" in a URL string.

on IsAnchorReference: string *url* return boolean

Returns TRUE if this url string matches the format of a relative URL which references an internal anchor within an HTML document. According to the current URL format, an anchor reference URL begins with the "#" mark. This method does not verify the validity of the anchor reference.

on MakeAbsolute: string *partial_url,* string *base_url* return string

Expands partial_url into its full form in the context of base_url. Returns the expanded URL. The internal anchor in partial_url is kept in the returned full form. However, the anchor in base_url is not.

on Escape: string *str* return string

Returns the escaped str from unacceptable characters using%.

on UnEscape: string *str* return string

Returns the unescaped str which was previously escaped using%.

on ExtractHost: string *url* return string

Returns the hostname extracted from the url.

on ExtractPort: string *url* return integer

Returns the port number extracted from the URL. Returns -1 if no port was found. (Currently this method returns -1 if a default port was found in the URL).

on ExtractPath: string *url* return string

Returns the path string extracted from the url.

Attributes

None

Activities

None

Example

Given a Uniform Resource Locator, IOURL provides methods to extract different components in the URL string.

```
string url1;
string url2;
string base_url;
string canon_url;
handle hURL;
string str;
integer port;

upon Construct
```

```
      {
       //--------------------------------------
       //Construct an IOurl
       //--------------------------------------
        hURL = new ‘Construct => IOurl;
       //--------------------------------------
       //child_url is a relative URL to be interpolated
       //using the base_url
       //--------------------------------------
       url1 = “cgi-bin/search?name=Dole department=CE”;
       url2 = “ sipb/sipb.html#Students “;
       base_url=“http://www.mit.edu/mit-activities.html#Computers “;

       url2 = {‘StripWhiteSpace, url2 } => hURL;
       base_url = {‘StripWhiteSpace, base_url } => hURL;

       echo (“\nPartial URL 1 is   :  “ + url1 + “\n”);
       echo (“\nPartial URL 2 is   :  “ + url2 + “\n”);
       echo (“Base URL is  :  “ + base_url + “\n”); echo (“\n”);
        //--------------------------------------
        //Given a parent url, expand the partial url to an
        //absolute url using  a base url.
        //--------------------------------------
        canon_url = {‘MakeAbsolute, url1, base_url} => hURL;
        echo (“Expanded URL1 is : “ + canon_url + “\n”);

        canon_url = {‘Escape, canon_url} => hURL;
        echo (“Escaped URL1 is “ + canon_url + “\n”);

        canon_url = {‘MakeAbsolute, url2, base_url} => hURL;
        echo (“Expanded URL2 is : “ + canon_url + “\n”);

        echo (“Expanded URL2 has the following Components: \n”);
        echo (“-----------------------------------------\n”);
        //--------------------------------------
        //Extract different components in the parent URL
        //--------------------------------------

        str = {‘ExtractAccessMethod, canon_url} => hURL;
        echo (“ACCESS METHOD: “ + str + “\n”);
        str = {‘ExtractHost, canon_url} => hURL;
        echo (“HOSTNAME: “ + str + “\n”);
        port = {‘ExtractPort, canon_url} => hURL;
        echo (“PORT NUMBER: “); echo (port); echo (“\n”);
        str = {‘ExtractPath, canon_url} => hURL;
        echo (“PATH/OBJECT REFERENCE: “ + str + “\n”);
        str = {‘ExtractAnchorName, canon_url} => hURL;
        echo (“ANCHOR: “ + str + “\n\n”);
        echo (“Good-Bye.\n”);
         ‘Exit => theApp;
   }
```

6.4.8 IOweb - *Abstract*

This abstract class provides client-side World-Wide Web protocol support. It manages a network connection to a Web server. Its subclasses include IOhttp and IOftp.

Superclasses

None

Methods

on Connect: string *hostname*, integer *port*

Establishes a connection with the server process on the specified host and port. You should call this method only after you use the default Constructor. You should call SendRequest to send a request.

on SendRequest: handle *hRequest*

Sends hRequest to the server. You should use this method if you had previously used the default Constructor and called Connect. "hRequest" must be a handle to an instance of an IOwebRequest subclass.

on Request

Sends a request to the server. It is assumed that the request info was previously specified in a constructor such as ConstructFromURL or ConstructFromRequest.

on GetResponse: return integer

Waits for the server to return a response in blocking mode. It returns the server response status code, or -1 on failure.

on GetEntity: return handle

If the response contains a data entity, this method returns a handle to an instance of a subclass of IOwebEntity. Otherwise, it returns a NULL handle. It should be noted that IOwebEntity mainly allows you to access the entity headers (i.e., metainformation about a data entity). If an entity body is returned from the server, you should construct an instance of an IOwebstream subclass from the IOwebEntity in order to read the data body.

on Close

Closes the connection with the server. Note: If you had constructed an IOwebStream to extract data from this connection, you must not close this IOweb until you are done reading from the data stream.

on GetURL: return string

Returns the URL string associated with this Connection session. For HTTP connections, if AutoRedirected is TRUE, this method returns the new URL used for this connection object.

on Good: return boolean

Returns TRUE if the connection is usable and that the last operation succeeded.

on Fail: return boolean

Returns TRUE if the last operation failed. The connection may still be ok.

on Bad: return boolean

Returns TRUE if something is wrong and the connection is unusable.

on Disconnected: return boolean

Returns TRUE if the connection is not active.

Attributes

None

Activities

None

Example

None

6.4.9 IOftp

This class provides client-side FTP protocol support. It manages a network connection to an FTP server. It is a subclass of IOweb.

Superclasses

Section 6.4.8, “IOweb - Abstract” page 211

Methods

upon Construct

Default constructor.You must call Connect, SendRequest, and GetResponse later.

upon ConstructFromURL: string *url*

Given a URL string, connects to the appropriate server. After this constructor, you can call Request to send a request to the server. It is assumed that the URL string contains sufficient information to send a request. The request defaults to GET. If you wish to specify methods other than GET in your request, use ConstructFromRequest.

upon ConstructFromRequest: handle *hRequest*

Given an HTTP request, connects to the appropriate server. After this constructor, you can call Request to send a request to the server. "hRequest" must be a handle to an instance of an IOftpRequest or its subclass thereof.

on GetProtocolName: return string

Returns the string "FTP".

Attributes

None

Activities

None

Example

This program illustrates all FTP-related classes.

```
string url;
handle connection;
handle request;
handle entity;
handle stream;
string data;
string content_type;
integer content_len;
integer response_status;
string  response_reason;
upon Construct
  {
    url = “ftp://ceci.mit.edu/pub/”;
    //-------------------------------------------------
    //Construct an FTP request from the URL
    //-------------------------------------------------
    request = new {‘Construct, url} => IOftpRequest;
    //-------------------------------------------------
    //Construct an IOftp and connects to the server
    //-------------------------------------------------
    connection = new {‘ConstructFromRequest, request} => IOftp;
    {‘Check_Status, “connection”} => self;

    {‘GetIt,  request} => self;
    ‘Close => connection;
    echo (“Closed FTP connection and stream. Good-Bye.\n”);
    ‘Exit => theApp;
  }
on GetIt : handle  aRequest
 {
    //-------------------------------------------------
    //Send the request
    //-------------------------------------------------
    echo (“Connected!!  Sending request...\n”);
    { ‘SendRequest, aRequest} => connection;
    {‘Check_Status, “sent request”} => self;
    //-------------------------------------------------
    //Get the response
    //-------------------------------------------------
    echo (“Requst sent.  Waiting for Response...\n”);
    response_status =  ‘GetResponse => connection;
    {‘Check_Status, “get response”} => self;
    //-----------------------------------------------
    //Get the Data Entity
    //-----------------------------------------------
    entity = ‘GetEntity => connection;
    if (entity == NULL)
    { echo (“Sorry no data entity returned in the response.\n”);
   ‘Exit => theApp;
 }
    content_type = ‘GetContentType => entity;
    //-----------------------------------------------
    //Get the Data Stream
    //-----------------------------------------------
    //Caution: connection object must be active for stream to work
    stream = new {‘ConstructFromConnection, connection, entity } =>
    IOftpStream;
    if (‘Fail => stream)
        {
           echo (“Error opening FTP stream.\n”);
        }
    else
 {
        //-------------------------------------------------------
        //If we have plaintext, directory, or HTML, get the data.
        //-------------------------------------------------------
     if (content_type == “text/plain” ||
    content_type == “text/ftp-directory” ||
         content_type == “text/html”)
        { data = ““;
          while (! ‘Eof => stream)
            { data = data + ‘ReceiveStringLine => stream + “\n”; }
         echo(“Received ftp Data is:\n”); echo(data); echo(“\n”);
        }
     else
       { echo (“Ignore “ + content_type +
             “ type in this test program.\n”);
       }
       }
    ‘Close => stream;
     delete stream;
  } //end of method GetIt
//---------------------------------------------------------------
//Check_Status
//----------------------------------------------------------------
on Check_Status : string where
  {
    if (‘Fail => connection)
       {  echo (“Sorry “ + where + “ failed. Good-Bye.\n”);
     ‘Close => connection;
     delete connection;
     ‘Exit => theApp;
  }
  }
```

6.4.10 IOhttp

This class provides client-side HTTP/1.0 protocol support. It manages a network connection to an HTTP server. It is a subclass of IOweb.

Superclasses

Section 6.4.8, "IOweb - Abstract" page 211

Methods

upon Construct

Default constructor. You must call Connect, SendRequest, and GetResponse later.

upon ConstructFromURL: string *url*

Given a URL string, connects to the appropriate server. After this constructor, you can call Request to send a request to the server. It is assumed that the URL string contains sufficient information to send a request. The request defaults to GET. If you wish to specify methods other than GET in your request, use ConstructFromRequest.

upon ConstructFromRequest: handle *hRequest*

Given an HTTP request, connects to the appropriate server. After this constructor, you can call Request to send a request to the server. "hRequest" must be a handle to an instance of an IOhttpRequest or its subclass thereof.

on AutoRedirected: return boolean

TRUE if we have been automatically redirected to another URL.

on ResponseReason: return string

Returns the response reason returned from the server. The reason string usually explains why an HTTP request was not fulfilled by the server.

on GetProtocolName: return string

Returns the string "HTTP".

Attributes

None

Activities

None

Example

This program illustrates all HTTP-related classes.

```
string url;
  handle connection;
  handle request;
  handle entity;
  handle stream;
  string data;
  string content_type;
  integer content_len;
  integer response_status;
  string  response_reason;
  //-----------------------------------------------------------
  //Constructor
  //-----------------------------------------------------------
  upon Construct
    {
      //substitute this URL to your favorite http url for testing.
       url = “http://abelard.mit.edu/”;

      //----------------------------------------------
      //Construct an HTTP request from the URL
      //----------------------------------------------
      request = new {‘Construct, url} => IOhttpRequest;

      //The default method is GET, but you can set the
      //method to something else, e.g. HEAD, by doing this:
      //{‘SetMethod, “HEAD”} => request;

      //Let’s see what our request line looks like...
      echo (“Full HTTP Request line is \n”);
      echo (‘RequestLine => request);

      //----------------------------------------------
      //Construct an IOhttp and connects to the server
      //----------------------------------------------
      connection = new {‘ConstructFromRequest, request} => IOhttp;

      //Check if we have successfully connected
      {‘Check_Status, “connection”} => self;

      //----------------------------------------------
      //Send the request
      //----------------------------------------------
      echo (“Connected!!  Sending request...\n”);
      ‘Request => connection;

      //Check if request has been sent successfully
      {‘Check_Status, “sent request”} => self;

      //----------------------------------------------
      //Get the response
```

```
        //-----------------------------------------------
        echo (“Requst sent.  Waiting for Response...\n”);
        response_status =  ‘GetResponse => connection;

        {‘Check_Status, “get response”} => self;

        response_reason = ‘ResponseReason => connection;
        echo (“Response Status Code: “ + response_status + “ \n”);
        echo (“Response Reason: “ + response_reason + “ \n”);

        //-----------------------------------------------
        //Get the Data Entity
        //-----------------------------------------------
        //If we got a response, get the Data Entity
        entity = ‘GetEntity => connection;

        //it’s possible that the response does not contain any entity
        if (entity == NULL)
        { echo (“Sorry no data entity returned in the response. \n”);
       ‘Exit => theApp;
    }

       //Get some metainformation on this data entity
       content_type = ‘GetContentType => entity;
       content_len = ‘GetContentLen => entity;

       echo (“Content Type: “ + content_type + “ \n”);
       echo (“Content Length: “); echo (content_len); echo (“\n”);

       if (! (‘HasBody => entity))
       { echo (“Sorry, no data body included in the response.\n”);
         ‘Close => connection;
          ‘Exit => theApp;
       }

        //-----------------------------------------------
        //Get the Data Stream
        //-----------------------------------------------
        //Caution: connection object must be active for stream to work
        stream = new {‘ConstructFromConnection, connection, entity }
                        => IOhttpStream;

        if (‘Fail => stream)
            {
               echo (“Error opening HTTP stream.\n”);
            }
        else
    {
            //-----------------------------------------------
              //If we have plaintext or HTML, get the data.
              //-----------------------------------------------
         if (content_type == “text/plain” ||
             content_type == “text/html”)
            { data = ““;
```

```
                while (! ‘Eof => stream)
                  {
               data = data + ‘ReceiveStringLine => stream + “\n”;

                    if (‘Eof => stream)
                 { echo (“end of stream!\n”); break; }
                   }

               echo(“Received HTTP Data is:\n”); echo(data); echo(“\n”);
              }
           else  //ignore any other types of data in this test program
             {
            echo (“Ignore data of “ + content_type +
                      “ type in this test program.\n”);
             }
             }
          ‘Close => stream;
          ‘Close => connection;

          echo (“Closed HTTP connection and stream. Good-Bye.\n”);
          ‘Exit => theApp;
        }
      //---------------------------------------------------------------
      //Check_Status
      //---------------------------------------------------------------
      on Check_Status : string where
        {
          if (‘Fail => connection)
             {  echo (“Sorry “ + where + “ failed. Good-Bye.\n”);
           ‘Close => connection;
           ‘Exit => theApp;
        }
        }
```

6.4.11 IOwebRequest - *Abstract*

This abstract class represents a client-side request to a World-Wide Web server. Its subclasses include IOhttpRequest and IOftpRequest.

Superclasses

None

Methods

on SetMethod: string *method*

Use this method to specify the request operation. For HTTP request, the methods may be GET, POST, or HEAD. For FTP, the only supported operation in this release is GET, which is the default. No validity checking of the method is performed. If not set, the default method is GET.

on GetMethod: return string

Returns the name of the request method contained in this request structure.

on SetEntity: handle *hEntity*

Use this method to specify the data Entity you wish to send to the server as part of the request. This is necessary for HTTP method such as POST -- mostly used for sending HTML fill-out form content to the server. "hEntity" should be a handle to an instance of an IOwebEntity subclass.

on GetURL: return string

Returns the URL string used to construct this request object.

on GetHost: return string

Returns the hostname as specified in this request structure. The hostname is usually extracted from the URL string from which this request was constructed.

on GetPort: return integer

Returns the port number specified in this request structure. The port number is usually extracted from the URL string of this request. If no port number is specified in the URL, a default port number (80 for HTTP) is used.

Attributes

None

Activities

None

Example

None

6.4.12 IOftpRequest

Represents an FTP request structure to an FTP server. This class is a subclass of IOwebRequest.

Superclasses

Section 6.4.11, “IOwebRequest - Abstract” page 218

Methods

upon Construct: string *url*

Default constructor which constructs an FTP request using the specified URL.

Attributes

None

Activities

None

Example

For a sample program using this IOftpRequest see "IOftp" on page 212.

6.4.13 IOhttpRequest

Represents an HTTP/1.0 request structure. This class allows you to construct an HTTP request based on a URL string. In particular, it allows you to specify various HTTP request header fields. A request header is not sent if its value is not set. This class is a subclass of IOwebRequest.

Note: If you want to send an HTTP POST request, you should construct an instance of this class, and set the method to POST. You should also construct an IOhttpEntity which contains the POST data, and then use SetEntity method in this class to enclose the Entity as part of the POST request.

Superclasses

Section 6.4.11, "IOwebRequest - Abstract" page 218

Methods

upon Construct: string *url*

Default constructor which constructs an HTTP request using the specified URL.

on SetUserAgent: string *agentName*

Use this method to specify the user agent originating the request. The default value for this field is "Experimental-HTTP-Client". Please refer to HTTP/1.0 specification for detail.

on GetUserAgent: return string

Returns the value of the User Agent header field.

on SetFrom: string *fromAddress*

Sets the From header field which contains an Internet email address of the user who controls the requesting user agent. Please refer to HTTP/1.0 specification for detail.

on GetFrom: return string

Returns the value of the From field.

on RequestLine: return string

Returns the full request line in HTTP/1.0 format. This method is mainly for debugging purposes and is subject to changes by final release.

on RequestHeaders: return string

Returns the formatted request header fields (not including the Entity headers in HTTP/1.0 format. This method is mainly for debugging purposes and is subject to changes.

Attributes

None

Activities

None

Example

For an example using IOhttpRequest see "IOhttp" on page 215.

6.4.14 IOwebEntity - *Abstract*

Represents a World-Wide Web data entity. An IOwebEntity can be enclosed within a Web request as part of the request data. It can also be part of a Web response data returned from the server. Its subclasses include IOhttpEntity and IOftpEntity.

Superclasses

None

Methods

on SetContentType: string *type*

Use this method to specify the content type of this entity. This method is only meaningful if this entity is to be included as part of a Web request. The content type string should follow the MIME content type format, e.g., text/plaintext, text/html, and image/gif, and so forth. Note: For POST HTML form data, the content type is usually set to application/x-www-form-urlencoded.

on GetContentType: return string

Returns the content type of this entity.

Attributes

None

Activities

None

Example

None

6.4.15 IOftpEntity

Represents an FTP data entity. Currently, an IOftpEntity can only be part of the response data from the server. It cannot be part of a request structure, since we do not yet support "put" for FTP. This class is a subclass of IOwebEntity.

Superclasses

Section 6.4.14, “IOwebEntity - Abstract” page 221

Methods

upon Construct

Default constructor.

Attributes

None

Activities

None

Example

For a sample program using the IOftpEntity class, see “IOftp” on page 212.

6.4.16 IOhttpEntity

Represents an HTTP data entity. An entity consists of entity headers (metainformation) and entity body (content). An HTTP entity may be enclosed within an HTTP request or an HTTP response message. If the entity is to be part of an HTTP request, you can set both the entity headers and data content. If the entity is part of an HTTP response from a network connection, this class allows you to get the entity headers, but not the entity body. You must construct an IOhttpStream from this entity in order to retrieve data from the network connection. This class is a subclass of IOwebEntity.

Superclasses

Section 6.4.14, “IOwebEntity - Abstract” page 221

Methods

upon Construct

Default constructor.

on GetContentLen: return integer

Returns the content length of this Entity.

on SetContentString: string *content*

Use this method to specify the data content of this Entity. This method is only meaningful if this Entity is to be included as part of an HTTP request. Use IOurl Escape method to encode your data. This method is considered experimental, and is subject to changes by the final release.

on HasBody: return boolean

Returns TRUE if this Entity has an Entity Body; otherwise, FALSE. (Use this method to check whether you should construct an IOhttpStream to access the Entity Body.)

Attributes

None

Activities

None

Example

For a sample program using IOhttpEntity, **see "IOhttp" on page 215** .

6.4.17 IOwebStream - *Abstract*

This abstract class represents an input data stream from a World-Wide Web connection. Its sub-classes include IOhttpStream and IOftpStream. Normally a Web stream contains media-specific data, and should be given to an appropriate media element to load the data. However, it is also possible to use the various receive methods below to get the data directly at the *ADL* level. (Note: these receive methods are experimental and are subject to changes.)

Superclasses

None

Methods

on OpenFromURL: string *url*

Given a URL, establishes a connection, sends a request, and gets a response from the server. The default request to the server is to "get" or "retrieve" the document referenced by the URL.

on OpenFromRequest: handle *hRequest*

The action of this method is equivalent to OpenFromURL, except that you can specify a more complicated request structure via "hRequest". For example, in the case of an HTTP request, it is possible to set the HTTP method to POST or HEAD in hRequest. "hRequest" must be a handle to an instance of an IOwebRequest subclass.

on OpenFromConnection: handle *hConnection,* handle *hEntity*

This method is used when you had previously established a connection and obtained a valid response from a server using an instance of IOweb subclass. "hConnection" must be a handle to an instance of an IOweb subclass. "hEntity" must be a handle to an IOwebEntity subclass.

on GetURL: return string

Returns the URL string associated with this data stream. If URL redirection was performed (HTTP only), this method returns the final URL used to retrieve the data at hand.

on Close

Closes this http data stream.

on Good: return boolean

Returns TRUE if the stream is usable and that the last operation on the stream succeeded.

on Fail: return boolean

Returns TRUE if the last operation on the stream failed. The stream may still be usable.

on Bad: return boolean

Returns TRUE if the stream is unusable.

on Eof: return boolean

Returns TRUE if end-of-file flag is set on the stream.

on ReceiveBoolean: return boolean

on ReceiveInteger: return integer

on ReceiveReal: return real

on ReceiveStringLine: return string

on ReceiveStringWord: return string

on ReceiveList: return list

on ReceiveAny: return any

Attributes

None

Activities

None

Example

None

6.4.18 IOftpStream

This class represents an input data stream from an FTP connection. It is a subclass of IOwebStream.

Superclasses

Section 6.4.17, “IOwebStream - Abstract” page 223

Methods

upon Construct

Default constructor. Constructs an IOftpStream without opening the stream. You must call OpenFromURL, OpenFromRequest, or OpenFromEntity to open the stream.

upon ConstructFromURL: string *url*

Given a URL string, constructs and opens an FTP data stream. This constructor is equivalent to calling Construct and then OpenFromURL.

upon ConstructFromRequest: handle *hRequest*

Given a handle to an FTP request, constructs and open an FTP data stream. This constructor is equivalent to calling Construct and then OpenFromRequest.hRequest must be a handle to IOftpRequest or its subclass thereof.

upon ConstructFromConnection: handle *hConnection*, handle *hEntity*

Given a handle to an FTP connection and an FTP Entity, constructs an FTP data stream to access the body of the Entity from the connection. This constructor is equivalent to calling Construct and then OpenFromConnection. "hConnection" must be a handle to an instance of an IOftp or its subclass thereof. "hEntity" must be a handle to an instance of an IOftpEntity or its subclass thereof.

on GetContentType: return string

Returns the media type of the FTP data Entity contained in this stream. If the data at hand contains directory information, the content type is text/ftp-directory. This method is experimental and is subject to changes.

on ReceiveDirectory: return list

This method reads and parses the directory information contained in the FTP stream. I returns the parsed directory listing as a list. The returned list consists of sublists, each of which represents an entry/item in directory listing. Each sublist contains the following components:

(1) a string to indicate the item type, which can be "dir", "link", or "file";

(2) a string to indicate item or file name; and (3) an integer to indicate item or file size.

Attributes

None

Activities

None

Example

```
string url;
handle stream;
string data;
string content_type;
integer content_len;

upon Construct
  {
    //substitute this url to your favorite ftp url for testing
    url = “ftp://ceci.mit.edu/pub/”;
    //---------------------------------------------------------
    //Construct an FTP stream without opening it
    //---------------------------------------------------------
    stream = new ‘Construct => IOftpStream;

    //---------------------------------------------------------
    //Open the stream by giving it a URL. OpenFromURL will connect
    //to the server, sends a request to, and gets a response back.
    //---------------------------------------------------------
    {‘OpenFromURL, url} => stream;

    //---------------------------------------------------------
    //Check the status.
    //---------------------------------------------------------
     if (‘Fail => stream)
        {
           echo (“Failed in opening FTP stream.\n”);
           ‘Exit => theApp;
        }
 content_type = ‘GetContentType => stream;

     //-------------------------------------------
     //if we have a plaintext or html, use
     //ReceiveStringLine to get the data...
```

```
          //----------------------------------------
          if (content_type == “text/plain” ||
         content_type == “text/ftp-directory” ||
         content_type == “text/html”)
       {  data = “”;
          while (! ‘Eof => stream)
            { data = data + ‘ReceiveStringLine => stream + “\n”; }
           echo (“Received FTP Data is:\n”); echo (data); echo (“ \n”);
        }
          ‘Close => stream;
          echo (“Closed FTP stream. Good-Bye.\n”);
          ‘Exit => theApp;
    }
```

6.4.19 IOhttpStream

This class represents an input data stream from an HTTP connection. It is a subclass of IOwebStream.

Superclasses

Section 6.4.17, “IOwebStream - Abstract” page 223

Methods

upon Construct

Default constructor. Constructs an IOhttpStream without opening the stream. You must call OpenFromURL, OpenFromRequest, or OpenFromEntity to open the stream.

upon ConstructFromURL: string *url*

Given a URL string, constructs and opens an HTTP data stream. This constructor is equivalent to calling Construct and then OpenFromURL.

upon ConstructFromRequest: handle *hRequest*

Given a handle to an HTTP request, constructs and open an HTTP data stream. This constructor is equivalent to calling Construct and then OpenFromRequest. hRequest must be a handle to IOhttpRequest or its subclass thereof.

upon ConstructFromConnection: handle *hConnection*, handle *hEntity*

Given a handle to an HTTP connection and an HTTP Entity, constructs an HTTP data stream to access the body of the Entity from the connection. This constructor is equivalent to calling Construct and then OpenFromConnection. "hConnection" must be a handle to an instance of an IOhttp or its subclass thereof. "hEntity" must be a handle to an instance of an IOhttpEntity or its subclass thereof.

on GetContentType: return string

Returns the media type of the HTTP data Entity contained in this stream.

on GetContentLen: return integer

Returns the expected size of the Entity (or data) body contained in this stream. The return result of this method should be used with caution, as not all HTTP data streams have the content length field set. A return result of zero should be interpreted as either that the stream is unavailable *or* that the content length field is not specified.

Attributes

None

Activities

None

Example

```
string url;
handle stream;
string data;
string content_type;
integer content_len;

upon Construct
  {
    //substitute this url to your favorite http url for testing
    url = “http://abelard.mit.edu/”;

    //-------------------------------------------------------
    //Construct an HTTP stream without opening it
    //-------------------------------------------------------
    stream = new ‘Construct => IOhttpStream;

    //-------------------------------------------------------
    //Open the stream by giving it a URL. OpenFromURL will connect
    //to the server, sends a request to, and gets a response from it.
    //-------------------------------------------------------
    {‘OpenFromURL, url} => stream;

    //-------------------------------------------------------
    //Check the status. For HTTP streams, it is a failure if there is
    //no data entity body in the response from the server.
    //-------------------------------------------------------
     if (‘Fail => stream)
        {
           echo (“Failed in opening HTTP stream.\n”);
           ‘Exit => theApp;
        }
     //-----------------------------------------
     //Check the content type and content length
     //of the data entity in this stream
     //-----------------------------------------
     content_len  = ‘GetContentLen  => stream;
```

```
          content_type = ‘GetContentType => stream;

          echo (“Content-Type : “ + content_type + “\n”);
          echo (“Content-Len : “); echo (content_len); echo (“ \n”);
          //----------------------------------------
          //if we have a plaintext or html, use
          //ReceiveStringLine to get the data...
          //----------------------------------------
          if (content_type == “text/plain” ||
         content_type == “text/html”)
       {  data = ““;
          while (! ‘Eof => stream)
            {
               data = data + ‘ReceiveStringLine => stream + “\n”;
               if (‘Eof => stream)
              { echo (“end of stream!\n”);  }
              }
           echo(“Received HTTP Data is:\n”); echo(data); echo(“\n”);
        }
          ‘Close => stream;
          echo (“Closed HTTP stream. Good-Bye.\n”);
          ‘Exit => theApp;
```

6.4.20 XNstream

This class is a network input/output stream object with XDR data representation.

Superclasses

None

Methods

upon Construct

Default constructor.

upon ListenConstruct: integer *port*

Waits for connection on specified port.

upon ConnectConstruct: string *host*, integer *port*

Attempts to make a connection to the specified port on the specified host.

on Listen: integer *port*

on Connect: integer *port*, string *host*

Same as ListenConstruct and ConnectConstruct, but not constructors.

on SendBoolean: boolean *val*

on SendInteger: integer *val*

on SendReal: real *val*

on SendString: string *val*

on SendList: list *val*

on SendAny: any *val*

Writes the specified value to the network connection.

on ReceiveBoolean: return boolean

on ReceiveInteger: return integer

on ReceiveReal: return real

on ReceiveString: return string

on ReceiveList: return list

on ReceiveAny: return any

Reads a value of the specified type from the network connection.

on Close

Closes connection.

on Good

Returns a status of stream. If TRUE, stream is healthy and connection is alive.

Attributes

None

Activities

None

Example 1

This program demonstrates receiving a list from a network stream. The applications ends when the list has been received.

```
uses “nro.adl”@”StdLib”;
class Message : ActivityManager
{
  integer portNumber;
  vanillaNro {‘Create, ‘ConnectReady, self, ‘GetMessage, FALSE} =>
   connectNro;
//* used to notify object when message arrives
  handle hNetNotify;
  handle hMessageStream; /* used to send and receive messages */
  list ActivityInfo = {{‘ReceiveMessage, {“message”}}};

  upon Construct: integer port
  {
    portNumber = port;
    hNetNotify = new {‘CreateFromPort, portNumber} => IOnwNotify;
    {‘Subscribe, &connectNro} => hNetNotify;
  }

  on GetMessage: boolean cd
  {
    any message;
    hMessageStream = ‘AcceptXN => hNetNotify;
    {‘Unsubscribe, &connectNro} => hNetNotify;
    delete hNetNotify;
    message = ‘ReceiveAny => hMessageStream;
    ‘Close => hMessageStream;
    delete hMessageStream;
    hNetNotify = new {‘CreateFromPort, portNumber} => IOnwNotify;
    {‘Subscribe, &connectNro} => hNetNotify;
    {‘TriggerNotification, ‘ReceiveMessage, {message}}=> self;
  }

  on Destroy
  {
    {‘Unsubscribe, &connectNro} => hNetNotify;
    delete hNetNotify;
  }
};

upon Construct
{
  integer portnum = 8900;
  Message {‘Construct, portnum} => myMessage;
  nro{‘Create, ‘ReceiveMessage, self, ‘GetMessage, TRUE}=> messNro;
  {‘Subscribe, &messNro} => myMessage;
  {‘GetMessage, TRUE} => myMessage;
}
on GetMessage: any cd, list names, list vals
{
   echo (vals);}
```

Example 2

This program demonstrates sending a list over a network stream.

```
uses “nro.adl”@”StdLib”;

class Message : ActivityManager
{
  integer portNumber;
  vanillaNro {‘Create, ‘ConnectReady, self, ‘GetMessage, FALSE} =>
   connectNro;
// used to notify object when message arrives
  handle hNetNotify;
// used to send and receive messages
  handle hMessageStream;
  list ActivityInfo = {{‘ReceiveMessage, {“message”}}};

```

```
    upon Construct: integer port
    {
      portNumber = port;
      hNetNotify = new {‘CreateFromPort, portNumber} => IOnwNotify;
      {‘Subscribe, &connectNro} => hNetNotify;
    }

    on SendMessage: any message, string host, integer port
    {
      {‘Unsubscribe, &connectNro} => hNetNotify;
      delete hNetNotify;
      hMessageStream = new {‘ConnectConstruct, host, port}
     => XNstream;
      {‘SendAny, message} => hMessageStream;
      delete hMessageStream;
      hNetNotify = new {‘CreateFromPort, portNumber} => IOnwNotify;
      {‘Subscribe, &connectNro} => hNetNotify;
    }

    on Destroy
    {
      {‘Unsubscribe, &connectNro} => hNetNotify;
      delete hNetNotify;
    }

  };

  upon Construct
  {
   /* Define the port number & host where receiving end is running */
    integer portnum = 8900;
    string tohost = “anyhost.your.domain”;
    string msgstring = “Hello from a remote AthenaMuse”;

    Message {‘Construct, portnum} => myMessage;
    nro{‘Create, ‘ReceiveMessage, self, ‘GetFromPeer, FALSE}
           =>messNro;
    {‘Subscribe, &messNro} => myMessage;
    {‘SendMessage, msgstring, tohost, portnum } => myMessage;}
```

6.5 External Processes

The External Processes (XT) wrapped classes are designed to allow the user of ADL to have access to other computational processes outside of AM2. This function can even allow AM2 to spawn other applications and send them input and output. Documentation for the following classes appear in this section:

Note that the class inheritance tree diagram for the External Commands (XT) wrapped classes is flat, meaning that that these classes do not have superclasses, thus no inheritance diagram is provided here.

6.5.1 XTcommand

This class implements the execution of an external command in a fashion similar to the UNIX "system" routine.

Superclasses

Section 6.1.2, "Activity Manager - Abstract" page 116

Section 6.1.3, "Attribute Manager - Abstract" page 118

Methods

upon Create: list *cmd*

Executes the command specified in the argument cmd. This constructor blocks on the execution of the external command. The list cmd must be a list of strings with at least one element. The first string on the list represents the command name, and the following elements, if present, represent command line arguments. The interpretation of the command list is platform dependent and is normally left to the standard platform execution environment. Under UNIX, this would be the user's shell.

upon CreateNoBlock: list *cmd*

Executes the command specified in the argument cmd asynchronously. Under UNIX, this corresponds to executing the command "in the background". This constructor does not block on the execution of the external command. The argument must be formatted as in the Create method.

on Abort

Kills an external command initiated with the ‘CreateNoBlock constructor. This method executes in a platform dependent manner, and it does not guarantee smooth termination or cleanup of the external command. The spawned external command should normally be terminated through their own interface.

on Done: return boolean

Returns TRUE if the external command has terminated and FALSE otherwise.

on GetStatus: return integer

The returned value is a status variable indicating the success or failure of the external command. The value 0 indicates successful execution on all platforms; non-zero values are platform dependent error codes. If an UNSET value is returned, then the external command has not terminated, i.e., in the cases in which the ‘Done message would return FALSE.

Attributes

None

Activities

None

Example

```
/* Demonstrates the use of XTcommand to run external programs */
anonymous: XFtop
{
      XFbutton but1
      {  x=20; y=100; width=400; height=40;
         label="Create Non Blocking External Process";
       };
      XFbutton but2
      {  x=20; y=200; width=400; height=40;
         label="Create Blocking External Process";
       };
upon Construct
       { width=440; height=350;
         but1.Pressed = {'but1Pressed, self};
         but2.Pressed = {'but2Pressed, self};
       }
on but1Pressed
       {   /*This creates a Telnet Session on Windows*/
             new {'CreateNoBlock, {"telnet"}} => XTcommand;
       }
on but2Pressed
       {   /*This creates a Telnet Session on Windows*/
             new {'Create, {"telnet"}} => XTcommand;
       }
}top;
```

6.5.2 XTprocFilter (only on UNIX)

This class implements the execution of an external command that functions as a co-process to the *ADL* script creating the XTprocFilter. That is, it both reads a stream generated by the calling *ADL* script and generates an output stream read by the script.

Superclasses

None

Methods

upon Create: list *cmd*

Executes the command specified by cmd in such a way that the input to and output from the command is tied to a read-write IOpipe whose handle is stored in the member hPipe.

on Abort

Kills the external command. This should not be the standard method of terminating the external process. Normal termination should be triggered by closing the IOpipe pointed to by hPipe.

on GetStatus: return integer

The returned value is a status variable indicating the success or failure of the external command. The value 0 indicates successful execution on all platforms; positive non-zero values are platform dependent error codes. ‘GetStatus returns UNSET if one of the IOpipes to or from the external process is still open. This return value except in the UNSET case is identical to the value returned by ‘Close => hPipe once the pipe has closed.

Attributes

None

Activities

Activity	Keys	Description
hPipe	handle	handle to the IOpipe object opened when the XTprocFilter was created

Figure 6.50: XTprocFilter Activities

Example

None

6.5.3 XTprocSink (only on UNIX)

This class implements the execution of an external command that reads a stream generated by the calling *ADL* script. It thus resembles the UNIX *popen* routine in “w”rite mode.

Superclasses

None

Methods

upon Create: list *cmd*

Executes the command specified by cmd in such a way that the input to the command is tied to a writable IOpipe whose handle is stored in the member hTo.

on Abort

Kills the external command. This should not be the standard method of terminating the external process. Normal termination should be triggered by closing the IOpipe pointed to by hTo.

on GetStatus: return integer

The returned value is a status variable indicating the success or failure of the external command. The value 0 indicates successful execution on all platforms; non-zero values are platform dependent error codes. ‘GetStatus returns UNSET if the IOpipe to the external process is still open. This return value except in the UNSET case is identical to that returned by ‘Close => hTo if the pipe has already been closed.

Attributes

None

Activities

Activity	Keys	Description
hTo	handle	handle to the IOpipe object opened when the XTprocSink was created

Figure 6.51: XTprocSink Attributes

Example

```
// Decode a rot-13 message using tr, a standard UNIX utility
XTprocSink { ‘Create, { “tr”, “a-zA-Z”, “n-za-mN-ZA-M” } }
                    => myOutFilter;
integer status;
myOutFilter.hTo << “Zrffntr va n obggyr” << Endl;
status = ‘Close => myOutFilter.hTo;
```

6.5.4 XTprocSource (only on UNIX)

This class implements the execution of an external command that generates an output stream read by the calling *ADL* script. It thus resembles the UNIX *popen* routine in “r”ead mode.

Superclasses

None

Methods

upon Create: list *cmd*

Executes the command specified by cmd in such a way that the output from the command is tied to a readable IOpipe whose handle is stored in the member hFrom.

on Abort

Kills the external command. This should not be the standard method of terminating the external process. Normal termination should be triggered by the external process itself and should be sensed by detecting an EOF on the pipe associated with hFrom.

on GetStatus: return integer

The returned value is a status variable indicating the success or failure of the external command. The value 0 indicates successful execution on all platforms; non-zero values are platform dependent error codes. ‘GetStatus returns UNSET if the IOpipe from the external process is still open. This return value except in the UNSET case is identical to the value returned by ‘Close => hFrom once the pipe has closed.

Attributes

None

Activities

Activity	Keys	Description
hFrom	handle	handle to the IOpipe object opened when the XTprocSource was created

Figure 6.52: XTprocSource Attributes

Example

None

6.6 Database

The Database (DB) wrapped classes provide multi-database support for object-oriented multimedia. The addition of database functionality allows the application author to decouple the application code from the application data, allowing the easy update of data and reuse of code. It also allows the application to provide easy access to many existing data repositories. Therefore, it is very important that an application provide easy access to many existing data repositories. It is also important that an application be able to interact, perhaps simultaneously, with a wide variety of existing database systems. The advantages of providing a generic database interface for use by the application author are twofold. The same application code can be used to access multiple database systems and there is no need for the author to spend time learning several different database access interfaces. AM2 DBclasses support the following database packages:

- **UNIX** - Oracle, OS2, PostGress, UniSQL and MSQL which is freeware and makes it possible to incorporate small databases without UniSQL.
- **NT** - Oracle, Dbase, FoxPro, MS Access and any other relational database with ODBC support. ODBC drivers are required for running *AM2* on NT.

Documentation for the following classes appear in this section:

Note that the class inheritance tree diagram for the Database (DB) wrapped classes is relatively flat, meaning that these classes share very few inheritance relationships, thus no inheritance diagram is provided here.

6.6.1 DBdatabase

This is the principle class and represents a connection to a component database. Methods are available on the class to perform schema identification and modification, to control query execution, transaction management and to store application objects in the component database.

Superclasses

Section 6.1.2, “Activity Manager - Abstract” page 116

Section 6.1.3, “Attribute Manager - Abstract” page 118

Methods

on AddAttribute: string *className,* string *attrName,* string *attr* return boolean

Adds an attribute attrName of type attrType to the class className. In a relational database this is equivalent to adding a field to a table. Returns TRUE if the attribute is added to the class, otherwise FALSE.

on Commit

Sends a commit transaction message to the database which saves all changes made to the database since the last commit and automatically begins a new transaction. Commit must be called before a DBdatabase instance is destroyed if changes are to be saved. Only applies to those database systems that support transactions.

on CreateAppObj: string *className,* list *arguments* return handle

This method returns a handle to an application object of type className constructed on the heap. The method is used to create instances of classes retrieved from the database. For example, it can retrieve a row from a database table or the results from a query. The number and type of the values in arguments determines which constructors are used. If there are no arguments, the default constructor is used.

on CreateClass: string *className,* list *attributes* return boolean

Causes the creation of a new class called className in the schema database. The list attributes contains a list of lists. Each of these sublists is made up of two strings. The first item specifies the attribute name and the second item specifies the attribute type. Returns TRUE if the class is created, FALSE if creation fails.

on CreateObj: string *className,* list *attributes,* list *data* return handle

Creates an instance of the class className in the database and returns a handle to the newly created database object. The returned handle will be a handle to an object of class DBobject. The list attributes contains a list of attribute names for which initial data values will be specified and the data contains the corresponding data values in the order specified in the list of attribute names.

on CreateSubClass: string *className*, list *superClasses*, list *attributes* return boolean

Similar to CreateClass, but allows the specification of a list of superclasses, which contains the list of class names from which the newly created class should inherit. The list may contain no class names. The list of attributes contains additional attribute names for which data values will be entered. Returns true upon successful creation of the subclass classname, and false otherwise.

on DeleteClass: string *className* return boolean

Removes the definition of the class named className from the database schema. Returns TRUE if successful, otherwise FALSE. Also necessarily removes all objects of this class.

on Execute: handle *hQuery* return handle

Executes the query specified by hQuery, a handle to an object of type DBquery. The query specification is pre-processed appropriately for the component database type before the query is executed. A handle to a DBcursor object is retuned from which the results may be retrieved.

on ExecuteStr: string *query* return handle

The string query is send directly to the database for execution. No pre-processing of the query is carried out. A handle to a DBcursor object is returned.

on GetAllClasses: return list

returns a list of the names of all the classes/tables in the component database

on GetBaseClasses: return list

Returns a list of the names of all the base classes (classes that have no superclasses) that are defined in the database schema.

on GetPersistentRoot: string *rootName* return handle

Returns a handle to the database object pointed to by the persistent root named rootName, or NULL if the root does not exist in the database. The handle is to an object of class DBobject.

on GetPersistenRoots: return list

Returns a list of the names of all the persistent roots defined in the database. The list will be empty if no roots are defined or if the database does not support the notion of persistent roots.

on Login: string *userName*, string *password* return boolean

Enables the user to specify a username and password when connecting to the database. This method is usually called before a connection to the database is opened. However, it may be called before and/or after Open. It can also be called later to change the username under which database operations are carried out. Returns TRUE if the username and password are accepted successfully and false otherwise.

on Open: string *databaseName*, string *server*, string *location*, string *type* return boolean

Opens a connection to the component database named `databaseName` that is located on the machine specified in the string `server` and can be located by means of the string `location`.

If the string containing `server` is an empty string then the database is presumed to reside on the local machine. Similarly, `location` is an empty string when it is not required to specify the component database. `The` string `type` specifies which type of database is being used. Currently, the supported types for UNIX are ODBC, O2, PG95, UniSQL, and MSQL. The supported types for Windows NT include Oracle, dBase, Foxpro and MS Access. A description of each type available is located after the class interface descriptions. The method returns `TRUE` if a connection is established and `FALSE` if the connection fails.

on Store: handle *hObject* return boolean

Stores an ADL object instance, referenced by the handle hObject, in the database so that it can be retrieved later. The class information for the object to be stored is stored first only if this is the first instance of that class to be stored in the open database. Returns TRUE if the object is stored successfully.

on StoreAs: handle *hObject*, string *objName* return boolean

Similar to Store, but stores the object specified by the handle hObject along with a persistent name objName. This persistent name, objName can be used as an identifier during retrieval.

on StoreClassOnly: handle *hObject* return boolean

Stores only the *ADL* class definition for the object retrieved by the handle `hObject`. It doesn't store individual instance information. Returns TRUE if the class definition is stored successfully.

on RefreshStructure

Reread the database schema definition from the database. This is only necessary if updates are made outside of the multidatabase environment. Changes made within the environment are automatically reflected in the schema.

on Retrieve: handle *hObject,* list *theList* return handle

Retrieves an *ADL* object instance from the database which is referenced by the database handle hObject. The list is used to hold special constructor arguments, such as handle to a parent widget when a widget object is being retrieved. Returns a handle to the retrieved object.

on RetrieveByName: string *className*, string *objName,* list *theList* return handle

Retrieves an ADL object instance from the database which is of class className and identified by the persistent name objName. If the object cannot be found in the database a NULL handle is returned.

on RetrieveClassOnly: string *className*, handle *parentClass* return handle

Retrieves the definition of the application class className from the database and loads it into the environment so that instances of that class can be created. The class is created as a nested class of parentClass. A handle to the class is returned. This handle will be NULL if the class is not successfully created.

Attributes

None

Activities

None

Example

```
DBdatabase dbase;          // variable declarations
list classes, attrs;
string class;
boolean done;

// login to the database with a username and password
done = {‘Login, ‘user, ‘password} => dbase;
// open the unisql database called myDatabase
// the unisql server must be running
done = {‘Open, ‘myDatabase, ‘UNISQL} => dbase;
if(done){
   classes = ‘GetAllClasses => dbase
   for class in classes{
    // list of class attributes
      attrs = {‘GetAttributes, class} => dbase;
   }
// create an instance of class person and give the
// attribute name an initial value of “joe”
// the class and attribute must be valid for the
// database “my Database”
{‘CreateObj, ‘person, {‘name}, {‘joe}} => dbase;
// save the changes to the database
‘Commit => dbase;
}
```

6.6.2 DBclass

This class represents a class definition within a component database. This class has superclass, attribute, method, key and extent properties associated with it. Methods are provided to access these properties.

Superclasses

None

Methods

on GetAttributes: return list

A list of attributes defined for the class is returned. The list is composed of two sublists. The first sublist is a list of attribute names. The second is a list of the data types of these attributes.

on GetMethods: return list

Returns a list of all the methods defined for the class. The returned list contains a sublist for each method. This sublist itself contains three elements. The first element is the name of the method. The second is a string specifying the return type of the method. The third element is a list of strings specifying, in order, the parameter types for the method.

on GetSubClasses: return list

Returns a list of the names of all the subclasses of the class.

on GetSuperClasses: return list

Returns a list of the names of all the superclasses of the class.

Attributes

None

Activities

None

Example

None

6.6.3 DBobject

This class represents a database object. The object will be an instance of a particular database class, which can be found from querying the object. The object has methods which allow manipulation of its attributes and methods.

Superclasses

None

Methods

on Name: return string

Returns the class name of the object class to which the object belongs.

on GetAttribute: string *attribute_name* return any

Returns the value of the specified attribute for that object.

on SetAttribute: string *attribute*, any *value* return boolean

Set the value of the named attribute to value and returns TRUE if successful assignment.

on Call: string *method_name*, list *arguments* return any

Calls the specified method on the object using the list of arguments specified and returns the result of the method invocation.

on Drop: return boolean

Deletes the object from the database and returns TRUE if the action was successful.

Attributes

None

Activities

None

Example

```
DBdatabase dbase;         // variable declarations
handle hObj, hCursor;
list result;

// don’t forget to open the database etc.
// execute a query that returns an object handle
hCursor = {‘ExecuteStr, “select person from person”} => dbase;
result = ‘Next => hCursor;
hObj = at(1, result);
// fine the value of the name attribute
echo({‘GetAttribute, ‘name} => hObj);
// delete the object from the database
‘Drop => hObj;
// save the changes
‘Commit => hObj;
```

6.6.4 DBset

This class represents the set data type. A set is used to hold a collection of data types. Duplicate data types are allowed within the set. To the ADL programmer, a set can be viewed as a list of database objects or values.

Superclasses

None

Methods

upon Create: list *values*

Creates a DBset and initializes its contents to the values found in the list. The contents of the list must be a database-recognized type, such as integer, real, etc. (toDTobject, DTdate etc.).

on GetList: return list

Returns the set in the form of a list.

on SetList: list *values*

Sets the contents of the set to the values in the list.

on AddElement: any *value*

Adds an element to the set of the specified type and value.

Attributes

None

Activities

None

Example

```
DBset {‘Create, {‘dum,’dee}} => set;
// add the integer 3 to the set
{‘AddElement, 3} => set;
values = ‘GetList => set;
// values should now be {‘dum, ‘dee, 3}
```

6.6.5 DBcursor

This class is used to access the results of a query execution. The class also provides information as to the format of the query result set. Each entry in the result set is represented as a list. The cursor class can be thought of as a pointer that moves forward and backwards in this list.

Superclasses

None

Methods

on Success: return boolean

Returns TRUE if the query executed successfully, otherwise FALSE. This value should be checked before attempting to retrieve the results.

on RowCount: return integer

Returns the number of entries in the result set.

on ColumnCount: return integer

This method returns the number of elements in an entry of the result set.

on TypeList: return list

Returns the data types of the elements of a result entry.

on Next: return list

Moves the cursor forward one entry in the result set and returns the value of that entry. Initially Next returns the data values for the first row of the cursor. If Next is used beyond the number of result rows in the cursor, an empty list is returned.

on Prev: return list

Moves the cursor to the previous entry in the result set and returns that entry. If Prev is used back beyond the first result row of the cursor, an empty list is returned.

on OpenAt: integer *position* return list

Moves the cursor to the result entry at position and returns that entry. If it is outside the range of the cursor, an empty list is returned.

on IsLast: return boolean

Returns TRUE if the cursor is currently positioned at the last result in the set. A subsequent execution of the method Next would result in an error.

on Position: return integer

Returns the position to which the cursor currently points. First row is at position 1.

Attributes

None

Activities

None

Example

```
handle hCursor;            // variable declarations
hCursor={‘ExecuteStr, “select name, age from
   person where name=’John’”}=>dbase;
//assuming there are three instances of person that match
//the set would be
//{{‘John, 14}, {‘John, 9}, {;John, 12}}
```

6.6.6 DBquery

This class represents the database query. The multidatabase uses a standard query format that is based on SQL-92 with extended functionality to accommodate object database systems. A typical SQL query has four parts, each part represented by one line in the sample query below. The query finds the name, age and salary of all people called John who are working on project 3345 and orders the result set by salary amount. The keyword in each line is given in bold face.

select name, age salary

from person, project

where name=’John’ and projectID=3345

order by 3

The query class enables the query to be specified as a string or built up from its component parts. The query is subject to pre-processing by the multidatabase system.

Superclasses

None

Methods

on SetQryFrom: list *fromList*

Sets the list of class names from which the query result will be generated.

on SetQryOrderBy: list *orderList*

Sets the order in which the results will appear.

on SetQrySelect: list *selectList*

Sets the list of attributes from which the query result will be generated.

on SetQryText: string *queryText*

Sets the text of the entire query. The query will automatically be broken down into its component parts.

on SetQryWhere: list *whereList*

Sets the list of conditions which govern result generation.

on GetQryOrderBy: return list

Returns the order in which the results will be listed.

on GetQrySelect: return list

Returns the list of attributes from which the query result will be generated.

on GetQryText: return string

Returns the complete text of the query. This will be automatically generated from the component parts if necessary.

on GetQryWhere: return list

Returns the list of conditions which governs result generation.

on Bind: integer *n,* any *value*

A parameter in a query can be specified by a question mark (?) when the query is constructed. This mark must be replaced by an appropriate value before the query is executed. This method replaces the parameter marked by the nth question mark with value.

on BindAll: list *values*

This function replaces all the question marks in the query with the values from the supplied list in order.

Attributes

None

Activities

None

Example

```
DBquery q;
// don’t forget to open the database etc.
// example of constructing queries from lists
{‘SetSelect, {‘name, ‘age}} => q;
{‘SetFrom, {‘person}} => q;
hCursor = {‘Execute, &q} => dbase;
// example of query binding
{‘SetQuery, “select person from person where name = ?”} => q;
{‘BindAll, {‘joe}} => dbase;
hCursor = {‘Execute, &q} => dbase;
```

6.6.7 DBbinary

This class represents the binary data type. It is assumed that the data is either stored directly as bytes or as a reference to an external file that holds the bytes. If the data is stored in an external file, then the file location can be retrieved. The location has three components, a name to identify the machine on which the file resides, a path to locate the file on that machine, and the name of the file itself. Currently, since there is no means of returning a pointer to a binary stream of data to the ADL program directly, the only option with binary data is to store it in a temporary file.

Superclasses

None

Methods

upon Create: string *pathName*, string *fileName*, string *hostName* return handle

Returns a handle to a DBbinary object where the binary information is stored in the file locations specified by the method parameters.

upon CreateAsBinary: string *pathName*, string *fileName*, string *hostName* return handle

Creates a DBbinary object where the binary information is read from the file location specified by the method parameters and stored internally to the object.

on GetFile: return string

If the binary data is stored as a reference to an external file. GetFile returns the external filename.

on GetHost: return string

If the binary data is stored as a reference to an external file. GetFile returns machine name where the file is stored.

on GetPath: return string

If the binary data is stored as a reference to an external file. GetPath returns a path to the external file.

on IsBinary: return boolean

Returns TRUE if the binary data is stored internally to the object. Returns FALSE if the binary data is stored as a reference to an external file.

Attributes

None

Activities

None

Example

None

6.6.8 DBmedia

This class represents media data that is stored either internally in binary form or as a reference to an external file.

Superclasses

Section 6.6.7, "DBbinary" page 249

Methods

upon Create: string *pathName*, string *fileName*, string *hostName* return handle

Returns a handle to a DBmedia object.

Attributes

None

Activities

None

Example

None

6.6.9 DBimage

This class represents image data specifically.

Superclasses

Section 6.6.7, "DBbinary" page 249

Methods

upon Create: string *pathName*, string *fileName*, string *hostName* return handle

Returns a handle to a DBimage object that is located on the hostname machine at the location pathname in filename.

on GetImage: return handle

Returns a handle to an MMimage object that has been constructed from the information contained in the DBimage object.

Attributes

None

Activities

None

Example

None

6.6.10 DBdate

This class is used to represent the date data type. The data type is composed of three components, an integer from 1 to 12 representing the month, an integer from 1 to 31 representing the day, and a four digit number representing the year.

Superclasses

None

Methods

upon Create: integer *month*, integer *day*, integer *year* return handle

Creates a DBdate and initializes the value of the date.

on SetDate: integer *day*, integer *month*, integer *year*

Sets the value of the date.

on GetDateStr: return string

Returns the date in the form of a string such as ‘13/10/1994’.

on GetDateList: return list

Returns the date in the form of a list of integers, {month, day, year}.

Attributes

None

Activities

None

Example

```
DBdate {‘Create, 5, 23, 1994} => date;
echo(‘GetDateStr => date);
```

6.6.11 DBtime

This class represents the time data type. The data types has three components, an integer between 0 and 23 representing the hour, an integer between 0 and 59 representing the minute and an integer between 0 and 59 representing the seconds.

Superclasses

None

Methods

upon Create: integer *hour*, integer *minute*, integer *second* return handle

Creates a DBtime and initializes the value of the time.

on SetTime: integer *hour*, integer *minute*, integer *second*

Sets the value of the time.

on GetTimeStr: return string

Returns the time (24 hour clock) in the form of a string such as ‘14:03:22’.

on GetTimeList: return list

Returns the date in the form of a list of integers, {hour, minute, second}.

Attributes

None

Activities

None

Example

```
DBtime {‘Create, 5, 15, 32} => dtime;
echo(‘GetTimeStr => dtime);
```

6.6.12 DBtimestamp

This class represents the timestamp data type which consist of a time and a date together represented by a data value of type DBtime and a data value to type DBdate.

Superclasses

None

Methods

upon Create: handle *date*, handle *time* return handle

Creates a DBtimestamp and initializes the value of the timestamp.

on GetTstampList: return list

Returns a list composed of two elements, a handle to a DBdate object and a handle to a DBtime object.

on SetTstamp: handle *date*, handle *time*

Sets the value of the timestamp.

on GetTstampStr: return string

Returns the timestamp in the form of a string such as ‘05:04:03 13/10/1994’.

Attributes

None

Activities

None

Example

```
DBtimestamp {‘Create, &date, &dtime} => tstamp;
echo(‘GetTstampStr => tstamp);
```

6.6.13 DBmonetary

This class is used to represent the monetary data type. Currently the only information available is the amount of money represented by an object of this class. Currency information is not yet incorporated into the class.

Superclasses

None

Methods

upon Create: real *amount*

Creates a DBmonetary and initializes the amount of the monetary value.

on SetAmount: real *amount*

Sets the amount of the monetary value.

on GetAmount: return real

Returns the amount of the monetary value.

Attributes

None

Activities

None

Example

```
DBmonetary {‘Create, 14.01} => money;
echo(‘GetAmount => money);
{‘SetAmount, 18.99} => money;
echo(‘GetAmount => money);
```

6.7 Data Structures

The wrapped Data Structures (DS) wrapped classes of *AM2* provide generic optimized data structures. The user could reimplement these using lists but the implementation would be less efficient. Data structure classes are easy to wrap, and a good place for the novice to begin to augment the set of *AM2* wrapped classes. Documentation for the following classes appear in this section:

Note that the Data Structures wrapped classes do not inherit from each other, thus no inheritance diagram is provided here.

6.7.1 DSqueue

This class implements a traditional first-in-first-out (FIFO) queue.

Superclasses

None

Methods

on Enqueue: any *value*

Adds the value to the end of the queue.

on Dequeue: return any

Returns the first value and removes it from the queue.

on First: return any

Returns the first value but does not remove it from the queue.

on IsEmpty: return boolean

Returns TRUE if the queue is empty.

on Unparse: return string

Returns a human-readable description of all entries.

on Clear

Clears the queue of elements.

Attributes

None

Activities

None

Example

```
on Assert: boolean condition
{
if (! condition)
    {
      die(“assertion failed!”);
    }
}

upon Construct
{
  DSqueue q;
  integer i = 6;
  string  s = “heather”;
  list    l = {“this”, “is”, “a”, “list”};
  {“Assert”, “IsEmpty” => q} => theApp;
  {“Enqueue”, 6} => q;
  {“Assert”, !(“IsEmpty” => q)} => theApp;
  {“Enqueue”, “heather”} => q;
  {“Enqueue”, {“this”, “is”, “a”, “list”}} => q;
  {“Assert”, i == “First” => q} => theApp;
  {“Assert”, i == “Dequeue” => q} => theApp;
  {“Assert”, s == “Dequeue” => q} => theApp;
  {“Assert”, l == “Dequeue” => q} => theApp;
  {“Assert”, “IsEmpty” => q} => theApp;
}
on Init
{
  DSqueue q;
  integer i = 6; string s = “heather”;
  list l = {“this”, “is”, “a”, “list”};
  “IsEmpty” => q;   // TRUE
  echo(“Unparse” => q);
  {“Enqueue”, 6} => q;
  echo(“Unparse” => q);
  “IsEmpty” => q;   // FALSE
  {“Enqueue”, “heather”} => q;
  echo(“Unparse” => q);
  {“Enqueue”, {“this”, “is”, “a”, “list”}} => q;
  echo(“Unparse” => q);
  i == “First” => q;    // TRUE
  i == “Dequeue” => q;  // TRUE
  s == “Dequeue” => q;  // TRUE
  l == “Dequeue” => q;  // TRUE
  “IsEmpty” => q;   // TRUE
}
```

6.7.2 DSstack

This class implements a traditional last-in-first-out (LIFO) stack.

Superclasses

None

Methods

on Push: any *value*

Adds the value to the top of the stack.

on Pop: return any

Returns the top value and removes it from the stack.

on Top: return any

Returns the top value but does not remove it from the stack.

on IsEmpty: return boolean

Returns TRUE if the stack is empty.

on Unparse: return string

Returns a human-readable description of all entries.

on Clear

Clears the stack of elements.

Attributes

None

Activities

None

Example

```
on Assert: boolean condition
{
  if (! condition)
    {
      die(“assertion failed!”);
    }
}

upon Construct
{
  DSstack stack;
  integer i = 6;
```

```
    string  s = “heather”;
    list    l = {“this”, “is”, “a”, “list”};

    {“Assert”, “IsEmpty” => stack} => theApp;

    {“Push”, 6} => stack;

    {“Assert”, !(“IsEmpty” => stack)} => theApp;

    {“Push”, “heather”} => stack;
    {“Push”, {“this”, “is”, “a”, “list”}} => stack;

    {“Assert”, l == “Top” => stack} => theApp;
    {“Assert”, l == “Pop” => stack} => theApp;
    {“Assert”, s == “Pop” => stack} => theApp;
    {“Assert”, i == “Pop” => stack} => theApp;

    {“Assert”, “IsEmpty” => stack} => theApp;
  }

  on Init
  {
    DSstack stack;
    integer i = 6;
    string  s = “heather”;
    list    l = {“this”, “is”, “a”, “list”};

    “IsEmpty” => stack;   // TRUE
    echo(“Unparse” => stack);

    {“Push”, 6} => stack;
    echo(“Unparse” => stack);

    “IsEmpty” => stack;   // FALSE

    {“Push”, “heather”} => stack;
    echo(“Unparse” => stack);
    {“Push”, {“this”, “is”, “a”, “list”}} => stack;
    echo(“Unparse” => stack);
```

Appendix A Built-In Functions for *ADL*

This appendix lists the functions that are built-in to *ADL* for operations on the base and compound types. We discuss their use in the *ADL* chapter (**Section 3.12, "Built-in Function Calls" page 29**). The available functions and their operations *will* change as we experiment to find what's most useful for application writers.

A.1 Notation

We list each function name, followed by its argument list in parenthesis, separated by commas with the type for each given. In the following descriptions, we use *sequence* to describe something that can be either a string or a list, and *number* to describe something that can be either an integer or a real. We use *any* to describe something that can be one of the following types: boolean, integer, real, string, handle, interval, list, time, or vtype.

A.2 Function Descriptions

A.2.1 Input/Output

`echo (any argument)`

Effects:	prints a human-readable representation of argument to standard output.
Returns:	argument.

`read ()`

Effects:	reads a string up to white-space from standard input.
Returns:	the string that was read.

`die (string message)`

Effects:	prints the contents of message to standard error, then exits with exit code
Returns:	die does not return

A.2.2 Time and Date

`localTime()`

Returns:	A time value whose hours, minutes, seconds, and milliseconds are set to the current local time.Platforms that cannot provide a given level of granularity should return 0 for that part of the time. For instance, if a machine cannot provide milliseconds, `localTime` may return the current time with milliseconds set to 0.
Note:	AthenaMuse2 relies on the system's idea of the current local time for this function. If you are not happy with the value returned, please check that the time is set properly on your machine.

`localDate()`

Returns:	A list containing four integer elements:
	1.day of week, 1 - 7, 1 == Sunday, etc.
	2.day of month, 1 - 31
	3.month of year, 1 - 12
	4.year, 1900 -
Note:	AthenaMuse2 relies on the system's idea of the current date for this function. If you are not happy with the value returned, please check that the date is set properly on your machine.

A.2.3 Conversion

`toBoolean(any source)`

Requires:	a boolean, a string containing a "TRUE" or "FALSE" (any capitalization), an integer, a real, or a time.
Effects:.	ERtype error if the type cannot reasonably be converted to a boolean.
Returns:	a boolean representation of the value in `source`.

`toInteger(any source)`

Requires:	an integer, a string containing an integer, a real that is smaller than the maximum size of an integer, a boolean, or a time (converts to number of milliseconds).
Effects:	ERsemantic error if the value in `source` is too large for an integer.ERtype error if the value in `source` cannot reasonably be converted to an integer.
Returns:	an integer representation of the value in `source`.

`toInterval(any source)`

Requires:	An interval, a string containing an interval constant, or a list of the format: {{boolean,integer/real},{boolean,integer/real}}.
Effects:	ERtype error if the value in `source` cannot reasonably be converted to an interval.
Returns:	the interval representation of the value in `source`.

`toTime(any source)`

Requires:	A time, an integer (milliseconds), a real (rounded to integer milliseconds), a string, or a list with one to four integer elements (representing the least significant parts of the time, i.e. 2 elements implies seconds and milliseconds).
Effects:	ERtype error if the value in `source` cannot reasonably be converted to a time.
Returns:	the time representation of the value in `source`.

`toList(any source)`

Requires:	A list, a string containing a list with constant elements, or an interval.
Effects:	ERtype if the value in `source` cannot reasonably be converted to a list.
Returns:	the list representation of the value in `source`.

`toReal(any source)`

Requires:	A real, an integer, a boolean, or a string representing a constant real in the same format as that recognized by the parser
Effects:	ERtype if the value in `source` cannot reasonably be converted to a real.
Returns:	A real number representing the value in `source`.

`toString(any source)`

Returns:	A string representation of `source`. If `source` is a handle this call produces a fatal error.

A.2.4 Type Query

- isInteger(any value)
- isBoolean(any value)
- isHandle(any value)
- isInterval(any value)

- isList(any value)
- isReal(any value)
- isString(any value)
- isTime(any value)
- isVtype(any value)

Returns: TRUE if value is of the given type, FALSE if not.

`getType(any value)`

Returns: The vtype of the argument `value`.

`canConvert(string source, vtype conversionType)`

Returns: TRUE if the `source` can be converted to `conversionType`, FALSE if not. Put another way, `canConvert` returns TRUE if the appropriate conversion built-in (`toBoolean`, `toInteger`, etc.) for `conversionType` succeeds when given `source` as an argument.

A.2.5 Sequences (lists and strings)

Lists

`at(integer index, sequence source)`

Returns: If `source` is a list, the item at `index` is returned. If `source` is a string, a one character string containing the character at `index` is returned. The index is counted from 1, not 0.

Notes: If index exceeds the length of source, an `ERsemantic` error is generated. This behavior may change.

`first(sequence source)`

Returns: The first element of the list if `source` is a list or the first character of it is a string. If `source` is empty, an empty sequence of the same type as `source` (list or string) is returned.

`rest(sequence source)`

Returns: A sequence of the same type (list or string) containing all but the first element in source. If the sequence is empty or only contains a single element or character, the returned sequence will be empty.

`isEmpty(sequence test)`

Returns: TRUE if `test` contains no items (list) or characters (string), FALSE otherwise.

`length(sequence source)`

Returns: The number of elements (list) or characters (string) in `source`.

`extract(sequence source,integer start,integer length)`

Returns: A sequence containing the elements of source with indices from `start` to `start+length-1`. Sequence indices start at 1, not 0.

Notes: If an attempt is made to extract beyond the end of a sequence, a shorter sequence will be returned containing only those items from start to the end of the sequence.

`find(any key, sequence source)`

Returns: If `source` and `key` are strings, returns index of first character of first occurrence of `key` in `source`, or 0 if `key` is not found in `source` (since indices in sequences start with 1). If `source` is a string, but `key` is not, an `ERsemantic` error is generated. If `source` is a list, returns index of first member of `source` found to be equal to `key`, or 0 if no match is found.

Notes: lists are *not* searched recursively, only top level members are compared, and no type conversion is performed, i.e. 1.0 != 1.

Strings

`split(string source, string delim)`

Returns: the list of substrings from `source` that remains after every occurrence of `delim` is deleted from it. If two instances of `delim` occur immediately adjacent to each other or if `source` starts or ends with an occurrence of `delim`, then an empty string is inserted into the list at the appropriate position.

`isAlpha(string source)`

Returns: TRUE if all the characters in the `source` string are one of [A-Z, a-z]

`toUpper(string source)`

Returns: a string identical to `source` except that all lowercase alphabetic characters will have been changed to uppercase.

Example: `toUpper("teSt sTriNG!")` returns `"TEST STRING!"`

`toLower(string source)`

Returns: a string identical to `source` except that all uppercase alphabetic characters will have been changed to lowercase.

A.2.6 Mathematical

The mathematical built-ins that follow can produce six categories of errors:

- `DOMAIN`: implies an inappropriate argument, e.g., `sqrt`(–1);
- `SING`: implies an argument or arguments which produce a singularity in the result, e.g., `log`(0);
- `OVERFLOW`: implies the result exceeds the maximal value that the return type can express;
- `UNDERFLOW`: implies that the result is smaller in magnitude than the return type can express;
- `TLOSS`: total loss of significance — most implementations do not produce this error
- `PLOSS`: partial loss of significance

By default, `DOMAIN` and `SING` errors are treated as fatal, and all others are ignored. The developer may change this behavior by sending a `'SetFatalErrors` message to `theApp`. Please see _ for further information.

`random(interval range)`

Returns: A random number in `range`. If both limits of range are integers, an integer will be returned; otherwise a real will be returned.

`sqrt(number val)`

Returns: The nonnegative square root of `val` expressed as a real. A `DOMAIN` error occurs if `val` is negative.

`pow(number base, number exp)`

Returns: `base` raised to the power `exp` expressed as a `real. exp` may be a real if `base` is nonnegative. A `DOMAIN` error occurs if `base` is negative and `exp` is not an integral value, and it may occur when `base` is 0 and `exp` is less than or equal to 0. This built-in may also generate `OVERFLOW` or `UNDERFLOW` errors.

`exp(number val)`

Returns: The exponential function of `val`, e^{val}, expressed as a real. An `OVERFLOW` or `UNDERFLOW` error may occur.

`log(number val)`

Returns: The natural logarithm of `val` expressed as a `real`. A `DOMAIN` error will occur if `val<0.0` and a `SING` error if `val==0`.

`log10(number val)`

Returns: The base 10 logarithm of `val` expressed as a `real`. A `DOMAIN` error will occur if `val<0.0` and a `SING` error if `val==0`.

`cos(number val)`

Returns: The cosine of `val` (interpreted in radians) expressed as a `real`.

`sin(number val)`

Returns: The sine of `val` (interpreted in radians) expressed as a `real`.

`tan(number val)`

Returns: The tangent of `val` (interpreted in radians) expressed as a `real`

`acos(number val)`

Returns: The principal value of the arc cosine of `val` (interpreted in radians) expressed as a `real` in the range $[0,\pi]$. A `DOMAIN` error will occur if `val` does not fall with in the range `[-1,1]`.

`asin(number val)`

Returns: The principal value of the arc sine of `val` (interpreted in radians) expressed as a `real` in the range $[-\pi/2,\pi/2]$. A `DOMAIN` error will occur if `val` does not fall with in the range `[-1, 1]`.

`atan(number val)`

Returns: The principal value of the arc tangent of `val` (interpreted in radians) expressed as a `real` in the range $[-\pi/2,\pi/2]$.

`atan2(number y, number x)`

Returns: The principal value of the arc tangent of `y/x` expressed as a `real` in the range $[-\pi/2,\pi/2]$. A DOMAIN occur error will occur if both `y` and `x` are 0.

cosh(number val)

Returns: The hyperbolic cosine of `val` expressed as a real. An `OVERFLOW` error may occur.

sinh(number val)

Returns: The hyperbolic sine of `val` expressed as a real. An `OVERFLOW` error may occur.

tanh(number val)

Returns: The hyperbolic tangent of `val` expressed as a real.

integerPart(number val)

Returns: The integer part of `val` expressed as a `real`. The returned value always has the same sign as `val.`

fractionPart(number val)

Returns: The fraction part of `val` expressed as a `real`. The returned value always has the same sign as `val.`

ceil(number val)

Returns: The smallest integral value not less than `val` expressed as a `real`.

fabs(number val)

Returns: The absolute value of `val` expressed as a `real`.

floor(number val)

Returns: The largest integral value not larger than `val` expressed as a `real`.

fmod(number x, number y)

Returns: The floating-point (`real`) remainder of x/y. That is, if $r = \mathrm{fmod}(x, y)$, then $\exists i$, such that $x = (i \times y) + r$, r has the same sign as *x,* the magnitude or r is less than that of y. A `DOMAIN` error may occur if `y==0`.

e()

Returns: The constant `e` at the significance of a `real`.

pi()

Returns: The constant `pi` at the significance of a `real`.

A.2.7 Handles

`isValid(handle h)`

Returns: TRUE if the handle is non-NULL and points to a *valid* object or variable, otherwise FALSE.

A.2.8 Classes and Inheritance

`isKindOf(handle hObject, handle hClass)`

Returns: TRUE if `hObject` points to an object (as opposed to a variable of base, compound or complex type), `hClass` points to a wrapped or *ADL* class, and `hObject` points to an instance of the class pointed to by `hClass` or to a class derived from it.

Notes: Pointers to classes are generally derived from expressions using the operators `theClass` or `classOf` **(see Section 3.22, "Metaclass Operations" page 43).**

`className(handle hClass)`

Returns: The string name of the class pointed to by the handle `hClass` or the null string if the class is `anonymous`.

Notes: Pointers to classes are generally derived from expressions using the operators `theClass` or `classOf` (**see Section 3.22, "Metaclass Operations" page 43**).

`isDirectBaseOf( handle hSuspectedBaseClass, handle hKidClass )`

Returns: boolean

Effects: If hSuspectBaseClass is a pointer to a direct base class of the class pointed to by hKidClass, returns TRUE, else FALSE.

Warning: It is a fatal error for either argument to be NULL or to not point to a class.

A.2.9 Networking

`userName()`

Returns: A string containing the name of the user executing AthenaMuse2.

`hostName()`

Returns: If Domain Name Service is enabled on the system, the fully qualified hostname is returned. Otherwise, the host table entry for the machine is returned. This may or may not be a fully qualified hostname depending on the format of the host table for the system.

Appendix B Creating Wrapped Classes

This chapter explains how to create your own wrapped classes using the wrap script, a facility intended for C++ programmers who want to extend the capabilities of *AM2* by installing their own code in the system. Most *AM2* users do not need this facility.

The chapter describes the model the wrap script uses and the details of how to wrap a C++ class to make it visible in the *ADL*. It also describes the model of inheritance the wrap script provides and describes the features that support it.

The purpose of the wrap script is to help the programmers publish C++ classes to the *ADL* as wrapped classes. Once published, a wrapped class is almost indistinguishable from a user-defined class in the *ADL*. For this reason, it is important that you understand the differences between wrapped classes and user-defined classes. To learn more about the two, see **Section 3.23, "Wrapped Classes" page 44** before continuing. If you are not familiar with the wrapped classes that come with the *ADL*, see **Chapter 6, "Wrapped Class Reference"**.

B.1 Wrap Script Model

There are essentially three parts to publishing a class: publishing its name, publishing its members, and publishing its methods.[1] This information is necessary for *AM2* to create a subclass, instantiate a class, and operate on a C++ class. You provide the information by writing a wrap file that describes the class to be wrapped. Once it is written, you run the wrap script, sometimes referred to as the wrapper, and generate the wrapped class.

For instance, to wrap a C++ class called IntStack as a wrapped class called DSIntStack, you write a file called `DSIntStackRO.wrp` and run the wrap script. The wrapper generates the class named DSIntStackRO, writing the files `DSintStackRO.h` to declare it and `DSintStackRO.cc` to implement it. The command line interface to the wrap script determines where to place the generated files. It is described in **Section B.3.1, "Command Line Interface" page 287**.

1 The wrap script also has an inheritance mechanism that provides an inheritance relationship between wrapped classes and simplifies the specification and maintenance of the wrap code itself. We postpone the description of this mechanism until after we cover the basics.

We call the C++ class that we want to wrap the foundation class. In our example, the class IntStack is the foundation class. We encourage programmers to think of the wrapper class as a translation layer only, turning *ADL* messages and member assignments into calls to an instance of the foundation class. In the example, when you create an instance of DSintStackRO, the DSintStackRO constructor creates a corresponding IntStack instance. *ADL* messages such as `Push` and `Pop` translate into calls on that instance of IntStack. By making it easy for a wrapped object to use an instance of the foundation class, the wrapper encourages programmers to put very little into the wrapped class.

B.2 How to Wrap a C++ Class

B.2.1 Lexical Conventions

The wrap script ignores white space in a wrap file. However, some parts of the description must be on lines by themselves, such as delimiters above and below sections containing C++ code not meant for interpretation by the wrap script. The delimiter is a line containing at least 3 consecutive equal signs (=). We recommend using long sequences to provide a good visual break. Both multi-line (`/*... */`) and single line (`//`) comments are allowed in these blocks, since the compiler does not interpret the code. For example:

```
==========================
// Uninterpreted C++ code belongs in here.
/* The bar above and below it must be on lines by themselves. */
==========================
```

Outside of these uninterpreted blocks use single line (`//`) comments to annotate the wrap file, anywhere but the delimiter (====) lines.

B.2.2 The Wrap File: an Example

In the following example, we use the wrapped class DSIntStackRO to look at each part of a wrap file. We explain the components of this file in **Section B.2.3, “An Explanation of the Parts of a Wrap File” page 274**.

```
notice
======================================================================
/*
 *
 *  $Header: /mit/ceci/1/aybee/devl/doc/firstRel/RCS/DSintStackRO.wrp,
v 1.1 1995/06/02 21:23:23 aybee Exp aybee $
```

```
 *
 * Copyright 1993, 1994, 1995 Massachusetts Institute of Technology.
 * All rights reserved.
 * AthenaMuse is a registered trademark of the Massachusetts
 * Institute of Technology.
 */
======================================================================

foundation: IntStack
wrapped:    DSintStack
module:     ExampleModule
abstract:   false

header
======================================================================
#include <IntStack.h>
/*
 * IntStack.h hypothetically includes something like...
 *
 *   class IntStack
 *   {
 *     public:
 *       int  getHeight();
 *       void push(int);
 *       int  pop ();
 *       int  isEmpty();
 *    };
 *
 */
======================================================================

source
======================================================================
#include <control/EXmodule.h>
#include <control/DSintStackRO.h>
#include <adl/ERsemantic.h>
======================================================================

superclasses
{
}

codeFragments
{
constructor
======================================================================
mpWrapped = new IntStack;
======================================================================
}

members
{
private:
```

```
published:
  name
  {
    get: default
    set: default
  }
  height
  {
    get: custom
    set: none
  }
}

methods
{
private:
  void localDestroy()
  ======================================================================
  if (mpWrapped)
     delete mpWrapped;
  ======================================================================

  UTvalue _Get_height() const
  ======================================================================
  return mpWrapped->getHeight();
  ======================================================================

published:
  void Push( integer_t newTop )
  ======================================================================
  /* effects: puts newTop on the top of the stack */
  mpWrapped->push( newTop );
  ======================================================================

  integer_t Pop()
  ======================================================================
  /* returns: removes top value from the stack and returns it */
  assert( ! mpWrapped->isEmpty() );
  return mpWrapped->pop();
  ======================================================================
}
```

B.2.3 An Explanation of the Parts of a Wrap File

Notice

This section lets you protect your intellectual property by housing things such as the copyright notice and the RCS `$Header$` string.[2] The wrap script places a copy of this section at the top of

[2] RCS is a source code control system widely used by programmers.

both the generated header file and the generated source code file. We usually place the contents of the file named “standard_header” in this section.

Foundation or FoundationRO Declaration

The wrap script encourages programmers to implement a wrapped class by using an instance of a foundation class. As encouragement, the wrapper provides a member named `mpWrapped` in the generated class. You can use this member to point to an instance of the wrapped class. The member `mpWrapped` can be either a C++ pointer or a reference counted pointer.

This is a required field and you must use one of these declarations to tell the wrap script the type of the foundation class:

```
foundation: className
```

This line must contain the designation `foundation`: followed by the name of foundation class. The `mpWrapped` member is of type `className`. In the few cases where there is no underlying foundation class to support the wrapped class, you can specify `void` as the class name.

```
foundationRO: classNameRO
```

This line must contain the designation `foundationRO:` followed by the name of foundation class that includes the `RO` suffix. The `mpWrapped` member is of type `classNameRO`.

For your convenience the wrap script provides two macros `WRAP_FOUNDATION` and `WRAP_FOUNDATION_PTR`, defined as the name of the foundation class and the type of a pointer to that class respectively. These become especially useful when you want to take advantage of wrapper inheritance, as discussed in **Section B.2, “Inheritance Model of the Wrap Script” page 280**.

The Wrapped Class Name

```
wrapped: nameOfWrappedClassInADL
```

This is a required field and must contain the designation `wrapped:` followed by the name of the wrapped class in the *ADL*. The latter must be a legal *ADL* identifier. (See **Section 3.2, “Identifiers” page 14**.)

Module Declaration

```
module: moduleName
```

This is a required field and must contain the module name followed by a legal *ADL* identifier. See **Section B.2, “Inheritance Model of the Wrap Script” page 280** for an explanation about the module mechanism that *AM2* uses. You must add a line such as `UT_FORCE_LOAD(nameOfWrappedClassInADL)` to the appropriate module source file to force the linker to pull in the newly wrapped class.

Abstract Declaration

```
abstract: boolean
```

This is a required field. The `boolean` must be either `TRUE` or `FALSE`. If true, this class may not be instantiated in the *ADL*. This becomes useful when you use wrapped class inheritance and have base classes that should not be instantiated.

Can You Create a Subclass in the *ADL*?

`adlSubclassable: boolean`

This is an optional field. The `boolean` must be either `true` or `false`. If true, *ADL* programmers may use this class as a direct base of any *ADL* class. Usually, only abstract classes may not have a subclass in the *ADL*. As a result, the default value for this characteristic is the logical *not* of the previous declaration `abstract`. If a class is abstract, then the default is that you cannot create a subclass in the *ADL*; if a class is not abstract, then the default is that you can create a subclass in the *ADL*.

The Header Section

At compile-time, wrap script inserts the `header` section into the generated header file before declaring the class. Use this section to include declarations for required types, most notably the declaration of the foundation class.

The Source Section

At compile-time, the wrap script inserts the `source` section into the generated source file before defining the methods of the generated class. Use this file to include header files for classes used in the implementation. The wrap script checks that there is an include statement that appears to include the generated header file, and exits with an error message if it does not find this include statement. For instance, in our stack example it checks that there is an `include` statement that looks something like `#include <...DSintStackRO.h>,` where the ellipsis represents any (or no) directory specification.

Code Fragments

You may need to insert code sections in various places, as described in this section. To do so, create a section called `codeFragments` as shown in the example in **Section B.2.2, “The Wrap File: an Example” page 272**. Legal names for these sections include:

constructor and preConstructor

The wrap script inserts code in the constructor fragment into the C++ constructor for the wrapped object. The code looks like the following:

```
wrappedClassNameRO(const MCprogramObjectRP& pParent,
      const MCinstanceRP& pDerived= gkpNullInstPtr)
      mpWrapped( 0 )
 {
   // ...contents of “preConstructor” fragment here...
   // ...contents of “constructor” fragment here...
 }
```

constructorFromFoundation

The wrap script provides a constructor for the wrapped class when a `constructorFromFoun-dation` fragment is present. The code looks like the following:

```
wrappedClassNameRO(WRAP_FOUNDATION_PTR pWrapped,
      const MCprogramObjectRP& pParent,
      const MCinstanceRP& pDerived=gkpNullInstPtr)
 {
   mpWrapped = pWrapped;
   // ...contents of preConstructor fragment here...
   // ...contents of constructorFromFoundation fragment here...
 }
```

wrappedIsReady

The contents of `wrappedIsReady` goes in a fragment similar to:

```
 void wrappedIsReady() { /* fragment code here */ }
```

A wrapped object should call the `wrappedIsReady()` function when the foundation object it wraps is ready. This is very useful in the case of wrapper inheritance, further discussed in **Section B.2, “Inheritance Model of the Wrap Script” page 280**.

preDestroy and postDestroy

If code fragments for `preDestroy` and `postDestroy` exist, their contents go in methods similar to:

```
void preDestroy()     { /* fragment code here */ }
void postDestroy()    { /* fragment code here */ }
```

The wrapped object’s Destroy method then has the following body:[3]

```
{
preDestroy();            //only inserted if preDestroy exists
localDestroy();           //only inserted if localDestroy exists
mpWrapped = NULL;
postDestroy();           //only inserted if postDestroy exists
MCcxxObjectRO::Destroy();//normal upchained Destroy call
}
```

Members

Use the members section to introduce members in the generated class. See **Section B.2.2, “The Wrap File: an Example” page 272** for the syntax for introducing members.

[3] The Destroy method is part of the Reference Counting mechanism and is beyond the scope of this chapter.

The members of a wrapped class have four protection levels: *private*, *protected*, and *public* in C++, and *published* in the wrapper. The wrap script prepares members in the published category, making them available to the *ADL* run-time engine. You can separate member descriptions into protection levels by placing the protection level name followed by a colon, such as

```
private:
```

on a separate line. All subsequent lines until the next protection level designation or the final delimiter (`}`) are given that protection level. The wrap script does not interpret lines with a C++ protection level of private, protected, or public. They go directly into a similarly protected section of the generated header.

Published members appear to the *ADL* as instance members of the published class. There are two generic operations performed on simple members: `get` and `set`. Use the `get` operation when an *ADL* program needs the value of a member, and the `set` operation when the program assigns a value to the member. The wrap script allows you to specify how to handle `get` and `set` operations on each published member: by using the default accessor, by using a custom accessor, or by preventing access.

Published members have a special format, as shown in the following example:

```
memberName
{
   get: accessStyle
   set: accessStyle
}
```

Put each statement on a line by itself and designate the access style to be one of the following: `default`, `custom`, or `none`.

If a member's access style is `default`, the wrap script automatically provides the generated class with a member of type `MClvalue` for storing that value. The member's name is its published name preceded by an `m`. In our example, the wrap script would provide the wrapped class with a member named `mname`.

Sometimes a member doesn't really exist as a member of the wrapped class. For instance, in our example, the foundation IntStack class already maintains the height, so it would be wasteful for us to maintain that information in the wrapped class too. For cases such as this, the wrap script allows you to specify a custom access method for a member. If you specify a custom `get` statement for a member, you must provide a method with the following signature, where *memberName* represents the appropriate member's name:

```
UTvalue _Get_memberName() const
```

In our example, we provide a method called `_Get_height` that asks its foundation object for the current height. If you specify a custom `set method`, you must provide a method with the following signature:

```
void _Set_memberName( const UTvalue& newVal )
```

If it made sense to set the height of a stack, you could specify that the `set` accessor of `height` be `custom` and provide a method named `_Set_height`.

You can tell the wrap script to specify an access style of `none` to handle the case where you do not want to allow `get` and `set` operations on members. In our example, `height` has a `set` access method of `none`, causing a fatal error for in any *ADL* program that attempts to set it.

What methods should the wrapped class have?

Use the methods section to describe the methods, or operations, for the wrapped class. See **Section B.2.2, "The Wrap File: an Example" page 272** for the syntax for specifying methods. The methods of a wrapped class have the same four protection levels as members: *private*, *protected*, and *public* in C++, and *published* in the wrapper. The wrap script prepares methods in the published category, making them available to the *ADL* run-time engine.

Each of the protection levels is optional and contains method descriptions. A method description consists of the method's C++ signature, followed by a block of uninterpreted C++ code (set off by === delimiters). The wrap script extracts the signatures and places them in the proper protection level section of the generated header, and places the definition in the generated source file. If you specify the keyword `inline`, the wrap script places the body of the method in the header too. Specify the keyword `ctor` to mark an *ADL* constructor. Marking a method with `ctor` is equivalent to using the keyword `upon` in the *ADL*. The keyword `local` marks a method as it does in the *ADL*. *ADL* subclasses of the wrapped class do not inherit the method, but wrapped subclasses do.

The wrap script exports methods listed in the published section to the *ADL* as methods with a corresponding name and *ADL* signature. The following types are the only legal arguments and return types for published methods:

C++ type	*ADL type*
Eboolean	boolean
integer_t	integer
UThandle	handle
UTinterval	interval
UTlist	list
real_t	real
UTstring	string
UTtime	time
UTvalue	any
UvalueType	vtype

Figure B.1 Arguments and Return Types for Published Methods

As a special case, you can also have the return type `void`. It is equivalent to not declaring a return type in the *ADL*.

B.2 Inheritance Model of the Wrap Script

The wrap script implements inheritance in a manner analogous to the way it is done by hand — it essentially copies-and-pastes inherited descriptions into derived wrap descriptions. In this section we explain the copying and pasting, the restrictions on creating a subclass, and the hooks that we provide to allow wrap descriptions to work when pasted into subclasses.

Frequently, the foundation classes that we want to wrap have an inheritance relationship. Let's suppose that we want to wrap a class called PrintableIntStack. This new class is a subclass of the IntStack class that we discuss in the example in Section 1.2.2, and it adds the method `getPrintString` to its interface, which returns a string that describes the stack.

Using only the features described in the previous section, we would have to create a new wrap file that includes all of the wrap description of the existing DSintStack wrapped class referred to in Section 1.1, but which has PrintableIntStack as its foundation class and `getPrintString` as a published method. As you might expect, this leads to maintenance nightmares, especially since wrapped classes often have more methods and members than DSintStack does.

Let's look at this example two ways: first without inheritance and then with inheritance.

B.2.1 Example without Inheritance

```
notice
=====================================================================
/*
 *  $Header: /mit/ceci/1/aybee/devl/doc/firstRel/RCS/
DSprintableIntStackROsansInheritance.wrp,v 1.1 1995/06/02 21:23:23 aybee Exp
aybee $
 *
 * Copyright 1993, 1994, 1995 Massachusetts Institute of Technology.
 * All rights reserved.
 * AthenaMuse is a registered trademark of the Massachusetts
 * Institute of Technology.
 */
=====================================================================
foundation: IntStack
wrapped:    DSprintableIntStack
module:     ExampleModule
abstract:   false

header
====================================================================
#include <PrintableIntStack.h>
/*
 * PrintableIntStack.h hypothetically includes something like...
 *
 * #include <PrintableIntStack.h>
 *
 *   class PrintableIntStack: public IntStack
 *   {
 *     public:
 *       char* getPrintString();
 *    };
 *
 */
======================================================================

source
======================================================================
#include <control/EXmodule.h>
#include <control/DSprintableIntStackRO.h>
#include <adl/ERsemantic.h>
======================================================================

codeFragments
{
constructor
======================================================================
mpWrapped = new PrintableIntStack;
======================================================================
}



members
{
private:


published:
  name
  {
    get: default
    set: default
  }
  height
  {
    get: custom
    set: none
  }
}

methods
{
private:
  void localDestroy()
  ======================================================================
  if (mpWrapped)
     delete mpWrapped;
  ======================================================================

  UTvalue _Get_height() const
  ======================================================================
  return mpWrapped->getHeight();
  ======================================================================

published:
  void Push( integer_t newTop )
  ======================================================================
  /* effects: puts newTop on the top of the stack */
  mpWrapped->push( newTop );
  ======================================================================

  integer_t Pop()
  ======================================================================
  /* returns: removes top value from the stack and returns it */
  assert( ! mpWrapped->isEmpty() );
  return mpWrapped->pop();
  ======================================================================

  UTstring GetPrintString( )
  ======================================================================
  /* effects: puts newTop on the top of the stack */
  return mpWrapped->GetPrintString();
  ======================================================================
}
```

B.2.2 Example with Inheritance

```
notice
======================================================================
/*
 *
 *  $Header: /mit/ceci/1/aybee/devl/doc/firstRel/RCS/DSprintableInt-
StackRO.wrp,v 1.1 1995/06/02 21:23:23 aybee Exp aybee $
 *
 * Copyright 1993, 1994, 1995 Massachusetts Institute of Technology.
 * All rights reserved.
 * AthenaMuse is a registered trademark of the Massachusetts
 * Institute of Technology.
 */
======================================================================

foundation: PrintableIntStack
wrapped:    DSprintableIntStack
module:     ExampleModule
abstract:   false

header
======================================================================
#include <PrintableIntStack.h>
/*
 * PrintableIntStack.h hypothetically includes something like...
 *
 * #include <PrintableIntStack.h>
 *
 *   class PrintableIntStack: public IntStack
 *   {
 *     public:
 *       char* getPrintString();
 *    };
 *
 */
======================================================================

source
======================================================================
#include <control/EXmodule.h>
#include <control/DSprintableIntStackRO.h>
#include <adl/ERsemantic.h>
======================================================================

superclasses
{
  DSintStack
}

codeFragments
{
constructor
======================================================================
mpWrapped = new PrintableIntStack;
======================================================================
}

methods
{
published:
  UTstring GetPrintString( )
  ======================================================================
  /* effects: puts newTop on the top of the stack */
  return mpWrapped->GetPrintString();
  ======================================================================
}
```

B.2.3 An Explanation of the Wrap Inheritance Description

For a discussion of the notice, wrapped, module, and abstract fields, see **Section B.2.3, “An Explanation of the Parts of a Wrap File” page 274**.

Superclasses

This section of the description lists the wrap descriptions that are superclasses for this class. See **Section B.2.2, "Example with Inheritance" page 282** for the format for specifying superclasses. Each line between the brackets should contain the name of a superclass description.

The wrap script uses each superclass name listed to find the corresponding description. It looks for a file named "superclassNameRO.wrp" somewhere in the wrap path. In our example, the wrapper looks for `DSintStackRO.wrp`. The wrapper ensures that the *ADL* run-time engine thinks of each listed class as a superclass of the given class.

Foundation: and FoundationRO:

To allow descriptions to inherit the code from their superclass descriptions, it is necessary that all code that manipulates the `mpWrapped` member work with the `mpWrapped` member of the subclass as well. For this reason, a wrap description can inherit only from a wrap description whose foundation class is a void or a C++ superclass of its foundation class. This provides the required C++ type safety since the wrap script is not intelligent enough to check this relationship at wrap-time. It generates code that attempts to cast from a NULL pointer to the subclass's WRAP_FOUNDATION to a pointer to the superclasses's WRAP_FOUNDATION. If the foundation classes do not have the proper relationship, this code intentionally causes a compile-time error; if all goes well, the compiler lists the error as occurring on the line containing the foundation declaration.

For example, DSprintableIntStack is a subclass of DSintStack. Their foundation classes are PrintableIntStack and IntStack, respectively, so PrintableStack must be a subclass of IntStack. The code that the wrap script generates to check the pointer types is as follows:

```
        (IntStack*) tempPtr = (PrintableIntStack*) NULL;
```

Header and Source Sections

Subclasses inherit the contents of the header and source sections of their superclasses by concatenation. These sections appear in an order that ensures that a class' header section does not appear until after the header sections of its superclasses.[4] We refer to this as the *proper* order.

Members

A subclass inherits members. The wrap script places the contents of the member section of a superclass of a wrapped class in the subclass in the proper order. Subclasses cannot have members with the same name as their superclasses' members. The wrap script detects duplicate published member names, but since the wrapper does not interpret the contents of the C++ member sections, duplicates in those sections are not be caught until compile-time. Note that members are placed at the same protection level as in the superclass and are not moved to another level as in C++ subclasses. That is, inherited private members are listed as private members of the subclass and *are* accessible to that class.

[4] The ActivityManager class is an exception. To work around a bug in the HP compiler having to do with templates and the order in which they and their declarations are seen, the source section of the ActivityManager class is always placed last in its subclasses.

Methods

A subclasses inherits methods, and it is possible to override inherited methods. Providing a method with the same name as a method provided by a superclass overrides that method. Just as in the *ADL*, methods are inherited in a depth-first fashion. Unlike the *ADL*, local methods (such as *ADL* constructors) *are* inherited. This is to encourage wrap class writers to provide a consistent *ADL* constructor interface.[5]

The contents of `constructor` and `constructorFromFoundation` are not inherited. This is in keeping with the C++ treatment of constructors, and it is partly attributable to the special meaning of the `mpWrapped` member. No matter how many wrapper descriptions a class inherits from, it has only one `mpWrapped` member. The wrap script provides the `preConstructor` and `wrap-pedIsReady` fragments (which are inherited by concatenation in the proper order) to allow a superclass to initialize its members before and after `mpWrapped` is constructed and ready for operation.

Useful Details

In addition to defining WRAP_FOUNDATION and WRAP_FOUNDATION_PTR, as described above, the wrapper provides WRAP_IS_ABSTRACT and WRAP_IS_ADLSUBCLASSABLE, which are either 0 (false) or 1 (true) depending on whether or not the wrapped class is abstract and adlSubclassable respectively.

In an effort to cut down on the size of wrapped classes, the wrapper does not completely fill out the class description for abstract, non-adlSubclassable classes since no instances of them will ever be created. We call those ANAS classes. In addition, it does not include all of the inherited header and source declarations. Unfortunately, hidden in the source sections are the includes of the generated header file and the include of the module in which the class is generated. Instead of including the whole source section to provide the correct include statements, the wrapper uses some heuristics to find and include only the needed include directives.

It is possible that the wrap script's attempted short cuts will be too effective and break something. The optional wrap directive `ANAScuts: boolean` keeps the wrapper from taking short cuts in any class for which it is set to `true,` as well as in all of its subclasses. This feature is provided *only* as an escape hatch in the unlikely event that the heuristics fail. If you have to use this escape hatch, please let us know.

5 It also a big win in large wrapped hierarchies, such as the XFwidgets, where we do not have to respecify the ADL constructor over and over again.

B.3 The Wrap Script and the Macintosh

The wrapper generates code for the Macintosh. This involves three things:

1) `#pragma once` goes in the generated .h file as follows:

```
       #ifdef macintosh
       #pragma once
       #endif // macintosh
```

2) The three random code blocks (`notice`, `source`, and `header)` go in the code in the following template:

```
   #ifdef macintosh
   macSection
   #else  // macintosh
   normalSection
   #endif // macintosh
```

The wrap script writes `normalSection` in the `.wrp` file, and treats `macSection` the same, with the include statements corrected for the Macintosh. This correction involves dropping all directory specifications in the lines of the include statement, except for those of the Rogue Wave™ class library. For example:

```
   #include <utils/UTtypes.h>    becomes  #include <UTtypes.h>
   #include <UTtypes.h>          remains  #include <UTtypes.h>
   #include <rw/cstring.h>       remains  #include <rw/cstring.h>
```

3) The wrap script adds `#pragma segment` commands to the generated .`cc` file. The default segment for all methods in the file is the same as the name of the wrapped class. The wrapper adds the following to the top of each generated `.cc` file (with the correct name substituted for XFbuttonRO).

```
   #ifdef macintosh
   #pragma segment XFbuttonRO
   #endif // macintosh
```

To handle future optimization on the Macintosh, you can specify a segment for each method by preceding the signature with the desired name enclosed in square brackets. For example:

```
[yourSegmentNameHere] UThandle requestContainer(UThandle hParent)
```

Note that the segment names set this way have effect only for that particular method.

B.3.1 Command Line Interface

The following is an example of using the wrapper. The command line:

```
$(TOOLS_DIR)/wrap $(WRAPFLAGS) XFbuttonRO.wrp $(WRAP_H_DIR) $(WRAP_CC_DIR)
```

causes the wrapper to generate:

```
$(WRAP_H_DIR)/XFbuttonRO.h
$(WRAP_CC_DIR)/XFbuttonRO.cc
```

The wrapper exits with zero status if it completes successfully, and with a non-zero status if there is a problem. The remaining sections cover the legal flags for the wrap script: `+depend`, `-l`, `-mac`, and `+Wdirectory.`

+depend

The wrapper does not generate its usual C++ header and source files when given the `+depend` flag. Instead, it outputs make-style dependencies for the source file it would have generated. For instance, the wrapper generates the following dependencies for the generated XFbuttonRO.cc file.

```
   ./XFbuttonRO.cc: ../../generic/src/ActivityManagerRO.wrp
   ./XFbuttonRO.cc: ../../generic/src/AttributeManagerRO.wrp
   ./XFbuttonRO.cc: ./XFwidgetRO.wrp
   ./XFbuttonRO.cc: ./XFcontainableRO.wrp
   ./XFbuttonRO.cc: ./XFsimpleRO.wrp
   ./XFbuttonRO.cc: ./XFfontableRO.wrp
   ./XFbuttonRO.cc: ./XFlabelRO.wrp
   ./XFbuttonRO.cc: XFbuttonRO.wrp
```

-l

The wrap script attempts to trick the compiler into reporting errors related to the `.wrp` file, although this mechanism is not always perfect. If you really cannot see the problem at a line, you may use the `-l` flag to turn off line numbering. With the `-l` flag, the compiler reports errors where it thinks they occur in the generated `.h` and `.cc` files. Note that you have to rewrap the file for the this flag to be useful.

-mac

By default, the wrapper creates code for the Macintosh platform. Using the `-mac` flag suppresses this behavior in case you ever have to read the generated files.

+Wdirectory

The `+Wdirectory` argument appends `directory` to the search path that the wrap scripts uses when it tries to find files describing superclass descriptions. It is the analog of the `-I` flag used by C/C++ compilers.

Index