The Amulet Reference Manuals

Brad A. Myers
Rich McDaniel, Alan Ferrency, Andy Mickish,
Alex Klimovitski, Amy McGovern

June, 1995
CMU-CS-95-166
CMU-HCII-95-102

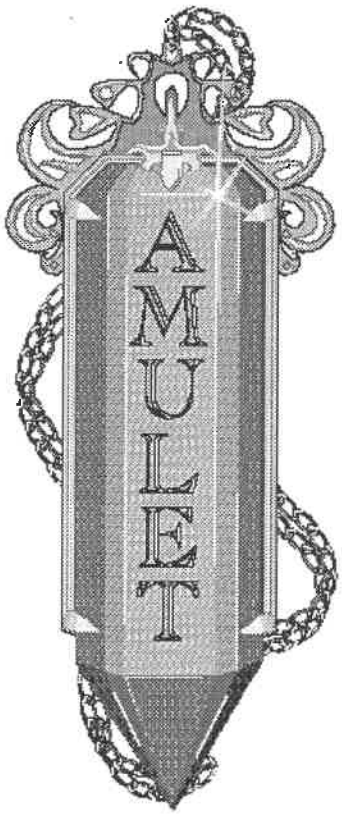
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

The Amulet User Interface Development Environment contains a comprehensive set of tools that make it significantly easier to design and implement highly interactive, graphical, direct manipulation user interfaces. Applications implemented in Amulet will run without modification on both Unix and PC platforms. Amulet provides a high level of support, while still being Look-and-Feel independent and providing applications with tremendous flexibility. Amulet currently provides a low-level toolkit layer, which is an object-oriented, constraint-based graphical system that allows properties of graphical objects to be specified in a simple, declarative manner, and then maintained automatically by the system. The dynamic, interactive behavior of the objects can be specified separately by attaching high-level "interactor" objects to the graphics. Higher-level tools are currently in production, which will allow user interfaces to be layed out without programming.

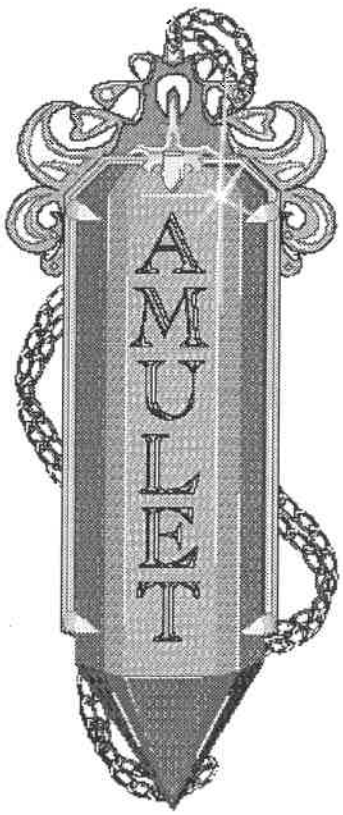
The Amulet toolkit is available for unlimited distribution by anonymous FTP. Amulet uses X/11 on Unix-based systems and the native Windows NT graphics on PC's. This document contains an overview with downloading and installation instructions, a tutorial, and a full set of reference manuals for the Amulet system.

This research was sponsored by NCCOSC under Contract No. N66001-94-C-6037, Arpa Order No. B326. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NCCOSC or the U.S. Government.



Keywords: User Interface Development Environments, User Interface Management Systems, Constraints, Prototype-Instance Object System, Widgets, Object-Oriented Programming, Direct Manipulation, Input/Output, Amulet, Garnet.

TABLE OF CONTENTS



1. Amulet Overview

This section provides an overview of Amulet, and contains retrieval and installation instructions.

This research was sponsored by NCCOSC under Contract No. N66001-94-C-6037, Arpa Order No. B326. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NCCOSC or the U.S. Government.

1.1 Introduction

The Amulet research project in the School of Computer Science at Carnegie Mellon University is creating a comprehensive set of tools which make it significantly easier to create graphical, highly-interactive user interfaces. The lower levels of Amulet are called the "Amulet Toolkit," and these provide mechanisms that allow programmers to code user interfaces much more easily. Amulet stands for Automatic Manufacture of Usable and Learnable Editors and Toolkits.

This manual describes version 1.0 of Amulet.

Amulet is written in C++ and can be used with either Unix systems running X Windows or PC's running Windows NT. Therefore, Amulet is quite portable to various environments. Computers currently running Amulet include Suns, HPs and PCs.

Amulet is being actively developed with these compilers:

- gcc 2.6.3
- ObjectCenter 2.1.0
- Visual C++ 2.0

There is a known bug in the gcc 2.5.8 compiler involving premature destruction of local variables that prevents it from being able to compile Amulet.

Amulet provides support for color and gray-scale displays. Amulet runs on X/11 R4 through R6, using any window manager such as mwm, uwm, twm, etc. It does not use any X toolkit (such as Xtk or TCL). It has also been implemented using the native Windows graphics system on the PC. Because of stack-size limitations on the 16-bit Windows operating system, Amulet requires PC users to run Windows NT or Windows 95.

More details about Amulet are available in the Amulet home page on the World Wide Web:

```
http://www.cs.cmu.edu/Web/Groups/amulet/amulet-home.html
```

which is updated periodically.

A previous project named Garnet was developed by the same people who are now writing Amulet. It has features similar to those in Amulet, but is implemented in Lisp. For information about Garnet, please refer to the Garnet home page:

```
http://www.cs.cmu.edu/Web/Groups/garnet/garnet-home.html
```

1.2 Amulet Mailing List

There is a mailing list called amulet-users@cs.cmu.edu where users and developers exchange information about Amulet. Topics include user questions and software releases. To be added to the list, please send your request to amulet-users-request@cs.cmu.edu.

Please send questions about installing Amulet to amulet@cs.cmu.edu.

You can also send bug reports directly to amulet-bugs@cs.cmu.edu. This mail is read only by the Amulet developers.

Another mailing list, amulet-interest@cs.cmu.edu, is available for people who are interested in learning only about new releases of Amulet.

1.3 How to Retrieve and Install Amulet

Amulet is available for free by anonymous FTP. It is in the public domain, so there are no licensing restrictions.

There are different instructions for obtaining the software depending on whether it will be installed on a PC or a Unix system. You will get only PC-specific or Unix-specific files, depending on which package you download.

These instructions assume that a C++ compiler such as gcc, CC, or Visual C++ has been installed properly on your system, and you know the location of your window manager libraries, etc. *Amulet is known to not compile in gcc 2.5.8, due to a compiler bug.*

1.3.1 The Amulet Manual

The Amulet manual is distributed separately from the Amulet source code. It is available in raw postscript format, or compressed using zip (for the PC) or compress (for UNIX). To download the Amulet manual, FTP to ftp.cs.cmu.edu (128.2.206.173) and login as "anonymous" with your e-mail address as the password. Type "cd /usr0/anon/project/amulet/amulet" (note the double amulet's). Do not type a trailing "/" in the directory name, and do not try to cd there in multiple steps, since the intermediate directories are probably protected from anonymous access.

Set the mode of your FTP connection to binary: Some versions of FTP require you to type "binary" at the prompt, and others require something like "set mode binary".

At the "ftp>" prompt, get the manual file you require using the get command: "ammanual.ps" is raw postscript, "ammanual.zip" is PC zip format, and "ammanual.Z" is UNIX compress format.

1.3.2 Installation on a PC

1.3.2.1 Retrieving and Unpacking the ZIP Files

The source code and documentation for Amulet 1.0 is available in `ZIP` files that make it easy to install Amulet on a PC. To download the Amulet `ZIP` files, FTP to `ftp.cs.cmu.edu` (128.2.206.173) and login as "anonymous" with your e-mail address as the password.

Type "`cd /usr0/anon/project/amulet/amulet`" (note the double `amulet`'s). Do not type a trailing "`/`" in the name of the directory, and do not try to `cd` there in multiple steps, since the intermediate directories are probably protected from anonymous access.

Set the mode of your FTP connection to binary: Some versions of FTP require you to type "`binary`" at the prompt, and others require something like "`set mode binary`".

At the "`ftp>`" prompt, do "`get amulet.zip`". This will retrieve a file that was prepared with the command `zip`, which can easily be expanded on a PC.

Quit FTP, and unzip `amulet.zip` into the directory where you want it to reside, preserving the directory structure. For example, if you use `pkunzip` and want to have Amulet files in the `C:\AMULET` directory, type at the DOS prompt "`pkunzip -d amulet.zip C:\`".

It is easiest to use Amulet in the base directory `\AMULET`. The turotial and installation instructions assume you installed amulet in `C:\AMULET`. The provided `.mak` files expect to find all your source files in that directory structure. If you place your Amulet directory somewhere else, you will need to change all the makefiles to find your source code elsewhere. The easiest but most tedious way to do this is to use Visual C++ to remove all the old files from the project, and add all the new ones.

People familiar with PC nmake files may directly edit the `.mak` files to specify the location of their files. This is somewhat dangerous, because changing the wrong parts of the makefile may confuse Visual C++ and cause it to reject the file. To manually edit the makefile, use your favorite editor to search and replace all occurances of `C:\amulet` with your preferred pathname (for example, `D:\blah\blah\amulet`).

Next you should set the environment variable `AMULET_DIR` to the directory where you installed Amulet. Go to the Program Manager, and open the Control Panel. Choose System, and add `AMULET_DIR = C:\amulet` (substitute the appropriate pathname) to the User Environment Variables section. Amulet uses this environment variable to look for its dynamic link libraries, as well as some bitmap files in the demo programs. If the `AMULET_DIR` is not set, Amulet looks in `C:\amulet` by default.

Change to the directory C:\AMULET\BIN\ and launch the program GWSTREAM.EXE, then close it again. This will register the program GWStreams with the Windows program database. (This program will be used if you run the Amulet tests, discussed below.)

1.3.2.2 Configuring Visual C++

In the Visual C++ menubar, choose "Tools", "Options", and "Directories" to access the search paths. Add the Amulet include directory C:\AMULET\INCLUDE to the include path, and add the Amulet library directory C:\AMULET\LIB to the library path. (If you installed Amulet in some other directory, be sure to specify that directory instead.)

1.3.2.3 The Amulet Library Files

The PC distribution comes with a compiled version of the Amulet library files amulet.lib and amuletd.lib. These are located in the directory AMULET\LIB. You can use these libraries without recompiling them. The following paragraphs describe how to compile the Amulet library if you want to test your Amulet installation, or if you ever need to recompile the library..

Launch the Visual C++ application, and open the Visual C++ project file C:\AMULET\BIN\AMULET.MAK. You can do this all in one step by double-clicking on the AMULET.MAK file from the File Manager.

Choose the Build All option. Compiling Amulet may generate over 500 warnings in Visual C++, but there should not be any errors.

AMULET.MAK generates either AMULETD.LIB for a "debugging" version of the Amulet library file, or AMULET.LIB for a release build. Choose the version to compile in the "Targets" choice of the "Project" menu. The standard Amulet library does not include the debugger file inspectr.cpp, since this is not needed in all Amulet applications. To use the inspector, you should include this file in your project explicitly.

Once you have generated either AMULETD.LIB or AMULET.LIB, you are ready to write your own Amulet programs and link them to the Amulet library. Section 1.3.2.4 discusses how you can build and run some of the Amulet demos and test programs. Your first experience with Amulet programming should involve the Amulet Tutorial, which includes a starter program and instructions to acquaint you with the compiling process. When you are ready to write your own Amulet program, see the PC-specific Section 1.3.2.6 below for instructions about linking your new program to the Amulet library.

1.3.2.4 Compiling Test Programs and Demos

There are about 10 test programs included in Amulet that test the lower levels of Amulet: the ORE object system, GEM graphics routines, OPAL graphical objects, Interactors event handlers, and Widgets. The .MAK files for these tests appear in the BIN\ directory. While you can build and run these programs to test your installation of Amulet, they are not intended to be particularly good examples of Amulet coding style. The test programs

use the GWStreams program, discussed below. The makefiles for these programs assume you've installed Amulet in \AMULET, so if you want to try these programs and have installed Amulet elsewhere, you'll have to change the makefiles.

There are also some example programs in AMULET\SAMPLES*. Executables are provided in the PC distribution for each of these programs. These programs are more like you would write as an actual Amulet user, and are intended to be exemplary code. Each of these programs have their own .MAK file in their subdirectory, and depend on the Amulet library file AMULET.LIB (see above). To regenerate a binary for any of these files, just open the project file for the program, located in its AMULET\SAMPLES* subdirectory, and build it.

See Section 1.4 for instructions and discussions about these demos and test programs.

1.3.2.5 Using GWStreams to Simulate a Terminal Window

The Amulet test programs were designed in a Unix environment, and require a simulated terminal window for some output. The special utility GWStreams provides this terminal window, and is called from the file GWSTREAM.EXE during execution of a test program. When a test program asks you to type a character to continue, you will need to type into the terminal window. Because GWStreams is not a real Unix terminal, you may need to input something (just pressing Enter is enough) even if an Amulet sample program says "Press Ctrl-D". If you encounter problems with too much tracing output (which could cause some Amulet samples to crash on slower PCs), simply lock the output stream using GWStreams controls or close GWStreams or its "STDIO" document window. You can later unlock output or relaunch GWStreams. Output will be resumed.

1.3.2.6 Writing and Compiling New Programs Using Amulet

When you are ready to build your first Amulet program, you will need to set up your Visual C++ project as follows:

- Create a new Visual C++ non-MFC project: In "General Settings" for the project, choose "Not using MFC".
- Add your program to the project.
- Make sure the Amulet include and library paths are set in Visual C++, as discussed in Section 1.3.2.3.
- In settings for the C/C++ preprocessor, define NEED_BOOL and SHORT_NAMES. Also define _MSC_VER and _WINDOWS if they are not already defined.
- In settings for the linker input:
    - Add the AMULETD.LIB library if you want to use the "debugging" version, or AMULET.LIB for a streamlined version.
    - If you want your program to handle terminal-style I/O, add the GWSTRMD.LIB library for the "debugging" version, or GWSTRM.LIB for a streamlined version. If your program does not input any data from the terminal, and doesn't need any terminal output, use the GWSTRM_D.LIB or GWSTRM_.LIB libraries instead.

Now you are ready to build your project.

1.3.2.7 PC filenames

For historical and practical reasons, we are still using 8 character file names for the Windows NT Amulet files. This manual generally refers to UNIX or machine independant file names. The PC filenames are logically shortened versions of the UNIX filenames. The following table describes the correspondence between PC and UNIX filenames.

UNIX filename	PC filename
priority_list.h	prty_lst.h
standard_slots.h	std_slot.h
symbol_table.h	symb_tbl.h
value_list.h	val_lst.h
object_advanced.h	object_a.h
formula_advanced.h	form_a.h
opal_advanced.h	opal_a.h
inter_advanced.h	inter_a.h
widgets_advanced.h	widgts_a.h
priority_list.cc	prty_lst.cpp
standard_slots.cc	std_slot.cpp
symbol_table.cc	symb_tbl.cpp
value_list.cc	val_lst.cpp
testobject.cc	testobj.cpp
testcolor.cc	testcolr.cpp
test_utils.h	testutil.h
test_utils.cc	testutil.cpp
testinput.cc	testinpt.cpp
testlineprops.cc	testline.cpp
testwinprops.cc	testwinp.cpp
testwisizes.cc	testwins.cpp
testpoints.cc	testpts.cpp
testsubwins.cc	testsubw.cpp
testtrans.cc	testtran.cpp
testMS1.cc	testms1.cpp
testfonts.cc	testfont.cpp
testlines.cc	testline.cpp
command_basics.cc	cmnd_bas.cpp
inter_basics.cc	intr_bas.cpp
inter_choice.cc	intr_chc.cpp
inter_move_grow.cc	intr_mvg.cpp
inter_new_points.cc	intr_npt.cpp
inter_text.cc	intr_txt.cpp
testinter.cc	testintr.cpp
inspector.cc	inspectr.cpp
button_widgets.cc	btn_wdgt.cpp
scroll_widgets.cc	scl_wdgt.cpp
testwidgets.cc	testwidg.cpp
text_widgets.cc	text_wid.cpp
goodbye_button.cc	bye_butn.cpp
goodbye_inter.cc	bye_intr.cpp

1.3.3 Installation in Unix

1.3.3.1 Retrieving and Unpacking the TAR Files

The source code and documentation for Amulet 1.0 is available in compressed tar files that make it easy to install Amulet on a Unix machine. To download the Amulet tar files, FTP to `ftp.cs.cmu.edu` (128.2.206.173) and login as "anonymous" with your e-mail address as the password.

Type "`cd /usr0/anon/project/amulet/amulet`" (note the double `amulet`'s). Do not type a trailing "`/`" in the name of the directory, and do not try to `cd` there in multiple steps, since the intermediate directories are probably protected from anonymous access.

Set the mode of your FTP connection to binary: Some versions of FTP require you to type "`binary`" at the prompt, and others require something like "`set mode binary`".

At the "`ftp>`" prompt, do "`get amulet.tar.Z`". This will retrieve a compressed tar file that can easily be expanded on a Unix system.

Quit FTP, and type "`uncompress amulet.tar.Z`" and "`tar -vxf amulet.tar`" at the Unix prompt to generate the `amulet/` directory tree.

1.3.3.2 Setting your Environment Variables

The Amulet Makefiles have been written so that all Amulet users must set two environment variables in their Unix shell before they can compile any program. Consistent binding of these variables by all Amulet users ensures that the installed Amulet binaries will always be compatible with user programs. Once Amulet has been installed and programs are being written that only depend on its library file, it would be possible for users to write their own Makefiles without regard to these variables. However, we recommend that all Amulet users at a site continue to use consistent values of environment variables to facilitate upgrading to future versions of the system.

There are two environment variables that all Amulet users must set in their Unix shell before they can compile any program. Typically, these will be set in your `.login` file:

- `AMULET_DIR` -- Set this to the root directory of the Amulet software hierarchy. For example, "`setenv AMULET_DIR /usr/amickish/amulet`".

- `AMULET_VARS_FILE` -- Set this to the particular `Makefile.vars.*` file appropriate for your compiler and machine. This file will be included by the main Amulet `Makefile`. Try compiling Amulet with this variable set to one of the pre-defined files first. If the compilation does not finish smoothly, you probably need to make changes to the variables in `Makefile.vars.custom` -- see instructions below, under "Customizing the Makefile Variables".

- **If you are at CMU**, then there may be a sub-makefile already defined that is appropriate for you. Set your `AMULET_VARS_FILE` environment variable to one of the following values:

```
Makefile.vars.gcc.Sun   -- For gcc on a Sun
Makefile.vars.gcc.HP    -- For gcc on an HP
Makefile.vars.CC.Sun    -- For ObjectCenter's CC on a Sun
Makefile.vars.CC.HP     -- For ObjectCenter's CC on an HP
```

  For example, your `.login` might include the lines

```
setenv AMULET_DIR        /afs/cs/project/amickish/work/amulet
setenv AMULET_VARS_FILE  Makefile.vars.CC.HP
```

- **If you are <u>not</u> at CMU**, and none of the above `Makefile.vars.*` files are appropriate, then you should set your `AMULET_VARS_FILE` variable to this file:

```
Makefile.vars.custom    -- For any other configuration
```

  For example, your `.login` might include the lines

```
setenv AMULET_DIR        /usr/amickish/amulet
setenv AMULET_VARS_FILE  Makefile.vars.custom
```

  Only edit `amulet/bin/Makefile.vars.custom` while installing Amulet. See "Customizing the Makefile.vars.custom Variables" below for a guide to the Makefile variables.

1.3.3.3 Generating the Amulet Library File

After you have set your environment variables, `cd` into the `bin/` directory and invoke `make`, with no arguments. This generates many object files, and eventually `libamulet.a` will be deposited in the `lib/` directory.

If the compile procedure is interrupted by an error, you probably need to customize the Makefile variables for your platform. Set your `AMULET_VARS_FILE` environment variable to `Makefile.vars.custom`, and refer to Section 1.3.3.6. Change some of the switches in `Makefile.vars.custom`, and recompile. If you are unable to compile Amulet after trying different combinations of compiler switches, please send mail to `amulet-bugs@ cs.cmu.edu` and we will try to make the Amulet code more portable.

Once you have generated `libamulet.a`, you are ready to write your own Amulet programs and link them to the Amulet library. Your first experience with Amulet programming should involve the Amulet Tutorial, which includes a starter program and instructions to acquaint you with the compiling process. When you are ready to write your own Amulet program, see the Unix-specific Section 1.3.3.5 below for instructions about linking your new program to the Amulet library.

1.3.3.4 Compiling Test Programs and Examples

From the `bin/` directory, doing "`make all`" generates about 10 executable binaries that test the lower levels of Amulet: ORE object system, GEM graphics routines, OPAL graphical objects, Interactors event handlers, and Widgets. You can run these programs directly, such as "`./testgem`". While you can build and run these programs to test your installation of Amulet, they are not intended to be particularly good examples of Amulet coding style.

There are also some example programs in `amulet/samples/*`. These programs are more like you would write as an actual Amulet user, and are intended to be exemplary code. Each of these programs have their own Makefile in their subdirectory, and depend on the Amulet library file `libamulet.a` (see above). To generate binaries for these files, just `cd` into their subdirectory and invoke `make` with no parameters.

See Section 1.4 for instructions and discussions about these demos and test programs.

1.3.3.5 Writing and Compiling New Programs Using Amulet

It is important to set your `AMULET_DIR` and `AMULET_VARS_FILE` environment variables, and to retain the structure of the sample Makefiles in your local version. By keeping the line "`include $(AMULET_DIR)/bin/Makefile.vars`" at the top of your Makefile, and continuing to reference the Amulet Makefile variables such as `FLAGS` and `CC`, you will be assured of generating binary files compatible with the Amulet libraries.

When you are ready to write a new program using Amulet, it is easiest to start with an example Makefile. For example, you could copy the contents of `samples/tutorial/` into your new directory, and edit the Makefile and sample file to begin your project. From the start, you will be able to just invoke `make` in that directory to generate a binary for your Amulet program. The examples in the Tutorial all start from this point, and can be used as models for your programs.

1.3.3.6 Customizing the Makefile.vars.custom Variables

Although C++ is supposed to be a standardized language, there are differences among compilers and machine architectures that still require different source code. Conditional code is built into Amulet that depends on the definition of several variables at compile-time, which you set according to your compiler and computer. This section is a guide to the variables that control the conditional Amulet code. If, after juggling the compiler variables documented below, you are still not able to compile Amulet, please send mail to `amulet-bugs@cs.cmu.edu` so we can try to make the Amulet code more portable.

Amulet is known to not compile in `gcc` 2.5.8, due to a compiler bug.

Before changing any of the Makefile variables, you should try compiling Amulet once (see Section 1.3.3.3, above). If the procedure does not terminate smoothly, you should have some indication of what switches need to be added or changed. Make sure that your AMULET_VARS_FILE environment variable is set to Makefile.vars.custom, and bring this file up in an editor (found in amulet/bin/). This is the only file that should change.

The variables that control conditional Amulet code are defined with -D compiler switches. For example, we have found that the CC libraries on Suns do not provide the standard function memmove(), so we have to define it ourselves in Amulet. The Amulet version of memmove() is only defined when the compiler switch -DNEED_MEMMOVE is included in the compile call (which declares the variable NEED_MEMMOVE). By adding or removing the -DNEED_MEMMOVE switch, you control whether Amulet defines memmove().

The interface for defining these variables is the FLAGS list in Makefile.vars.custom. For each variable VAR, you would include the switch -DVAR in the FLAGS list to define the variable, or simply leave out the switch to avoid defining the variable. By iteratively adding or removing these variables from your FLAGS list and recompiling Amulet, you should be able to install Amulet on your system.

Compiler Variables:

HP -- Including -DHP in the FLAGS list will cause some type casting required for HP's, inappropriate for other machines.

GCC -- Including -DGCC in the FLAGS list causes different header files to be referenced than when Amulet is compiled with CC or Visual C++.

NEED_BOOL -- Including -DNEED_BOOL in the FLAGS list causes Amulet to define the bool type. This type is pre-defined in gcc.

NEED_MEMMOVE -- Including -DNEED_MEMMOVE in the FLAGS list causes Amulet to define its own memmove() function. This function is missing in some C libraries.

NEED_STRING -- Including -DNEED_STRING in the FLAGS list causes Amulet to include the standard header file strings.h in special places required by some versions of the CC compiler.

DEBUG -- Including -DDEBUG in the FLAGS list causes Amulet debugging code to be compiled into your binaries. This allows you to turn on tracing of internal Amulet behavior, which might help you understand why a program behaves unexpectedly. Not defining DEBUG will make your binaries slightly smaller, and you will not be able to turn on tracing.

Makefile Variables:

CC -- Your compiler, such as /usr/local/bin/gcc.

LD -- Your linker, such as /bin/ld.

OP -- Options you want to pass to your compiler, such as -g which tells gcc to put debugging information in the binaries.

FLAGS -- A list of switches to pass to the compiler, including the Amulet compiler variables listed above and any include paths not pre-defined for your compiler.

LIBS -- A list of libraries to link, such as -lX11 and -lg++.

XLIB Pathnames:

Some compilers on Unix systems do not know where to find the XLIB library and include files. You may need to include the pathnames for your XLIB files in the FLAGS list in Makefile.vars.custom. The include path should be provided with the -I switch (pronounced "eye"), and the library path should be provided with the -L switch.

For example, if your XLIB files reside in /usr/include/X11R5/X11/ and /usr/lib/X11R5/X11/, your FLAGS definition might look like this:

```
FLAGS  = -I$(AMULET_DIR)/include -DGCC -DDEBUG -DHP \
         -I/usr/include/X11R5  -L/usr/lib/X11R5
```

Note that a backslash is required at the end of a line when the definition of a Makefile variable continues on the next line.

1.4 Test Programs and Demos

The procedure for compiling and executing the demos and test programs is different depending on your platform. See section 1.3 for PC-specific and Unix-specific instructions for installing Amulet and compiling the demos.

`hello` -- This program creates a window and displays "hello world" in it. You can exit by hitting control-ESC.

`goodbye_button`
-- This program creates a button widget on the screen which quits when pressed.

`goodbye_inter`
-- This program displays "goodbye world" in a window, and an interactor causes the program to quit when the text is clicked on.

`tutorial` -- This program just creates a window. It is the starting point for all the examples in the Tutorial.

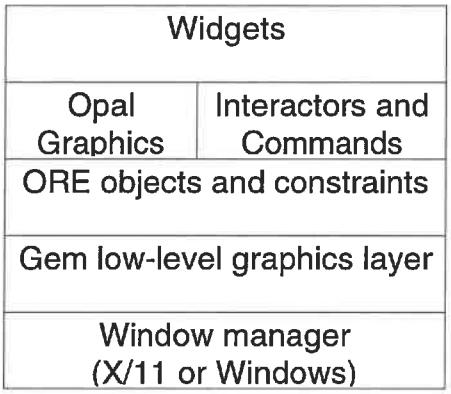
`checkers` -- This is a two-player checkers game that demonstrates the multi-screen capabilities of Amulet in Unix. You can run it on two screens by supplying the names of the displays when the program is executed, as in "`checkers my_machine:0.0 your_machine:0.0`". On the PC, you can still run this program, but it can only be displayed on one screen.

`space` -- This program demonstrates many features of Amulet: constraints, bitmaps, polylines, widgets, and scrolling windows. Some of the interactions to try are:

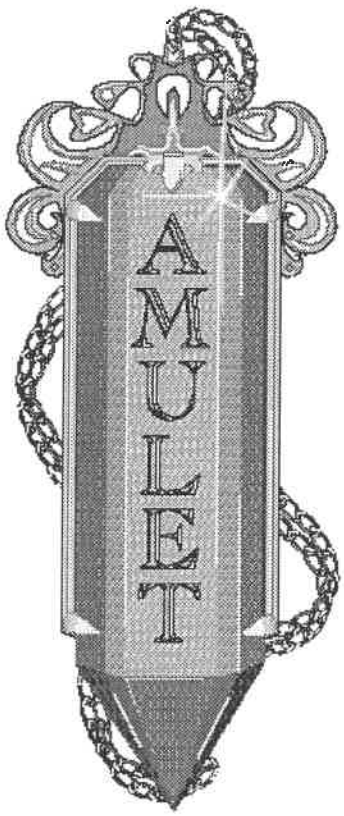
- Leftdown in the background of the Short-Range Scan creates a ship
- Leftdown on an existing ship moves it
- Middledown+drag (or META-leftdown+drag on PC) from one ship to another draws phasers and destroys the destination ship
- Rightdown+drag from one ship to another establishes a tractor beam which stays attached to the ships through constraints
- Dragging the white rectangle in the Long-Range Scan changes the visible area in the Short-Range Scan. You can scroll the visible area with the scroll-bars or by dragging the white feedback rectangle in the Long-Range Scan.

1.5 Parts of Amulet

Amulet is divided into various layers, which each have their own interface (and chapter of this manual). The overall picture is shown below.



The Gem layer provides a machine-independent interface so the rest of Amulet is independent of the particular window manager in use. Most programmers will not need to use Gem. Ore provides a prototype-instance object system and constraint solving that is used by the rest of Amulet. Opal provides an object-oriented interface to the output graphics, and the Interactors and Command objects handle input processing. At the top are a set of Widgets, including scroll bars, buttons, menus and text input fields.



2. Amulet Tutorial

Abstract

Amulet is a user interface development environment that makes it easier to create highly interactive, direct manipulation user interfaces in C++ for Windows NT or Unix X/11. This tutorial introduces the reader to the basic concepts of Amulet. After reading this tutorial and trying the examples with a C++ compiler, the reader will have a basic understanding of: the prototype-instance system of objects in Amulet; how to create windows and display graphical objects inside them; how to constrain the positions of objects to each other using formulas; how to use interactors to define behaviors on objects (such as selecting objects with the mouse); how to collect objects together into groups; how to use the Amulet widgets; and how to use some of the debugging tools in Amulet.

This research was sponsored by NCCOSC under Contract No. N66001-94-C-6037, Arpa Order No. B326. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NCCOSC or the U.S. Government.

2.1 Setting Up

2.1.1 Install Amulet in your Environment

Before beginning this tutorial, you should have already installed Amulet in your computing environment, according to the instructions in the Amulet Overview. If you are using Unix, you should set your `AMULET_DIR` and `AMULET_VARS_FILE` environment variables according to the installation instructions. The Visual C++ environment will be automatically set up for PC users through a predefined project file, mentioned below. This tutorial assumes that you are familiar with `C++` and with the editor and compiler environments on your system.

In this tutorial, you will be introduced to the most common programming layer of Amulet, where you can create windows, graphical objects, interactors, and widgets. It includes code examples that can you can type in and compile yourself, along with discussions of Amulet programming techniques.

There is another programming interface to Amulet at the Gem layer (the Graphics and Events Module). By accessing the Gem layer, you can explicitly call the functions that Amulet uses to draw objects on the screen. Most Amulet users will not need to call Gem functions directly, because Amulet objects redraw themselves automatically once they are added to a window. Gem is only needed by programmers who cannot get sufficient performance using the higher layers.

2.1.2 Copy the Tutorial Starter Program

Throughout this tutorial, you will be typing and compiling code to observe its behavior. A starter program is installed with Amulet in the directory `samples/tutorial/` in Unix, or in `samples\tutorial\` in Windows NT. By following the instructions in this tutorial, you will iteratively edit and recompile this program in your local area while learning about Amulet.

If you are using Unix, copy the `tutorial/` directory and its contents into your local filespace. Your copy of the directory should contain the `Makefile` and `tutorial.cc`. From the `tutorial/` directory, invoke `make` which will generate `tutorial.o` and `tutorial`. You should be able to execute the `tutorial` binary which creates a window in the upper-left corner of the screen. Exit the program by placing the mouse in the Amulet window (and clicking in the window to make it active, if necessary) and type the Amulet escape sequence, `SHIFT-ESC`.

If you are using Windows NT, copy the `tutorial\` directory and its contents into your local filespace. If Amulet is installed in your C:\AMULET directory, you might copy the tutorial files with the following sequence of commands:

```
mkdir tutorial
copy c:\amulet\samples\tutorial\*.* tutorial
```

Your copy of the directory should contain the source code `tutorial.cpp` and the project file `tutorial.mak`. From the File Manager, double-click on `tutorial.mak` to launch Visual C++ and load the project file. If you change the tutorial working directory, you'll have to change the makefile so Visual C++ can find your source code. Add the files `tutorial.cpp`, and `src\debug\inspectr.cpp`. If you choose "Build" from the "Project" menu, and then run the program, a window will be created in the upper-left corner of the screen. To exit the program, click the mouse in the Amulet window to make it active, and type the Amulet escape sequence, `SHIFT-ESC`.

If you have trouble copying the starter program or generating the `tutorial` executable, there may be a problem with the way that Amulet was installed at your site. Consult the Amulet Overview for detailed instructions about installing Amulet. In Windows NT, the GW Streams program must be executing to see output or type input for an Amulet program.

2.1.3 Amulet Header Files

The only header file you need to include in your Amulet programs is `amulet.h`, which will include all of the others for you. However, some programmers like to look at header files, and they can be found in the `include/amulet/` directory if you are using Unix, and in `include\amulet\` if you are using Windows NT.

Amulet header files fall into two categories, Basic, and Advanced. Basic header files define all the standard objects, functions, and global variables that Amulet users might need to write various applications. Advanced header files define lower level Amulet functions and classes which would be useful to an advanced programmer interested in creating custom objects in addition to the default objects Amulet supports. Most users will only ever include the Basic header files.

2.1.3.1 Basic header files

The header file `amulet.h` includes all the headers most amulet programmers will ever need to use: `standard_slots.h`, `value_list.h`, `gdefs.h`, `idefs.h`, `opal.h`, `inter.h`, and `widgets.h`. Here is a summary of the basic header files, listing the major objects, classes, and functions they define.

- `gdefs.h`: `Am_Style`, `Am_Font`, `Am_Point_List`, `Am_Image_Array`
- `idefs.h`: `Am_Input_Char`, default `Am_Input_Char`'s
- `object.h`: `Am_Object`, `Am_String`, `Am_Value`, `Am_Slot`, `Am_Instance_Iterator`, `Am_Slot_Iterator`, `Am_Part_Iterator`
- `standard_slots.h`: standard slot names, `Am_Register_Slot_Key`, `Am_Register_Slot_Name`, `Am_Get_Slot_Name`, `Am_Slot_Name_Exists`
- `opal.h`: default `Am_Style`'s, default `Am_Font`'s, `Am_Screen`, `Am_Graphical_Object`, `Am_Window`, `Am_Rectangle`, `Am_Roundtangle`, `Am_Line`, `Am_Arrow`, `Am_Polygon`, `Am_Arc`, `Am_Text`, `Am_Bitmap`, `Am_Group`, `Am_Map`, default constraints, `Am_Initialize`, `Am_Cleanup`, `Am_Beep`,

Am_Move_Object, Am_To_Top, Am_To_Bottom, Am_Create_Screen, Am_Update, Am_Update_All, Am_Do_Events, Am_Main_Event_Loop, Am_Exit_Main_Event_Loop, default Am_Point_In_ functions, Am_Translate_Coordinates, Am_Merge_Pathname

- formula.h: Am_Formula, Am_Declare_Formula, Am_Define_Formula, Am_Declare_Value_Formula, Am_Define_Value_Formula
- value_list.h: Am_Value_List
- text_fns.h: all text editing functions, Am_Edit_Translation_Table
- inter.h: Am_Interactor, Am_Choice_Interactor, Am_New_Points_Interactor, Am_One_Shot_Interactor, Am_Move_Grow_Interactor, Am_Text_Edit_Interactor, Am_Where_Functions, interactor debugging functions, Am_Command, Am_Choice_Command, Am_Move_Grow_Command, Am_New_Points_Command, Am_Edit_Text_Command, Am_Undo_Handler, Am_Single_Undo_Object, Am_Multiple_Undo_Object
- widgets.h: Am_Border_Rectangle, Am_Button, Am_Button_Panel, Am_Checkbox_Panel, Am_Radio_Button_Panel, Am_Menu, Am_Menu_Bar, Am_Button_Command, Am_Menu_Line_Command, Am_Scroll_Command, Am_Vertical_Scroll_Bar, Am_Horizontal_Scroll_Bar, Am_Scrolling_Group, Am_Text_Input_Widget
- debugger.h: Am_Initialize_Inspector, Am_Inspect, Am_Text_Inspect

2.1.3.2 Advanced header files

All other header files are considered Advanced header files. They support advanced Amulet features, such as user-defined objects, daemons, constraints, and so on. Most users should never include these files explicitly. For users who will be using Amulet's advanced features, here is a brief summary of the contents of each advanced header file.

- gem.h: Am_Drawonable, Am_Input_Event, Am_Input_Event_Handlers, Am_Region
- am_io.h: Am_TRACE
- object_advanced.h: Am_Demon_Queue, Am_Demon_Set, Am_Constraint, Am_Constraint_Iterator, m_Dependancy_Iterator, Am_Slot_Advanced, Am_Object_Advanced, Am_Constraint_Context, Ore_Initialize
- priority_list.h: Am_Priority_List_Item, Am_Priority_List
- symbol_table.h: Am_Symbol_Table
- types.h: NULL, Am_Error, Am_Wrapper, Am_WRAPPER_DECL, Am_WRAPPER_IMPL, Am_WRAPPER_DATA_DECL, Am_WRAPPER_DATA_IMPL
- formula_advanced.h: Am_Formula_Advanced, Am_Depends_Iterator
- opal_advanced.h: Am_Aggregate, advanced opal object slots, Am_Draw_Method, Am_Draw, Am_Invalid_Method, Am_Invalidate, Am_Point_In_Obj_Method, Am_Translate_Coordinates_Method, Am_State_Store, Am_Item_Function, Am_Invalid_Rectangle_Intersect, Am_Window_ToDo
- inter_advanced.h: Am_Initialize_Interactors, Am_Action_Function, Am_Four_Ints_Data, Am_Interactor_Input_Event_Notify, Am_Inter_Tracing, Am_Get_Filtered_Input, Am_Modify_Object_Pos,

Am_Choice_Command_Set_Value, interactor Am_*_Action functions, Am_Command functions, Undo handler functions

- widgets_advanced.h: Computed_Colors_Record, Am_Button_Command functions, Am_Checkbox, Am_Radio_Button_Item, Am_Menu_Item, widget drawing functions, widget formula constraints, other widget support functions

2.2 The Prototype-Instance System

The most common action performed in Amulet is to create objects and display them in windows on the screen. Am_Object is a fundamental data type in Amulet. Lines, circles, groups and windows are all objects. These are all *prototype* objects -- you make *instances* of these objects, and customize them to have your desired size and position, as well as other graphic qualities such as filling styles and line styles. The instances will have default property values determined by their prototypes if you don't supply values yourself. Thus, Amulet is a prototype-instance system.

For a complete list of all of Amulet's default prototype objects, see Chapter 8, Summary of Exported Objects and Slots.

2.2.1 Objects and Slots

The properties of an Amulet object are stored in its *slots*. A rectangle's slots contain values for its left, top, width, height, line-style, filling-style, etc. In the following code, a rectangle is created and its slots are set with new values (it is not necessary to type in this code, it is just for discussion):

```
Am_Object my_rect = Am_Rectangle.Create ("my_rect")
   .Set (Am_LEFT, 20)
   .Set (Am_TOP, 20)
   .Set (Am_LINE_STYLE, Am_Black)
   .Set (Am_FILL_STYLE, Am_Red);

int my_left = my_rect.Get (Am_LEFT);    //  my_left has value 20
```

The Set operation sets the values of slots, and the corresponding Get operation retrieves them. Set takes a *slot key*, like Am_LEFT, and a new value to store in the slot. Get takes a slot key and returns the value stored in the slot. A slot key is just an index into the set of slots in an object.

There are many pre-defined slot keys used by Amulet objects, all starting with the "Am_" prefix, declared in the header file standard_slots.h. Slot keys that you create for your own use need to be declared with special Amulet functions, as in:

```
Am_Slot_Key MY_SLOT  = Am_Register_Slot_Name ("MY_SLOT");
```

There are many examples of setting and retrieving slot values throughout this tutorial.

An important difference between C++ classes and Amulet objects is that Amulet allows the dynamic creation of slots in objects. A program can add and remove slots from an object as required by the given situation. In C++ classes, only the class's data can be modified at runtime -- modifying a class's structure would require recompiling.

2.2.2 Dynamic Typing

Another difference between Amulet objects and C++ classes involves the type restrictions of the values being stored. In C++ classes, you are restricted to defining member variables of a specific type, and you can only store data of that type in the variables. In contrast, Amulet uses *dynamic typing*, where the type of a slot is determined by the value currently stored in it. So, any slot can hold any type of data.

Amulet achieves dynamic typing by overloading the `Set` and `Get` operators. There are versions of `Set` and `Get` that handle most simple C++ types including `int`, `float`, `double`, `char`, and `bool`. They also handle more general types like strings, Amulet objects, functions, and `void*`. Other types are encapsulated in a type called `Am_Wrapper`, which allows C++ data structures to be stored in slots.

For example, if your code contains:

```
int i = obj.Get(Am_LEFT);
```

then Amulet looks in the slot `Am_LEFT` and if the value there is an integer, it is assigned to `i`. If the value there is *not* an integer, however, this causes an error. If you do not know what type a slot contains, you can get the slot into a generic `Am_Value` type or ask the slot what type it contains using `obj.Get_Slot_Type` (these are explained in the ORE manual).

Unfortunately, some compilers cannot always correctly determine the type to use for a slot access, and you might have to cast values that are passed to `Set` or retrieved by `Get`. The following instruction casts the value retrieved from `obj`'s `Am_LEFT` slot into an `int`, so that it can be compared to another `int`:

```
if ((int)obj.Get(Am_LEFT)) > 10) {
...
}
```

Some compilers even get confused with perfectly unambiguous expressions involving `Am_Objects` and other types of wrappers. In this case, you need to put the variable declaration in a separate statement from its assignment:

```
~~Am_Object obj = my_object.Get(Am_PARENT); //fails on some compilers~~
Am_Object obj; //using two statements works!
obj = my_object.Get(Am_PARENT);
```

2.2.3 Inheritance

When instances are created, an inheritance link is established between the prototype and the instance. *Inheritance* is the property that allows instances to get values from their

prototypes without specifying those values in the instances themselves. For example, if we set the filling style of a rectangle to be gray, and then we create an instance of that rectangle, then the instance will also have a gray filling style. Naturally, this leads to an inheritance hierarchy among the objects in the Amulet system. In fact, there is one root object in Amulet -- the `Am_Graphical_Object` -- that all graphical objects are instances of. **Figure 2-1** shows some of the objects in Amulet and how they fit into the inheritance hierarchy.

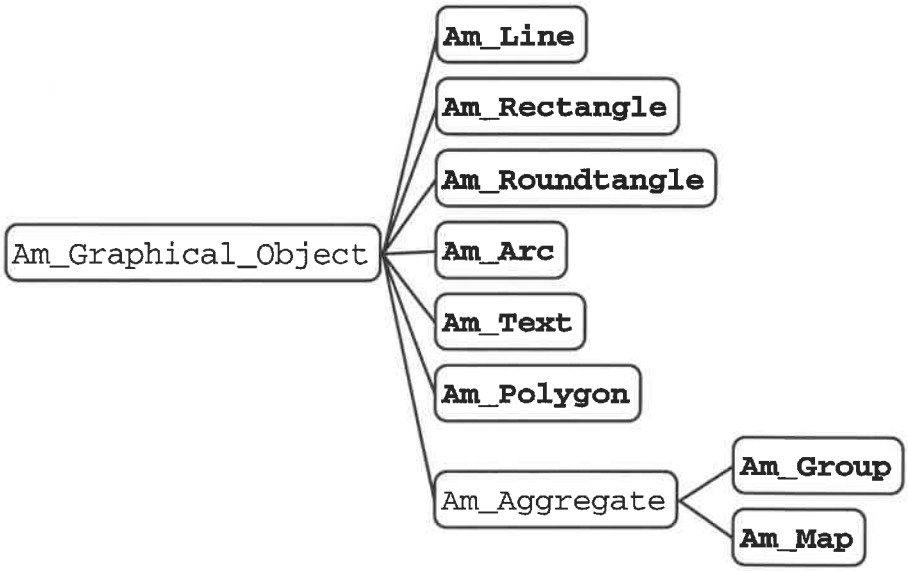


Figure 2-1: The inheritance hierarchy among some of the Amulet prototype objects. Objects shown in bold are used by all Amulet programmers, while the others are internal, and intended to be accessed only by advanced users. All of the standard shapes in Amulet are instances of the `Am_Graphical_Object` prototype. As an example of inheritance, the `Am_Map` and `Am_Group` objects are both special types of aggregates, and they inherit most of their properties from the `Am_Aggregate` prototype object. The Widgets (the Amulet gadgets) are not pictured in this hierarchy, but most of them are instances of the `Am_Aggregate` object.

To see an example of inheritance, let's create an instance of a window and look at some of its inherited values. If you have followed the instructions in Section 2.1.2, you should have the file `tutorial.cc` (Unix) or `tutorial.cpp` (PC) in your local area. On Unix, bring up `tutorial.cc` in an editor, or on the PC double-click on the `tutorial.cpp` icon in the Visual C++ Project Window, to show the code pictured in **Figure 2-2**.

If you have not already compiled this file, do so now. In UNIX, invoke `make` in your `tutorial/` directory to generate the `tutorial` binary. On the PC, select "Build" from the "Project" menu. Execute `tutorial` to create a window in the upper-left corner of the screen.

```
#include <amulet/amulet.h>
#include <amulet/debugger.h>

main (void)
{
  Am_Initialize ();

  Am_Object my_win = Am_Window.Create ("my_win")
    .Set (Am_LEFT, 20)
    .Set (Am_TOP, 50);

  Am_Screen.Add_Part (my_win);

  Am_Initialize_Inspector (my_win);  // Allow Inspector -- see Section 2.7.1
  Am_Main_Event_Loop ();             // Process events -- see Section 2.5.5
  Am_Cleanup ();                     // Destroy Amulet prototypes and classes
}
```

Figure 2-2: The initial contents of `tutorial.cc` / `tutorial.cpp`

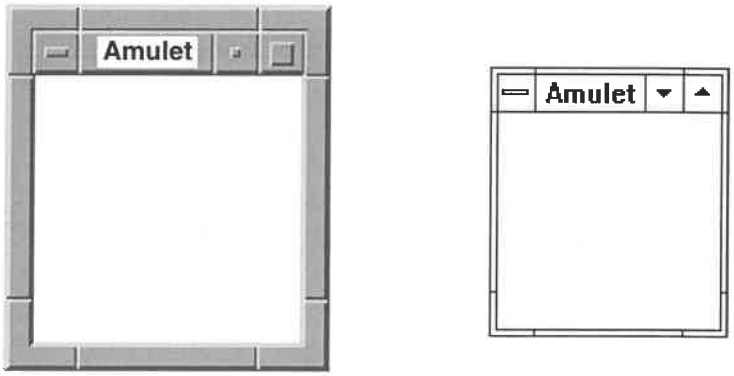


Figure 2-3: The window created by `tutorial`, as displayed by the Motif window manager in Unix X/11 (left), and by Windows NT on the PC (right).

The `tutorial` program creates an object called `my_win`, which is an instance of `Am_Window`. A value of 20 was installed in its `Am_LEFT` slot and 50 in its `Am_TOP` slot. These values are reflected in the position of the window on the screen.

To check that the slot values are correct, bring up the Amulet `Inspector` to examine the slots and values of the window. Move the mouse over the window and press the `F1` key. The Amulet `Inspector` will pop up a window that displays the slots and values of `my_win`, as shown in **Figure 2-4**. You will see many slots displayed, some of which are internal and not intended for external use. In general, the slots with "*"s in their names are internal slots. Many of the other slots are "advanced" and should not be needed by most programmers. Chapter 8 of this manual called "Summary of Exported Objects" lists the primary exported slots of the main Amulet objects.

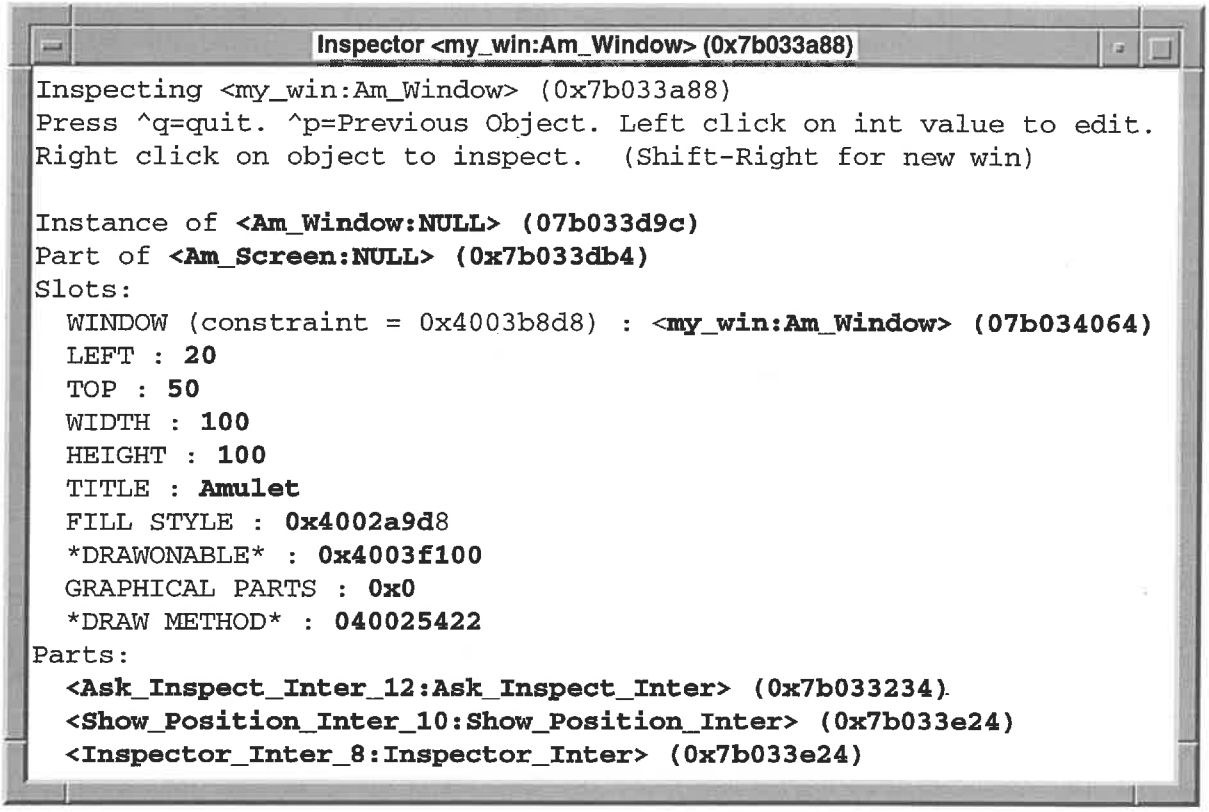


Figure 2-4: The Amulet Inspector displaying the slots and values of `my_win`. (The set of slots actually displayed has been abridged in this picture so that it will fit on the page.)

The `Am_LEFT` and `Am_TOP` slots of `my_win` shown in the `Inspector` contain the expected values. Additionally, the `Am_WIDTH` and `Am_HEIGHT` slots contain values that were not set by the `tutorial` program. These values were *inherited* from the prototype. That is, they were defined in the `Am_Window` object when it was created, and now `my_win` inherits those values as its own. The `Inspector` shows they are inherited by displaying the slots in blue. We could, however, override these inherited values.

Currently, the inspector does not update the display when the values change (say because you move the window with the mouse). However, you can explicitly update the inspector by typing control-r in the inspector window.

To exit the tutorial program and destroy the Amulet window, position the mouse over the window (and click to select the window, if necessary) and type `SHIFT-ESC`.

Let's change the width and height of `my_win` using `Set`, the function that sets the values of slots. Edit the source code, and immediately after the definition of `my_win` add the following lines:

```
my_win.Set (Am_WIDTH, 200)
      .Set (Am_HEIGHT, 400);
```

Notice that we can cascade the calls to `Set` without placing semi-colons at the end of each line. `Set` makes this possible by returning the object that is being changed, so that the return value of `Set` can be used without intermediate binding. After compiling and executing the file, and hitting `F1` to invoke the `Inspector`, you can see that we have successfully overridden the `Am_WIDTH` and `Am_HEIGHT` slots in `my_win` with our local values.

The counterpart to `Set` is `Get`, which retrieves values from slots. The `Inspector` uses `Get` on `my_win` to obtain the values to print in the `Inspector` window. We can use `Get` directly by typing the following code into the source code, after the definition of `my_win`:

```
int left  = my_win.Get (Am_LEFT);
int width = my_win.Get (Am_WIDTH);
cout << "left == "  << left  << endl;
cout << "width == " << width << endl;
```

Try deleting the code we used to set the left, top, width, and height of the window, and see what values are printed by the `cout` statement.

The inheritance hierarchy which was partially pictured in **Figure 2-1** is traced from the leaves toward the root (from right to left) during a search for a value. Whenever we use `Get` to retrieve the value of a slot, the object first checks to see if it has a local value for that slot. If there is no value for the slot in the object, then the object looks to its prototype to see if it has a value for the slot. This search continues until either a value for the slot is found or the root object is reached. When no inherited or local value for the slot is found, an error is raised. This might occur if you are asking for a slot from the wrong object.

2.2.4 Instances

Typically, all the objects displayed in a window are instances of other objects. In `tutorial`, `my_win` is an instance of `Am_Window`. Let's create several instances of graphical objects and add them to `my_win`. First, make sure that your window is large enough, at least 200x200. You could change your definition of `my_win` to look something like this:

```
Am_Object my_win = Am_Window.Create ("my_win")
  .Set (Am_LEFT, 20)
  .Set (Am_TOP, 50)
  .Set (Am_WIDTH, 200)
  .Set (Am_HEIGHT, 200);

Am_Screen.Add_Part (my_win);      // Puts my_win on the screen
```

Now we can create several graphical objects and add them to the window. Type the following code into the `tutorial` program, then recompile and execute `tutorial`.

```
Am_Object my_arc = Am_Arc.Create ("my_arc")
  .Set (Am_LEFT, 10)
  .Set (Am_TOP, 10);

Am_Object my_text = Am_Text.Create ("my_text")
  .Set (Am_LEFT, 80)
  .Set (Am_TOP, 30)
  .Set (Am_TEXT, "This is my_text");

Am_Object my_rect = Am_Rectangle.Create ("my_rect")
  .Set (Am_LEFT, 10)
  .Set (Am_TOP, 100)
  .Set (Am_WIDTH, 180)
  .Set (Am_HEIGHT, 80)
  .Set (Am_FILL_STYLE, Am_Red);

my_win.Add_Part (my_arc)
  .Add_Part (my_text)
  .Add_Part (my_rect);
```

The circle, text, and rectangle will be displayed in the window. You can position the mouse over any of the objects and hit `F1` to display the slots of the object in the `Inspector`. If you hit `F1` while the mouse is over the background of the window, you will raise the `Inspector` for the window itself. While inspecting `my_win`, you can see at the bottom of the `Inspector` display that the new objects have been added as *parts* of the window.

Amulet supplies a large collection of objects that you can make instances of, including the basic graphical primitives like rectangles and circles, and the standard widgets like menus, buttons and scroll bars. Chapter 8, called "Summary of Exported Objects" at the end of the manual, lists the main exported Amulet objects you can make instances of.

2.2.5 Prototypes

When programming in Amulet, inheritance among objects can eliminate a lot of duplicated code. If we want to create several objects that look similar, we could create each of them from scratch and copy all the values that we need into each object. However, inheritance allows us to define these objects more efficiently, by creating several similar objects as instances of a single prototype.

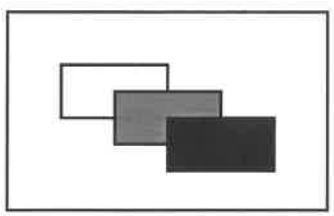


Figure 2-5: Three instances created from one prototype rectangle.

To start, look at the picture in **Figure 2-5**. We are going to define three rectangles with three different filling styles and put them in a window. Using your current version of `tutorial`, make sure it will create a window of size at least 200x200.

Now let's consider the design for the rectangles. The first thing to notice is that all of the rectangles have the same width and height. Therefore, we will create a prototype rectangle which has a width of 40 and a height of 20, and then we will create three instances of that rectangle. To create the prototype rectangle, type the following.

```
Am_Object proto_rect = Am_Rectangle.Create ("proto_rect")
  .Set (Am_WIDTH, 40)
  .Set (Am_HEIGHT, 20);
```

This rectangle will not appear anywhere, because it will not be added to the window. But now we need to create the three actual rectangles that will be displayed. Since the prototype has the correct values for the width and height, we only need to specify the left, top, and filling styles of our instances.

```
Am_Object r1 = proto_rect.Create ("r1")
  .Set (Am_LEFT, 20)
  .Set (Am_TOP, 20)
  .Set (Am_FILL_STYLE, Am_White);

Am_Object r2 = proto_rect.Create ("r2")
  .Set (Am_LEFT, 40)
  .Set (Am_TOP, 30)
  .Set (Am_FILL_STYLE, Am_Gray_Stipple);

Am_Object r3 = proto_rect.Create ("r3")
  .Set (Am_LEFT, 60)
  .Set (Am_TOP, 40)
  .Set (Am_FILL_STYLE, Am_Black);

my_win.Add_Part(r1)
  .Add_Part(r2)
  .Add_Part(r3);
```

After you recompile and execute, you can see that the instances `r1`, `r2`, and `r3` have inherited their width and height from `proto_rect`. You may wish to use the `Inspector` to verify this. With these three rectangles still in the window, we are ready to look at another important use of inheritance by changing values in the prototype.

Bring `proto_rect` up in the `Inspector` by selecting it from a display of one of the instances. That is, if `r1` is already displayed in the `Inspector`, the line "Instance of `<proto_rect:Am_Rectangle>`" will appear at the top of the `Inspector` window, with `proto_rect` shown in bold. When you click the right mouse button on `proto_rect`, the contents of the `Inspector` window will be replaced by the slots and values of `proto_rect`. You could also hold down the shift key while clicking the right mouse button over `proto_rect` to bring up its display in a new `Inspector` window.

The `Inspector` displays values of slots in bold, and these values can be edited to change the properties of the object being inspected. (Currently, only the editing of integer values is supported, but eventually all kinds of values will be editable through the `Inspector`). When you click the left mouse button in an integer value, a cursor appears and you can use standard Amulet text editing commands to change the value. Here is a brief summary of text editing commands (note: "`^f`" means "hold down `CONTROL` and press the 'f' key).

`^f` or `rightarrow` -- forward one character
`^b` or `leftarrow` -- backward one character
`^a` -- go to beginning of line
`^e` -- go to end of line
`^h`, `DELETE`, `BACKSPACE` -- delete previous character
`^w`, `^DELETE`, `^BACKSPACE` -- delete previous word
`^d` -- delete next character
`^u` -- delete entire string
`^k` -- kill (or delete) rest of line
`^y`, `INSERT` -- insert the contents of the cut buffer into the string at the current point
`^c` -- copy the current string into the cut buffer
`^g` -- aborts editing and returns the string to the way it was before editing started
`leftdown` (inside the string) -- move the cursor to the specified point

All other characters go into the string (except other control characters which beep).

By editing the values in the `Inspector` window, change the width of `proto_rect` to 30 and change its height to 40. The result should look like the rectangles in **Figure 2-6**. Just by changing the values in the prototype rectangle, we were able to change the appearance of all its instances. This is because the three instances inherit their width and height from the prototype, even when the prototype changes.

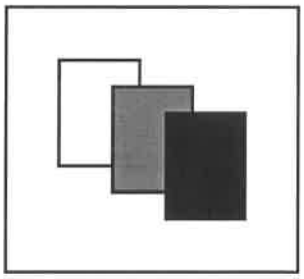


Figure 2-6: The instances change whenever the prototype object changes.

For our last look at inheritance in this section, let's override the inherited slots in one of the instances. Suppose we now want the rectangles to look like **Figure 2-7**. In this case, we only want to change the dimensions of one of the instances. Bring `r3` (the black rectangle) up in the `Inspector`, and change the value of its width slot to 100.

The rectangle `r3` now has its own value for its `Am_WIDTH` slot, and no longer inherits it from `proto_rect`. If you change the width of the prototype again, the width of `r3` will not be affected. However, the width of `r1` and `r2` will change with the prototype, because they still inherit the values for their `Am_WIDTH` slots. This shows how inheritance can be used flexibly to make specific exceptions to the prototype object.

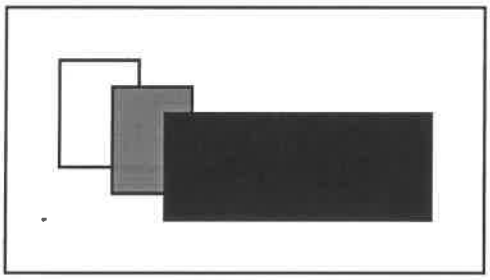


Figure 2-7: The width of `r3` is overridden by a local value, and is no longer inherited from the prototype.

2.2.6 Default Values

Because of inheritance, all instances of Amulet prototype objects have reasonable default values when they are created. As we saw in Section 2.2.3, the `Am_Window` object has its own `Am_WIDTH` value. So, if an instance of it is created without an explicitly defined width, the width of the instance will be inherited from the prototype, and it can be considered a default value. Section 8 contains a complete list of Amulet objects and the default values of their slots.

2.2.7 Destroying Objects

After objects have fulfilled their purpose, it is appropriate to destroy them. All objects occupy space in memory, and continue to do so until explicitly destroyed (or the program terminates). A `Destroy()` method is defined on all objects, so at any point in a program you can do *`obj`*`.Destroy()` to destroy *obj*.

When you destroy a graphical object (like a line or a circle), it is automatically removed from any window or group that it might be in and erased from the screen. Destroying a window or a group will destroy all of its parts. Destroying a prototype also destroys all of its instances.

2.2.8 Unnamed Objects

Sometimes you will want to create objects that do not have a particular name. For example, you may want to write a function that returns a rectangle, but it will be called repeatedly and should not return multiple objects with the same name. In this case, you should return an unnamed rectangle from the function, and allow Amulet to generate a unique name for you.

As an example, the following code creates unnamed objects and displays them in a window. Instead of supplying a quoted name to `Create`, we invoke it with no parameters.

```
Am_Object obj;
for (int i=0; i<10; i++) {
  obj = Am_Rectangle.Create()
    .Set (Am_LEFT, i*10)
    .Set (Am_TOP, i*10);
  my_win.Add_Part (obj);
}
```

When no name string is supplied to `Create`, Amulet generates a unique name for the object being created. In this case, something like `<Am_Rectangle_5:Am_Rectangle>`. This name has a unique number as a suffix that prevents it from being confused with other rectangles in Amulet.

2.3 Graphical Objects

2.3.1 Lines, Rectangles, and Circles

The Opal module provides different graphical shapes including circles, rectangles, roundtangles, lines, text, bitmaps, and polygons. Each graphical object has special slots that determine its appearance, which are fully documented in chapter 4 and summarized in section 8.4. (For example, `Am_Line` uses the slots `Am_X1`, `Am_Y1`, `Am_X2`, and `Am_Y2`.) Examples of creating instances of graphical objects appear throughout this tutorial.

2.3.2 Groups

In order to put a large number of objects into a window, we might create all of the objects and then add them, one at a time, to the window. However, this is usually not how we organize the objects conceptually. For example, if we were to create a sophisticated interface with tool palettes, icons with labels, and feedback objects, we would not want to add each line and rectangle directly to the window. Instead, we would think of creating each palette from its composite rectangles, then creating the labeled icons, and then adding each assembled group to the window.

Grouping objects together like this is the function of the `Am_Group` object. Any graphical object can be part of a group - lines, circles, rectangles, widgets, and even other groups. Usually all the parts of a group are related in some way, like all the selectable icons in a tool palette.

Groups define their own coordinate system, meaning that the left and top of their parts is offset from the origin of the group. Changing the position of the group *translates* the position of all its parts. However, there is no complementary feature of *scaling* groups to change the size of all the parts. Groups also *clip* their parts to the bounding box of the

group, meaning that objects outside the left, top, width, or height of the group are not drawn.

In Amulet terminology, a group is the *owner* of all of its *parts*. The `Add_Part()` and `Remove_Part()` methods are used to add and remove parts. You can optionally provide a slot key (a slot name, such as `Am_LEFT`) in an `Add_Part()` call. If a slot key is provided, then in addition to becoming a part of the group, the new part will be stored in that slot of the group. Parts with slot keys are instantiated when instances of an existing group are created, but parts without a key are not. It is often convenient to provide slot keys for parts so that functions and formulas can easily access these objects in their groups.

Objects may be added directly to a window or to a group which, in turn, has been added to the window. When groups have other groups as parts, a group hierarchy is formed.

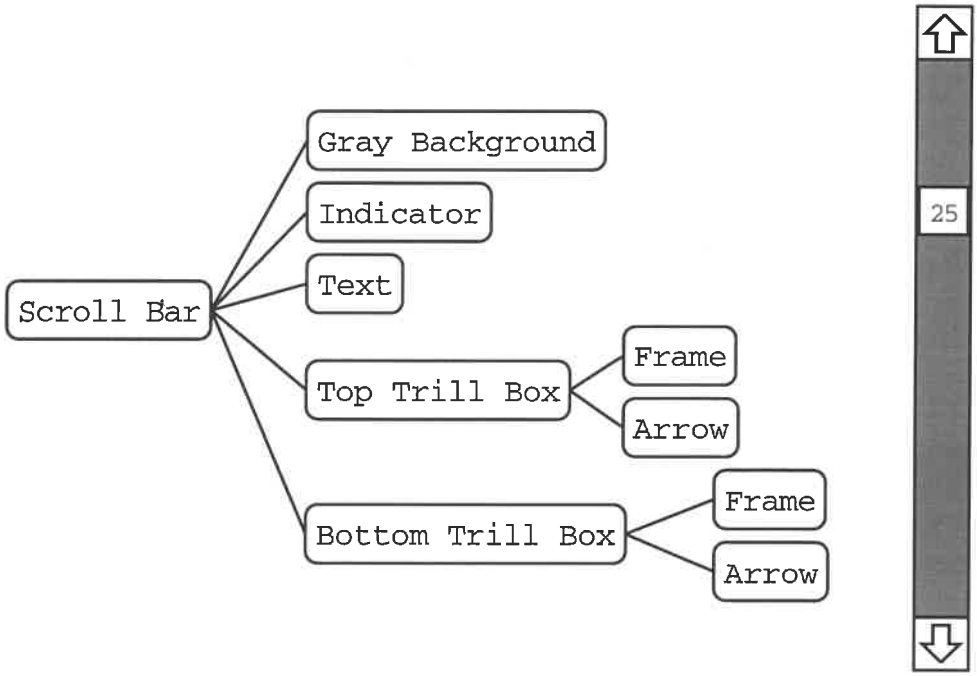


Figure 2-8: One possible hierarchy for the objects that make up a scroll bar.

In the scroll bar hierarchy, all of the leaves correspond to shapes that appear in the scroll bar. The leaves are always Amulet graphic primitives, like rectangles and text. The nodes `Top_Trill_Box` and `Bottom_Trill_Box` are both groups, each with two parts. And, of course, the top-level `Scroll_Bar` node is a group.

This group hierarchy should not be confused with the inheritance hierarchy that was discussed earlier. Parts of a group do not inherit values from their owners. Instead,

relationships among groups and their parts must be explicitly defined using constraints, a concept which will be discussed shortly in this tutorial.

2.3.3 Am_Group

`Am_Group` and `Am_Map` are used to form groups of other objects. They both define their own coordinate system, so that their parts are offset from the origin of the group.

You may create a group and add components to it in distinct steps, or you can use the cascading style of method invocation to perform all the `Set` and `Add_Part` operations in one expression. **Figure 2-9** shows an example of a group that contains an arc and a rectangle.

```
// Declared at the top-level, outside of main()
// You may install new slots in any object, but if they are not pre-defined Amulet slots, starting with the
"Am_" prefix, then
// you must define them seperately at the top-level. See Section 2.2.1.
Am_Slot_Key ARC_PART  = Am_Register_Slot_Name ("ARC_PART");
Am_Slot_Key RECT_PART = Am_Register_Slot_Name ("RECT_PART");
...

  // Defined inside of main()
  Am_Object my_group = Am_Group.Create ("my_group")
    .Set (Am_LEFT, 20)
    .Set (Am_TOP, 20)
    .Set (Am_WIDTH, 100)
    .Set (Am_HEIGHT, 100)
    .Add_Part(ARC_PART, Am_Arc.Create ("my_circle")
                        .Set (Am_WIDTH, 100)
                        .Set (Am_HEIGHT, 100))
    .Add_Part(RECT_PART, Am_Rectangle.Create ("my_rect")
                        .Set (Am_WIDTH, 100)
                        .Set (Am_HEIGHT, 100)
                        .Set (Am_FILL_STYLE, Am_No_Style));

  // Instances of my_group
  Am_Object my_group2 = my_group.Create ("my_group2")
    .Set (Am_LEFT, 150);

  Am_Object my_group3 = my_group.Create ("my_group3")
    .Set (Am_TOP, 150);

  // Don't forget to add the graphical objects to the window!
  my_win.Add_Part (my_group)
    .Add_Part (my_group2)
    .Add_Part (my_group3);
```

Figure 2-9: A group with two components, and two instances of that group.

The `Add_Part()` method is reminiscent of `Set` -- it takes an optional slot key and an object to install in the group. In addition to making the object an official part of the group, it is installed in the given slot in the group. Thus, the objects `my_circle` and

my_rect are stored in slots ARC_PART and RECT_PART of my_group. The slots ARC_PART and RECT_PART are *pointer* slots because they point to other objects. These slots provide immediate access to these objects through my_group, which is useful when defining constraints among the objects. Once installed, the parts can be retrieved by name from the group with the methods Get() and Get_Part().

When an instance of my_group is created, all of the named parts are duplicated in the new group. Thus, my_group2 and my_group3 are groups with the same structure as my_group, but at different positions.

2.3.4 Am_Map

A map is a kind of group that has similar parts, all generated from a single prototype. In an Am_Map, a single object is defined to be an *item-prototype*, and instances of this object are generated according to a set of items. See the Opal chapter for details and examples of maps.

2.3.5 Windows

Any object must be added to a window in order for it to be shown on the screen. Or, the object must be added to a group that, in turn, has been added to a window. All objects in a window are continually redrawn as necessary while the Am_Main_Event_Loop() is running (see Section 2.5.5).

As shown in previous examples, objects are added to windows using the Add_Part() method. Subwindows can also be attached using Add_Part(), using exactly the same syntax for adding groups or other graphical objects.

2.4 Constraints

In the course of putting objects in a window, it is often desirable to define relationships among the objects. For example, you may want the tops of several objects to be aligned, or you might want a set of circles to have the same center, or you may want an object to change color if it is selected. Constraints are used in Amulet to define these relationships among objects.

Constraints can be arbitrary C++ code, and can contain local variables and calls to functions. They may also have side effects on unrelated data structures with no ill effect, including setting slots and creating and destroying other Amulet objects.

Although all the examples in this section use constraints on the positions of objects, it should be clear that constraints can be defined for filling styles, strings, or any other property of an Amulet object. Many examples of constraints can be found in the following sections of this tutorial.

2.4.1 Formulas

A formula is an explicit definition of how to calculate the value for a slot. If we want to constrain the top of one object to be the same as another, then we define a formula for the `Am_TOP` slot of the dependent object. With constraints, the value of one slot always *depends* on the value of one or more other slots, and we say the formula in that slot has *dependencies* on the other slots.

An important point about constraints is that they are constantly maintained by the system. That is, they are evaluated once when they are first created, and then they are continually *re-evaluated* when any of their dependencies change. Thus, if several objects depend on the top of a certain rectangle, then all the objects will change position whenever the rectangle is moved.

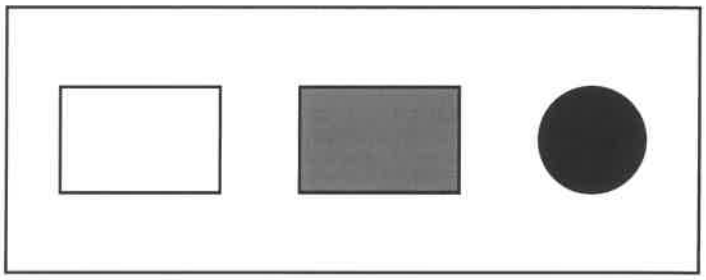


Figure 2-10: Three objects that are all aligned with the same top. The top of the gray rectangle is constrained to the white rectangle, and the top of the black circle is constrained to the top of the gray rectangle.

2.4.2 Declaring and Defining Formulas

There are several macros that are used to define formulas. These macros expand to conventional function definitions, but with special context information that Amulet uses to keep track of the constraint's dependencies. The particular macro you should use to define your formula depends on the type of the value to be returned from the formula.

`Am_Define_Formula (`*`type, formula_name`*`)` -- General purpose: returns specified *`type`*

`Am_Define_No_Self_Formula (`*`type, function_name`*`)` -- General purpose: returns specified *`type`*. Used when the formula does not reference the special `self` variable, so compiler warnings are avoided.

`Am_Define_Value_Formula (`*`formula_name`*`)` -- Return type is `Am_Value`

`Am_Define_Value_List_Formula (`*`formula_name`*`)` -- Return type is `Am_Value_List`

`Am_Define_Object_Formula (`*`formula_name`*`)` -- Return type is `Am_Object`

```
Am_Define_Style_Formula (formula_name) -- Return type is Am_Style
Am_Define_Font_Formula (formula_name) -- Return type is Am_Font
Am_Define_Point_List_Formula (formula_name) -- Return type is
            Am_Point_List
```

There are complimentary declaration macros that you can use in your own header files, when a formula is defined in one source file and referenced in another.

```
Am_Declare_Formula (type, formula_name)
Am_Declare_No_Self_Formula (type, formula_name)
Am_Declare_Value_Formula (formula_name)
Am_Declare_Value_List_Formula (formula_name)
Am_Declare_Object_Formula (formula_name)
Am_Declare_Style_Formula (formula_name)
Am_Declare_Font_Formula (formula_name)
Am_Declare_Point_List_Formula (formula_name)
```

2.4.3 An Example of Constraints

As our first example of defining constraints among objects, we will make the window in **Figure 2-10**. Let's begin by creating the white rectangle at an absolute position, and then create the other objects relative to it.

The constraints in the following examples will reference global values, and it is essential that the object variables and formulas be defined at the top-level of the program, outside of `main()`. Create the window and the first box with the following code.

```
// Defined at the top-level, outside of main()
Am_Object my_win, white_rect, gray_rect, black_arc;
...

  // Defined inside main()

  // Create the window and display it on the screen
  my_win = Am_Window.Create ("my_win")
    .Set (Am_LEFT, 20)
    .Set (Am_TOP, 50)
    .Set (Am_WIDTH, 260)
    .Set (Am_HEIGHT, 100);
  Am_Screen.Add_Part (my_win);

  // Create the white rectangle
  white_rect = Am_Rectangle.Create ("white_rect")
    .Set (Am_LEFT, 20)
    .Set (Am_TOP, 30)
    .Set (Am_WIDTH, 60)
    .Set (Am_HEIGHT, 40)
    .Set (Am_FILL_STYLE, Am_White);

  // Add the rectangle to the window
  my_win.Add_Part (white_rect);
```

We are now ready to create the other objects that are aligned with `white_rect`. We could simply create another rectangle and a circle that each have their top at 30, but this would lead to extra work if we ever wanted to change the top of all the objects, since each object's `Am_TOP` slot would have to be changed individually. If we instead define a relationship that depends on the top of `white_rect`, then whenever the top of `white_rect` changes, the top of the other objects will automatically change, too. Define and use a constraint that depends on the top of `white_rect` as follows:

```
// Define this at the top-level, outside of main()
Am_Define_Formula (int, top_of_white_rect) { // The formula is named top_of_white_rect
  return white_rect.GV (Am_TOP);               //   and returns an int
}
...

  // Define this inside main(), after white_rect
  gray_rect = Am_Rectangle.Create ("gray_rect")
    .Set (Am_LEFT, 110)
    .Set (Am_TOP, Am_Formula::Create (top_of_white_rect))
    .Set (Am_WIDTH, 60)
    .Set (Am_HEIGHT, 40)
    .Set (Am_FILL_STYLE, Am_Gray_Stipple);

  my_win.Add_Part (gray_rect);
```

Without specifying an absolute position for the top of the gray rectangle, we have constrained it to always have the same top as the white rectangle. The formula in the `Am_TOP` slot of the gray rectangle was defined using the macro `Am_Define_Formula` and

Am_Formula::Create(). The Am_Define_Formula macro helps to define a function to be used as a constraint. The formula is named top_of_white_rect, and returns an int.

The macro GV() means "get value", and it is just like Get(), except that GV() causes a dependency to be established on the referenced slot, so that the formula will be reevaluated when the value in the referenced slot changes. Typically, you will always want to use GV() inside of formulas, and Get() inside of normal functions.

To see if our constraint is working, bring up the Inspector on white_rect by hitting F1 while the mouse is positioned over the white rectangle. Change the top of white_rect and notice how the gray rectangle stays aligned with its top. This shows that the formula in gray_rect is being re-evaluated whenever its depended values change.

Now we are ready to add the black circle to the window. We have a choice of whether to constrain the top of the circle to the white rectangle or the gray rectangle. Since we are going to be examining these objects closely in the next few paragraphs, let's constrain the circle to the gray rectangle, resulting in an indirect relationship with the white one. Define another constraint and the black circle with the following code.

```
// Define this at the top-level, outside of main()
Am_Define_Formula (int, top_of_gray_rect) {
  return gray_rect.GV (Am_TOP);
}
...

  // Define this inside main(), after gray_rect
  black_arc = Am_Arc.Create ("black_arc")
    .Set (Am_LEFT, 200)
    .Set (Am_TOP, Am_Formula::Create (top_of_gray_rect))
    .Set (Am_WIDTH, 40)
    .Set (Am_HEIGHT, 40)
    .Set (Am_FILL_STYLE, Am_Black);

  my_win.Add_Part (black_arc);
```

At this point, you may want to inspect the white rectangle again and change its top just to see if the black circle follows along with the gray rectangle.

2.4.4 Values and constraints in slots

What happens if you set the Am_TOP of the gray rectangle now? The default for most slots, including the Am_TOP slot of Am_Rectangle, is for the value you set to replace the formula you put in the slot. Try setting the Am_TOP of the gray rectangle now, by hitting F1 over the rectangle, and editing the value of its Am_TOP slot. You should see that the rectangle will now have a different position which will not be recalculated, because the formula that was in the slot has been destroyed.

In some slots of certain objects, such as the button widgets, there are formulas in the slots by default which are required to maintain proper behaviour of the objects. If the formulas were destroyed, the object would no longer work as expected. These slots have

a special flag set which tells Amulet to keep the formula around even if you set the slot with a new value, and to reevaluate the formula if any of its dependencies change. Setting these slots with a new value does not replace the formula in the slot, it simply overrides the current cached value of the formula.

Any slot can be set so that formulas will not be destroyed when the slot is set. This is currently an advanced feature of Amulet, which is described in the advanced section of the ORE chapter.

2.4.5 Constraints in Groups

As mentioned in Section 2.3.3, parts can be stored in pointer slots of their group, making it easier for the parts to reference each other. Additionally, the owner is set in each part as they are added to a group. In this section, we will examine how pointer slots and variations on the GV function can be used to communicate among parts of a group.

The group we will use in this example will make the picture of concentric shapes in Figure 2-11. Suppose that we want to be able to change the size and position of the shapes easily, and that this should be done by setting as few slots as possible.

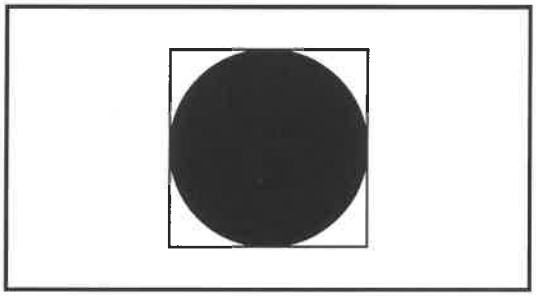


Figure 2-11: A group with two parts.

From the picture, we see that the dimensions of the rectangle are the same as the diameter of the circle. Therefore, it will be helpful to put slots for the size and position at the top-level of the group, and have the parts reference these top-level values through formulas.

```
// Declared at the top-level, outside of main()
Am_Slot_Key ARC_PART  = Am_Register_Slot_Name ("ARC_PART");
Am_Slot_Key RECT_PART = Am_Register_Slot_Name ("RECT_PART");

//self is an Am_Object parameter to all formulas that holds the object the constraint is in.
// The Am_Define_Formula macro expands to define self and some other necessary variables.
Am_Define_Formula (int, owner_width) {
  return self.GV_Owner().GV(Am_WIDTH);
}

Am_Define_Formula (int, owner_height) {
  return self.GV_Owner().GV(Am_HEIGHT);
}
...

  // Defined inside of main()
  Am_Object my_group = Am_Group.Create ("my_group")
    .Set (Am_LEFT, 20)
    .Set (Am_TOP, 20)
    .Set (Am_WIDTH, 100)
    .Set (Am_HEIGHT, 100)
    .Add_Part(ARC_PART, Am_Arc.Create ("my_circle")
        .Set (Am_WIDTH,  Am_Formula::Create (owner_width))
        .Set (Am_HEIGHT, Am_Formula::Create (owner_height)))
    .Add_Part(RECT_PART, Am_Rectangle.Create ("my_rect")
        .Set (Am_WIDTH,  Am_Formula::Create (owner_width))
        .Set (Am_HEIGHT, Am_Formula::Create (owner_height))
        .Set (Am_FILL_STYLE, Am_No_Style));
```

Both parts of `my_group` get their position and dimensions from the top-level slots in `my_group`. The reference to `my_group` from the arc is through the `GV_Owner()` function, which links the part to its group. The special variable `self` is used in the formulas to reference slots within the object that the formula is installed on. The arc's left and top are relative to the origin of `my_group`, so as it inherits a position of (0,0) from the `Am_Arc` prototype, it will appear at (20,20) in the window.

Finally, notice that the parts do not "inherit" any values from their owner. Adding parts to a group sets up a *group* hierarchy, where values travel back-and-forth over constraints, not inheritance links. If you want a part to depend on values in its owner, you have to define constraints.

The slot names for the parts could have been used to define the constraints, also. Instead of asking its owner for its dimensions, the rectangle part could have asked the arc for its dimensions. In this example the result would be the same, but here are alternate definitions for the rectangle's width and height formulas to illustrate the use of aggregate pointer slots:

```
Am_Define_Formula (int, arc_width) {
  return self.GV_Owner().GV_Part(ARC_PART).GV(Am_WIDTH);
}

Am_Define_Formula (int, arc_height) {
  return self.GV_Owner().GV_Part(ARC_PART).GV(Am_HEIGHT);
}
```

2.5 Interactors

Graphical objects do not directly respond to any input events. Instead, you create invisible "interactor" objects and attach them to objects to respond to input. Sometimes you may just want a function to be executed when the mouse is clicked, but often you will want changes to occur in the graphics depending on the actions of the mouse. Examples include moving objects around with the mouse, editing text with the mouse and keyboard, and selecting an object from a given set.

Interactors are described in detail in chapter 5, and a summary of interactors can be found in the object summary, section 8.6. It is important to note that all of the widgets (Section 2.6 and chapter 6) come with their interactors already attached. Therefore, you do not need to create interactors for the widgets.

The fundamental way that the interactors communicate with graphical objects is that they set slots in the objects in response to mouse movements and keyboard key strokes. That is, they generate side effects in the objects that they operate on. For example, the `Am_Move_Grow_Interactor` sets the left, top, width, and height slots of objects. The `Am_Choice_Interactor` sets the `Am_SELECTED` and `Am_INTERIM_SELECTED` slots to indicate when an object is currently being operated on. You might define formulas that depend on these special slots, causing the appearance of the objects (i.e., the graphics of the interface) to change in response to the mouse. The examples in Sections 2.5.2, 2.5.3, and 2.5.4 show how you can use interactors this way.

A more advanced way to use interactors (and widgets) is through their command objects (Section 2.5.5). Command objects contain methods that support undo, help, and selective enabling of operations associated with interactors and widgets. They can also contain a custom function that will be executed whenever the user operates the interactor or widget.

Figure 2-12 shows the general data flow when input events occur: the user hits a keyboard key or a mouse event, which is passed to the window manager. The Gem layer of Amulet converts it into a machine-independent form and passes it to the Interactors which finds the right interactor object to handle the event. Each interactor has an embedded command object that causes the appropriate action to take place. If this interactor is part of a widget, then the command object in the interactor calls the widget's command object. Eventually, some graphics will be modified in the Opal layer, which is automatically transformed into drawing calls at the Gem level, and then to the window manager.

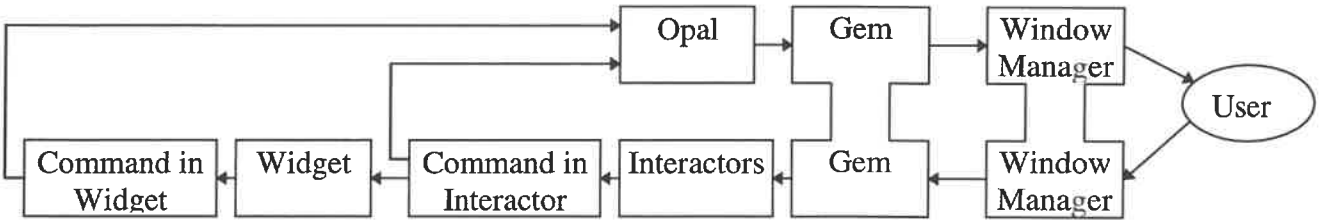


Figure 2-12: The data flow when events come from the user.

In this section we will see some examples of how to change graphics in conjunction with interactors. Section 2.7.2 describes how to use an important debugging function for interactors called `Am_Set_Inter_Trace()`. Although this tutorial only gives examples of using the `Am_One_Shot_Interactor` and `Am_Move_Grow_Interactor`, there are examples of interactors in demo and test programs included with the Amulet files. For example, see `samples/demo_space/demo_space.cc` in your Amulet source files (instructions for compiling and running these programs are in the Overview chapter).

2.5.1 Kinds of Interactors

The design of the interactors is based on the observation that there are only a few kinds of behaviors that are typically used in graphical user interfaces. Currently, Amulet supports five types of interactive behavior, which allows a wide variety of user actions in an interface. Below is a list of the available interactors.

- `Am_Choice_Interactor` - This is used to choose one or more from a set of objects. The user is allowed to move around over the objects (getting *interim feedback*) until the correct item is found, and then there will often be *final feedback* to show the final selection. The `Am_Choice_Interactor` can be used for selecting among a set of buttons or menu items, or choosing among the objects dynamically created in a graphics editor.
- `Am_One_Shot_Interactor` - This is used whenever you want something to happen immediately, for example when a mouse button is pressed over an object, or when a particular keyboard key is hit. Like the `Am_Choice_Interactor`, the `Am_One_Shot_Interactor` can be used to select among a set of objects, but it will not provide interim feedback—the one where you initially press will be the final selection. The `Am_One_Shot_Interactor` is also useful in situations where you are not selecting an object, such as when you want to get a single keyboard key.
- `Am_Move_Grow_Interactor` - This is useful in all cases where you want a graphical object to be moved or changed size with the mouse. It can be used for moving and growing objects in a graphics editor.

- `Am_New_Points_Interactor` - This interactor is used to enter new points, such as when creating new objects. For example, you might use this to allow the user to drag out a rubber-band rectangle for defining where a new rectangle should go.
- `Am_Text_Edit_Interactor` - This supports editing the text string of a text object. It supports a flexible *key translation table* mechanism so that the programmer can easily modify and add editing functions. The built-in mechanisms support basic text editing behaviors.
- `Am_Rotate_Interactor` - This interactor will support rotating graphical objects. It is not yet implemented.
- `Am_Gesture_Interactor` - This interactor will support free-hand gestures, such as drawing an X over an object to delete it, or encircling a set of objects to be selected. It is not yet implemented.
- `Am_Animation_Interactor` - This interactor will support animations and time-based events. It is not yet implemented.

2.5.2 The Am_One_Shot_Interactor

In this example, we will perform an elementary operation with an interactor. We will create a window with a white rectangle inside, and then create an interactor that will make it change colors when the mouse is clicked inside of it. First, make sure you have working code that creates a window (maybe from Section 2.2.4), then add the following definitions to your program. Remember to add the rectangle to your window using `Add_Part()`.

```
// Defined at the top-level, outside of main()
Am_Define_Style_Formula (compute_fill) {
  // bool is a Boolean type defined by Amulet.  Often you need to cast the
  // value returned from GV, since a slot can contain any type of object.
  if ((bool) self.GV (Am_SELECTED))
    return Am_Black;
  else
    return Am_White;
}
...

  // Defined inside main()
  Am_Object changing_rect = Am_Rectangle.Create ("changing_rect")
    .Set (Am_LEFT, 20)
    .Set (Am_TOP, 30)
    .Set (Am_WIDTH, 60)
    .Set (Am_HEIGHT, 40)
    .Set (Am_SELECTED, false)  // Set by the interactor
    .Set (Am_FILL_STYLE, Am_Formula::Create (compute_fill));

  my_win.Add_Part (changing_rect);
```

From the definition of the `compute_fill` formula, you can see that if the `Am_SELECTED` slot in `changing_rect` were set to `true`, then its color would turn to black. You can test

this by bringing up the Inspector on changing_rect, and editing the value of the slot. Conveniently, setting the Am_SELECTED slot is one of the side effects of the Am_One_Shot_Interactor. The following code defines an interactor which will set the Am_SELECTED slot of an object, and attaches it to changing_rect.

```
Am_Object color_inter = Am_One_Shot_Interactor.Create ("color_inter");
changing_rect.Add_Part (color_inter);
```

Now you can click on the rectangle repeatedly and it will change from white to black, and back again. From this observation, and knowing how we defined the compute_fill formula of changing_rect, you can conclude that the Am_One_Shot_Interactor is setting (and clearing) the Am_SELECTED slot of the object. This is one of the functions of this type of interactor.

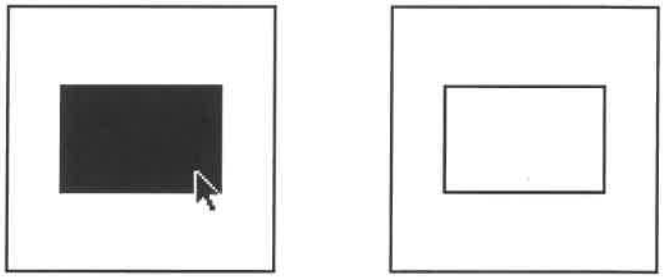


Figure 2-13: The rectangle changing_rect when its Am_SELECTED slot is false (the default), and when it is set to true by the interactor (when the mouse is clicked over it).

2.5.3 The Am_Move_Grow_Interactor

From the previous example, you can see that it is easy to change the graphics in the window using the mouse. We are now going to define several more objects in the window and create an interactor to move and grow them. The following code creates a prototype circle and several instances of it.

```
Am_Object moving_circle = Am_Arc.Create ("moving_circle")
  .Set (Am_WIDTH, 40)
  .Set (Am_HEIGHT, 40)
  .Set (Am_FILL_STYLE,  Am_No_Style);

Am_Object objs_group = Am_Group.Create ("objs_group")
  .Set (Am_WIDTH, Am_Width_Of_Parts)
  .Set (Am_HEIGHT, Am_Height_Of_Parts)
  .Add_Part (moving_circle.Create())
  .Add_Part (moving_circle.Create().Set (Am_LEFT, 50))
  .Add_Part (moving_circle.Create().Set (Am_LEFT, 100));
```

Now let's create an instance of the Am_Move_Grow_Interactor which will cause the moving circles to change position. The following interactor, when added to objs_group, works on all the parts of that group.

```
Am_Object objs_mover = Am_Move_Grow_Interactor.Create ("objs_mover")
  .Set (Am_START_WHERE_TEST, (Am_Object_Proc*)&Am_Inter_In_Part);

objs_group.Add_Part(objs_mover);
```

Notice that the interactors by default operate directly on the object they are attached to (with `Add_Part`), as shown in the `changing_rect` example. However, by setting the `Am_START_WHERE_TEST` slot to `Am_Inter_In_Part`, you can have an interactor work on any part of a group. A few other tests are provided in `inter.h`, or you can write your own start-where-test procedure to return whatever object is appropriate.

Now if you press and drag in any of the circles, they will follow the mouse. This is because the interactor sets the left and top slots of the objects it acts on.

While the program is running, you can bring up the `Inspector` on the `objs_mover` interactor and change the value of some of its slots to alter its behavior. (You will have to inspect a circle first, then inspect its owner, and then find the interactor in its list of parts to activate the `Inspector` for `objs_mover`.) Click in the value field of the `objs_mover`'s `Am_GROWING` slot and change the value to `true`. Now dragging the circles will cause them to change size rather than move.

2.5.4 A Feedback Object with the Am_Move_Grow_Interactor

Now let's add a feedback object to the window that will work with the moving circles. In this case, the feedback object will appear whenever we click on and try to drag a circle. The mouse will actually drag the feedback object, and then the real circle will jump to the final position when the mouse is released.

Our feedback object will be a circle with a dashed line. The `feedback_circle` object defined below will have its left, top, and visible slots set by the interactor. Given our `moving_circle` prototype, the feedback object is easy to define:

```
Am_Object feedback_circle = moving_circle.Create ("feedback_circle")
  .Set (Am_LINE_STYLE, Am_Dashed_Line)
  .Set (Am_VISIBLE, false);

my_win.Add_Part (feedback_circle);


// The definition of the interactor, with feedback object
Am_Object objs_mover = Am_Move_Grow_Interactor.Create ("objs_mover")
  .Set (Am_START_WHERE_TEST, (Am_Object_Proc*)&Am_Inter_In_Part)
  .Set (Am_GROWING, true)    // Makes the circles grow instead of move
  .Set (Am_FEEDBACK_OBJECT, feedback_circle);

objs_group.Add_Part (objs_mover);

// Don't forget to add feedback_circle and objs_mover to the right owners!
```

The Am_VISIBLE slot of feedback_circle is initialized to false, because we do not want it visible unless it is being used by objs_mover. The interactor will set the Am_VISIBLE slot to true and false as it is needed. Now when you move or grow the circles with the mouse, the feedback object will follow the mouse, instead of the real circle following it directly.

2.5.5 Command Objects

All interactors and widgets have command objects associated with them stored as the Am_COMMAND part. Command objects contain functions that determine what the interactor will do as it operates. For example, you can store a function in a command object that will be executed as the interactor runs in order to cause side-effects in your program. You can also store functions in a command object to support undo, help, and selective enabling of operations. There is a library of pre-defined command objects, so you can often use a command object from the library without writing any code. See demo_space and testinter for examples of how to use command objects.

Most interactors have three different things that they do: they directly modify the associated graphical objects (like setting the Am_SELECTED slot), they set the Am_VALUE field of their attached command object, and they call the Am_DO_ACTION method of the attached command object. Widgets, such as menus or scroll bars, do the last two of these operations, plus changing their appearance on the screen.

To use the Am_VALUE slot of the command object, you can establish a constraint from your object to the Am_VALUE slot of the Am_COMMAND part in the interactor or widget.

Putting a "call-back" procedure in a command object is somewhat more complicated with interactors and widgets. Currently, the Am_DO_ACTION procedure in the command object causes the interactor or widget to operate correctly. If you replace the procedure, the object may not do its normal actions. Therefore, you should make sure the standard code is called in addition to your code. (This will probably be fixed in the next release.) You can do this by adding a call to the prototype's method inside your procedure before your custom code. For example, suppose we are creating a custom Am_DO_ACTION for a Am_Move_Grow_Interactor:

```
void my_do_action (Am_Object command_obj) {
// First, call the standard do action from the command object in the Am_Move_Grow_Interactor.
Am_Call(Am_Object_Proc, Am_Move_Grow_Interactor.Get_Part(Am_COMMAND),
                Am_DO_ACTION, (command_obj));

//now do custom stuff here
...
}
```

Am_Call is a macro that takes a procedure type, an object to get the procedure out of, the slot the procedure is stored in and the parameters to the procedure. Note that the parameters have to be in an extra set of parentheses. Here, we need to get the Am_DO_ACTION out of the standard move-grow interactor's command object.

Remember that you must set the Am_DO_ACTION and access the Am_VALUE slot of the Command object in the Interactor or widget, and *not* access those slots directly in the Interactor or widget themselves. Thus:

```
my_inter.Get_Part(Am_COMMAND).Get(Am_VALUE); //right
my_inter.Get_Part(Am_COMMAND)
   .Set(Am_DO_ACTION, my_do_action); //right
my_inter.Get(Am_VALUE); //WRONG
my_inter.Set(Am_DO_ACTION, my_do_action); //WRONG
```

2.5.6 The Am_Main_Event_Loop

In order for interactors to perceive input from the mouse and keyboard, the main event loop must be running. This loop constantly checks to see if there is an event, and processes it if there is one. The automatic redrawing of graphics also relies on the main-event-loop. Exposure events, which occur when one window is uncovered or *exposed*, cause Amulet to refresh the window by redrawing the objects in the exposed area.

All Amulet programs should call two routines at the end of `main()`. `Am_Main_Event_Loop()` should be called, followed by `Am_Cleanup()`, which destroys the resources Amulet allocated. Your program will continue to run until Amulet perceives the escape sequence, which by default is SHIFT-ESC. Typically, your program will have some sort of Quit button. This should call `Am_Exit_Main_Event_Loop()`, which will cause the main-event-loop to terminate.

2.6 Widgets

The Amulet Widgets are a set of ready-made gadgets that can be be added directly to a window or a group just like other graphical objects. You do not have to define separate interactors to operate the gadgets -- they already have their own interactors. They have slots that can be set in order to customize their apperance and behavior. Generally, they are controls that are commonly found in an interface including scroll bars, menus, buttons and editable text fields. Section 8.9 summarizes the widget objects, and Chapter 6 discusses them all in detail.

The Widgets will eventually be available in several versions, simulating the look-and-feel of the standard widgets available in Motif, Windows, and Macintosh toolkits. To date, however, only the Motif style of widgets have been implemented in Amulet; they work on all platforms, but always look like Motif widgets. Examples of the widgets can be found in several Amulet demos (including `demo_space`, found in the `samples/demo_space` subdirectory of your Amulet source files -- see the Overview chapter for instructions on compiling and running Amulet demos).

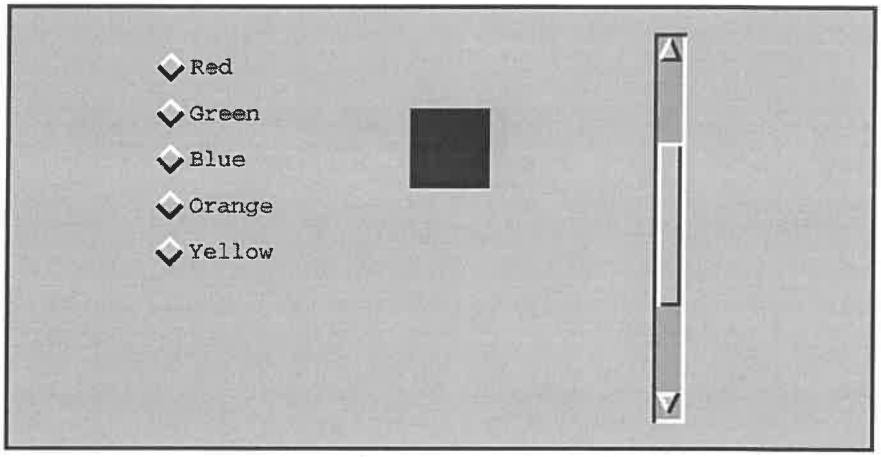


Figure 2-14: A panel of radio buttons and a vertical scroll bar, affecting a rectangle.

In this section we will use a radio button panel and a scroll bar to change the appearance of a rectangle.

There are two ways to interact with widgets. You can define a formula to depend on the value of the command object in the widget, or you can define a function that will be executed by the widget's command object whenever the user operates the widget.

The code below defines the radio button panel pictured in **Figure 2-14**. As mentioned above, in section 2.5.5, all widgets store their value in the Am_VALUE slot of the command object stored as the Am_COMMAND part. Here, we define a formula for the filling style of the rectangle that depends on the value of the button panel. This formula is reevaluated every time the buttons are operated, so the rectangle changes color.

```
// Declared at the top-level, outside of main()
Am_Object color_buttons, color_rect;

// Declared at the top-level, outside of main()
Am_Define_Style_Formula (color_from_panel) {
  Am_Object cmd = color_buttons.GV_Part(Am_COMMAND);
  Am_String s;
  s = cmd.GV (Am_VALUE);
  if ((const char*)s) {
    if (strcmp(s, "Red") == 0) return Am_Red;
    else if (strcmp(s, "Blue") == 0) return Am_Blue;
    else if (strcmp(s, "Green") == 0) return Am_Green;
    else if (strcmp(s, "Yellow") == 0) return Am_Yellow;
    else if (strcmp(s, "Orange") == 0) return Am_Orange;
    else return Am_White;
  }
  else return Am_White;
}...
```

```
// Defined inside main()
color_buttons = Am_Radio_Button_Panel.Create("color_buttons")
  .Set (Am_LEFT, 10)
  .Set (Am_TOP, 10)
  .Set (Am_ITEMS, Am_Value_List ()      // An Am_Value_List supports an arbitrary list
        .Add("Red")                     //   of dynamically typed values
        .Add("Blue")
        .Add("Green")
        .Add("Yellow")
        .Add("Orange"))
  .Set (Am_FILL_STYLE, Am_Motif_Gray);

// Defined inside main()
color_rect = Am_Rectangle.Create("color_rect")
    .Set(Am_LEFT, 100)
    .Set(Am_TOP, 50)
    .Set(Am_WIDTH, 50)
    .Set(Am_HEIGHT, 50)
    .Set(Am_FILL_STYLE, Am_Formula::Create(color_from_panel));

my_win.Add_Part (color_buttons)
  .Add_Part (color_rect);
```

Now let's create the scroll bar to change the position of the rectangle. As with the button panels, we could define a formula that depends on the value of the widget's command object. Instead, let's use the Am_DO_ACTION of the scroll bar's command object to call a function each time the widget is operated.

```
// Defined at the top-level, outside of main()
Am_Object my_scrollbar;

// Defined at the top-level, outside of main()
void my_scrollbar_do (Am_Object cmd) {
  int value = cmd.Get(Am_VALUE);
  color_rect.Set (Am_TOP, 20 + value);
  //Now, call the standard do action from the command object in the Am_Vertical_Scroll_Bar.
  Am_Call(Am_Object_Proc, Am_Vertical_Scroll_Bar.Get_Part(Am_COMMAND),
                    Am_DO_ACTION, (cmd));
}
...

// Defined inside main()
  my_scrollbar = Am_Vertical_Scroll_Bar.Create ("my_scrollbar")
    .Set (Am_LEFT, 250)
    .Set (Am_TOP, 10)
    .Set (Am_SMALL_INCREMENT, 5)
    .Set (Am_LARGE_INCREMENT, 20)
    .Set (Am_VALUE_1, 0)
    .Set (Am_VALUE_2, 100);

  my_scrollbar.Get_Part(Am_COMMAND).Set(Am_DO_ACTION, my_scrollbar_do);

  my_win.Add_Part (my_scrollbar);
```

2.7 Debugging

2.7.1 The Inspector

The `Inspector` is an important tool for examining properties of objects. This tool is accessible in Amulet by adding the "`#include <amulet/debugger.h>`" statement at the top of your file, and adding the file `src\debug\inspectr.cpp` to your project if you're using Visual C++. A call to `Am_Initialize_Inspect(`*win*`)` is required on each *win* that you want to be able to launch the `Inspector` from. This call installs interactors in the window that make it sensitive to the `F1` keystroke.

If your keyboard does not have an `F1` key, or hitting it does not seem to do anything, you can start the `Inspector` manually by calling the function `Am_Inspect(`*obj*`)` with the object you want to inspect as its argument. The function `Am_Text_Inspect(`*obj*`)` is similar, except that it prints the object's slots and values to `stdout` instead of popping up an interactive window.

The `Inspector` shows all inherited and local slots of an object, with the inherited slots shown in blue. Editing an inherited value causes the value to become local, and changes its color from blue to black. Currently, only the editing of integer values is supported, but in future versions of Amulet you will be able to edit all types of values. .

In the `Inspector` window, clicking the right mouse button over a value that is an object will cause that object's slots and values to be displayed in the window. To display an object in a new window instead, hold down the `SHIFT` key while pressing the right mouse button over its name in the `Inspector` window. When you are finished with the `Inspector`, you can type `^q` (hold down the `CONTROL` key and press 'q') to make the `Inspector` windows disappear.

The `Inspector` does not keep the display up-to-date with the objects. To refresh the display, hit `^r`. Other commands include `^i` to turn on interactor tracing (Section 2.7.2) and `^p` to redisplay he previous object shown in the `Inspector` window.

2.7.2 Tracing Interactors

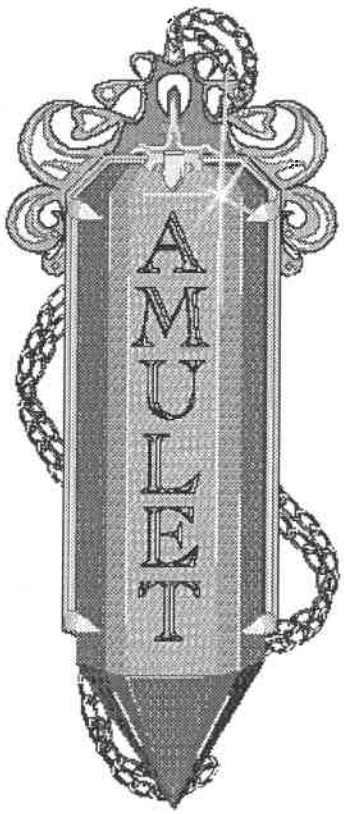
The interactors and default commands provide a number of mechanisms to help programmers debug their interactions. The primary one is a *tracing* mechanism that supports printing to standard output (`cout`) whenever an "interesting" interactor event happens. Amulet supplies many options for controlling when printout occurs, as described below (full details are in the Interactors chapter). You can either set these parameters in your code and recompile, or they can be dynamically changed as your application is running, if you have a C++ interpreter like ObjectCenter, or from the interactive `Inspector` (see Section 2.7.1).

```
typedef enum { Am_INTER_TRACE_NONE, Am_INTER_TRACE_ALL,
          Am_INTER_TRACE_EVENTS, Am_INTER_TRACE_SETTING,
          Am_INTER_TRACE_PRIORITIES, Am_INTER_TRACE_NEXT,
          Am_INTER_TRACE_SHORT } Am_Inter_Trace_Options;

void Am_Set_Inter_Trace(); //prints current status
void Am_Set_Inter_Trace(Am_Inter_Trace_Options trace_code);
void Am_Set_Inter_Trace(Am_Object* inter_to_trace);
void Am_Clear_Inter_Trace();
```

By default, tracing is off. Each call to `Am_Set_Inter_Trace` *adds* tracing of its parameter to the set of things being traced (except for `Am_INTER_TRACE_NONE` which clears the entire trace set). The options for `Am_Set_Inter_Trace` are:

- no parameters: If `Am_Set_Inter_Trace` is called with no parameters, it prints out the current tracing status.
- `Am_INTER_TRACE_NONE`: If `Am_Set_Inter_Trace` is called with zero or `Am_INTER_TRACE_NONE`, then it sets there to be nothing being traced. This is the same as calling `Am_Clear_Inter_Trace`.
- `Am_INTER_TRACE_ALL`: Traces everything. The `Inspector` command `^i` does this.
- `Am_INTER_TRACE_EVENTS`: Only prints out the incoming events, and not what happens as a result of those events. When you trace anything else, Amulet automatically also adds `Am_INTER_TRACE_EVENTS` to the set of things to trace, so you can tell why things are being updated.
- `Am_INTER_TRACE_SETTING`: This very useful option just shows which slots of which objects are being set by interactors and commands. It is very useful for determining why an object slot is being set.
- `Am_INTER_TRACE_PRIORITIES`: This prints out changes to the priority levels.
- `Am_INTER_TRACE_NEXT`: This turns on tracing of the next interactor to be executed. This is very useful if you don't know the name of the interactor to be traced.
- `Am_INTER_TRACE_SHORT`: This prints out only the name of the interactors which are run.
- an interactor: This prints lots of information about the execution of that one interactor.



3. ORE Object and Constraint System

The Amulet Object System

Abstract

This chapter describes "ORE", the object and constraint level of Amulet. ORE allows programmers to create objects as instances of other objects, and define constraints among objects that keep properties consistent. For advanced users and researchers, ORE allows demons to be defined on various object operations, slot inheritance to be controlled, and even entirely new constraint solvers to be written.

This research was sponsored by NCCOSC under Contract No. N66001-94-C-6037, Arpa Order No. B326. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NCCOSC or the U.S. Government.

3.1 Introduction

This is the chapter for the Amulet object and constraint system, nicknamed ORE which stands for Object Registering and Encoding. (Naturally, ORE is a reverse engineered acronym so any day its meaning may change without notice.) This portion of the manual covers the basic operation and use of ORE and its facilities. The basic operation of ORE covers general use of objects and the kinds of values that can be stored in them. Also covered is how to make and use *formulas* that can be used to attach values together. At the end of this chapter, the means for writing new kinds of value types called wrapper types is covered.

ORE is used in Amulet as the means for representing all higher-level graphical concepts. Rectangles, for instance, are created using an exported object called `Am_Rectangle`. The process for moving a rectangle is also represented as an object. It is called `Am_Move_Grow_Interactor`. Lower-level graphical features like colors and fonts are not ORE objects in Amulet. Instead they are "wrapper types" or "wrapper values," "wrapper objects," or just plain "wrappers." What makes wrappers different from regular C++ objects is that they contain data that derives from the class `Am_Wrapper`. This makes it easy to fetch and store them in ORE objects.

The coding style for ORE objects is declarative. That means the values and behaviors of objects are specified mostly at the time an object gets created by storing initial values and declaring constraints in the needed slots. All high level objects defined in Amulet are designed to be used in a declarative way. Normal programming practice consists of choosing the kinds of objects your program needs, finding out what slots these objects contain and what the semantics of the slots do, and finally assigning values to the slots and grouping the objects together to make the final application.

Much of the concepts and processes in ORE are derived from the Garnet object system called KR. KR differs from ORE in that it was originally designed to be a artificial intelligence knowledge representation framework. The graphics came later and KR underwent an evolution that made it more compatible with the demands of a graphical system. ORE begins where KR left off. In ORE, some KR features were abandoned like multiple inheritance. Many of the good KR features that only made it into KR in the last couple releases have been put into ORE right from the start. These features include dynamic type checking and an efficient algorithm for formula propagation and evaluation. And, of course, there are many brand new features in ORE that were never part of KR. Things like the owner-part hierarchy and the ability to install multiple constraint solvers which are hoped to become very useful to Amulet programmers.

ORE features like setting up type checking and writing a new constraint solver are quite advanced and are covered in Section 3.10 of this chapter. These sorts of features are not necessary for the novice Amulet programmer to make working applications, but are intended to be used by system programmers or researchers that want to extend Amulet.

3.2 Include Files

The various objects, types and procedures described in this chapter are spread out though several `.h` files. Typically, you will include `amulet.h` in your code which automatically includes all the ones you need. This chapter tells you where various names are defined so you can look up their exact definitions. The main include files relevant to ORE are:

- `standard_slots.h`: The functions for defining slot names, and the list of Amulet-defined slots
- `objects.h`: All of the basic object methods, the `Am_Value` type, slot and part iterators, and the `Am_Call` macros
- `objects_advanced.h`: Needed if you use any of the advanced features discussed in section 3.10.
- `value_list.h`: Defines the type Am_Value_List and all its related methods.

3.3 Objects and Slots

The Amulet object system, ORE, supports a "prototype-instance" object system. Essentially, an object is a collector for data in the form of "slots." Each slot in an object is similar to a field in a structure. Each slot stores a single piece of data for the object. A `Am_Rectangle` object, for example, has a separate slot for its left, top, width, and height.

An ORE object is different from a C++ object in many ways. The slots of ORE objects are dynamic. A program can add and remove slots as required by the given situation. Whole new types of objects can be created on demand without requiring anything to be recompiled. In C++, only the object's data can be modified and not its structure without recompiling. Furthermore in ORE, the types stored into each slot can change. For instance, the `Am_VALUE` slot can hold an integer at one time, and then a string later. ORE keeps track of the current type stored in the slot, and supports full integration with the C++ type system, including dynamic type checking.

3.3.1 Get and Set

The basic operations performed on slots are `Get` and `Set`. A slot is essentially a key–value pair. `Get` takes a slot key and returns the slot's value. `Set` takes a key and value and replaces the slot's previous value with the one given. Creating new slots is done by performing `Set` using a key that has not been used before in that object. Another way to think of an object is as a name space for slot keys. A single key can have only one value in a given object.

```
my_object.Set (Am_LEFT, 5);  // Set left slot to value 5
int position = my_object.Get (Am_TOP);  // Get the value of slot Am_TOP
my_object.Set (Am_ANGLE1, 45.3);  // Set the angle1 slot to float 45.3
```

Calling `Get` on a slot which does not exists raises an error. Make sure you initialize objects with values to avoid this problem. To test whether a slot exists yet, you can use the `Get_Slot_Type` method on objects (see Section 3.3.3), or else use the `Am_Value` form of `Get` (see Section 3.3.10).

If you are interested in how this is implemented, there are multiple versions of `Set` and the compiler differentiates automatically. However, because C++ does not allow methods to be disambiguated on their return type, the method `Get` actually returns a `Am_Slot` reference which in turn has a multitude of casting operators to convert it to the type one desires. This is usually invisible to Amulet programmers. One should never save the `Am_Slot` reference directly because slots are dynamic entities and a `Am_Slot` stored in a variable quite literally grows stale.

3.3.2 Slot Keys

A slot key in ORE is simply an unsigned integer. An example of a slot key is `Am_LEFT`. `Am_LEFT` is actually the integer 100 as defined in `standard_slots.h`, but one uses the name `Am_LEFT` because it is more descriptive. The slot `Am_LEFT` is used in all the graphical objects like rectangles, circles, and windows and it represents the leftmost position of that object. Potentially, the slot key, 100, could be used in another object with semantics completely different from those used in graphical objects, in essence 100 could be a key besides `Am_LEFT`. However, ORE provides mechanisms to avoid this kind of inconsistency and makes certain that integers and slot names map one to one. The string names associated with slots are mainly used for debugging. For example, they are printed out by the inspector. The string names for slots are not used during normal program execution.

Programmers can define new slot keys for their own use by using functions defined in ORE (in `standard_slots.h`). There are four essential functions to do this: `Am_Register_Slot_Name`, `Am_Register_Slot_Key`, `Am_Get_Slot_Name`, and `Am_Slot_Name_Exists`.

`Am_Register_Slot_Name` is the major function for defining new slot keys. The function returns a key which is guaranteed not to conflict with any other key chosen by this function (it is actually just a simple counter). The return value is normally stored in a global variable which is used throughout the application code. If the string name passed already has a key associated with it, `Am_Register_Slot_Name` will return the old key rather than allocating a new one. Thus, `Am_Register_Slot_Name` can also be used to look up a name to find out its current key assignment.

```
Am_Slot_Key MY_FOO_SLOT = Am_Register_Slot_Name ("My Foo Slot");
```

We recommend that programmers define their slots this way, as shown in the various example programs.

Am_Register_Slot_Key is for directly pairing a number with a name. This is useful for times when one does not want to use a global variable to store the number returned by Am_Register_Slot_Name. The number and name chosen must be known beforehand not to conflict with any other slot key chosen in the system. The range of numbers that programmers are allowed to use for their own slot keys is 10000 to 29999. Numbers outside that range are allocated for use by Amulet. The number of new slot keys needed by an application is likely to be small so running out of numbers is not likely to be a problem. The main concern will be conflicting with numbers chosen by other applications written in Amulet.

```
#define MY_BAR_SLOT 10500

Am_Register_Slot_Key (MY_BAR_SLOT, "My Bar Slot");
```

The functions Am_Get_Slot_Name and Am_Slot_Name_Exists are used for testing the library of all slot keys for matches. This is especially useful when generating keys dynamically from user request.

```
const char* name = Am_Get_Slot_Name (MY_BAR_SLOT);
cout << "Slot " << MY_BAR_SLOT << " is named " << name << endl;

if (Am_Slot_Name_Exists (name)) cout << "Slot already exists\n";
```

3.3.3 Slot Types

The value of Am_LEFT in a graphical object is an integer specifying a pixel location. Hence slot values have types, specifically the Am_LEFT slot has integer type in graphical objects. The type of a slot's value is determined by whatever value is stored in the slot. A slot can potentially have different types of values at different times depending on how the slot is used, but a given value has only one type so that a slot has only one type at a time. Thus, slots are "dynamically typed" like variables in Lisp.

The types supported in ORE are the majority of the simple C++ types including integer, float, double, character, and boolean. Also supported are some more high-level types like string, ORE object, a function type, and void pointer. Although void* can be used to store any type of object, ORE supports a type called Am_Wrapper which is used to encapsulate C++ classes and structures so that general C++ data can be stored in slots while still maintaining a degree of type checking.

`Am_NONE` -- the value of the slot does not exist
`Am_WRAPPER` -- used for encapsulating generic C++ classes
`Am_OBJECT` -- an ORE object
`Am_INT` -- signed integer
`Am_LONG` -- signed long integer
`Am_BOOL` -- the type `bool`
`Am_FLOAT` -- floating point value
`Am_DOUBLE` -- double precision floating point value
`Am_CHAR` -- a single character
`Am_STRING` -- stores a character string as class `Am_String` (section 3.3.6), which is readily converted into a `const char*`
`Am_VOIDPTR` -- `void*` in Unix and `unsigned char*` in Windows. A `typedef` called `Am_Ptr` can be used as a cast to make code portable between Unix and Windows
`Am_PROC` -- the type `void (*) (Am_Object)`, used for storing procedures like call backs and methods

Figure 3-1: ORE Types for Slot Values, defined by `Am_Slot_Type` in `objects.h`

```
my_object.Set (Am_LEFT, 50);
Am_Slot_Type slot_type = my_object.Get_Slot_Type (Am_LEFT);
// slot_type == Am_INT
my_object.Set (Am_FILL_STYLE, Am_Blue);
Am_Slot_Type slot_type = my_object.Get_Slot_Type (Am_FILL_STYLE);
// slot_type == Am_WRAPPER
```

3.3.4 The Basic Types

As shown by the examples above, the `Set` and `Get` operators are overloaded so that the normal built-in C++ primitive types can be readily used in Amulet. This section discusses some details of the primitive types, and the next few sections discuss some specialized types.

Usually, the C++ compilers can tell the appropriate types of slots from the various declarations. Thus, the compiler will correctly figure out which `Set` to use for each of the following:

```
my_object.Set (Am_LEFT, 50);  //uses int
my_object.Set (Am_TEXT, "Foo"); //uses Am_STRING
my_object.Set (Am_PERCENT_VISIBLE, 0.75); //uses Am_FLOAT
long lng = 600000
my_object.Set (Am_VALUE_1, lng);
```

However, in some cases, the compiler cannot tell which version to use. In these cases, the programmer must put in an explicit type cast:

```
 //without cast, compiler doesn't know whether to use bool, int, void*, ...
if((bool)my_object.Get (Am_VISIBLE)) ...
 //without cast, compiler doesn't know whether to use int, long or float
int i = 5 + (int)my_object.Get(Am_LEFT);
```

Unfortunately, some compilers seem to get confused with perfectly unambiguous expressions involving Am_Objects and Am_Wrappers. In this case, you need to put the variable declaration in a separate statement from its assignment:

```
~~Am_Object obj = my_object.Get(Am_PARENT); //fails on some compilers~~
Am_Object obj; //using two statements works!
obj = my_object.Get(Am_PARENT);
```

Am_INT is the same as Am_LONG on Unix machines (32 bits), but on Windows an Am_INT is only 16 bits, so you should be careful to use long whenever the value might overflow 16 bits if you want to have portable code.

The Am_Ptr type (defined in types.h) should be used where-ever you would normally use a void* pointer, because Visual C++ cannot differentiate void* from some more specific pointers used by Amulet. Am_Ptr is defined as void* in Unix and unsigned char* in Windows.

3.3.5 Bools

Amulet makes extensive use of the bool type supplied by some C++ compilers (gcc). For compilers that do not support it (Visual C++, ObjectCenter, etc.), Amulet defines bool as int and defines true as 1 and false as 0, so you can still use bool in your code. When bools are supported by the compiler, Amulet knows how to return a bool from any kind of slot value. For example, if a slot contains a string and you convert it to a bool, it will return true if there is a string and false if the string is null. However, for compilers that do not support bool, conversion to an int is *not* provided, so counting on this conversion is a bad idea. Instead, it would be better to get the value into a Am_Value type and test that for Valid (see section 3.3.10).

3.3.6 The Am_String Class

The Am_String type (defined in object.h) allows simple, null terminated C++ strings (char*) to be conveniently stored and retrieved from slots. It is implemented as a form of wrapper (see section 3.3.9). An Am_String can be created directly from a char* type, likewise it can be compared directly against a char*. Because Am_String is a Am_Wrapper which is a reference counted structure, the programmer need not worry about the string's memory being deallocated in a local variable even if an object slot that holds a pointer to the same string gets destroyed.

The Am_String class will make a copy of the string if the programmer wants to modify its contents. The Am_String class does not allow the programmer to perform destructive modification on the string's contents.

Listed below are the basic methods defined for Am_String:

```
Am_String ()
Am_String (const char* initial)
```

The constructor that takes no parameters essentially creates a NULL char pointer. It is not equivalent to the string "". The second constructor creates the Am_String object with a C string as its value. The C string must be '\0' terminated so as to be usable with the standard string functions like strcpy and strcmp. The Am_String object will allocate memory to store its own copy of the string data.

```
operator const char* ()
operator char* ()
```

These casting operators make it easy to convert a Am_String to the more manipulable char* format. When a programmer casts to const char*, the string cannot be modified so no new memory needs to be allocated. When the programmer casts to char*, however, the copy of the string stored in object slots are protected by making a local copy that can be modified. The modified string can be set back to an object slot by calling Set.

3.3.7 Storing Methods in Slots

ORE treats methods (procedures) stored in slots exactly the same as data. Thus, method slots can be dynamically stored, retrieved, queried and inherited like all other slots. A few macros are provided for making the use of methods easier.

Methods stored in slots are normally all typed as Am_Object_Proc which is specifically declared as:

```
typedef void Am_Object_Proc (Am_Object context);
```

It is up to the programmer to keep straight what the actual number of parameters needed by the actual method stored in the slot. (This will probably be fixed in the next version.) Typically, there will be a typedef of the actual signature of the method you are storing, and you use this name with the Am_Call macro to invoke the method.

3.3.8 Calling methods

Am_Call (defined in object.h) takes the type, the object to get the method from, the slot name, and the list of parameters to the function *in an extra pair of parentheses:*

```
//This is defined in inter.h.  See the interactors chapter for what it does
typedef Am_Object Am_Where_Function (Am_Object inter, Am_Object object,
                Am_Object event_window, Am_Input_Char ic, int x, int y);

//store my specific where function called My_Where_Test into a slot
inter.Set (Am_START_WHERE_TEST, (Am_Object_Proc*)My_Where_Test);

//call the function in that slot
Am_Call (Am_Where_Function, inter, Am_START_WHERE_TEST,
                (inter, current_ic, 20, 50));
```

`Am_Call` does nothing if the procedure is not found (if the slot does not exist or contains `NULL`). You will get a run-time error from Amulet if the slot contains a non-procedure value, and usually a segmentation fault or something if the procedure is not of the same type as the type passed to `Am_Call`.

There is also `Am_Function_Call` which can be used for methods in slots that return a value. This is used as:

```
Am_Function_Call (Func_type, object, slot, return_variable, (args));
```

where the return_variable is the name of the variable into which the return of the function is to be assigned. For example:

```
Am_Function_Call (Int_Func_Type, obj2, COUNT_METHOD_SLOT, i, (obj2));
```

3.3.9 Using Wrapper Types

Although you could store C++ objects into ORE slots as a `void*`, ORE provides the `Am_Wrapper` type to "wrap" the C++ objects. `Am_Wrappers` provide dynamic type checking and memory management to the objects. These wrapper objects add a degree of safety to slots without sacrificing the dynamic aspects. Making new wrapper types is discussed in Section 3.10.2 and requires some care. On the other hand, using wrapper types is extremely simple. Notable wrapper types in Amulet are `Am_Style`, `Am_Font`, `Am_String`, `Am_Value_List` (see section 3.7), and especially `Am_Object` itself. Getting and setting a wrapper is syntactically identical to getting and setting an integer.

```
Am_Style blue (0.0, 0.0, 1.0); // Am_Style is a wrapper type
my_object.Set (Am_FILL_STYLE, blue); // using a wrapper with Set
Am_Style color;
color = my_object.Get (Am_FILL_STYLE); // a wrapper with Get
```

Even though both `Am_Object` and `Am_String` are both wrapper types, ORE still distinguishes their type as separate from other wrapper types. For `Am_Object`, the slot type `Am_OBJECT` is used, for `Am_String`, the type `Am_STRING` is used. This aids in distinguishing these two very common kinds of values from other wrappers.

3.3.9.1 Standard Wrapper Methods

Amulet wrappers provide a number of useful methods for querying about their state and for testing whether a given `Am_Wrapper*` belongs to a given class. These methods are common across all wrapper objects that Amulet provides. The methods are also available when programmers build their own wrapper objects using the standard macros.

The first thing that all built-in wrappers have is not really a method but a special `NULL` object. The name of the `NULL` object is `Am_No_`*Typename* where *typename* is replaced by the actual name for the type. Examples are `Am_No_Font`, `Am_No_Object`, and `Am_No_Style`. All of the `NULL` wrapper types are essentially equivalent to a `NULL` pointer. To test whether a wrapper is `NULL` or not one uses the method `Valid()`. If a wrapper is not valid, then it should not be used to perform operations.

```
// Here the code checks to see that my_obj is not a NULL pointer by using the Valid method.
Am_Object my_obj = other_obj.Get (MY_OBJ);
if (my_obj.Valid ()) {
  my_obj.Set (OTHER_SLOT, 6);
}
```

Sometimes a programmer uses a slot to hold multiple different wrapper types. To distinguish whether a wrapper pointer is of a given type the `Test` method is available. `Test` is a static method so it can be called directly.

```
// Here the Test method is used to test which kind of wrapper type the
// value holds.
Am_Value_List my_list;
Am_Object my_object;
Am_Value val = obj.Get (MY_VALUE);
if (Am_Value_List::Test (val))
  my_list = val;
else if (Am_Object::Test (val))
  my_object = val;
```

3.3.10 Using Am_Value

Most of the time a programmer knows precisely what sort of value is stored in a slot. For these situations, the most convenient form of `Get` is the one that returns the value directly. For times when the programmer wants to call `Get` on a slot but does not know what type the slot contains (or possibly if there is a slot there at all), one can either query the type of the slot using the `Get_Slot_Type` method for objects, or the programmer can `Get` the value into a `Am_Value` structure. The `Am_Value` is a union type for all the ORE types. It is always valid to use the `Am_Value` style of `Get`. If the slot does not exist, `Get` will set the `Am_Value` with type `Am_NONE`.

```
int i_value; float f_value;
Am_Value value;
my_object.Get (SOME_SLOT, value);  // Get the value regardless of type
if (value.type == Am_INT)          // the type field contains the type of value retrieved
  i_value = value;                 // Am_Value defines many casting operators
else if (value.type == Am_FLOAT)   // as assignment and constructors to aid
  f_value = value;                 // setting and retrieving the value from
                                   // the Am_Value
```

The `Am_Value` type has a number of methods, including printing (<<), `==`, `!=` and `Valid`. `Valid` returns true if the slot value existed (not `Am_NONE`) and if the value in the slot is not zero:

```
Am_Value value;
my_object.Get (SOME_SLOT, value);  // Get the value regardless of type
if (value.Valid()) {
      // then it is safe to use value
    ...
}
```

3.4 Inheritance: Creating Objects

The inheritance style of ORE objects is prototype - instance (as opposed to C++ which is class - instance). A prototype - instance object model means that objects are used directly as the prototypes of other objects. There is no distinction between instances and classes, in essence there are only instances. Specialization of sub objects into new types is performed by adding slots to the sub-object or changing the contents of existing slots defined in the prototype.

Here is an example of creating an ORE object and setting some of its slots:

```
Am_Object my_rectangle = Am_Rectangle.Create ("my rect")
   .Set (Am_LEFT, 10)
   .Set (Am_TOP, 20)
   .Set (Am_WIDTH, 100)
   .Set (Am_HEIGHT, 150)
   ;
```

A major style convention in ORE is to write an object all in one expression. This is so that the programmer need not repeat the name of objects over and over. This works because `Set` returns the original object. The main components of the creation action involves:

- Choose a prototype object. In the above case, the prototype is `Am_Rectangle` which is defined as one of the Amulet graphical objects.
- Call the `Create` method on the prototype. The optional parameter to `Create` is a string which is used if one prints out the name of the object and can be used for other sorts of debugging. The alternative to `Create` is the `Copy` method which is described below.
- Set the initial values of slots. This includes making new slots if desired.

Although this manual uses the one expression convention for brevity and to familiarize programmers with its use, it would be just as correct to write out each individual `Create` and `Set` call on its own line.

```
Am_Object my_rectangle;
my_rectangle = Am_Rectangle.Create ("my rect");
my_rectangle.Set (Am_LEFT, 10);
my_rectangle.Set (Am_TOP, 20);
my_rectangle.Set (Am_WIDTH, 100);
my_rectangle.Set (Am_HEIGHT, 150);
```

Objects inherit all the slots of their prototype that they do not specifically set locally. Thus, if the `Am_Rectangle` object defines a color slot called `Am_LINE_STYLE` with a value of `Am_Black`, then my_rectangle will also have a `Am_LINE_STYLE` slot with the same value. If a slot is inherited, it will change value if the prototype's value changes. Thus, if `Am_LINE_STYLE` of `Am_Rectangle` is set to `Am_Blue`, then `my_rectangle`'s `Am_LINE_STYLE` will also change. However, the `Am_LEFT` of `my_rectangle` will not change if the `Am_LEFT` of `Am_Rectangle` is set because `my_rectangle` sets a local value for that slot. See Section 3.10.4 for a discussion about how you can control the inheritance of slots.

The inheritence of Amulet objects' print names is dealt with slightly differently. Objects created with a name parameter to the Create() call have that name. Objects created without a string parameter get the name of their prototype, with a number appended to distinguish between the objects when their names are printed.

The root of the inheritance tree is `Am_Root_Object`. Programmers will typically create instances of the pre-defined objects exported by the various Amulet files (as shown in the examples in this manual), but `Am_Root_Object` is useful if you are defining application-specific objects, say for your own data structures.

The `Copy` method can also be used to make new objects. Instead of being an instance of the prototype object, a copied object will become a sibling of the object. Every slot in the prototype is copied in the same manner as in the original. If a slot is local in the original, it will be local in the copy likewise if the slot is inherited, the copy will also be inherited.

Other useful methods relevant to inheritance include:

- `obj.Is_Instance_Of (obj2);` returns true if `obj` is an instance of `obj2`.
- `Am_Object proto = obj.Get_Prototype ();` returns the prototype for the object.
- `bool inherited = obj.Is_Slot_Inherited (SLOT_KEY);` Returns `true` if the slot is not local, so the value is inherited from a prototype.
- `obj.Destroy ();` Destroys the object and all its parts.
- `const char* name = obj.Get_Name ();` returns the string name of the object defined when the object was created. Objects also define `operator<<` so they can

be used in C++ `cout` statements.

- `obj.Print_Name (ostream&);` prints the string name of the object to the given stream. Acts just as the << operator.
- Objects can be tested for == and != with other objects
- `obj.Remove_Slot (SLOT_KEY);` removes the slot from the object.

3.5 Parts

In ORE, it is possible to make objects become *part* of another object. The subordinate object is called a "part" of the containing object which is called the "owner." The part-owner relationship is used heavily in Amulet programs.

The Opal level of Amulet defines the `Am_Window` and `Am_Group` objects which are designed to hold graphical parts (rectangles, circles, text, etc.). Thus, if you want to make a composite of graphical objects, they should be added as part of a window or group. Non-graphical objects can be made parts of any kind of object. Thus, an Interactor object can be a part of a rectangle, group, or any other object. Similarly, any kinds of objects that an application defines can be added as parts of any objects. So for graphical objects, the "owner" of any graphical object will be a window or a group, but the owner of an interactor or application object can be any kind of object. We decided not to support adding graphical objects directly as parts of other graphical objects (so that you cannot add a rectangle directly as a part of a circle; instead, you would create a group object and add the rectangle and circle as parts of the group). Behaviors like translating the coordinates of the parts is handled by Opal level routines, so would not happen for non-graphical parts added as parts.

3.5.1 Parts Can Have Names

A very important distinction among parts is whether or not the part is named. A "named" part has a slot key. One can generate a key for a part the same way that they are generated for slots. When a part is named, it becomes possible to refer to it by that name in the method `Get_Part` (or by regular `Get`) and it also takes on other properties. If the part is unnamed, then the part cannot be accessed from the owner except through the part iterator (section 3.8) and perhaps by reading its name from a list like the `Am_GRAPHICAL_PARTS` slot in graphical objects (described in the Opal chapter).

In some ways, a named part of an object is like a slot that contains an object. The named part has a value and can have dependencies just like a slot. Parts are different from slots in that their type can only be an object and that any particular object can only be assigned as a part to only one owner. Parts cannot be generated by a constraint and the inheritance mechanism for parts is not as sophisticated as that for slots.

New names for parts are define the same way as new slot keys:

```
Am_Slot_Key MY_FOO_PART = Am_Register_Slot_Name ("My Foo Part");
Am_Object my_obj = Am_Group.Create("My_Obj")
        .Add_Part(MY_FOO_PART, Am_Rectangle.Create("foo"));  //named part
```

3.5.2 How Parts Behave With Regard To Create and Copy

The other very important distinction between named and unnamed parts is if an instance or copy is made of an object which has named parts, then instances are made for each of the parts also. Unnamed parts are *not* created in the instance, and regular slots which contain objects will *share* the same object. Thus:

```
Am_Object my_obj = Am_Group.Create("My_Obj")
        .Add_Part(MY_PART, Am_Rectangle.Create("foo"))  //named part
        .Add_Part(Am_Circle.Create())                   //un-named part
        .Set(Am_PARENT, other_object);  //slot containing an object

Am_Object my_obj2 = my_obj.Create();
// my_obj2 now has a rectangle part called MY_PART which is an instance of foo.
// It does not have a circle part, since that part was un-named in the prototype.
// The Am_PARENT slot of both my_obj and my_obj2 point to the same object, other_object.
```

When an object is copied, all parts are copied along with the object. Both named and unnamed parts are copied. If the parts had a name, the same name will be used but with a number appended to the end to distinguish it from the original.

3.5.3 Other Operations on Parts

Other methods on objects relevant to parts are listed below. In each of these, `MY_PART` is a part name.

- `Am_Object part = obj.Get_Part (MY_PART);` returns the part of `obj` named `MY_PART`.
- `Am_Object owner = obj.Get_Owner ();` returns the owner of `obj` (the object that `obj` is part of).
- `Am_Object part = obj.Get_Sibling (MY_PART);` is equivalent to `obj.Get_Owner().Get_Part(MY_PART);`
- `obj.Remove_From_Owner ();` makes `obj` no longer be a part of any object.
- `obj.Remove_Part (MY_PART);` removes from `obj` the part named `MY_PART`.
- `obj.Remove_Part (part);` removes object `part` from owner `obj`
- `obj.Is_Part_Of (Am_Object object);` returns true if `obj` is a part of `object`. Objects are considered to be parts of themselves: `obj.Is_Part_Of(obj)` returns `true`.
- `obj.Get_Key (Am_Object object);` if the object is a named part, then this will return the slot key name for the part.

For example:

```
Am_Slot_Key RECT = Am_Register_Slot_Name ("RECT");
  Am_Object my_window = Am_Window.Create ("a window")
    .Add_Part (RECT, Am_Rectangle.Create ("a rectangle"))
    .Add_Part (Am_Line.Create ("a line"));

  Am_Object rect = my_window.Get_Part (RECT);
  my_window.Remove_Part(RECT);
```

As mentioned above, when you destroy an object, all of its parts are destroyed also. Removing a part does not destroy the part.

3.6 Formulas

Formulas are the means for connecting together the values of slots. The use of formulas in ORE is the chief means for making Amulet declarative. With formulas, the programmer assigns the value of a slot to be dependent on the value of other slots. When the dependent slot changes, the value of the formula will be recomputed and the formula's slot will take on the computed value.

Often times this method of computing values from dependencies is called constraint maintenance. ORE's mechanisms for constraint maintenance are actually more general (and complicated) than the formula constraint mentioned here. The full ORE constraint mechanism will be described in a later revision of this manual. Suffice it to say for now that ORE allows more than one constraint system to be included in the system. The formula constraint is just one of many possible constraints that may be used in ORE. For example, Amulet currently also contains a "Web" constraint used to support multi-way interactions, but it is currently not finished or documented.

3.6.1 Formula Functions

An ORE formula consists of a C++ function that defines the dependencies of a slot and returns the value to set into the slot. The parameter list of a formula function is always the same two parameters: an `Am_Object` called `self`, which points to the object containing the slot, and an `Am_Constraint_Context&` called `cc`, which is an opaque handle (which means that its internals are not visible) to the state of the formula which used internally by the constraint system. The `cc` parameter is also used to distinguish the two forms of `Get`: one which just returns the value of the slot, discussed above, the other which returns the value and also sets up a dependency link. The return value of the function is of the same type that the slot will be when it takes on the returned value. There are also constraint versions of most of the `Get_XX` functions defined above, as shown by the following code example:

```
// Example of a formula function.  This formula returns a value for
// Am_LEFT which will center itself within its owner's dimensions.
int my_left_formula (Am_Constraint_Context& cc, Am_Object self) {
  Am_Object owner = self.Get_Owner (cc);
  int owner_width = owner.Get (cc, Am_WIDTH);
  int my_width = self.Get (cc, Am_WIDTH);
  return (owner_width - my_width) / 2;
}
```

The above example uses no macros so it is clear where variables are defined, and which methods take the special `cc` parameter. Because formula functions have such a generic format, the macro `Am_Define_Formula` is usually used to save some verbiage. Likewise, using the `cc` parameter in `Get` is common enough in a formula that macros like `GV` are available that automatically add the `cc` parameter. Using macros, the function above would look like the function below. (This particular function definition could easily be reduced to one line, as most Amulet programmers will quickly become adept at doing.)

```
// Example of a formula function.  This formula returns a value for
// Am_LEFT which will center itself within its owner's dimensions.
Am_Define_Formula (int, my_left_form) {
  Am_Object owner = self.GV_Owner ();
  int owner_width = owner.GV (Am_WIDTH);
  int my_width = self.GV (Am_WIDTH);
  return (owner_width - my_width) / 2;
}
```

It is noteworthy that there exists a `Set` version of `GV` called `SV`, that is used in other kinds of constraints, in particular for constraints with multiple outputs, like Web constraints. Though it is possible to use `SV` in a formula, it is generally easier to simply return the desired value. None of the formulas defined in the sample code use SV.

There are also macros for defining formulas that return many of the built-in wrapper types. For instance, the macro `Am_Define_Style_Formula (func_name)` returns the type `Am_Style`. Actually, all the wrapper formulas return the same type, `Am_Wrapper*`. It is somewhat confusing to look at a formula function that is supposed to return `Am_Style` or `Am_Object` and see that it returns `Am_Wrapper*`. So, these macros are available to make the code better reflect what is really intended. The various types of wrappers are described in the Opal chapter:

`Am_Define_Formula (`*`type, formula_name`*`)` -- General purpose: returns specified *`type`*

`Am_Define_No_Self_Formula (`*`type, function_name`*`)` -- General purpose: returns specified *`type`*. Used when the formula does not reference the special `self` variable, so compiler warnings are avoided.

`Am_Define_Value_Formula (`*`formula_name`*`)` -- Return type is `Am_Value`; used when the formula might return different types, described in Section 3.6.1.2

`Am_Define_Object_Formula (`*`formula_name`*`)` -- Return type is `Am_Object`

```
Am_Define_String_Formula (formula_name)  -- Return type is Am_String
Am_Define_Style_Formula (formula_name) -- Return type is Am_Style
Am_Define_Font_Formula (formula_name) -- Return type is Am_Font
Am_Define_Point_List_Formula (formula_name) -- Returns an Am_Point_List
Am_Define_Image_Formula (formula_name) -- Returns an Am_Image_Array
Am_Define_Value_List_Formula (formula_name) -- Returns an
            Am_Value_List
Am_Define_Cursor_Formula (formula_name) -- Returns an Am_Cursor
```

3.6.1.1 Declaring Formulas

In order to use a formula procedure outside of the file that defines it, C++ expects an external declaration. To facilitate making declarations of formula procedures, macros are available that spell out the parameters of the formula as an extern. The "Declare" macros take the same parameters as their "Define" counterparts. These are often used in `.h` files.

```
Am_Declare_Formula (int, my_int_form);
// same as:
//  extern int my_int_form (Am_Constraint_Context& cc, Am_Object self);

Am_Declare_Style_Formula (form2);
// same as:
//  extern Am_Wrapper* form2 (Am_Constraint_Context& cc, Am_Object self);
```

The set of declare forms is:

```
Am_Declare_Formula (type, formula_name)
Am_Declare_No_Self_Formula (type, formula_name)
Am_Declare_Value_Formula (formula_name)
Am_Declare_Object_Formula (formula_name)
Am_Declare_String_Formula (formula_name)
Am_Declare_Style_Formula (formula_name)
Am_Declare_Font_Formula (formula_name)
Am_Declare_Point_List_Formula (formula_name)
Am_Declare_Image_Formula (formula_name)
Am_Declare_Value_List_Formula (formula_name)
Am_Declare_Cursor_Formula (formula_name)
```

3.6.1.2 Formulas Returning Multiple Types

Some formulas do not return only one type of value. For these formulas, you don't want to define a single return type. To remedy this situation, the formula constraint system uses the `Am_Value` as a parameter instead of having a return value.

The value formula declaration is similar to the normal formula declaration except that its return value is void and it has one extra parameter of type Am_Value whose name is "value." The self and cc parameters are the same as in normal formulas and are used in exactly the same way. The standard macro for declaring a multi-type formula is Am_Define_Value_Formula, and there is an associated Am_Declare_Value_Formula.

```
Am_Define_Value_Formula (my_form) {
  if ((bool)self.GV (Am_SELECTED))
    value = 5;
  else
    value = Am_Blue;
}
```

To pass on a value from a different slot and without checking what type it is, something like the following can be used:

```
Am_Define_Value_Formula (my_copy_it_form) {
  other_obj.GVM(SOME_SLOT, value);  //value is what is returned from this formula
}
```

In the above example, if the slot Am_SELECTED is true, the formula will return an integer value of 5. If it is false, it returns the color blue.

To read a slot set with a value formula the programmer can use either the Am_Value form of Get or can call Get_Slot_Type.

3.6.2 Using GV

When the Get method is used with a constraint context (or equivalently, the GV macro is used), the constraint solver gets to decide how the actual Get is performed. The regular formula constraint solver sets up a dependency to the slot being fetched. Whenever the fetched slot changes value, the formula will automatically be updated by the system. Likewise, if the slot with the formula changes value due to the formula being updated, it too will cause dependent formulas to update. This continues recursively. The formula:

```
Am_Define_Formula (int, my_left) {
  int owner_width = self.GV_Owner ().GV (Am_WIDTH);
  int my_width = self.GV (Am_WIDTH);
  return (owner_width - my_width) / 2;
}
```

defines three slots as dependencies: the object's Am_OWNER slot (the GV_Owner macro expands into a GV on the Am_OWNER slot), the owner's Am_WIDTH slot, and the object's Am_WIDTH slot. A formula can change dependencies by simply calling GV on different slots. For instance, the above formula's dependencies will change if ever the object get moved to a new owner.

A programmer can still use a regular Get without a constraint context in a formula function, but the slot fetched will not become a dependency. Forgetting to use GV instead

of Get is a common mistake for Amulet programmers. If ever it seems that a formula is not updating when it is supposed to, check to make sure that GV is being used in the right places.

There exists a form of GV for all forms of Get but one, Get_Prototype. Since the prototype of an object is fixed and can never change, there is never a need to install a dependency to it. The full list of GV forms is:

- GV(slot) Get the specified slot, setting up a constraint.
- GVM(s1, value) Get the specified slot into the Am_Value parameter, setting up a constraint. Unfortunately, macros cannot be differentiated like procedures; hence the different name.
- GV_Owner() Get the owner of this object, setting up a constraint.
- GV_Part(slot) Get the specified part of this object, setting up a constraint.
- GV_Sibling(slot) Get the specified sibling of this object, setting up a constraint.

3.6.3 Putting Formulas into Slots

Once you have defined the formula function, you then need to create a formula object using that function, and then set the formula object into a slot. Typically, these are done in a single step, but it is also possible to re-use a formula object in multiple slots by declaring a variable that the result of the Am_Formula::Create is stored into.

The normal way to install a formula is as part of a create call. Remember that the second parameter to a Am_Define_Formula is the name of the function. This is then passed to the Am_Formula::Create call:

```
Am_Object my_rectangle = Am_Rectangle.Create ("my rect")
    .Set (Am_LEFT, Am_Formula::Create(my_left_form))
    ;
```

Formulas are evaluated eagerly, which means that the formula expression must always be evaluatable, even right after the formula is created. Therefore, it is wise to make sure that any slots that the formula references are Set before the formula is set into a slot. Furthermore, the formula expression should be written so that it checks to make sure that any pointer variables are valid before using them. For example, the following code makes sure that the owner is valid:

```
Am_Define_Formula (int, my_form) {
  Am_Object owner = self.GV_Owner();
    if (owner.Valid ())
        return owner.Get(Am_LEFT);
    else return 0;
}
```

3.6.4 Slot Setting and Inheritance of Formulas

When a slot is set, by default a formula in that slot is removed. Just like setting a slot with a new value removes the old value of the slot, setting a slot with a value or a new formula removes the old value that was there, even if the old value was a formula.

Like values, formulas are inherited from prototypes to their instances. However, the formula in the instance might compute a value from the formula in the prototype if the formula contains indirect links. For example, the formula that computes the width of a text object depends on the text string and font, and even though the same formula is used in every text object, they will all compute different values.

Sometimes, constraints from a prototype should be retained in instances *even if the local value is set.* This requires declaring that the slot is not Single_Constraint_Mode, which is an advanced feature covered in section 1.10.5.

3.7 Lists

In virtually any non-trivial program written in Amulet, one is going to use lists. For example, many widgets require a list of labels to be displayed. To make lists easy to manipulate and store into slots, ORE provides the standard `Am_Value_List` type, implemented as a form of wrapper. The operations on `Am_Value_Lists` are provided in the file `value_list.h`. Like slots, Lists can hold any type of value. In fact, a single list can contain many different types at the same time.

Because the `Am_Value_List` is a form of wrapper, it supports all the standard wrapper operations, including:

- assignment (=), which makes copy of the value list: `Am_Value_List l2 = l1;`
- test for equality (==), which tests whether the two lists contain values which are the same (thus, it goes through each list testing each element for ==).
- test for whether a wrapper is an Am_Value_List: `Am_Value_List::Test(wrap);`

3.7.1 Current pointer in Lists

In addition to the data, Am_Value_Lists also contain a pointer to the "current" item. This pointer is manipulated using the following functions:

- `void Start ();` Make first element be current. This is always legal, even if the list is empty.
- `void End ();` Make last element be current. This is always legal, even if the list is empty.
- `void Prev ();` Make previous element be current. This will wrap around to the end of the list when current is at the head of the list.

- `void Next ();` Make next element be current. This will wrap around to the beginning of the list when current is at the last element in the list.
- `bool First ();` Returns true when current element passes the first element.
- `bool Last ();` Returns true when current element passes the last element.

The standard way to iterate through all the items in a list in a forward order is:

```
for (my_list.Start (); !my_list.Last (); my_list.Next ()) {
}
```

Similarly, to go in reverse order, you would use:

```
for (my_list.End (); !my_list.First (); my_list.Prev ()) {
}
```

Note that the pointer is *not* initialized automatically in a list, so you should always call `Start` or `End` on the list before using the pointer.

3.7.2 Adding items to lists

There are two mechanisms for adding items to lists: either always at the beginning or end, or at the position of the pointer.

To add items at the beginning or end of the list, the `Add` method can be used. Since this does not use the pointer, you do not need to call start. The first parameter to `Add` is the value to be added, which can be any primitive type, an object, a wrapper, or `Am_Value`. The second parameter to `Add` is either `Am_TAIL` or `Am_HEAD` and defaults to `Am_TAIL` and controls where the value goes. `Add` returns the original `Am_Value_List` so multiple `Adds` can be chained together:

```
Am_Value_List l;
l.Add(3)
 .Add(4.0)
 .Add(Am_Rectangle.Create())
 .Add(Am_Blue)
 ;
```

To add items at the current position, the programmer must first initialize the pointer, using `Start` or `End` and then the `Insert` routine can be called. The second parameter to `Insert` specifies whether the new item should go `Am_BEFORE` or `Am_AFTER` the current item. There is no default for this parameter.

3.7.3 Other operations on Lists

The `Get` method will retrieve the item at the current position. Like object's `Get`, it will convert into the appropriate type, and there is a `Am_Value` version if you don't know what types the list contains. For example:

```
Am_Value v;
for (my_list.End (); !my_list.First (); my_list.Prev ()) {
  my_list.Get(v);
  cout << "List item type is " << v.type << endl << flush;
}
```

If you don't know the type of the current value in the list, you can use `Get_Type()` instead of using `Get` with an `Am_Value`.

`Set` can be used to change the current item in the list. This is different from `Insert` because `Set` deletes the old current item and replaces it with the new value.

The `Delete()` method will destroy the item at the current position. It is an error if no element is current. The current position is shifted to the element previous to the deleted one. `Make_Empty()` deletes all the items of the list.

`Am_Value_Lists` support a membership test, using the `Member` method. This starts from the current position, so **be sure to set the pointer before calling Member**. Thus, to find the first instance of 3 in a list, the following would be used:

```
l.Start();
if (l.Member(3)) cout << "Found a 3";
```

`Member` leaves the pointer at the found item, so calling `Set` or `Delete` will affect the found item. Calling `Member` again will find the next occurrence of the item.

`l.Length()` will return the current length of the list, and `l.Empty()` returns true if the list is empty.

3.8 Iterators

For efficiency, ORE does not allocate an `Am_Value_List` for some types of list-like information, and instead supplies an "iterator." The iterator object contains a current pointer and allows the programmer to examine all the elements. There are three kinds of iterators available in the basic use of objects: one for slots, one for instances, and another for parts. Each of the iterators has the same basic form, and the interface is essentially the same as for `Am_Value_Lists`. The main difference between them is the type of objects upon which is iterated.

3.8.1 Reading Iterator Contents

The iterator methods treat the list of items more like a linked list rather than an array. The main operations are `Start` and `Next`. `Start` places the iterator at the first element. `Next` moves the iterator to the next element. To find the value of the current element use `Get`. To initialize the list, one assigns the iterator with the object that contains whatever is to be iterated upon. For example:

```
cout << "The instances of " << my_object << " are:" << endl;
Am_Instance_Iterator iter = my_object;
for (iter.Start (); !iter.Last (); iter.Next ())  {
  Am_Object instance = iter.Get ();
  cout << instance << endl;
}
```

The first line of the example is used to initialize the iterator. The example prints out the instances of `my_object` so `my_object` is the object to assign to the iterator. To detect when the list is complete, the method `Last` is used which returns true when the last element of the list is passed and the iterator no longer has an element that is current.

3.8.2 Types of Iterators

To iterate over the parts of an object, use an `Am_Part_Iterator` which is initialized with the object for which you want the parts of. The `Get` method returns an `Am_Object` which is the part. For `Am_Groups` and `Am_Windows`, however, it is better to use the `Am_Value_List` stored in the `Am_GRAPHICAL_PARTS` slot instead of the part iterator.

To iterate over the instances of an object, use an `Am_Instance_Iterator` which is initialized with the object for which you want the instances of. The `Get` method returns an `Am_Object` which is the part.

To get all the slots (both inherited and local) of an object, use the `Am_Slot_Iterator`, which is initialized with the object for which you want the slots of. The `Get` method returns an `Am_Slot_Key` which is the name of the slot. You can use the object method `Is_Slot_Inherited` to see if the slot is inherited or not.

3.8.3 The Order of Iterator Items

When an iterator is first initialized, there is no particular order imposed on the list. The order of the elements will be such that a single element will not repeat if the list is read from beginning to end, but the order may change if the iterator is restarted from the beginning. (Opal keeps track of the Z (stacking or covering) order of the parts through a different mechanism, using the `Am_GRAPHICAL_PARTS` slot which contains an `Am_Value_List` of the graphical parts, sorted correctly).

When items are added to an iterator's list during the middle of searching the list, the items added are not guaranteed to be seen by the iterator. In other words, the iterator may skip them. The value returned by the `Length` method will be correct, but the only way to make certain all values have been seen is to restart the iterator.

Likewise, the order in which the elements are stored in each iterator is not guaranteed to be maintained when an item is *deleted* from the list. The iterators themselves cannot be used to destroy an item, but other methods like `obj.Destroy`, `obj.Remove_Part`, and `obj.Remove_Slot` will affect the contents of iterators that hold those values.

When an iterator has a slot or object as its current position, and that item gets removed, the affect on the iterator is not determined. An iterator can always be restarted by calling the `Start` method in which case it will operate as expected. Though an iterator will not likely cause a crash if its current item is deleted, continued use of it will cause some very odd results.

For iterators that iterate over objects (specifically `Am_Part_Iterator` and `Am_Instance_Iterator`), it is possible to continue using the iterator even when items are deleted. If the programmer makes certain that the iterator does not have the deleted object as the current position when the object is removed, then the iterator will remain valid. For example:

```
// Example: Remove all parts of my_object that are instances of Am_Line.
Am_Part_Iterator iter = my_object;
iter.Start ();
while (!iter.Last ()) {
  Am_Object test_obj = iter.Get ();
  iter.Next ();
  if (test_obj.Is_Instance_Of (Am_Line))
    test_obj.Remove_From_Owner ();
}
```

In the above example, the call to `Next` occurs before the call to `Remove_From_Owner`. If these methods calls were reversed, then iterator would go into an odd state and one would get undetermined results.

The `Am_Slot_Iterator` type does not have the same deletion properties as the object iterators. If a slot gets removed from an object being listed by a slot iterator (or the prototype of the object assuming the slot is defined there), then the affect on the iterator is undetermined. The slot iterator must be restarted whenever a slot gets added or removed from the list in order to guarantee that all slots are seen.

3.9 Errors

Whenever Amulet notices an error, it calls the `Am_Error` routine which prints out the error and then aborts the program. If you have a debugger running, it should cause the program to enter the debugger. Note that under Visual C++, you have to have the GWStreams application running to see the error output.

3.10 Advanced Features of the Object System

3.10.1 Destructive Modification of Wrapper Values

Some wrappers, like `Am_Style`'s are immutable, which means that once created, the programmer cannot change their values. Other wrapper objects, like `Am_Value_Lists` are mutable. The default Amulet interface copies the wrapper every time it is used and automatically destroys the copies when the programmer is finished with them (explained in section 3.10.2.2). This design guarantees both that the programmer will not

accidentally change a value still stored in a slot, and that there will not be a space leak. However, for the programmers that know what they are doing, it is unnecessarily wasteful since copies are not always required. This section discusses how you can modify a wrapper value without making a copy. See also the discussion of the implementation of wrappers, Section 3.10.2.

When you retrieve a wrapper value out of a slot, it points to the same value that is in the slot, and only when a destructive modification is made to the value is a copy made. Thus, most mutable wrappers provide a "make_unique" optional parameter in their data changing operations. The default for this parameter is always `true`, but if you call the procedure with it `false`, then the value you modify will be the same as the value stored in the slot. However, the object system will not know that you have changed the value, so you have to called `Note_Changed` on the object to tell it after you are done your modifications. This is needed so Amulet will know to redraw the object if necessary and notify any slots with a constraint dependent on this slot. For example:

```
Am_Value_List list = obj.Get(MY_LIST_SLOT);
list.Start();
list.Delete(false);  //destructively delete the first item
obj.Note_Changed(MY_LIST_SLOT);  //tell Amulet that I modified the slot
```

An important point to remember when doing destructive modification is to be careful that the wrapper is not shared among multiple objects. To save space, Amulet shares a wrapper between multiple objects, especially between a prototype and its instances. Thus, if you make an instance of `obj` in the example above called `obj_instance`, then both `obj_instance` and `obj` will have the same list in the `MY_LIST_SLOT`. In that case, the above code would modify the list for both `obj` and `obj_instance`, *but Amulet would not know about the change to* `obj_instance`. Thus, you should only do destructive modifications if you are sure the wrapper value is not shared (possibly due to your setting of the slot inheritance--see Section 3.10.4), or because you have explicitly set the wrapper value into the slot of an instance.

If the programmer cannot be sure that the slot contains a local copy of the wrapper, or knows for fact that it does not, then there is the option of using the `Make_Unique` method. `Make_Unique` is a method on `Am_Object` that takes a slot key as a parameter. The effect of `Make_Unique` is to make certain that whatever value is in that slot, it will be made local. After `Make_Unique` is called on the slot, the wrapper can be fetched without worry and destructively modified as desired.

```
// Example which uses Make_Unique to guarantee the uniqueness of MY_SLOT's value.
obj.Make_Unique (MY_SLOT);
Am_Value_List list = obj.Get (MY_SLOT);
list.Add (5, Am_TAIL, false);
obj.Note_Changed (MY_SLOT);
```

3.10.2 Writing a Wrapper Using Amulet's Wrapper Macros

You should consider creating a new type of wrapper whenever you need to store a C++ object into a slot of an Amulet object. A wrapper manages two things for its value. First

it has a simple mechanism that supports dynamic type checking, so you can make sure that you have the correct type of data. Second, wrappers add a reference counting scheme to the value to prevent the value's memory from being deleted while the value is still in use.

Wrappers are created in two layers. The outermost layer is the C++ object layer used by programmers to refer to the object. The type `Am_Style` is the object layer for the `Am_Style` wrapper. Inside the object layer is the data layer. For `Am_Style`, the type is called `Am_Style_Data`. Normally, programmers do not have access to the data layer. The object layer of the wrapper is used to manipulate the data layer which is where the actual data for the object is stored.

3.10.2.1 Creating the Wrapper Data Layer

Both the typing and reference counting is embodied by the definition of the class `Am_Wrapper` from which the data layer of all wrappers must be derived. The class `Am_Wrapper` is pure virtual with five methods that all wrappers must define. Most of the time, the programmer can use the pre-defined macros `Am_WRAPPER_DATA_DECL` and `Am_WRAPPER_DATA_IMPL` to define these five methods.

```
void Note_Reference ()
Am_Wrapper* Make_Unique ()
void Release ()
operator== (Am_Wrapper& test_value)
Am_ID_Tag ID ()
```

Of the five methods, three are used for maintaining the reference count and making sure that only a unique wrapper value is ever modified. These are `Note_Reference`, `Make_Unique`, and `Release`.

`Note_Reference` tells the wrapper object that the value is being referenced by another variable. The reference could be a slot or a local variable or anything else. The implementation of `Note_Reference` is normally to simply add one to the reference count.

`Release` is the opposite of `Note_Reference`. It says that the variable that used to hold the value does not any longer. Typical implementation is to reduce the reference count by one. If the reference count reaches zero, then the memory should be deallocated.

`Make_Unique` is the trickiest of these methods to understand. The basic idea is that a programmer should not be allowed to modify any wrapper value that is not unique. For example, if the programmer retrieves a `Am_Value_List` from a slot and adds an item to the list, this destructive modification should normally *not* affect the list that is still in the slot. The way to maintain this paradigm is for the method used to modify the wrapper's data to first call `Make_Unique`. If the reference count is one, the wrapper value is already unique and `Make_Unique` simply returns `this`. If the reference count is greater than one, then `Make_Unique` generates a new allocation for the value that is unique and returns it. Either way a unique wrapper value will be modified. Some wrapper types have boolean

parameters on their destructive operations that turn off the behavior of `Make_Unique` to allow the programmer to do destructive modifications (see Section 3.10.2.1).

The `operator==` method allows the object system to compare two wrapper values against one another. The system will automatically compare the pointers so all the == method must do is compare the actual data. Simply returning false is sufficient for most wrappers.

The final operator is used to handle a primitive dynamic typing system. Each wrapper type is assigned a number called an `Am_ID_Tag` which is just an unsigned integer. Integers are dispensed using the function `Am_Get_Unique_ID_Tag`. Normal procedure is to define a static member to the wrapper data class called `id` which gets set by calling `Am_Get_Unique_ID_Tag`. Then the ID method simply returns `id`.

All of the methods can be defined instantly by using the `Am_WRAPPER_DATA_DECL` and `Am_WRAPPER_DATA_IMPL` macros. The macros require that the user define at least two methods in the wrapper data class. The first required method is a constructor to be used by `Make_Unique`. The method is used to create a copy of the original value which can be modified without affecting the original. This can be done easily by making a constructor that takes a pointer to its own class as its parameter. Make sure that in all the data class constructors to initialize the `ref` member (defined by `Am_WRAPPER_DECL`) to 1. The second required method is a `operator==` that tests equality with a reference to the wrapper data class. The == method does not need to check to make sure the parameter is of the same type as itself because that is handled by the default implementation of the `operator==` that takes a `Am_Wrapper&` and which calls the specific == routine if the types are the same.

For example:

```
class Foo_Data : public Am_Wrapper {
    Am_WRAPPER_DATA_DECL (Foo_Data)
   public:
    Foo_Data (Foo_Data* prev)
    {
      ... // initialize member values
      refs = 1; // Do not forget this line!
    }
    operator== (Foo_Data& test_value)
    {
      ... // compare test_value to this
    }
   protected:
    ... // define own members
  };

// typically this part goes in a .cc file
  Am_WRAPPER_DATA_IMPL (Foo_Data, (this))
```

All the standard wrapper macros take the name of the type as their first parameter. If one wants the name of the wrapper type to be `Foo`, the data layer type must be name `Foo_Data` assuming the programmer is using the standard macros in the outer layer. The

Am_WRAPPER_DATA_IMPL macro takes a second parameter which is the parameter set to use when the Make_Unique method calls the data object's constructor. In the above case, "(this)" is used because the parameter to the Foo_Data constructor is equivalent to the this pointer in the Make_Unique method. That is, the Make_Unique method will sometimes have to create a new copy of the wrapper object. The new copy will be created using one of the object's constructors. In the above case, the programmer wants the constructor Foo_Data(Foo_Data* prev) to be used. This constructor requires Make_Unique to pass in its this pointer as the parameter. Therefore the parameter set declared in the macro Am_WRAPPER_DATA_IMPL is "(this)." If the programmer wanted a different constructor to be used, the parameter set put into the macro would be different.

3.10.2.2 Using The Wrapper Data Layer

The wrapper data layer is normally manipulated only by the methods in the wrapper outer layer. One can take the Am_Foo_Data* and manipulate it as a normal C++ object with the following caveat. One must be sure that the reference count is always correct. When one uses the data pointer directly, there are no functions called Add_Reference or Release for the programmer so it must be done in the code.

Consider the following example: The programmer wants to return a Am_Foo type but currently has a Am_Foo_Data* stored in his data.

```
// Here we move a Foo_Data pointer from one variable to another.  The
// object that lives in the first variable must be released and the
// reference object moving to the new variable must be incremented
Foo_Data* foo_data1;
Foo_Data* foo_data2;

//... assume that foo_data1 and foo_data2 are somehow set up with
// real values ...

if (foo_data1)
  foo_data1->Release ();
foo_data1 = foo_data2;
foo_data1->Note_Reference ();
```

To keep changes in a wrapper type local one must call the Make_Unique method on the data before one makes a change to it. If the wrapper designer wants to permit the wrapper user to make destructive modifications, a boolean parameter should be added to let the user decide which to do.

```
void Foo::Change_Data (bool destructive = false)
{
  if (!destructive)
    data = data->Make_Unique ();
  data->Modify_Somehow ();
}
```

Sometimes a programmer will want to use the wrapper data pointer outside the outer wrapper layer of the object. To convert from the wrapper layer to the data layer, one uses Narrow.

```
Foo my_foo;
Foo_Data* data = Foo_Data::Narrow (my_foo);
```

The way to test if a given wrapper is the same type as a known wrapper class is to compare IDs.

```
Am_Wrapper* some_wrapper_ptr = something;
Am_ID_Tag id = some_wrapper_ptr->ID ();
if (id == Foo_Data::Foo_Data_ID ())
  cout << "This wrapper is foo!" << endl;
```

The static method *TypeName*`_ID` is provided in the standard macros.

Other functions provided for wrapper data classes by the standard macro are `Is_Unique`, and `Is_Zero` with are boolean functions that query the state of the reference count.

3.10.2.3 Creating The Wrapper Outer Layer

The standard macros for building the wrapper outer layer assume that the class for the wrapper data is called *TypeName*`_Data` where *TypeName* is the name for the wrapper outer layer. Like the data layer macros, there are two outer layer macros, one for the class declaration part and one for the implementation.

```
// Building the outer layer of Foo. This bit normally goes in a .h file.
class Foo {   // Note there is no subclassing.
  Am_WRAPPER_DECL (Foo)
 public:
  Foo (params);
  Use ();
  Modify ();
};

 // This normally goes in the .cc file.
  Am_WRAPPER_IMPL (Foo)
```

The wrapper outer layer is given a single member named `data` which is a pointer to the data layer. In all the wrapper methods, one performs all operation on the `data` member.

```
// A Foo constructor - initializes the data member.
Foo::Foo (params)
{
  data = Foo_Data::Create (params);
}
```

For methods that modify the contents of the wrapper data, one must be sure that the data is unique. One uses the `Make_Unique` method to manage uniqueness.

```
Foo::Modify ()
{
  if (!data)
    Am_Error ("not initialized!");
  data = data->Make_Unique ();
  data->Modify ();
}
```

Methods that do not modify data do not need to force uniqueness so they can use the wrapper data directly.

```
Foo::Use ()
{
  if (!data)
    Am_Error ("not initialized!");
  data->Use ();
}
```

Somewhere in the code the wrapper will actually do something: calculate an expression, draw a box, whatever. Whether the programmer puts the implementation in the data layer or the outer layer is not important. Most wrapper implementations will compromise somewhere in between.

Putting a wrapper around an existing C++ class is not difficult. One can put the original class as a single piece of data for the wrapper data layer. If the programmer does not want to reimplement all the methods that come with the existing class, then provide a single function that returns the class and one can perform the methods on that. Be certain that `Make_Unique` is called before the existing object is returned. If the object can be destructively modified, then the wrapper must be made safe before the modifications occur. However, if the programmer wants the wrapper object to behave as if it were the original class then some reimplementation may be required.

3.10.3 Using Am_Object_Advanced

There are several extra methods that can be used on any Amulet object that are not available in the regular `Am_Object` class. A programmer can manipulate these methods by typecasting a regular `Am_Object` into a `Am_Object_Advanced` class. For instance, in order to retrieve the `Am_Slot_Advanced` form for a slot one can use `Get_Slot`:

```
Am_Object_Advanced obj_adv = (Am_Object_Advanced&)my_object;
Am_Slot_Advanced* slot = obj_adv.Get_Slot (Am_LEFT);
```

Needless to say, a programmer can break a good many things using the advanced object and slot classes. A general principle should be to use the advanced features for a short time right after an object is first created. After the object is manipulated to add or change whatever is needed, the object should never need to be cast to advanced again.

A number of methods in the advanced object class are used to fetch slots. The method `Get_Slot` retrieves a slot and returns it as the advanced class `Am_Slot_Advanced`. `Get_Slot` will always return a slot local to the object. If the slot was previously not local because it is still inherited or for other reasons, `Get_Slot` will make a placeholder slot locally in the object and return that. If the slot does not exist at all, `Get_Slot` will return `Am_NULL_SLOT`. There are two other methods used for fetching slots: `Get_Owner_Slot`, and `Get_Part_Slot`. These methods are similar to `Get_Slot` except they are to be used only for fetching part slots. It is entirely possible to use `Get_Slot` instead of these specialized methods, but the specialized methods are more efficient.

3.10.4 Controlling Slot Inheritance

An innovation in Amulet is that the programmer can control the inheritance of each slot. This is useful if you want to make sure that certain slots are not shared by a prototype and its children. For example, the Amulet `Am_Window` object has a slot that points to the actual X/11 or MS Windows window object associated with that window. This slot should not be shared by multiple objects. To control the inheritance, you need to include `object_advanced.h`. The choices are defined by the enum `Am_Inherit_Rule` and are:

- `Am_INHERIT`: The default. Slots are inherited normally from prototypes to instances, and subsequent changes to the prototype will affect all instances that do not override the value.
- `Am_LOCAL`: The slot is never inherited by instances. Thus, the slot will not exist in an instance unless explicitly set there.
- `Am_COPY`: When the instance is created, a copy is made of the prototype's value and this copy is installed in the instance. Any further changes to the prototype will not affect the instance.
- `Am_STATIC`: All instances will share the same value. Changing the value in one object will therefore affect all objects. We have found this to be rarely useful.

Setting the inheritance is an "advanced" feature, which means that you have to use an `Am_Slot_Advanced`. For example, to set the inheritance rule of the `Am_INTER_LIST` slot of `new_win` to be local, you need to do:

```
#include <amulet/object_advanced.h>  //for slot_advanced

  Am_Object_Advanced obj_adv = (Am_Object_Advanced&)new_win;
  Am_Slot_Advanced *slot = obj_adv.Get_Slot(Am_DRAWONABLE);
  slot->Set_Inherit_Rule(Am_LOCAL);
```

The default rule with which an object will create all new slots added to an object can be changed using the Set_Default_Inherit_Rule method. Likewise, the current inherit rule can be examined using Get_Default_Inherit_Rule.

```
((Am_Object_Advanced&)my_object).Set_Default_Inherit_Rule (Am_COPY);
Am_Inherit_Rule rule = my_adv_object.Get_Default_Inherit_Rule ();
```

3.10.5 Controlling Formula inheritance

For slots that are inherited normally, sometimes you still might want to control Formula inheritance separately. Remember from section 1.6.4 that instances inherit formulas from their prototypes, but that setting the instance's slot normally removes the inherited formulas. There are times, however, when constraints from a prototype should be retained in instances *even if the instance's value is set.* For example, the `Am_VALUE` slot of widgets contain formulas that compute the value based on the user's actions. However, programmers are also allowed to set the `Am_VALUE` slot if they want the widget

to reflect an application-computed value. In this case, the default formula in the Am_VALUE slot should *not* be removed if the programmer sets the slot.

To achieve this, the programmer must set the slot's Single_Constraint_Mode to false (the default is true). This is an advanced feature of Am_Slot_Advanced, so it would be done like:

```
Am_Object_Advanced obj_adv = (Am_Object_Advanced&)obj;
obj_adv.Get_Slot(Am_WIDTH)->Set_Single_Constraint_Mode (false);
```

Now, if obj contains a constraint, any instances of obj will always retain that constraint, even if another constraint or value is set into the instance. Furthermore, calls like Remove_Constraint on the instance's slot will still *not* remove the inherited constraint (though it will remove any additional constraints set directly into the instance).

3.10.6 Writing and Incorporating Demon Procedures

Amulet demons are special methods that are attached directly to an object or slot. The demons are for controlling behavior that objects must do frequently or that have a specific purpose in the object's function. The demons are written as procedures that are stored in an object's "demon set." The demon set is shared among objects that are inherited or copied from another.

The demon procedures that operate on an object have very specific purpose. There are five demons in total that can be overridden on the object level. Three of the demons deal with object creation and destruction, the other two are used to handle part management.

Demons that are attached to slots behave in a way similar to formulas. The slot demons are more generic than object level demons. Slot demons can detect when the slot value changes or is invalidated. Several slot demons can be assigned to a single slot making it possible for the slot to have multiple effects with a single event.

When a demon event occurs, the demon affected is put into the "demon queue" to be invoked later. All the demons put into demon queue are invoked, in order, whenever any slot is fetched by using Get. By invoking the demons on Get, Amulet can simulate the effects of eager evaluation because any demon that affects the value of different slots will invoked whenever a slot is queried.

3.10.6.1 Object Level Demons

The three demons that handle object creation and destruction are the create, copy, and destroy demons. Each demon is enqueued on its respective event. The create and copy demons get enqueued when the object is first created depending on whether the method Create or Copy was used to make the object. The destroy demon is not actually ever enqueued. Since the Destroy operation will cause the object to no longer exist, all

demons that are already enqueued will be invoked and then the destroy demon will be called directly. This allows the programmer to still read the object while the destroy procedure is running.

All of the creation/destruction demon procedures have the same parameter signiture. They all take just the single object that has been affected. The type of the procedure is `Am_Object_Demon`.

```
// Here is an example create demon that initializes the slot MY_SLOT to be zero.
void my_create_demon (Am_Object self)
{
  self.Set (MY_SLOT, 0);
}
```

Two object level demons are used to handle part-changing events. These are the add-part and the change-owner demons. The add-part and change-owner demons are always paired: the add-part demon for the part and the change-owner demon for the part's owner. Both demon procedures have the same parameter signature, three objects, which has the type `Am_Part_Demon`, but the semantics of each demon is different. The first parameter for both prodecures is the `self` object -- the object being affected by the demon. The next two objects represent the change that has occured. In the add-part demon, the second object parameter is object that is being removed or replaced. The third parameter is the object that is being added or is replacing the object in the second parameter. For the change-owner demon, the semantics are reversed -- instead of parts being added the second and third parameters represent the change that a part sees in its owner. The second parameter is the owner the part used to have, the third parameter is the new owner that has replaced the old owner.

```
// This owner demon checks to make sure that it's owner is a window.  Any
// other owner will cause an error.
void my_owner_demon (Am_Object self, Am_Object prev_owner,
                     Am_Object new_owner)
{
  if (!new_owner.Is_Instance_Of (Am_Window))
    Am_Error ("You can only add me to a window!");
}
```

The events that generate add-part and change-owner demon updates are the methods like `Add_Part`, `Remove_Part`, `Destroy`, and other methods that can change the object hiearchy. Note that a given add-part demon can always be associated with a corresponding change-owner demon. The correspondance is not neccessarily one to one because one can conceive of situations where one part is replacing another and thus two add-part calls can be associated with a single change-owner and vice versa.

Important note: The Opal and Interactor layers of Amulet define important demons for all of these object-level operations, so before setting a custom demon, the code should fetch the demon procedure currently stored in the demon set and call these in addition to the new demon. In a future release, we will make this more convenient to do.

3.10.6.2 Slot Level Demons

Slot demons are not given permanent names as the object level demons are. The slot demons are assigned a bit in a bit field to serve as their moniker. For space considerations, the actual slot demon procedure pointer is stored in the object. The bit is turned on in the slot to indicate that it should look up the procedure from the demon whenever a triggering event occurs.

There are two parameters that control a slot demon. The first parameter distinguishes what event will trigger the demon. Slot demons can be triggered by one of two slot messages, either the invalidate message or the value changed message. Most demons trigger on the value changed message because the demon's purpose is to note the change to other parts of the system. This can be used to implement an active value scheme with a slot. Triggering using invalidate message makes the demon act more like a formula. The demon can be used to revalidate the slot if desired. In fact, the "eager demon" uses precisely this mechanism to make Amulet formulas eager.

The other slot demon parameter is used to determine how often the demons will be triggered. Quite often several slots affect the same demon in the same object. For instance, in a graphical object, the `Am_TOP` and `Am_LEFT` slots both affect the position of the object. A demon that handles object motion only needs to be triggered once if either of these slots changes. For this case, we use the per-object style demon. Whenever multiple slots change in a single object, the per-object demon will only be enqueued once. Only after the demon has been invoked will it reset and be allowed to trigger again. The other style of demon activation is per-slot. In this case, the demons act independently on each slot they are assigned. The demon triggers once for each slot and after it is invoked, it will be reset. The per-slot demon does not check to see if other demons have already been enqueued for the same object.

The slot demon procedure takes as its only parameter the slot that triggered the demon. If the demon could have been triggered by more than one slot as can be the case when the demon is set to be per object, the slot provided is the very first one that triggered it.

```
// Here is an example slot demon. This demon does not do anything
// interesting, but it shows how the parameter can be used.
void my_slot_demon (Am_Slot_Advanced* slot)
{
  Am_Object_Advanced self = slot->Get_Owner ();
  self.Set (MY_SLOT, 0);
}
```

3.10.6.3 Modifying the Demon Set and Activating Slot Demons

To activate any demon, the demon procedure must be made known to the object. Objects keep the list of available procedure in a structure called the "demon set" which is defined by the class `Am_Demon_Set`. Objects inherit their demon set from their prototypes. The demon set, therefore, is shared between objects in order to conserve memory. To modify the demon set of an object, one must first make sure that the set is a local copy. The

demon set's `Create` method is used for making new sets. The programmer can pass a demon set that is already filled with demon procedures as a parameter to the `Create` method in order to make a copy.

```
// Here we will modify the demon set of my_adv_obj by first making
// a copy of the old set and modifying it. The new demon set is then
// placed back into the object.
Am_Demon_Set* prev_demons = my_adv_obj.Get_Demon_Set ();
Am_Demon_Set* my_demons = Am_Demon_Set::Create (prev_demons);
my_demons->Set_Object_Demon (Am_DESTROY_OBJ, my_destroy_demon);
my_adv_obj.Set_Demon_Set (my_demons);
```

Once the demon procedures are installed for object level demons, the demons will trigger on the next occurance of their corresponding event. Note that the create and copy demon's events have already occured for the prototype object in which the demon procedures are installed. However, instances of the prototype as well as new copies will cause the new procedures to run. To make the demon procedure run for the current prototype object, one must call the demon procedure directly.

The demon set holds all demon procedures for the object, including the demons used in slots. The slot demon are installed by assigning each demon a bit name that will be stored in the slot that activates the demon. The bit name is represented by its integer value so bit 0 is number 1, bit 5 is number 32 (hex 0x0020). Section 3.10.5.5 discusses how to allocate demon bits.

```
// Here we install a slot demon that uses bit 5. The slot demon's semantics are to activate when
// the slot changes value and only once object. Make sure that the demon set is local to the per
// object (see above section).
my_demons->Set_Slot_Demon (0x0020, my_slot_demon,
                           Am_DEMON_ON_CHANGE | Am_DEMON_PER_OBJECT);
```

After the demon procedure is stored, one can set the bits on the slot to tell it which demons belong to it.

```
// Here we set a new bit to a slot. To make sure we do not turn off
// previously set bits, we first get the old bits and bitwise-or the new.
Am_Slot_Advanced* slot = my_adv_obj.Get_Slot (MY_SLOT);
unsigned short prev_bits = slot->Get_Demon_Bits ();
slot->Set_Demon_Bits (0x0020 | prev_bits);
```

If one wants every new slot in an object to have certain demon bits set, one can change the default demon bits in the object.

```
// Make the new slot demon default.
unsigned short default_bits = my_adv_obj.Get_Default_Demon_Bits ();
default_bits |= 0x0020;
my_adv_obj.Set_Default_Demon_Bits (default_bits);
```

Another factor in slot demon maintainance is the "demon mask." The demon mask is used to control whether the presence of a demon bit in a slot will force the slot to make a temporary slot in every instance. A temporary slot is used by ORE to provide a local slot in an object even when the value of the slot is inherited. If a temporary slot is not

available, then there will be no demon run for that object. This is especially necessary when one wants inherited objects to follow the behavior of a prototype object. For instance, in a rectangle object, if one changes the Am_LEFT slot in the prototype, one would like a demon to be fired for the Am_LEFT slot in any instance of that prototype. That requires there to be a temporary slot for every instance. For demons that require a temporary slot put a one in the demon mask. For all other demons put zero. Bitwise-and is performed between the demon mask and the demon bits of every slot that is instanced. If the result is non-zero a temporary slot is created for the new instance.

```
// Setting the demon mask
unsigned short mask = my_adv_obj->Get_Demon_Mask ();
mask |= 0x0020;  // add the new demon bit.
my_adv_obj->Set_Demon_Mask (mask);
```

3.10.6.4 The Demon Queue

The demon queue is where demon procedure are put when their events occur. Objects store the demon queue in the same way that they store their demon set: the same queue is shared when objects are instanced or copied. However, Amulet never uses more than one demon queue. There is only one global queue for all objects. It is quite possible to make a new queue and store it in an object but it just never happens.

```
// Here is how to make a new queue.
// It is highly unlikely that anyone will ever do this.
Am_Demon_Queue* my_queue = Am_Demon_Queue::Create ();
my_adv_obj.Set_Queue (my_queue);
```

To find and manipulate the global demon queue, one can take any object and read its queue.

```
Am_Demon_Queue* global_queue =
          ((Am_Object_Advanced&)Am_Root_Object).Get_Queue ();
```

The demon queue has two basic operations: one can enqueue a new demon into the list and one can cause the queue to invoke. Invoking the queue causes all the demon procedures stored to be read out of the queue, in order, and executed. While the queue is invoking, it cannot be invoked recursively. This prevents the queue from being read out of order while a demon is still running.

The demon queue is automatically invoked in a few circumstances. First, the queue is invoked whenever the method Get is called on an object. This is to make sure that any demons that affect the slot being retrieved are brought up to date. Another time is when the Destroy method is called on the object for similar reasons. The other time invoke is called is in the updating cycle in main loop and other window updating procedures. When windows are updated the demon queue is invoked to update all the object values.

The demon queue is not a true queue in that it does not have a dequeue operation. The dequeue is wrapped up in the Invoke method. The queue does have a means for deleting

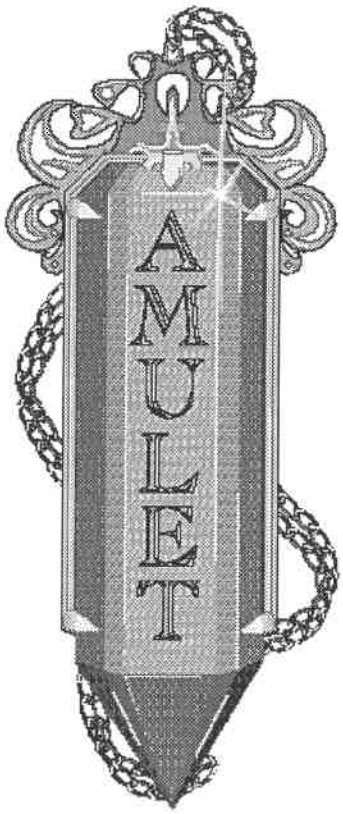
entries, though. One can delete all the demons that have a given slot or object as a parameter by using the `Delete` method. One can also delete the demons that pertain to a given object including the slots that the object contains by using the `Delete_Owned` method.

3.10.6.5 How to Allocate Demon Bits and the Eager Demon

In order to develop new slot demons, one must provide a bit name for it. Presently, Amulet does not provide a means for dispensing bit names for demons. To see if a demon bit is being used by an object it is quite possible to read the slot demons from the demon set and see which bits are not being used. This procedure should be very sufficient since one would not expect an object's demon set to be modified very often. Generally, only prototype objects need to be manipulated and one can often know which demons are set in a given prototype object.

Some of the demon bits are off limits to Amulet programmers. Amulet reserves two bits for use by the object system and another three bits for opal. The object system uses bits 0 and 1, opal uses bits 2, 3, and 4. Bits 5, 6, and 7 are available for programmers. Presently there are only the eight bits total available for slot demons.

Bit 0 in the object system is for the "eager demon." The eager demon is a default demon that all slots use. The demon is used simply to validate the slot whenever it becomes invalid. This makes the formula validation scheme eager hence the name. A programmer can turn off eager evaluation by turning off the eager bit in all the slots that one wants to be lazy. One can also set the eager demon procedure to be NULL in the demon set. When one is adding new demons to a slot one should be careful not to turn off the eager bit by accident.



4. Opal Graphics System

Abstract

This chapter describes simple graphical objects, styles, and fonts in Amulet. "Opal" stands for the Object Programming Aggregate Layer. Opal makes it easy to create and manipulate graphical objects. In particular, Opal automatically handles object redrawing when properties of objects are changed.

This research was sponsored by NCCOSC under Contract No. N66001-94-C-6037, Arpa Order No. B326. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NCCOSC or the U.S. Government.

4.1 Overview

This chapter describes the Opal graphical object system. The text assumes that the reader is familiar with the ideas of objects, slots, and constraints presented in the Amulet Tutorial and the ORE chapter.

4.1.1 Include Files

There are several data types documented in this chapter, declared in several different header files. You only need to include `amulet.h` at the top of your files, but some programmers like to look at header files. Here is a list of most of the objects and other data types discussed in this manual, along with the header file in which they are declared.

- `gdefs.h`: `Am_Style`, `Am_Font`, `Am_Point_List`, `Am_Image_Array`
- `opal.h`: default `Am_Style`'s, default `Am_Font`'s, `Am_Screen`, `Am_Graphical_Object`, `Am_Window`, `Am_Rectangle`, `Am_Roundtangle`, `Am_Line`, `Am_Arrow`, `Am_Polygon`, `Am_Arc`, `Am_Text`, `Am_Bitmap`, `Am_Group`, `Am_Map`, default constraints, `Am_Initialize`, `Am_Cleanup`, `Am_Beep`, `Am_Move_Object`, `Am_To_Top`, `Am_To_Bottom`, `Am_Create_Screen`, `Am_Update`, `Am_Update_All`, `Am_Do_Events`, `Am_Main_Event_Loop`, `Am_Exit_Main_Event_Loop`, default `Am_Point_In_` functions, `Am_Translate_Coordinates`, `Am_Merge_Pathname`
- `value_list.h`: `Am_Value_List`
- `text_fns.h`: all text editing functions, `Am_Edit_Translation_Table`

4.2 The Opal Layer of Amulet

Opal, which stands for the Object Programming Aggregate Layer, provides simple graphical objects for use in the Amulet environment. The goal of Opal is to make it easy to create and edit graphical objects. To this end, Opal provides default values for all of the properties of objects, so simple objects can be drawn by setting only a few parameters. If an object is changed, Opal automatically handles refreshing the screen and redrawing that object and any other objects that may overlap it. Objects in Opal can be connected together using *constraints*, which are relations among objects that are declared once and automatically maintained by the system. An example of a constraint is that a line must stay attached to a rectangle. Constraints are discussed in the Tutorial and the ORE chapter.

Opal is built on top of the Gem module, which is the Graphics and Events Module that refers to machine-specific functions. Gem provides an interface to both X windows and to Windows NT, so applications implemented with Opal objects and functions will run on either platform without modification. Gem is described in chapter 7.

To use Opal, the programmer should be familiar with the ideas of objects and constraints presented in the Tutorial. Opal is part of the Amulet library, so all objects discussed in this manual are accessible just by linking Amulet to your program (see the Overview manual for instructions). Opal will work with any window manager on top of X/11, such as mwm, uwm, twm, etc. Additionally, Opal provides support for color and gray-scale displays.

Within the Amulet environment, Opal forms an intermediary layer. It uses facilities provided by the ORE object and constraint system, and provides graphical objects that can be combined to build more complicated gadgets. Opal does not handle any input from the keyboard or mouse -- that is handled by the separate *Interactors* module, which is described in chapter 5. The Amulet Widgets, such as scroll-bars and menus, are partially built out of Opal objects and partly by calling Gem directly (for efficiency). Widgets are generally more complicated than the Opal objects, usually consisting of interactors attached to graphics, and are discussed in chapter 6.

4.3 Basic Concepts

4.3.1 Windows, Objects, and Groups

X/11 and Windows NT both allow you to create windows on the screen. In X they are called "drawables", and in Windows NT they are just called "windows". An Opal window is an ORE data structure that contains pointers to these machine-specific structures. Like in X/11 and Windows NT, Opal windows can be nested inside other windows (to form "sub-windows"). Windows clip all graphics so they do not extend outside the window's borders.

Also, each window forms a new coordinate system with (0,0) in the upper-left corner. The coordinate system is one-to-one with the pixels on the screen (each pixel is one unit of the coordinate system). Amulet windows are discussed fully in section 4.10.

The basics of object-oriented programming are beyond the scope of this chapter. The objects in Opal use the ORE object system, and therefore operate as a prototype-instance model. This means that each object can serve as a prototype (like a class) for any further instances; there is no distinction between classes and instances. Each graphic primitive in Opal is implemented as an object. When the programmer wants to cause something to be displayed in Opal, it is necessary to create instances of these graphical objects. Each instance remembers its properties so it can be redrawn automatically if the window needs to be refreshed or if objects change.

A *group* is a special kind of object that holds a collection of other objects. Groups can hold any kind of graphic object including other groups, but an object can only be in one group at a time. Therefore, groups form a pure hierarchy. The objects that are in a group are called *parts* of that group, and the group is called the *owner* of each of the parts. Groups, like windows, clip their contents to the bounding box defined by their

left, top, width, and height. Groups also define their own coordinate system, so that the positions of their parts are relative to the left and top of the group.

Objects are not drawn anywhere until they are added to a window. Windows are not drawn until they are added to the screen. Graphical objects can be added directly to a window, or they can be added to a group that is, in turn, part of a window.

Non-graphical objects, like Interactors and Command objects (and the application-specific objects) can be added as parts to any kind of object, but graphical objects can only be added as parts to an `Am_Window` or a `Am_Group`.

4.3.2 The "Hello World" Example

An important goal of Opal is to make it significantly easier to create pictures, hiding most of the complexity of the X/11 and Windows NT graphics models. Therefore, there are appropriate defaults for all properties of objects (such as the color, line-thickness, etc.). These only need to be set if the user desires to. All of the complexity of the X/11 and Windows NT graphics packages is available to the Opal user, but it is hidden so that you do not need to deal with it unless it is necessary to your task.

To get the string "Hello world" displayed on the screen (and refreshed automatically if the window is covered and uncovered), you only need the following simple program:

```
#include <amulet/amulet.h>

main (void)
{
  Am_Initialize ();

  Am_Screen
    .Add_Part (Am_Window.Create ("window")
      .Set (Am_WIDTH, 200)
      .Set (Am_HEIGHT, 50)
      .Add_Part (Am_Text.Create ("string")
        .Set (Am_TEXT, "Hello World!")));

  Am_Main_Event_Loop ();
  Am_Cleanup ();
}
```

This code, and a Makefile that compiles it, is provided in the Amulet source code in the directory `samples/hello/`. Note that the programmer never calls "draw" or "erase" methods on objects. This is a significant difference from other graphical object systems. Opal causes the objects to be drawn and erased at the appropriate times automatically.

Section 4.5 presents all the kinds of objects available in Opal.

4.3.3 Initialization and Cleanup

Amulet requires a call to `Am_Initialize()` before referencing any Opal objects or classes. This function creates the Opal prototypes and sets up bookkeeping information in Amulet objects and classes. Similarly, a call to `Am_Cleanup()` at the end of your program allows Amulet to destroy the prototypes and classes, explicitly freeing memory that might otherwise remain occupied.

4.3.4 The Main Event Loop

In order for interactors to perceive input from the mouse and keyboard, the main-event-loop must be running. This loop constantly checks to see if there is an event, and processes it if there is one. The automatic redrawing of graphics also relies on the main-event-loop. Exposure events, which occur when one window is uncovered or *exposed*, cause Amulet to refresh the window by redrawing the objects in the exposed area.

A call to `Am_Main_Event_Loop()` should be the second-to-last instruction in your program, just before `Am_Cleanup()`. Your program will continue to run until Amulet perceives the escape sequence, which by default is `CONTROL-ESC`. Typically, your program will have some sort of Quit button that calls the `Am_Exit_Main_Event_Loop()` routine, which will cause the main event loop to terminate.

4.3.5 Am_Do_Events

Normally, in response to an input event, applications will make some changes to graphics or their internal state and then return. Sometimes, however, an application might want to make a graphical change, have it seen by the user while waiting a little, and then make another change. This might be necessary to make something blink a few times, or for a short animation. To do this, an application must call `Am_Do_Events()` to cause Amulet to update the screen based on all the changes that have happened so far. Eventually, Amulet will contain an Animation Interactor, but it is not implemented yet.

You might also use `Am_Do_Events()` if you want your own event loop, for example, because you need to monitor non-Amulet queues, processes or inter-process-communication sockets. Calling `Am_Do_Events()` repeatedly in a loop will cause all the Interactors and Amulet activities to operate correctly, because the standard `Am_Main_Event_Loop` essentially does the same thing as calling `Am_Do_Events()` in a loop. `Am_Do_Events()` never blocks waiting for an event, but returns immediately whether there is anything to do or not. The return boolean from `Am_Do_Events` tells whether the whole application should quit or not. Therefore, a main-loop might look like:

```
main (void) {
  Am_Initialize ();
  ... // do the necessary set up and creating of objects

  // use the following instead of Am_Main_Event_Loop ();
  bool continue_looping = true;
  while (continue_looping) {
     continue_looping = Am_Do_Events();
     ... // check other queues or whatever
  }
  Am_Cleanup ();
}
```

4.4 Slots of All Graphical Objects

4.4.1 Left, Top, Width, and Height

All graphical objects have Am_LEFT, Am_TOP, Am_WIDTH, and Am_HEIGHT slots that determine their position and dimensions. Some objects have simple numerical values in these slots, and some have formulas that compute these values. Check the section below for a specific object to find its default values for these slots. All values must be ints.

4.4.2 Am_VISIBLE

In a graphical object, the Am_VISIBLE slot, which should contain a bool, controls whether the object is drawn in the window or group that it is a part of. In a window, Am_VISIBLE controls whether the window is drawn on the screen. To make a group and all of its parts invisible, it is sufficient to just set the Am_VISIBLE slot of the group to false.

Invisible objects are typically ignored by interactors and graphics routines. For example, you cannot use interactors to select an invisible object with the mouse, even if you click on the area where the invisible object would appear. Also, invisible parts of a group are generally not taken into account when the size of the group is computed.

4.4.3 Line Style and Filling Style

The Am_LINE_STYLE and Am_FILL_STYLE slots hold instances of the Am_Style class. If an object has a style in its Am_LINE_STYLE slot, it will have a border of that color. If it has a style in the Am_FILL_STYLE slot, it will be filled with that color. Other properties such as line thickness and stipple patterns are determined by the styles in these slots.

Usually you do not have to create customized instances of Am_Style to change the color of an object -- you can just use the predefined styles like Am_Red. Styles are fully documented in section 4.6.

Storing the special value Am_No_Style or NULL in either slot will cause the object to have no border or no fill.

4.4.4 Am_HIT_THRESHOLD and Am_PRETEND_TO_BE_LEAF

The Am_HIT_THRESHOLD, Am_PRETEND_TO_BE_LEAF, and Am_VISIBLE slots are used by functions which search for objects given a rectangular region or an (x,y) coordinate. For example, suppose a mouse click in a window should select an object from a group of objects. When the mouse is clicked, Amulet compares the location of the mouse click with the size and position of all the objects in the window to see which one was selected.

First of all, only visible objects can be selected this way. If an object's Am_VISIBLE slot contains false, it will not respond to events such as mouse clicks with conventional Interactors programming techniques.

The Am_HIT_THRESHOLD slot controls the sensitivity of functions that decide whether an event (like a mouse click) occurred "inside" an object. If the Am_HIT_THRESHOLD of an object is 3, then an event 3 pixels away from the object will still be interpreted as being "inside" the object. The default value of Am_HIT_THRESHOLD for all Opal objects is 0. **Note:** it is often necessary to set the Am_HIT_THRESHOLD slot of all groups *above* a target object; if an event occurs "outside" of a group, then the selection functions will not check the parts of the group.

When the value of a group's Am_PRETEND_TO_BE_LEAF slot is true, then the selection functions will treat that group as a leaf object (even though the group has parts). See Section 4.9.2 regarding the function Am_Point_In_Leaf. Also, consult the Interactors manual regarding the function Am_Inter_In_Leaf.

4.5 Specific Graphical Objects

The descriptions in this section highlight aspects of each object that differentiate it from other objects. Some properties of Opal objects are similar for all objects, and are documented in section 4.4. All of the exported objects in Amulet are summarized in Chapter 8.

4.5.1 Am_Rectangle

Am_Rectangle:

Slot	Default Value	Type
Am_VISIBLE	true	bool
Am_LEFT	0	int
Am_TOP	0	int
Am_WIDTH	10	int
Am_HEIGHT	10	int
Am_FILL_STYLE	Am_Black	Am_Style
Am_LINE_STYLE	Am_Black	Am_Style

4.5.2 Am_Line

Am_Line:

Slot	Default Value	Type
Am_LINE_STYLE	Am_Black	Am_Style
Am_X1	0	int
Am_Y1	0	int
Am_X2	0	int
Am_Y2	0	int
Am_LEFT	0	int
Am_TOP	0	int
Am_WIDTH	1	int
Am_HEIGHT	1	int
Am_VISIBLE	true	bool
Am_HIT_THRESHOLD	0	int

Am_X1, Am_Y1, Am_X2, Am_Y2, Am_LEFT, Am_TOP, Am_WIDTH, and Am_HEIGHT are constrained in such a way that if any one of them changes, the rest will automatically be updated to reflect that change. The Am_FILL_STYLE slot is ignored in Am_Line.

4.5.3 Am_Arc

Am_Arc: *(useful for circles, ovals, and arcs)*

Slot	Default Value	Type
Am_VISIBLE	true	bool
Am_LEFT	0	int
Am_TOP	0	int
Am_WIDTH	10	int
Am_HEIGHT	10	int
Am_ANGLE1	0	0..360
Am_ANGLE2	360	0..360
Am_FILL_STYLE	Am_Black	Am_Style
Am_LINE_STYLE	Am_Black	Am_Style

The slots Am_ANGLE1 and Am_ANGLE2 are used to specify the origin and terminus of the arc. The arc runs from Am_ANGLE1 counterclockwise for a distance of Am_ANGLE2 degrees. Am_ANGLE1 is measured from 0° at the center right of the oval (three o'clock), and Am_ANGLE2 is measured from Am_ANGLE1.

Arcs are filled as pie pieces to the center of the oval when a colored filling style is provided.

4.5.4 Am_Roundtangle

Am_Roundtangle: *(rectangle with rounded corners)*

Slot	Default Value	Type
Am_VISIBLE	true	bool
Am_LEFT	0	int
Am_TOP	0	int
Am_WIDTH	10	int
Am_HEIGHT	10	int
Am_RADIUS	Am_SMALL_RADIUS	Am_Radius_Flag or int
Am_FILL_STYLE	Am_Black	Am_Style
Am_LINE_STYLE	Am_Black	Am_Style

Instances of the Am_Roundtangle prototype are rectangles with rounded corners. The slots in this object are the same as Am_Rectangle, with the additional slot Am_RADIUS, which specifies the curvature of the corners. The value of the Am_RADIUS slot can either be an integer, indicating an absolute pixel radius for the corners, or an element of the enumerated type Am_Radius_Flag, indicating a small, medium, or large radius (see table below). The keyword values do not correspond directly to pixel values, but rather compute a pixel value as a fraction of the length of the shortest side of the bounding box.

Value of Am_RADIUS	Fraction
Am_SMALL_RADIUS	1/5
Am_MEDIUM_RADIUS	1/4
Am_LARGE_RADIUS	1/3

Figure 4-1 shows the meanings of the slots of Am_Roundtangle. If the value of Am_RADIUS is 0, the roundtangle looks just like a rectangle. If the value of Am_RADIUS is more than half the shortest side (which would mean there is not room to draw a corner of that size), then the corners are drawn as large as possible, as if the value of Am_RADIUS were half the shortest side.

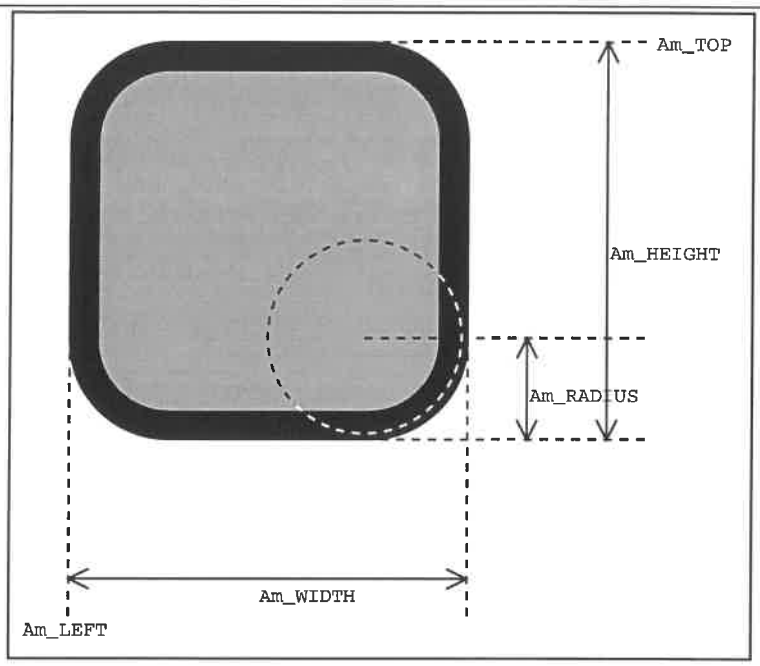


Figure 4-1: The parameters of a roundtangle.

4.5.5 Am_Polygon

Am_Polygon:

Slot	Default Value	Type
Am_VISIBLE	true	bool
Am_LEFT	<formula>	int
Am_TOP	<formula>	int
Am_WIDTH	<formula>	int
Am_HEIGHT	<formula>	int
Am_POINT_LIST	empty Am_Point_List	Am_Point_List

The interface to the Am_Polygon object is more complicated than other Opal objects. To specify the set of points that should be drawn for the polygon, you must first create an instance of Am_Point_List with all your (x,y) coordinates, and then install the point list in the Am_POINT_LIST slot of your Am_Polygon object.

Section 4.5.5.1 lists all of the functions that are available for the Am_Point_List class, including how to create point lists and add points to the list. Section 4.5.5.2 provides an example of how to create a polygon using the Am_Polygon object and the Am_Point_List class.

The formulas in the Am_LEFT, Am_TOP, Am_WIDTH, and Am_HEIGHT slots reevaluate whenever a new point list is installed in the Am_POINT_LIST slot. If a destructive modification is made to the point list class currently installed in the slot, such as adding a point to it with Add_Point(), then Note_Changed() will have to be called to cause the

formulas to reevaluate and to cause the object to be redrawn (for details about destructive modification of slot values and formula evaluation, see the ORE manual).

The shape defined by an `Am_Polygon` object does not necessarily have to be a closed polygon. To draw a regular polygon, the first and last points in the `Am_Point_List` should be the same. If they are not the same, and a fill style is provided, then the border of the fill style will be along an invisible line between the first and last points.

4.5.5.1 The Am_Point_List Class

`Am_Point_List` is a regular C++ class, defined in `gdefs.h`. Here is a list of the member functions that are available for the `Am_Point_List` class. The creator functions are used to make customized instances containing the list of (x,y) coordinates. The other functions are for adding, removing, and manipulating points within your specific instance. Section 4.5.5.2 contains an example of how to use the `Am_Point_List` class with the `Am_Polygon` object.

```
Am_Point_List (); // empty list: this constructor is implicitly called when you declare the type
                  //  of a point list variable, as in "Am_Point_List my_point_list;"
Am_Point_List (int x, int y); // first point: as in "Am_Point_List my_pl (25, 75)"
Am_Point_List (int *ar, int size); // copy from array: the array should be a flat list
                                   //  of values { x1 y1 x2 y2 ... xn yn }, where size is 2n

void Add_Point (int x, int y, int index); // Index is relative to number of (x,y) pairs.
                                          //  If there is already a point at index, it is
                                          //  replaced by the new point. Index can
                                          //  be any positive integer, regardless of
                                          //  the number of points already in the list.

void Delete_Point (int index); // index is relative to number of (x,y) pairs.

void Get_Extents (int& min_x, int& min_y, int& max_x, int& max_y);

int Size () const;  // returns number of (x,y) pairs in the list
```

The member functions of the `Am_Point_List` class are invoked using the standard C++ dot notation, as in "`my_point_list.Add_Point (10, 20);`".

4.5.5.2 Using Point Lists with Am_Polygon

The list of points for the polygon should be installed in an instance of `Am_Point_List`, and then that point list should be installed in the `Am_POINT_LIST` slot of your `Am_Polygon` object. The constructors for `Am_Point_List` allow you to initialize your point list with some, all, or none of the points that you will eventually use. After the point list has been created, you can add and remove points from it.

Here is an example of a triangle generated by adding points to an empty point list. The point list is then installed in an Am_Polygon object. To see the graphical result of this example, add the triangle_polygon object to a window.

```
Am_Point_List triangle_pl;
triangle_pl.Add_Point (15, 50, 0);
triangle_pl.Add_Point (45, 10, 1);
triangle_pl.Add_Point (75, 50, 2);

Am_Object triangle_polygon = Am_Polygon.Create ("triangle_polygon")
  .Set (Am_POINT_LIST, triangle_pl)
  .Set (Am_LINE_STYLE, Am_Line_2)
  .Set (Am_FILL_STYLE, Am_Yellow);
```

Here is an example of a five-sided star generated from an array of integers. To see the graphical result of this example, add the star_polygon object to a window.

```
static int star_ar[12] = {100, 0, 41, 181, 195, 69, 5, 69, 159, 181,
                          100, 0};

Am_Point_List star_pl (star_ar, 12);

Am_Object star_polygon = Am_Polygon.Create ("star_polygon")
  .Set (Am_POINT_LIST, star_pl)
  .Set (Am_FILL_STYLE, Am_No_Style);
```

4.5.6 Am_Text

Am_Text:

Slot	Default Value	Type
Am_VISIBLE	true	bool
Am_LEFT	0	int
Am_TOP	0	int
Am_WIDTH	<formula>	int
Am_HEIGHT	<formula>	int
Am_TEXT	""	Am_String
Am_FONT	Am_Default_Font	Am_Font
Am_CURSOR_INDEX	Am_NO_CURSOR	int
Am_LINE_STYLE	Am_Line_2	Am_Style
Am_FILL_STYLE	Am_No_Style	Am_Style
Am_X_OFFSET	<formula>	int
Am_INVERT	false	bool

The Am_Text object is used to display single lines of text in a specified font. The Am_TEXT slot holds the string to be displayed, and the Am_FONT slot holds an instance of Am_Font. Section 4.5.6.1 contains a list of predefined fonts and discusses how to create specific fonts.

The Am_WIDTH and Am_HEIGHT slots contain formulas that reevaluate according to the string and the font being displayed. These formulas are marked so that if the value of the slot is set, then the formula will be removed and from then on the string will stay the

constant value set there. In that case, the Am_X_OFFSET field contains a formula that scrolls the text string left and right to make sure the cursor is always visible.

The Am_CURSOR_INDEX contains an integer specifying the position of the cursor in the string. A value of zero places the cursor at the beginning of the string, and a value of Am_NO_CURSOR turns off the cursor.

The Am_LINE_STYLE slot controls the color of the string and the thickness of the cursor. If a style is provided in the Am_FILL_STYLE slot, then the background behind the text will be filled with that color. Setting Am_INVERT to true causes the line and filling styles to be switched, which is useful for "highlighting" text. If Am_INVERT is true but no fill style is provided, Amulet draws the text as white against a background of the line style color.

4.5.6.1 Fonts

Am_Font is a regular C++ class, defined in gdefs.h. Its creator functions are used to make customized instances describing the desired font. You can create fonts either with standard parameters that are more likely to be portable across different platforms, or by specifying the name of a specific font. The properties of fonts are: *family* (fixed, serif, or sans-serif), *face* (bold, italic, and/or underlined), and *size*. Am_Default_Font, exported from opah.h, is the fixed-width, medium-sized font you would get from calling the Am_Font constructor with its default values. Allowed values of the standard parameters appear below.

Constructors:

```
Am_Font (Am_Font_Family_Flag family = Am_FONT_FIXED,
         bool is_bold = false,
         bool is_italic = false,
         bool is_underline = false,
         Am_Font_Size_Flag size = Am_FONT_MEDIUM)

Am_Font (const char* the_name)
```

Pre-Defined Fonts:

Am_Default_Font - a fixed, medium-sized font

In the creator functions for Am_Font, the allowed values for the *family* parameter are:

- Am_FONT_FIXED -- a fixed-width font, such as Courier.
- Am_FONT_SERIF -- a variable-width font with "serifs", such as Times.
- Am_FONT_SANS_SERIF -- a variable-width font with no serifs, such as Helvetica.

The allowed values for the *size* parameter are:

- Am_FONT_SMALL -- a small size: about 10 pixels tall
- Am_FONT_MEDIUM -- a normal size: about 12 pixels tall
- Am_FONT_LARGE -- a large size: about 18 pixels tall

- Am_FONT_VERY_LARGE -- a larger size: about 24 pixels tall

4.5.6.2 Functions on Text and Fonts

There are additional functions that operate on Am_Text objects, strings, and fonts declared in the header file text_fns.h. These functions are included in the standard Amulet library, but are not automatically included by amulet.h because of their infrequent use. The Am_Text_Interactor uses these functions to edit strings. To access these functions directly, add the line "#include <amulet/text_fns.h>" at the top of your Amulet program.

4.5.6.3 Editing Text

Text editing is a feature provided by the Interactors module. To make a text object respond to mouse clicks and the keyboard, you need to define an instance of Am_Text_Interactor that operates on it. See the Interactors manual for details.

4.5.7 Am_Bitmap

Am_Bitmap:

Slot	Default Value	Type
Am_VISIBLE	true	bool
Am_LEFT	0	int
Am_TOP	0	int
Am_WIDTH	<formula>	int
Am_HEIGHT	<formula>	int
Am_LINE_STYLE	Am_Black	Am_Style
Am_FILL_STYLE	Am_No_Style	Am_Style
Am_IMAGE	Am_No_Image *(solid)*	Am_Image_Array

The interface to the Am_Bitmap object is more complicated than other Opal objects. To specify the image that should be drawn by this object, you must first create an instance of Am_Image_Array containing the image data, and then install the image in the Am_IMAGE slot of your Am_Bitmap object.

Section 4.5.7.1 lists all of the functions that are available for the Am_Image_Array class, including how to create images from data stored in files. Section 4.5.7.2 provides an example of how to display an image using the Am_Bitmap object and the Am_Image_Array class.

The formulas in the Am_WIDTH and Am_HEIGHT slots reevaluate whenever a new image is installed in the Am_IMAGE slot. If a destructive modification is made to the image class currently installed in the slot, such as by using Set_Bit(), then Note_Changed() will have to be called to cause the formulas to reevaluate and to cause the object to be redrawn (for details about destructive modification of slot values and formula evaluation, see the ORE manual).

4.5.7.1 The Am_Image_Array Class

Am_Image_Array is a regular C++ class, defined in gdefs.h. Here is a list of the member functions that are available for the Am_Image_Array class. The creator functions are used to make customized instances containing images described by data arrays or data stored in files. The other functions are for accessing, changing, and saving images to a file. Section 4.5.7.2 contains an example of how to use the Am_Image_Array class with the Am_Bitmap object. Section 4.6.3.7 discusses how to use the Am_Image_Array class with Am_Style.

```
Am_Image_Array (int percent);                    // Halftone pattern (see Section 4.6.2.2)

Am_Image_Array (unsigned int width,              // Solid rectangular image
                unsigned int height,
                int depth,
                Am_Style intial_color);

Am_Image_Array (const char* file_name);          // Pattern read from a file:
                                                 //  • On Unix, the file must be in X11 bitmap
                                                 //      format (i.e., generated by the Unix
                                                 //      bitmap utility).
                                                 //  • On the PC, the file must be in BMP or
                                                 //      GIF format
                                                 // For portable code, you should use #ifdef to
                                                 // load a different file depending on the platform
                                                 // (See the Amulet space demo and testgobs for
                                                 // examples.) Use XV to convert among formats.

void Get_Hot_Spot (int& x, int& y) const;
void Set_Hot_Spot (int x, int y);

// The size of an image will be zero until drawn, & depends on the window in which the image is displayed.
void Get_Size (int& width, int& height);

int Get_Bit (int x, int y);  // Not Implemented Yet
void Set_Bit (int x, int y, int val); // Not Implemented Yet
int Write_To_File (const char* file_name, // Not Implemented Yet
                   Am_Image_File_Format form);
```

4.5.7.2 Using Images with Am_Bitmap

To display an image whose description is stored in a file, you must first create an instance of Am_Image_Array initialized with the name of the file, and then install that image in the Am_IMAGE slot of your Am_Bitmap object.

The example below creates a bitmap for the Federation ship used in the Amulet space demo. Either the X11 or BMP file is read, depending on which platform the code has been compiled (the compiler variable _WINDOWS is defined in Visual C++ for the PC). To see the graphical result of this code, add the federation_bm object to a window.

```
// Use either the X11 or BMP format, depending on the platform
#if defined(_WINDOWS)
#define FEDERATION_FILE "lib/images/ent.bmp"
#else
#define FEDERATION_FILE "lib/images/ent"
#endif

Am_Image_Array federation_image =
  Am_Image_Array (Am_Merge_Pathname(FEDERATION_FILE)); // Merge the filename
                                                       // with the Amulet root
                                                       // directory (see Sec. 4.9.4)

Am_Object federation_bm = Am_Bitmap.Create ("federation_bm")
  .Set (Am_LEFT, 20)
  .Set (Am_TOP, 30)
  .Set (Am_IMAGE, federation_image);
```

4.6 Styles

The C++ class `Am_Style` is used to specify a set of graphical properties, such as color and line thickness. The `Am_LINE_STYLE` and `Am_FILL_STYLE` slots present in all graphical objects hold instances of `Am_Style`. The definition of the `Am_Style` class is in `gdefs.h`, and the predefined styles are listed in `opal.h`.

There are many predefined styles, like `Am_Red`, that provide the most common colors and thicknesses (see Section 4.6.1). You can also create your own custom styles just by calling the constructor functions for `Am_Style` with your desired parameters (see Sections 4.6.2 and 4.6.3). Whether you use a predefined style or create your own, you can set it directly into the appropriate slot of a graphical object. The style placed in the `Am_LINE_STYLE` slot of a graphical object controls the drawing of lines and outlines, and the style in the `Am_FILL_STYLE` slot controls the inside of the object.

Instances of `Am_Style` are immutable, and cannot be changed after they are created.

4.6.1 Predefined Styles

The most frequently used styles are predefined by Amulet. You can use any of the styles listed in this section directly in the `Am_LINE_STYLE` or `Am_FILL_STYLE` slot of an object.

Color styles: all have thickness zero (which really means 1--explained in the manual)

```
Am_Red          Am_Cyan         Am_Motif_Gray       Am_Motif_Light_Gray
Am_Green        Am_Orange       Am_Motif_Blue       Am_Motif_Light_Blue
Am_Blue         Am_Black        Am_Motif_Green      Am_Motif_Light_Green
Am_Yellow       Am_White        Am_Motif_Orange     Am_Motif_Light_Orange
Am_Purple                       Am_Amulet_Purple
```

Thick and dashed line styles: all are black

```
Am_Thin_Line    Am_Line_1       Am_Line_4           Am_Dashed_Line
Am_Line_0       Am_Line_2       Am_Line_8           Am_Dotted_Line
```

Stippled styles: all are black and white

```
Am_Gray_Stipple             Am_Opaque_Gray_Stipple
Am_Light_Gray_Stipple       Am_Diamond_Stipple
Am_Dark_Gray_Stipple        Am_Opaque_Diamond_Stipple
```

Special:

`Am_No_Style` - can be used in place of `NULL`

You can create a style of any other color, of any line thickness, by using the constructor functions in Section 4.6.2 and 4.6.3.

4.6.2 Creating Simple Line and Fill Styles

4.6.2.1 Thick Lines

To quickly create black line styles of a particular thickness, you can use the following special `Am_Style` creator function:

```
Am_Style::Thick_Line (unsigned short thickness);
```

For example, if you wanted to create a black line style 5 pixels thick, you could say "`black5 = Am_Style::Thick_Line (5)`". To specify the color or any other property simultaneously with the thickness, you have to use the full `Am_Style` creator functions discussed in Section 4.6.3. Section 4.6.3.2 explains the thickness.

4.6.2.2 Halftone Stipples

Stippled styles repeat a pattern of "on" and "off" pixels throughout a line style or fill style. A *halftone* is the most common type of stipple pattern, where the "on" and "off" bits are regularly spaced to create darker or lighter shades of a color. When mixing black and white pixels, for example, a 50% stipple of black and white bits will look gray. A 75% stipple will look darker, and a 25% stipple will look lighter. Some gray stipples are predefined in Amulet, and listed in Section 4.6.1. More complicated stipples, such as diamond patterns, are discussed in Section 4.6.3.7.

To create a simple halftone style with a regular stipple pattern, use this special `Am_Style` creator function:

```
Am_Style::Halftone_Stipple (int percent,
              Am_Fill_Solid_Flag fill_flag = Am_FILL_STIPPLED);
```

The *percent* parameter determines the shade of the halftone (0 is white and 100 is black). The *fill_flag* determines whether the pattern is transparent or opaque (see Section 4.6.3.6). In order to create a halftone that is one-third black and two-thirds white, you could say "`gray33 = Am_Style::Halftone_Stipple (33)`". There are only 17 different halftone shades available in Amulet, so several values for *percent* will map onto each built-in shade.

To specify the color or any other property simultaneously with the stipple, or to specify a more interesting stipple pattern, you have to use the full `Am_Style` creator functions discussed in Section 1.6.3.

4.6.3 Customizing Line and Fill Style Properties

Any property of a style can be specified by creating an instance of `Am_Style`. The properties are provided as parameters to the `Am_Style` constructor functions. All the parameters have convenient defaults, so you only have to specify values for the parameters you are interested in. Since styles are used for both line styles and fill styles, some of the parameters only make sense for one kind of style or the other. The parameters that do not apply in a particular situation are simply ignored.

```
Am_Style (float red, float green, float blue,          //color part
          short thickness = 0,
          Am_Line_Cap_Style_Flag cap_flag = Am_CAP_BUTT,
          Am_Join_Style_Flag join_flag = Am_JOIN_MITER,
          Am_Line_Solid_Flag line_flag = Am_LINE_SOLID,
          const char* dash_list = Am_DEFAULT_DASH_LIST,
          int dash_list_length = Am_DEFAULT_DASH_LIST_LENGTH,
          Am_Fill_Solid_Flag fill_flag = Am_FILL_SOLID,
          Am_Fill_Poly_Flag poly_flag = Am_FILL_POLY_EVEN_ODD,
          Am_Image_Array stipple = Am_No_Image)

Am_Style (const char* color_name,                      //color part
          short thickness = 0,
          Am_Line_Cap_Style_Flag cap_flag = Am_CAP_BUTT,
          Am_Join_Style_Flag join_flag = Am_JOIN_MITER,
          Am_Line_Solid_Flag line_flag = Am_LINE_SOLID,
          const char *dash_list = Am_DEFAULT_DASH_LIST,
          int dash_list_length = Am_DEFAULT_DASH_LIST_LENGTH,
          Am_Fill_Solid_Flag fill_flag = Am_FILL_SOLID,
          Am_Fill_Poly_Flag poly_flag = Am_FILL_POLY_EVEN_ODD,
          Am_Image_Array stipple = Am_No_Image)
```

The only required parameters for these style constructors are for the colors, discussed below. Before you read the details about what all the other parameters mean, be aware that most applications will just use the default values. All these parameters are discussed in Sections 1.6.3.1 through 1.6.3.7 below.

4.6.3.1 Color Parameter

The `Am_Style` constructor functions allow color to be specified in two ways: either with three *red*, *green*, *blue* values, or with a *color_name* such as "pink". The RGB values should be three `floats` between 0.0 and 1.0, where 1.0 is full on. Color names are looked up by the native graphics system (either X Windows or Windows NT) to get color indices. In X, the list of allowed color names is stored in the file `/usr/misc/lib/rgb.txt`, `/usr/misc/.X11/lib/rgb.txt`, or `/usr/lib/X11/rgb.txt`. However, if the X server does not find the color, a warning will be printed and black will be used instead.

4.6.3.2 Thickness Parameter

The *thickness* parameter holds the integer line thickness in pixels. There may be a subtle difference between lines with thickness zero and lines with thickness one. Zero thickness lines are actually drawn as one pixel wide, but they use a device-dependent line drawing algorithm, and therefore may be less aesthetically pleasing. They are also probably drawn much more efficiently. Lines with thickness one are drawn using the same algorithm with which all the thick lines are drawn. For this reason, a thickness zero line parallel to a thick line may not be as aesthetically pleasing as a line with thickness one.

For instances of `Am_Rectangle`, `Am_Roundtangle`, and `Am_Arc`, increasing the thickness of the line style will not increase the width or height of the object; the object will stay the same size, but the solid black boundary of the object will extend *inwards* to occupy more of the object. On the other hand, increasing the thickness of the line style of an `Am_Line` or `Am_Polygon` will increase the object's width and height; for these objects the thickness will extend *outward* on *both sides* of the line or polyline.

4.6.3.3 Cap_Flag Style Parameter

The *cap_flag* parameter determines how the end caps of line segments are drawn in X11 -- this parameter is ignored on the PC. Allowed values are elements of the enumerated type `Am_Line_Cap_Style_Flag`:

cap_flag	Result
`Am_CAP_BUTT`	Square at the endpoint (perpendicular to the slope of the line) with no projection beyond.
`Am_CAP_NOT_LAST`	Equivalent to `Am_CAP_BUTT`, except that for thickness 0 or 1 the final endpoint is not drawn.
`Am_CAP_ROUND`	A circular arc with the diameter equal to the thickness centered on the endpoint.
`Am_CAP_PROJECTING`	Square at the end, but the path continues beyond the endpoint for a distance equal to half of the thickness.

4.6.3.4 Join_Flag Style Parameter

The *join_flag* parameter determines how corners (where multiple lines come together) are drawn for thick lines as part of rectangle and polygon objects in X11 -- this parameter is ignored on the PC. This does not affect individual lines (instances of `Am_Line`) that are part of a group, even if they happen to have the same endpoints. Allowed values are elements of the enumerated type `Am_Join_Style_Flag`:

join_flag	Result
`Am_JOIN_MITER`	The outer edges of the two lines extend to meet at an angle.
`Am_JOIN_ROUND`	A circular arc with a diameter equal to the *thickness* is drawn centered on the join point.
`Am_JOIN_BEVEL`	Endpoints of lines are drawn with `Am_CAP_BUTT` style, with the triangular notch filled.

4.6.3.5 Dash Style Parameters

The *line_flag* parameter determines whether the line is solid or dashed, and how the spaces between the dashes should be drawn. Valid values are elements of the enumerated type `Am_Line_Solid_Flag`:

line_flag	Result
`Am_LINE_SOLID`	No dashes
`Am_LINE_ON_OFF_DASH`	Only the "on" dashes are drawn, and nothing is drawn in the "off" dashes.

The *dash_list* and *dash_list_length* parameters describe the pattern for dashed lines. The *dash_list* should be a `const char*` array that holds numbers corresponding to the pixel length of the "on" and "off" pixels. The default `Am_DEFAULT_DASH_LIST` value is `{4 4}`. A *dash_list* of `{1 1 1 1 3 1}` is a typical dot-dot-dash line. A list with an odd number of elements is equivalent to the list being appended to itself. Thus, the *dash_list* `{3 2 1}` is equivalent to `{3 2 1 3 2 1}`.

The following code defines a dash pattern with each "on" and "off" dash 15 pixels long. To see the result of this code, store the `thick_dash` style in the `Am_LINE_STYLE` slot of a graphical object.

```
static char thick_dash_list[2] = {15, 15};
Am_Style thick_dash ("black", 8, Am_CAP_BUTT, Am_JOIN_MITER,
             Am_LINE_ON_OFF_DASH, thick_dash_list);
```

4.6.3.6 Fill Style Parameters

The *fill_flag* determines the way "off" pixels in the stippled pattern (see Section 1.6.3.7) will be drawn. The "on" pixels are always drawn with the color of the style. Allowed values are elements of the enumerated type `Am_Fill_Solid_Flag`:

fill_flag	Result
`Am_FILL_SOLID`	Draw "off" pixels same as "on" pixels.
`Am_FILL_TILED`	Not implemented yet
`Am_FILL_STIPPLED`	Only the "on" pixels are drawn, and nothing is drawn for the "off" pixels (transparent stipple).
`Am_FILL_OPAQUE_STIPPLED`	Draw the "off" pixels in white.

The value of the *poly_flag* parameter should be an element of the enumerated type `Am_Fill_Poly_Flag`, either `Am_FILL_POLY_EVEN_ODD` or `Am_FILL_POLY_WINDING`. This parameter controls the filling for self-intersecting polygons, like the five-pointed star example in Section 1.5.5.2. For a better discussion of polygon filling, see any reasonable graphics textbook, or the X11 Protocol Manual.

4.6.3.7 Stipple Parameters

A stippled style consists of a small pattern of "on" and "off" pixels that is repeated throughout the border or filling of an object. The simplest stipple pattern is the halftone, discussed in Section 1.6.2.2. You should only need to specify the *stipple* parameter in the full `Am_Style` creator functions when you are specifying some other property (like color) along with a non-solid stipple, or you are specifying an unconventional image for your stipple pattern.

The value of the *stipple* parameter should be an instance of `Am_Image_Array`. An image array holds the pattern of bits, which can either be a standard halftone pattern or something more exotic. The creator functions and other member functions for `Am_Image_Array` are discussed in Section 1.5.7.1.

Here is an example of a colored style with a 50% halftone stipple, created using the halftone initializer for `Am_Image_Array`:

```
Am_Style red_stipple ("red", 8, Am_CAP_BUTT, Am_JOIN_MITER, Am_LINE_SOLID,
                      Am_DEFAULT_DASH_LIST, Am_DEFAULT_DASH_LIST_LENGTH,
                      Am_FILL_STIPPLED, Am_FILL_POLY_EVEN_ODD,
                      (Am_Image_Array (50)) );
```

Here is an example of a stipple read from a file. The "stripes" file contains a description of a bitmap image. On Unix, the "stripes" file must be in X11 bitmap format (i.e., generated with the Unix `bitmap` utility). On the PC, the "stripes" file must be in either `BMP` or `GIF` format. This implies that for portable image definitions, you should use the

`#ifdef` macro to load different files depending on your platform (see Section 1.5.7.2, the Amulet `space` demo and `testgobs` for more examples).

```
Am_Style striped_style ("black", 8, Am_CAP_BUTT, Am_JOIN_MITER,
                        Am_LINE_SOLID, Am_DEFAULT_DASH_LIST,
                        Am_DEFAULT_DASH_LIST_LENGTH, Am_FILL_STIPPLED,
                        Am_FILL_POLY_EVEN_ODD,
                        (Am_Image_Array ("stripes")) );
```

4.7 Groups

Am_Group:

Slot	Default Value	Type
Am_VISIBLE	true	bool
Am_LEFT	0	int
Am_TOP	0	int
Am_WIDTH	10	int
Am_HEIGHT	10	int
Am_GRAPHICAL_PARTS	empty Am_Value_List	Am_Value_List
Am_LAYOUT	NULL	<formula>
Am_X_OFFSET	0	int
Am_Y_OFFSET	0	int
Am_H_SPACING	0	int
Am_V_SPACING	0	int
Am_H_ALIGN	Am_CENTER_ALIGN	{Am_LEFT_ALIGN, Am_RIGHT_ALIGN, Am_CENTER_ALIGN}
Am_V_ALIGN	Am_CENTER_ALIGN	{Am_TOP_ALIGN, Am_BOTTOM_ALIGN, Am_CENTER_ALIGN}
Am_FIXED_WIDTH	Am_NOT_FIXED_SIZE	int
Am_FIXED_HEIGHT	Am_NOT_FIXED_SIZE	int
Am_INDENT	0	int
Am_MAX_RANK	false	int, bool
Am_MAX_SIZE	false	int, bool

Groups hold a collection of graphical objects (possibly including other groups). The objects in a group are called its *parts*, and the group is the *owner* of each component. The concept of part/owner relationships was introduced in the ORE manual, but groups treat their parts specially, by drawing them in windows.

Groups define their own coordinate system, meaning that the left and top of their parts is offset from the origin of the group. Changing the position of the group *translates* the position of all its parts. However, there is no complementary feature of *scaling* groups to change the size of all the parts.

Am_Width_Of_Parts Am_Height_Of_Parts Groups also *clip* their parts to the bounding box of the group, meaning that objects outside the left, top, width, or height of the group are not drawn. The default width and height of any group is only 10x10, so you must be careful to set the `Am_WIDTH` and `Am_HEIGHT` slot of your instances of `Am_Group`. The predefined constraints `Am_Width_Of_Parts` and `Am_Height_Of_Parts` may be used to compute the size of a group based on its parts.

Groups can lay out their parts in rows and columns, as in a list of menu items. See Section 1.7.2.

4.7.1 Adding and Removing Graphical Objects

The Am_GRAPHICAL_PARTS slot of a group contains a list of objects. This slot should be considered read-only, to be referenced perhaps in a situation involving iteration over graphical parts. Parts of a group should be manipulated with the following member functions defined on Am_Object:

```
Am_Object Add_Part (Am_Object new_part);
Am_Object Add_Part (Am_Slot_Key key, Am_Object new_part);

void Remove_Part (Am_Slot_Key key);
void Remove_Part (Am_Object part);
```

Parts can be *named* or *unnamed*, depending on whether a slot name is provided in the Add_Part() call. If a slot name is provided, then in addition to becoming a part of the group, the new part will be stored in that slot of the group. It is often convenient to name parts so that functions and formulas can easily access these objects in their groups. Also, named parts are reproduced in groups when instances of groups are created.

Of course, you can add non-graphical parts, like Interactors, to a group or any other kind of object, but only add graphical parts to groups or windows.

4.7.2 Layout

The simplest type of group is one that does not use a layout procedure. In a regular group, each part has its own left and top, which places it at some user-defined position relative to the left and top of the group.

However, it is often convenient for the group itself to lay out its graphical parts. For example, if the parts should all be in a row or column. Therefore, the Am_Group object can contain a formula in the Am_LAYOUT slot which lays out all of the parts. This formula operates by directly setting the Am_LEFT and Am_TOP of the parts. The following sections discuss this in more detail.

4.7.2.1 Vertical and Horizontal Layout

Amulet provides two built-in layout procedures:

```
Am_Constraint*  Am_Vertical_Layout
Am_Constraint*  Am_Horizontal_Layout
```

These layout procedures arrange the parts of a group according to the values in the slots listed below. To arrange the parts of a group in a vertical list (like a menu), set the Am_LAYOUT slot to Am_Vertical_Layout. You may then want to set other slots of the

group like Am_V_SPACING to control things like the spacing between parts or the number of columns.

These procedures install values in the Am_LEFT and Am_TOP slots of the parts of the group, overriding whatever values were there before.

The slots that control layout when using the standard vertical or horizontal layout procedures are:

Am_X_OFFSET - The horizontal space to leave between the origin of the group and the first part that is placed, measured in number of pixels (default is 0)

Am_Y_OFFSET - Same as Am_X_OFFSET, only vertical (default is 0)

Am_H_SPACING - The horizontal space to leave between parts, measured in pixels (default is 0)

Am_V_SPACING - Same as Am_H_SPACING, only vertical (default is 0)

Am_H_ALIGN - Justification for parts within a column: when a narrow part appears in a column with other wider parts, this parameter determines whether the narrow part is positioned at the left, center, or right of the column (default is Am_CENTER_ALIGN)

Am_V_ALIGN - Same as Am_H_ALIGN, only vertical (default is Am_CENTER_ALIGN)

Am_FIXED_WIDTH - The width of each column, probably based on the width of the widest part. When Am_NOT_FIXED_SIZE is used, the columns are not necessarily all the same width; instead, the width of each column is determined by the widest part in that column. (default is Am_NOT_FIXED_SIZE)

Am_FIXED_HEIGHT - Same as Am_FIXED_WIDTH, only vertical (default is Am_NOT_FIXED_SIZE)

Am_INDENT - How much to indent the second row or column (depending on horizontal or vertical orientation), measured in number of pixels (default is 0)

Am_MAX_RANK - The maximum number of parts allowed in a row or column, depending on horizontal or vertical orientation (default is false)

Am_MAX_SIZE - The maximum number of pixels allowed for a row or column, depending on horizontal or vertical orientation (default is false)

. For example, the following will create a column containing a rectangle and a circle:

```
Am_Object my_group = Am_Group.Create ("my_group")
    .Set (Am_LEFT, 10)
    .Set (Am_TOP, 10)
    .Set (Am_LAYOUT, Am_Vertical_Layout)
    .Set (Am_V_SPACING, 5)
    .Add_Part(Am_Rectangle.Create())
    .Add_Part(Am_Circle.Create());
```

4.7.2.2 Custom Layout Procedures

You can provide a customized layout procedure for arranging the parts of a group. The procedure should be defined as a constraint, using Am_Define_Formula or a related function, and the constraint should be installed in the Am_LAYOUT slot of the group. The parts of the group should be arranged as a side effect of evaluating the formula (the return value is ignored). This can be done by iterating over the Am_GRAPHICAL_PARTS list, and setting each part's Am_LEFT and Am_TOP slots appropriately.

4.8 Maps

Am_Map:

Slot	Default Value	Type
Am_VISIBLE	true	bool
Am_LEFT	0	int
Am_TOP	0	int
Am_WIDTH	Am_Width_Of_Parts	int
Am_HEIGHT	Am_Height_Of_Parts	int
Am_GRAPHICAL_PARTS	<formula>	Am_Value_List
Am_ITEMS	0	int, Am_Value_List
Am_ITEM_PROTOTYPE	Am_No_Object	Am_Object
Am_LAYOUT	NULL	<formula>
Am_X_OFFSET	0	int
Am_Y_OFFSET	0	int
Am_H_SPACING	0	int
Am_V_SPACING	0	int
Am_H_ALIGN	Am_CENTER_ALIGN	{Am_LEFT_ALIGN, Am_RIGHT_ALIGN, Am_CENTER_ALIGN}
Am_V_ALIGN	Am_CENTER_ALIGN	{Am_TOP_ALIGN, Am_BOTTOM_ALIGN, Am_CENTER_ALIGN}
Am_FIXED_WIDTH	Am_NOT_FIXED_SIZE	int
Am_FIXED_HEIGHT	Am_NOT_FIXED_SIZE	int
Am_INDENT	0	int
Am_MAX_RANK	false	int, bool
Am_MAX_SIZE	false	int, bool

The Am_Map object is a special kind of group that generates graphical parts based on a prototype description. Maps should be used when all the parts of a group are similar enough that they can be generated from one prototype object (for example, they are all rectangles, or all the same kind of group.). This part-generating feature of maps is often used in conjunction with the layout feature of groups, in a situation such as arranging the

selectable text items in a menu. For details on laying out the components of groups and maps, see Section 1.7.2.

You must set two slots in the map to control the parts that are generated:

- Am_ITEMS - The value should be either:
  - A number, specifying how many parts should be generated, or
  - An instance of Am_Value_List, containing elements corresponding to each part to be generated
- Am_ITEM_PROTOTYPE - A graphical object or group, to serve as the prototype for each part

Additionally, there are two slots automatically installed in each of the generated parts, that are useful for distinguishing the parts from each other. These slots can be referenced by formulas in the item-prototype to make each part different (see examples below).

Am_RANK - The position of this part in the list, from 0
Am_ITEM - The element of the map's Am_ITEMS list that corresponds to this part

The Am_RANK of each created part is set with the count of this part, so that the first part's Am_RANK is set to 0, the second part's Am_RANK is set to 1, and so on. If the Am_ITEMS slot of the map contains an Am_Value_List, then the Am_ITEM (note: singular) of each created part is set with the corresponding element of the list.

The following code defines a map whose Am_ITEMS slot is a number. The map generates 4 rectangles, whose fill styles are determined by the formula map_fill_from_rank. The formula computes a halftone fill from the value stored in the Am_RANK slot of the part, which was installed by the map as the part was created. This uses a vertical layout formula so the rectangles will be in a column.

```
// Formulas are defined at the top level, outside of main()
Am_Define_Style_Formula (map_fill_from_rank) {
int rank = self.GV (Am_RANK);
  return Am_Style::Halftone_Stipple (20 * rank);
}
...

  // Defined inside main()
  Am_Object my_map = Am_Map.Create ("my_map")
    .Set (Am_LEFT, 10)
    .Set (Am_TOP, 10)
    .Set (Am_LAYOUT, Am_Vertical_Layout)
    .Set (Am_V_SPACING, 5)
    .Set (Am_ITEMS, 4)
    .Set (Am_ITEM_PROTOTYPE, Am_Rectangle.Create ("map item")
          .Set (Am_FILL_STYLE, Am_Formula::Create (map_fill_from_rank))
          .Set (Am_WIDTH, 20)
          .Set (Am_HEIGHT, 20));
```

The next example defines a map whose Am_ITEMS slot contains a list of fill styles. The map generates 4 rectangles, whose fill styles are determined by the formula map_fill_from_item. The formula simply returns the value stored in the Am_ITEM slot of the part, which was installed by the map as the part was created.

```
// Formulas are defined at the top level, outside of main()
Am_Define_Style_Formula (map_fill_from_item)
{
  return self.GV (Am_ITEM);
}
...

  // Defined inside main()
  Am_Object my_map = Am_Map.Create ("my_map")
    .Set (Am_LEFT, 10)
    .Set (Am_TOP, 10)
    .Set (Am_LAYOUT, Am_Horizontal_Layout)
    .Set (Am_H_SPACING, 5)
    .Set (Am_ITEMS, Am_Value_List ()
                    .Add (Am_Motif_Blue)
                    .Add (Am_Motif_Green)
                    .Add (Am_Motif_Orange)
                    .Add (Am_Motif_Light_Blue))
    .Set (Am_ITEM_PROTOTYPE, Am_Rectangle.Create ("map item")
          .Set (Am_ITEM, Am_Black)
          .Set (Am_FILL_STYLE, Am_Formula::Create (map_fill_from_item))
          .Set (Am_WIDTH, 20)
          .Set (Am_HEIGHT, 20));
```

To add another item to the map in the second example, you could install a new list in the Am_ITEMS slot containing all the old items plus the new one:

```
Am_Value_List map_items = (Am_Value_List) my_map.Get (Am_ITEMS);
map_items.Add (Am_Gray_Stipple);
my_map.Set (Am_ITEMS, map_items);
```

A more efficient way to add an item to the list is to destructively modify the list that is already installed (note the use of the *false* parameter in the Add method for Am_Value_List):

```
Am_Value_List map_items = (Am_Value_List) my_map.Get (Am_ITEMS);
map_items.Add (Am_Gray_Stipple, false);
my_map.Note_Changed (Am_ITEMS);
```

The list in the Am_ITEMS slot can also be calculated with a formula, and the items in the map will change whenever the formula is reevaluated.

For more information on Am_Value_Lists, see section 3.7.

4.9 Methods on all Graphical Objects

4.9.1 Reordering Objects

As you add objects to a group or window, each new object by default is on top of the previous one. This is called the "Z" or "stacking" or "covering" order.

The following functions are useful for changing the order of an object among its siblings. For example, `Am_To_Top(obj)` will bring an object to the front of a all of the other objects in the same group or window. To promote an object just above a certain target object, use `Am_Move_Object (obj, target_obj, true)`. These functions work for windows as well as for regular graphical objects.

```
void Am_To_Top (Am_Object object);
void Am_To_Bottom (Am_Object object);

void Am_Move_Object (Am_Object object, Am_Object ref_object,
                     bool above = true);
```

4.9.2 Finding Objects from their Location

The following functions are useful for determining whether an object is under a given (x,y) coordinate:

```
Am_Object Am_Point_In_Obj (Am_Object in_obj, int x, int y,
                           Am_Object ref_obj);

Am_Object Am_Point_In_Part (Am_Object in_obj, int x, int y,
                            Am_Object ref_obj);

Am_Object Am_Point_In_Leaf (Am_Object in_obj, int x, int y,
                            Am_Object ref_obj);
```

`Am_Point_In_Obj()` checks whether the point is inside the object. It ignores covering (i.e., it just checks whether point is inside the object, even if the object is covered). If the point is inside, the object is returned; otherwise the function returns `NULL` (0). The coordinate system of *x* and *y* is defined with respect to *ref_obj*, that is, the origin of *x* and *y* is the left and top of *ref_obj*.

`Am_Point_In_Part()` finds the front-most (least covered) immediate part of `in_obj` at the specified location. If none, then it returns `NULL` (0). The coordinate system of *x* and *y* is defined with respect to *ref_obj*.

`Am_Point_In_Leaf()` is similar to `Am_Point_In_Part()`, except that the search continues to the deepest part in the group hierarchy (i.e., it finds the leaf-most object at the specified location). If (*x*,*y*) is inside the bounding box of `in_obj` but not over a leaf, it returns *in_obj*. The coordinate system of *x* and *y* is defined with respect to *ref_obj*. Sometimes you will want a group to be treated as a leaf in this search, like a button group in a collection of buttons. In this case, you should set the `Am_PRETEND_TO_BE_LEAF` slot

to true for each group that should be treated like a leaf. The search will not proceed through the parts of such a group, but will return the group itself.

`Am_Point_In_Part()` and `Am_Point_In_Leaf()` use the function `Am_Point_In_Obj()` on the parts.

4.9.3 Beeping

```
void Am_Beep (Am_Object window = Am_No_Object);
```

This function causes the computer to emit a "beep" sound. Passing a specific window is useful in Unix, when several different screens might be displaying windows, and you only want a particular screen displaying a particular window to beep.

4.9.4 Filenames

```
char *Am_Merge_Pathname (char *name);
```

`Am_Merge_Pathname()` takes a filename as an argument, and returns the full Amulet directory pathname prepended to that argument. For example, "`Am_Merge_Pathname ("lib/images/ent.bmp")`" will return the full pathname to the Unix-compatible Enterprise bitmap included with the Amulet source files.

```
bool Am_Translate_Coordinates (Am_Object src_obj, int src_x, int src_y,
                               Am_Object dest_obj, int& dest_x, int& dest_y,
                               Am_Constraint_Context& cc = *Am_Empty_Constraint_Context);
```

4.9.5 Translate Coordinates

`Am_Translate_Coordinates()` converts a point in one object's coordinate system to that of another object. It works for both windows and groups. If the objects are not comparable (like being on different screens or not being on a screen at all) then the function will return `false`. Otherwise, it will return `true` and *dest_x* and *dest_y* will contain the converted coordinates. Note that the coordinates are for the inside of *dest_obj*. This means that if *obj* was at *src_x*, *src_y* in *src_obj* and you remove it from *src_obj* and add it to *dest_obj* at *dest_x*, *dest_y* then it will be at the same physical screen position. Providing an `Am_Constraint_Context` parameter can be used in formulas to make the formula dependent on the relative positions of the objects.

Since each group and window defines its own coordinate system, you must use `Am_Translate_Coordinates` whenever you define a formula that depends on the left or top of an object that might be in a different group or window.

4.10 Windows

Am_Window:

Slot	Default Value	Type
Am_VISIBLE	true	bool
Am_LEFT	0	int
Am_TOP	0	int
Am_WIDTH	100	int
Am_HEIGHT	100	int
Am_GRAPHICAL_PARTS	empty Am_Value_List	Am_Value_List
Am_FILL_STYLE	Am_White	Am_Style
Am_MAX_WIDTH	0	int
Am_MAX_HEIGHT	0	int
Am_MIN_WIDTH	1	int
Am_MIN_HEIGHT	1	int
Am_TITLE	"Amulet"	char*
Am_ICON_TITLE	"Amulet"	char*
Am_ICONIFIED	false	bool
Am_USE_MIN_WIDTH	false	bool
Am_USE_MIN_HEIGHT	false	bool
Am_USE_MAX_WIDTH	false	bool
Am_USE_MIN_HEIGHT	false	bool
Am_QUERY_POSITION	false	bool
Am_QUERY_SIZE	false	bool
Am_LEFT_BORDER_WIDTH	0	int
Am_TOP_BORDER_WIDTH	0	int
Am_RIGHT_BORDER_WIDTH	0	int
Am_BOTTOM_BORDER_WIDTH	0	int
Am_CURSOR	NULL	Am_Cursor
Am_OMIT_TITLE_BAR	false	bool
Am_CLIP_CHILDREN	false	bool
Am_SAVE_UNDER	false	false

Objects can be added to windows with `Add_Part()`, just like with groups. All graphical objects added to a window will be displayed in that window. When a window is added as a part to another window, it becomes a *subwindow*. Subwindows usually do not have any window manager decoration (such as title bars).

4.10.1 Slots of Am_Window

The initial values of `Am_LEFT`, `Am_TOP`, `Am_WIDTH`, and `Am_HEIGHT` determine the size and position of the window when it appears on the screen. These slots can be set later to change the window's size and position. If the user changes the size or position of a window using the window manager (e.g., using the mouse), this will be reflected in the values for these slots.

The `Am_FILL_STYLE` determines the background color of the window. All parameters of `Am_Style` that affect fillings, including stipples, affect the fill style of windows (see section 1.6). This is more efficient than putting a window-sized rectangle behind all the other objects in the window.

When values are installed in the Am_MAX_WIDTH, Am_MAX_HEIGHT, Am_MIN_WIDTH, or Am_MIN_HEIGHT slots, and the corresponding Am_USE_MAX_WIDTH, Am_USE_MAX_HEIGHT, Am_USE_MIN_WIDTH, or Am_USE_MIN_HEIGHT slot is set to true, then the window manager will make sure the user is not allowed to change the window's size to be outside of those ranges. You can still set the Am_WIDTH and Am_HEIGHT to be any value, but the window manager will eventually clip them back into the allowed range.

When Am_QUERY_POSITION or Am_QUERY_SIZE are set to true, then the user will have the opportunity to place the window on the screen when the window is first added to the screen, clicking the left mouse button to position the left and top of the window, and dragging the mouse to the desired width and height.

The border widths applied to the window by the window manager are stored in the Am_LEFT_BORDER_WIDTH, Am_TOP_BORDER_WIDTH, Am_RIGHT_BORDER_WIDTH, and Am_BOTTOM_BORDER_WIDTH. These slots should be considered read-only, set by Amulet as the window becomes visible on the screen.

When the Am_CURSOR slot is set with an instance of Am_Cursor, then the mouse pointer will change when it moves into this window according to the bitmaps in the Am_Cursor.

The Am_OMIT_TITLE_BAR slot tells whether the Amulet window should have a title bar. If the slot has value false (the default), and the window manager permits it, then the window will have a title bar; otherwise the window will not have a title bar.

In the rare case when you want to have graphics drawn on a parent window appear over the enclosed (child) windows, you can set the Am_CLIP_CHILDREN slot of the parent to be true. Then any objects that belong to that window will appear on top of the window's subwindows (rather than being hidden by the subwindows).

When the Am_SAVE_UNDER slot is set to true, then the window manager is instructed to save the graphics appearing under the window at all times, so that when it destroyed or made invisible, the graphics under the window will be redrawn quickly. Using this feature requires slightly more memory than otherwise.

4.10.2 Am_Screen

As mentioned in Section 1.3.1, windows are not visible until they are added to the screen. The Am_Screen object can be thought of as a root window to which all top-level windows are added. In the "hello world" example of Section 1.3.2, the top-level window is added to Am_Screen with a call to Add_Part().

Am_Screen can be used in calls to Am_Translate_Coordinates() to convert from window coordinates to global coordinates and back again.

4.11 Predefined formula constraints:

Opal provides a number of constraints that can be put into slots of objects that might be useful. Some of these constraints were described in previous sections.

`Am_Width_Of_Parts` - Useful for computing the width of a group: returns the distance between the group's left and the right of its rightmost part. You might put this into a group's `Am_WIDTH` slot.

`Am_Height_Of_Parts` - Analogous to `Am_Width_Of_Parts`, but for the `Am_HEIGHT`.

`Am_Right_Is_Right_Of_Owner` - Useful for keeping a part at the right of its owner. Put this formula in the `Am_LEFT` slot of the part.

`Am_Bottom_Is_Bottom_Of_Owner` - Useful for keeping a part at the bottom of its owner. Put this formula in the `Am_TOP` slot of the part.

`Am_Center_X_Is_Center_Of_Owner` - Useful for centering a part horizontally within its owner. Put this formula in the `Am_LEFT` slot of the part.

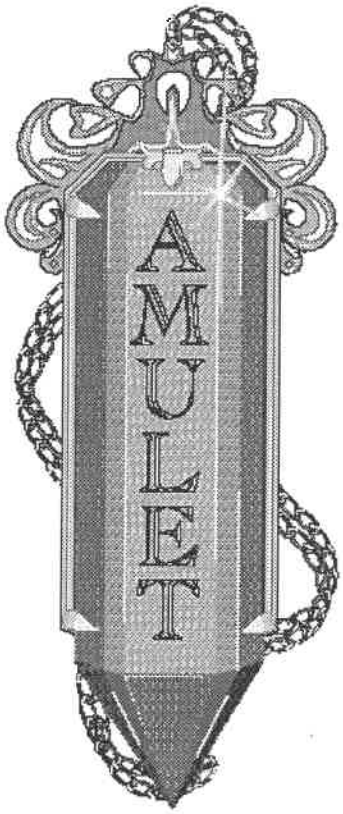
`Am_Center_Y_Is_Center_Of_Owner` - Useful for centering a part vertically within its owner. Put this formula in the `Am_TOP` slot of the part.

`Am_Center_X_Is_Center_Of` - Useful for horizontally centering `obj1` inside `obj2`. Put this formula in the `Am_LEFT` slot of `obj1`, and put `obj2` in the `Am_CENTER_X_OBJ` slot of `obj1`.

`Am_Center_Y_Is_Center_Of` - Useful for vertically centering `obj1` inside `obj2`. Put this formula in the `Am_TOP` slot of `obj1`, and put `obj2` in the `Am_CENTER_Y_OBJ` slot of `obj1`.

`Am_Horizontal_Layout` - Constraint which lays out the parts of a group horizontally in one or more rows. Put this into the `Am_LAYOUT` slot of a group.

`Am_Vertical_Layout` - Constraint which lays out the parts of a group vertically in one or more columns. Put this into the `Am_LAYOUT` slot of a group.



5. Interactors and Command Objects for Handling Input

Abstract

Graphical objects in Amulet do not respond to input events; they are purely output. When the programmer wants to make an object respond to a user action, an *Interactor* object is attached to the graphical object. The built-in types of Interactors usually enable the programmer to simply choose the correct type and fill in a few parameters. The intention is to significantly reduce the amount of coding necessary to define behaviors.

When an Interactor or a *widget* (see the Widgets chapter) finishes its operation, it allocates a *Command object* and then invokes the "do" method of that Command object. Thus, the Command objects take the place of *call-back procedures* in other systems. The reason for having Command objects is that in addition to the "do" method, a Command object also has methods to support undo, help, and selective enabling of operations. As with Interactors, Amulet supplies a library of Command objects so that often programmers can use a Command object from the library without writing any code.

This research was sponsored by NCCOSC under Contract No. N66001-94-C-6037, Arpa Order No. B326. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NCCOSC or the U.S. Government.

5.1 Include Files

The primary include files that control the Interactors and Command objects are `inter.h` for the main, top-level objects and procedures, `idefs.h` for the definitions specific to input events, and `inter_advanced.h` for what you might need if you are going to create your own custom Interactors. Of course, all the slots of the Interactor and Command objects are defined in `standard_slots.h`. Some of the functions and types needed to customize the text editing Interactor are defined in `text_fns.h`.

5.2 Overview of Interactors and Commands

The graphical objects created with Opal do not respond to input devices: they are just static graphics. In order to handle input from the user, you create an "Interactor" object and attach it to the graphics. The Interactor objects have built-in behaviors that correspond to the normal operations performed in direct manipulation user interfaces, so usually coding interactive interfaces is quick and easy using interactors. However, like programming with constraints, programming with Interactors requires a different "mind set" and the programming style is probably different than what most programmers are used to.

All of the Interactors are highly parameterized so that you can control many aspects of the behavior simply by setting slots of the Interactor object. For example, you can easily specify which mouse button or keyboard key starts the interactor. In order to affect the graphics and connect to application programs, each Interactor has multiple protocols. For example, the "Move-Grow" interactor, for moving graphical objects with the mouse, explicitly sets the `Am_LEFT` and `Am_TOP` slots of the object, and also calls the `Am_DO_ACTION` method stored in the Command object attached to the Interactor. Therefore, there are multiple ways to use an Interactor, to give programmers flexibility in what they need to achieve.

When an Interactor or a *widget* (see the Widgets chapter) finishes its operation, it allocates a *Command object* and then invokes the "do" method of that Command object. Thus, the Command objects take the place of *call-back procedures* in other systems. The reason for having Command objects is that in addition to the "do" method, a Command object also has methods to support undo, help, and selective enabling of operations. Each Interactor and Widget has a Command object as the part named `Am_COMMAND`, and Interactors set the `Am_VALUE` and other slots in its command object, and then call the `Am_DO_ACTION` method. This and other methods in the Command objects implement the functionality of the Interactors.

5.3 Standard Operations

We hope that most normal behaviors and operations will be supported by the Interactors and Command objects in the library. This section discusses how to use these. If you find that the standard operations are not sufficient, then you may need to create your own

Command objects or even your own Interactors. These advanced features are discussed in sections 5.5 and 5.7.

5.3.1 Designing Behaviors

The first task when designing the interaction for your interface is to choose the desired behavior. The first choice is whether one of the built-in widgets provides the right interface. If so, then you can choose the widget from the Widgets chapter and then attach the appropriate Command object to the widget. The widgets, such as buttons, scroll bars and text-input fields, combine a standard graphical presentation with an interactive behavior. If you want custom graphics, or you want an application-specific graphical object to be moved, selected or edited with the mouse, then you will want to create your own graphics and Interactors.

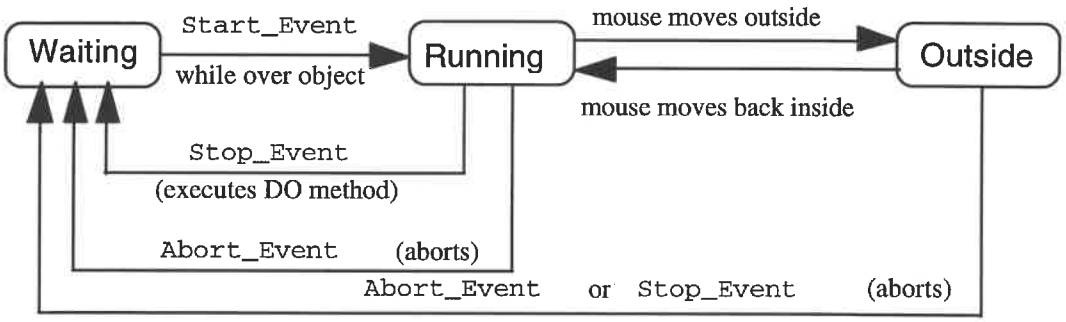
The first step in programming an Interactor is to pick one of the fundamental built-in styles of behavior that is closest to the interaction you want. The current choices are (these are exported in `inter.h`):

- `Am_Choice_Interactor`. This is used to choose one or more from a set of objects. The user is allowed to move around over the objects (getting *interim feedback*) until the correct item is found, and then there will often be *final feedback* to show the final selection. The `Am_Choice_Interactor` can be used for selecting among a set of buttons or menu items, or choosing among the objects dynamically created in a graphics editor.
- `Am_One_Shot_Interactor`. This is used whenever you want something to happen immediately when an event occurs, for example when a mouse button is pressed over an object, or when a particular keyboard key is hit. Like the `Am_Choice_Interactor`, the `Am_One_Shot_Interactor` can be used to select among a set of objects, but the `Am_One_Shot_Interactor` will not provide interim feedback—the object where you initially press will be the final selection. The `Am_One_Shot_Interactor` is also useful in situations where you are not selecting an object, such as when you want to get a single keyboard key.
- `Am_Move_Grow_Interactor`. This is useful in all cases where you want a graphical object to be moved or changed size with the mouse. It can be used for dragging the indicator of a scroll bar, or for moving and growing objects in a graphics editor.
- `Am_New_Points_Interactor`. This Interactor is used to enter new points, such as when creating new objects. For example, you might use this to allow the user to drag out a rubber-band rectangle for defining where a new object should go.
- `Am_Text_Edit_Interactor`. This supports editing the text string of a text object. It supports a flexible *key translation table* mechanism so that the programmer can easily modify and add editing functions. The built-in mechanisms support basic text editing behaviors.
- `Am_Rotate_Interactor`. This Interactor will support rotating graphical objects. It is not yet implemented.

- `Am_Gesture_Interactor`. This Interactor will support free-hand gestures, such as drawing an X over an object to delete it, or encircling the set of objects to be selected. It is not yet implemented.
- `Am_Animation_Interactor`. This Interactor will support animations and time-based events. It is not yet implemented.

5.3.2 General Interactor Operation

Once an Interactor is created and its parameters are set (see Section 5.3.3), the programmer will then attach the Interactor to some object in a window (see Section 5.3.3.1.5). Amulet then waits for the user to perform the Interactor's start event (for example by pressing the left mouse button, see Section 5.3.3.1) over the graphical object to which the Interactor is attached. `Am_One_Shot_Interactors` then immediately execute their Command's `DO` method and go back to waiting. Other types of Interactors, however, usually show interim feedback while waiting for a specific stop event (for example, left mouse button up). While Interactors are operating, the user might move the mouse outside the Interactor's operating area, in which case the Interactor stops working (for example, a choice Interactor used in a menu should turn off the highlighting if the mouse goes outside the menu). If the mouse goes back inside, then the Interactor resumes operation. If the abort_event is executed while an Interactor is running, then it is aborted (and the Command's `DO` method is not executed). Similarly, if the stop event is executed while the mouse is outside, the Interactor also aborts. The operation is summarized by the following diagram.



Multiple Interactors can be running at the same time. The way this works is that each Interactor keeps track of its own status, and for each input event, Amulet checks which Interactor or Interactors are interested in the event. The appropriate Interactor(s) will then process that event and return.

5.3.3 Parameters

Once the programmer has chosen the basic behavior that is desired, then the various parameters of the specific Interactor must be filled in. The next sections give the details of these parameters. Some, such as the start and abort events, are shared by all Interactors, and other parameters, such as gridding, are specific to only a few types of Interactors.

The parameters are set as normal slots of the objects. The names of the slots are defined in `standard_slots.h` and are described below. As an example, the following creates a choice Interactor called "Select It" assigned to the variable `select_it` and sets its start event to the middle mouse button. See the ORE chapter for how to create and name objects.

```
select_it = Am_Choice_Interactor.Create("Select It")
              .Set(Am_START_WHEN, "MIDDLE_DOWN");
```

5.3.3.1 Events

One of the most important parameters for all Interactors are the input events that cause them to start, stop and abort. These are encoded as an `Am_Input_Char` which are defined in `idefs.h`. Normally, you do not have to worry about these since they are automatically created out of normal C strings, but you can convert a string into an `Am_Input_Char` for efficiency, or if you want to set or access specific fields.

Note: Do not use a C++ `char` to represent the events. It must be a C string or an `Am_Input_Char` object.

5.3.3.1.1 Event Slots

There are three slots of Interactors that can hold events: `Am_START_WHEN`, `Am_ABORT_WHEN`, and `Am_STOP_WHEN`.

`Am_START_WHEN` determines when the Interactor begins operating. The default value is `"LEFT_DOWN"` with no modifier keys (see section 5.3.3.1.3) but with any number of clicks (see section 5.3.3.1.4). So by default, all interactors will operate on both single and double clicks.

`Am_ABORT_WHEN` allows the Interactor to be aborted by the user while it is operating. The default value is `"CONTROL_g"`. Aborting is different from undoing since you abort an operation *while it is running*, but you undo an operation *after it is completed.*

`Am_STOP_WHEN` determines when the Interactor should stop. The default value is `"ANY_MOUSE_UP"` so even if you change the start_when, you can often leave the stop_when as the default value.

5.3.3.1.2 Event Values

In any of these slots, you can provide an `Am_Input_Char`, a string in the format described below, or the special values `true` or `false`. The value `true` matches any event, and `false` will never match any event. You might use false for example in the `Am_ABORT_WHEN` slot of an Interactor to make sure it is never aborted.

The general form for the events is a string with the modifiers first and the specific keyboard key or mouse button last. The specific keys include the regular keyboard keys, like `"A"`, `"z"`, `"["`, and `"\""` (use the standard C++ mechanism to get special characters into the string). The various function and special keys are generally named the same thing as their label, such as `"F1"`, `"R5"`, `"HELP"`, and `"DELETE"`. Sometimes, keys have multiple markings, in which case we usually use the more specific or textual marking, or sometimes both markings will work. Also, the arrow keys are always called `"LEFT_ARROW"`, `"UP_ARROW"`, `"DOWN_ARROW"`, and `"RIGHT_ARROW"`. Note that keys with names made out of multiple words are separated by underscores. For keyboard keys, we currently only support operations on the button being pressed, and no events are generated when the button is released. You can specify any keyboard key with the special event `"ANY_KEYBOARD"`. You can find out what the mapping for a keyboard key is by running the test program `testinput`, which is in the `src/gem` directory. We have tried to provide appropriate mappings for all of the keyboards we have come across, but if there are keyboard keys on your keyboard that are not mapped appropriately, then please send mail to `amulet@cs.cmu.edu` and we will add them to the next release.

For the mouse buttons, we support both pressing and releasing. The names of the mouse buttons are `"LEFT"`, `"MIDDLE"` and `"RIGHT"` (on a 2-button mouse, they are `"LEFT"` and `"RIGHT"` and on a 1-button mouse, just `"LEFT"`), and you must append either `"UP"` or `"DOWN"`. Thus, the event for the left button down is `"LEFT_DOWN"`. You can specify any mouse button down or up using `"ANY_MOUSE_DOWN"` and `"ANY_MOUSE_UP"`.

5.3.3.1.3 Event Modifiers

The modifiers can be specified in any order and the case of the modifiers does not matter. The currently supported modifiers are:

`shift_` Either of the keyboard shift keys, or the caps-lock key is being held down. For regular letters, you can also just use the upper case. Thus, `"F"` is equivalent to `"SHIFT_f"`. However, do not use shift to try to get the special characters. Therefore `"Shift_5"` is *not* the same as `"%"`.

`control_` The control key is being held down.

`meta_` The meta key is the diamond key on Sun keyboards, the `EXTEND-CHAR` key on HPs, the option key on Macintosh keyboards, and the `ALT` key on PC keyboards. On other Unix keyboards, it is generally whatever is used for "meta" by Emacs and the window manager.

`any_` This means that you don't care which modifiers are down. Thus `"any_f"` matches `"shift_f"` as well as `"F"` and `"meta_control_shift_f"`. Note that `"ANY_KEYBOARD"` or `"ANY_MOUSE_DOWN"` also specifies any modifiers.

5.3.3.1.4 Multiple Clicks

Amulet supports the detection of multiple click events from the mouse. To double-click, the user must press down on the same mouse button quickly two times in succession. The clicks must be faster than `Am_Double_Click_Time` (which is defined in gem.h) milliseconds, which defaults to 250 milliseconds.

On the PC, Amulet detects single and double clicks, and on Unix Amulet will detect up to five clicks. The multiple clicks are named by preceding the event name with the words `"DOUBLE_"`, `"TRIPLE_"`, `"QUAD_"`, and `"FIVE_"`. For example, `"double_left_down"`, or `"shift_meta_triple_right_down"`. When the user double clicks, a single click event will still be generated first. For example, for the left button, the sequence of received events will be `"LEFT_DOWN"`, `"LEFT_UP"`, `"DOUBLE_LEFT_DOWN"`, `"DOUBLE_LEFT_UP"`. The `"ANY_"` prefix can be used to accept any number of clicks, so `"ANY_LEFT_DOWN"` will accept single or multiple clicks with any modifier held down.

5.3.3.1.5 Am_Input_Char type

The `Am_Input_Char` is defined in `idefs.h`. It is a regular C++ object (*not* an Amulet object). It has constructors from a string or from the various pieces:

```
Am_Input_Char (const char *s); //from a string like "META_LEFT_DOWN"
Am_Input_Char (short c = 0, bool shf = false,
               bool ctrl = false,
               bool meta = false, Am_Button_Down down = Am_NEITHER,
               Am_Click_Count click = Am_NOT_MOUSE,
               bool any_mod = false);
```

It can be converted to a string, to a long (which is only useful for storing the `Am_Input_Char` into a slot of an object) or to a character (which returns 0 if it is not an normal ascii character). An `Am_Input_Char` will also print to a stream as a string. If `ic` is an `Am_Input_Char`:

- `ic.As_String(char *s);` convert to a string by writing into `s`, which should be at least `Am_LONGEST_CHAR_STRING` characters long.
- `(long)ic;` convert `ic` into a long, for storing it into a slot.
- `char c = ic.As_Char();` Returns a char if `ic` represents a simple ascii character, otherwise returns 0
- `cout << ic;` you can print an `Am_Input_Char` directly.

The member variables of an `Am_Input_Char` are:

```
typedef enum { Am_NOT_MOUSE = 0,  //When not a mouse button.
               Am_SINGLE_CLICK = 1, //Also for mouse moved, with Am_NEITHER.
               Am_DOUBLE_CLICK = 2, Am_TRIPLE_CLICK = 3,
               Am_QUAD_CLICK = 4, Am_FIVE_CLICK = 5, Am_MANY_CLICK = 6,
               Am_ANY_CLICK = 7  // when don't care about how many clicks
             } Am_Click_Count;

typedef enum { Am_NEITHER = 0, Am_BUTTON_DOWN = 1,
               Am_BUTTON_UP = 2, Am_ANY_DOWN_UP = 3} Am_Button_Down;

short code; // the base code.
bool shift;  // whether these modifier keys were down
bool control;
bool meta;
bool any_modifier; //true if don't care about modifiers
Am_Button_Down button_down;  // whether a down or up transition.
                             // For keyboard, only support down.
Am_Click_Count click_count;  // 0==not mouse, otherwise # clicks
```

5.3.3.2 Graphical Objects

5.3.3.2.1 Start_Where

For an Interactor to become active, it must be added as a part to a graphical object which is part of a window. To do this, you use the regular Add_Part method of objects. For example, to make the `select_it` Interactor defined above in section 5.3.3 select the object `my_rect`, the following code could be used:

```
my_rect.Add_Part(select_it);
```

Interactors can be added as parts to any kind of graphical object, including primitives (like rectangles and strings), groups, and windows. You can add multiple Interactors to any object, and they can be interspersed with graphical parts for groups and windows. Interactors can be removed or queried with the standard object routines for parts. If you plan to make instances of the object and wish to have an instance made of the Interactor also, then the `Add_Part` call should also contain a slot name (see the ORE chapter). For example:

```
Am_Slot_Key INTER_SLOT = Am_Register_Slot_Name ("INTER_SLOT");
my_rect.Add_Part(INTER_SLOT, select_it); // named part
rect2 = my_rect.Create(); //rect2 will have its own which is an instance of select_it
```

It is very common for a behavior to operate over the *parts* of a group, rather than just on the object itself. For example, a choice Interactor might choose any of the items (parts) in a menu (group), or a move_grow Interactor might move any of the objects in the graphics window. Therefore, the slot `Am_START_WHERE_TEST` can hold a function to determine where the mouse should be when the start-when event happens for the

Interactor to start. The built-in functions for the slot (from `inter.h`) are as follows. Each of these returns the object over which the Interactor should start, or NULL if the mouse is in the wrong place so the Interactor should not start.

- `Am_Inter_In`: If the mouse is inside the object the Interactor is part of, this returns that object. This is the default.
- `Am_Inter_In_Part`: The Interactor should be part of a group or window object. This tests if the mouse is in a part of that group or window object, and if so, returns the part of the group or window the mouse is over.
- `Am_Inter_In_Leaf`: This is useful when the Interactor is part of a group or window which contains groups which contain groups, etc. It returns the lowest level object the mouse is over. If you want `Am_Inter_In_Leaf` to return a group rather than a part of the group, set the `Am_PRETEND_TO_BE_LEAF` slot of the group to be `true`.
- `Am_Inter_In_Text`: If the mouse is inside the object the Interactor is part of, and that object is an instance of `Am_Text`, then returns that object. This is useful for `Am_Text_Interactors`.
- `Am_Inter_In_Text_Part`: If the mouse is in a part of the object the Interactor is part of, and that the part the mouse is over is an instance of `Am_Text`, then returns that part. This is useful for `Am_Text_Interactors`.
- `Am_Inter_In_Text_Leaf`: If the mouse is in a leaf of the object the Interactor is part of, and that leaf part is an instance of `Am_Text`. This is useful for `Am_Text_Interactors`.

If none of these functions returns the object you are interested in, then you are free to define your own function. It should be of the form `Am_Where_Function` and return the object that the Interactor should manipulate, or `NULL` if none.

Note that this means that the Interactor may actually operate on an object *different* from the one to which it is attached. For example, Interactors will often be attached to a group but actually modify a part of that group. With a custom `Am_START_WHERE_TEST` function, the programmer can have the Interactor operate on a completely independent object.

5.3.3.2.2 Running_Where

Section 5.3.2 mentioned that Interactors can be defined so that they stop operating when the mouse goes outside of their active area. The active area is defined by the value of the `Am_RUNNING_WHERE_OBJECT` slot. This slot should contain either a graphical object or `true`, which means anywhere (so the Interactor never goes outside). The default for most Interactors for this slot is `true`, but for Choice Interactors, this slot contains a constraint that makes it have the same object as where the Interactor starts. To refine where the Interactor should be considered outside, the programmer can also supply a value for the `Am_RUNNING_WHERE_TEST` slot, which defaults to `Am_Inter_In` except for

choice Interactors, where it contains a constraint that uses the same function as the Am_START_WHERE_TEST.

5.3.3.3 Active

It is often convenient to be able to create a number of Interactors, and then have them turn on and off based on the global mode or application state. The Am_ACTIVE slot of an Interactor can be set to `false` to disable the Interactor, and it can be set to `true` to re-enable the Interactor. By default, all Interactors are active. The default value in this slot is a constraint that depends on the Am_ACTIVE slot of the command object in the interactor (see Section 5.5). Setting the Am_ACTIVE slot is more efficient than creating and destroying the Interactor. The Am_ACTIVE slot can also be set with a constraint that returns `true` or `false`.

5.3.4 Top Level Interactor

The top level Interactor has the following default values. Most of these are advanced features and are discussed in Section 5.4.

Am_Interactor:

Slot	Default Value	Type	
Am_START_WHEN	Am_Input_Char ("LEFT_DOWN")	Am_Input_Char	
Am_START_WHERE_TEST	Am_Inter_In	Am_Where_Function	
Am_ABORT_WHEN	Am_Input_Char ("CONTROL_g")	Am_Input_Char	
Am_RUNNING_WHERE_OBJECT	true	Am_Object, bool	
Am_RUNNING_WHERE_TEST	Am_Inter_In	Am_Where_Function	
Am_STOP_WHEN	Am_Input_Char ("ANY_MOUSE_UP")	Am_Input_Char	
Am_ACTIVE	true	bool	*// Section **Error! Reference source n found.***
Am_START_OBJECT	0	Am_Object	*// Section **Error! Reference source n found.***
Am_START_CHAR	0	Am_Input_Char	*// Section **Error! Reference source n found.***
Am_CURRENT_OBJECT	0	Am_Object	*// Section **Error! Reference source n found.***
Am_RUN_ALSO	false	bool	*// Section**Error! Reference source n found.***
Am_PRIORITY	1.0	float	*// Section **Error! Reference source n found.***
Am_OTHER_WINDOWS	NULL	Am_Value_List or Am_Window	*// Section **Error! Reference source n found.***
Am_WINDOW	NULL	Am_Window	*Set with current window*
Am_COMMAND	Am_Command	Am_Command	

5.3.5 Specific Interactors

All of the interactors and command objects are summarized in Chapter 8. The next sections discuss each one in detail.

5.3.5.1 Am_Choice_Interactor

Am_Choice_Interactor:

Slot	Default Value	Type	
Am_START_WHEN	Am_Input_Char ("LEFT_DOWN")	Am_Input_Char	
Am_START_WHERE_TEST	Am_Inter_In	Am_Where_Function	
Am_ABORT_WHEN	Am_Input_Char ("CONTROL_g")	Am_Input_Char	
Am_RUNNING_WHERE_OBJECT	<formula>	Am_Object, bool	*// computes own*
Am_RUNNING_WHERE_TEST	<formula>	Am_Where_Function	*// same as start*
Am_STOP_WHEN	Am_Input_Char ("ANY_MOUSE_UP")	Am_Input_Char	
Am_HOW_SET	Am_CHOICE_TOGGLE	Am_Choice_How_Set	
Am_FIRST_ONE_ONLY	false	bool	*// whether menu-* *// or button-like*
Am_COMMAND	Am_Choice_Command	Am_Command	

The choice Interactor is used whenever the programmer wants to choose one or more out of a set of objects, such as in a menu or to select objects in a graphics window. The standard behavior allows the programmer to choose whether one or more objects can be selected, and special slots called Am_INTERIM_SELECTED and Am_SELECTED of these objects are set by the default Command in the choice Interactor. Typically, the programmer would define constraints on the look of the object (e.g. the color) based on the values of these slots. Note that Am_INTERIM_SELECTED and Am_SELECTED are set in the *graphical object* the Interactor operates on, not in the Interactor itself.

5.3.5.1.1 Special Slots of Choice Interactors

Two slots of choice Interactors can be set to customize its behavior:

- Am_HOW_SET: This controls whether a single or multiple values will be selected. Legal values are from the following type:

```
typedef enum { Am_CHOICE_SET, Am_CHOICE_CLEAR, Am_CHOICE_TOGGLE,
               Am_CHOICE_LIST_TOGGLE } Am_Choice_How_Set;
```

  These mean:

  - Am_CHOICE_SET: the object under the mouse becomes selected, and the previously selected object is de-selected (useful for single selection menus and radio buttons). Unlike Am_CHOICE_TOGGLE, clicking on an already-selected object leaves it selected.
  - Am_CHOICE_CLEAR: the object under the mouse becomes de-selected. This is rarely useful.
  - Am_CHOICE_TOGGLE: if the object under the mouse is selected, it becomes deselected, otherwise it becomes selected and any previous object become de-selected. This is useful when you want zero or one selection (the user is able to turn off the selection).
  - Am_CHOICE_LIST_TOGGLE: if the object under the mouse is selected, then it is de-selected, otherwise it becomes selected, but other objects are left alone. This allows multiple selection, and is useful for check boxes.

The default value for the Am_HOW_SET slot is Am_CHOICE_TOGGLE.

- Am_FIRST_ONE_ONLY: If false (the default), then the selection is free to move from one item in the group to another, as in menus. If true, then only the initial object the mouse is over can be manipulated, and the user must release outside and then press down in another object to change objects. This is how radio button and check box widgets work on most systems.

5.3.5.1.2 Standard operation of the Am_Choice Command

Am_Choice_Command:

Slot	Default Value	Type
Am_START_ACTION	Am_Choice_Command_Start	
Am_INTERIM_DO_ACTION	Am_Choice_Command_Interim_Do	Am_Object_Proc*
Am_DO_ACTION	Am_Choice_Command_Do	Am_Object_Proc*
Am_UNDO_ACTION	Am_Choice_Command_Undo	Am_Object_Proc*
Am_UNDO_THE_UNDO_ACTION	Am_Choice_Command_Undo_ The_Undo	Am_Object_Proc*
Am_ABORT_ACTION	Am_Choice_Command_Abort	Am_Object_Proc*
Am_LABEL	"choice interactor"	Am_String
Am_ACTIVE	true	bool
Am_INTERIM_VALUE	0	Am_Object
Am_OLD_INTERIM_VALUE	0	Am_Object
Am_OLD_VALUE	0	any
Am_VALUE	0	any

As the Choice_Interactor is operating, it calls the various methods of the Command object stored in its Am_COMMAND slot. The default Command object, Am_Choice_Command, uses the following mechanisms to show the operation. If this is not sufficient for your needs, then you need to create a custom Command object, as described in Section 5.5.

As the Interactor moves over various graphical objects, the Am_INTERIM_SELECTED slot of the object is set to true for the object which is under the mouse, and false for all other objects. Typically, the graphical objects that the Interactor affects will have a constraint to the Am_INTERIM_SELECTED slot from the Am_FILL_STYLE or other slot. At any time, the Interactor can be aborted by typing the key in Am_ABORT_WHEN (the default is "controlg"). When the stop_when event occurs, the Am_INTERIM_SELECTED slot is set to false, and the Am_HOW_SET slot of the Interactor is used to decide how many objects are allowed to be selected (as explained above). The objects that should end up being selected have their Am_SELECTED slot set to true, and the rest of the objects have their Am_SELECTED slot set to false. Also the Am_VALUE slot of the Command object (in the Am_COMMAND slot of the Interactor) will contain the current value. If Am_HOW_SET is not Am_CHOICE_LIST_ TOGGLE, then the Am_VALUE slot will either contain the selected object or NULL (0). If Am_HOW_SET is Am_CHOICE_LIST_TOGGLE, then the Am_VALUE slot of the Command object will contain an Am_Multi_Value containing the list of the selected objects (or it will be the empty list).

The Undo_Action of the default Command object in the Am_Choice_Interactor simply resets the Am_SELECTED slots of the selected object(s) and the Am_Value of the Command object to be as they were before the Am_Choice_Interactor was run.

5.3.5.1.3 Simple Example

See the file testinter.cc for lots of additional examples of uses of Interactors and Command objects. The following Interactor works on any object which is directly a part of the window. Due to the constraints, if you press the mouse down over any rectangle created from rect_proto that is in the window, it will change to having a thick line style when they mouse is over it (when it is "interim-selected"), and they will turn white when the mouse button is release (and it becomes selected).

```
Am_Define_Style_Formula (rect_line)  {
  if ((bool)self.GV (Am_INTERIM_SELECTED)) return thick_line;
  else return thin_line;
}
Am_Define_Style_Formula (rect_fill)  {
  if ((bool)self.GV (Am_SELECTED)) return Am_White;
  else return self.GV (Am_VALUE);  //the real color
}
rect_proto = Am_Rectangle.Create ("rect_proto")
   .Set (Am_WIDTH, 30)
   .Set (Am_HEIGHT, 30)
   .Set (Am_SELECTED, false)
   .Set (Am_INTERIM_SELECTED, false)
   .Set (Am_VALUE, Am_Purple)  //put the real color here
   .Set (Am_FILL_STYLE, Am_Formula::Create (rect_fill))
   .Set (Am_LINE_STYLE, Am_Formula::Create (rect_line))
   ;
select_inter = Am_Choice_Interactor.Create("choose_rect")
               .Set (Am_START_WHERE_TEST, &Am_Inter_In_Part);
window.Add_Part (select_inter);
```

5.3.5.2 Am_One_Shot_Interactor

The Am_One_Shot_Interactor is used when you want something to happen immediately on an event. For example, you might want a Command to be executed when a keyboard key is hit, or when the mouse button is first pressed. The parameters and default behavior for the Am_One_Shot_Interactor are the same as for a Am_Choice_Interactor, in case you want to have an object be selected when the start_when event happens. The programmer can choose whether one or more objects can be selected, and the slots Am_INTERIM_SELECTED and Am_SELECTED of these objects are set by the default Command in the Am_One_Shot_ Interactor .

The slots for the Am_One_Shot_Interactor are identical to those for the Am_Choice_ Interactor (see above).

5.3.5.2.1 Simple Example

In this example, we create a `Am_One_Shot_Interactor` which calls the `change_setting` function when any keyboard key is hit in the `window`. The `unchange_setting` function will be used to undo this action. The programmer would write the `change_setting` and `unchange_setting` functions (see Section 5.5.3.1 for how to write command procedures that won't break the Interactors).

```
Am_Object how_set_inter =
  Am_One_Shot_Interactor.Create("change_settings")
        .Set(Am_START_WHEN, "ANY_KEYBOARD")
        ;
Am_Object cmd = how_set_inter.Get(Am_COMMAND);
cmd.Set(Am_DO_ACTION, (Am_Object_Proc*)&change_setting)
   .Set(Am_UNDO_ACTION, (Am_Object_Proc*)&unchange_setting);
window.Add_Part (how_set_inter);
```

5.3.5.3 Am_Move_Grow_Interactor

The `Am_Move_Grow_Interactor` is used to move or change the size of graphical objects with the mouse. The default Command object in the `Am_Move_Grow_Interactor` directly sets the appropriate slots of the object to cause it to move or change size. For rectangles, circles, groups and most other objects, the default Command object sets the `Am_LEFT`, `Am_TOP`, `Am_WIDTH` and `Am_HEIGHT`. For lines (more specifically, any object whose `Am_AS_LINE` slot is true), the Command object may instead set the `Am_X1`, `Am_Y1`, `Am_X2` and `Am_Y2` slots.

Am_Move_Grow_Interactor:

Slot	Default Value	Type	
Am_START_WHEN	Am_Input_Char ("LEFT_DOWN")	Am_Input_Char	
Am_START_WHERE_TEST	Am_Inter_In	Am_Where_Function	
Am_ABORT_WHEN	Am_Input_Char ("CONTROL_g")	Am_Input_Char	
Am_RUNNING_WHERE_OBJECT	true	Am_Object, bool	
Am_RUNNING_WHERE_TEST	Am_Inter_In	Am_Where_Function	
Am_GROWING	false	bool	
Am_AS_LINE	<formula>	bool	
Am_FEEDBACK_OBJECT	0	Am_Object	*// interim feedback*
Am_GRID_X	0	int	
Am_GRID_Y	0	int	
Am_GRID_ORIGIN_X	0	int	
Am_GRID_ORIGIN_Y	0	int	
Am_GRID_PROC	0	Am_Custom_Gridding_Proc	
Am_WHERE_ATTACH	Am_ATTACH_WHERE_HIT	Am_Move_Grow_Where_Attach	Am_ATTACH_ .. {WHERE_HIT, NW, N, NE, E, SE, S, SW, W, END_1, END_2, CENTER}
Am_MINIMUM_WIDTH	0	int	
Am_MINIMUM_HEIGHT	0	int	
Am_MINIMUM_LENGTH	0	int	
Am_COMMAND	Am_Move_Grow_Command	Am_Command	

5.3.5.3.1 Special Slots of Move_Grow Interactors

- Am_GROWING: If false or zero, then object is moved without changing its size (or for lines, without changing the orientation or length). If true or non-zero, then adjusts the size (or a single end-point for a line). The default is false.
- Am_AS_LINE: If false or zero, then treats the object as a rectangle and adjusts the Am_LEFT, Am_TOP, Am_WIDTH and Am_HEIGHT slots. If true or non-zero, then if the object is being changed size, then sets the Am_X1, Am_Y1, Am_X2 and Am_Y2 slots (lines can be moved by setting their Am_LEFT and Am_TOP slots). The default is a formula that looks at the value of the Am_AS_LINE slot of the object the Interactor is modifying.
- Am_FEEDBACK_OBJECT: If NULL (0) (the default), then the actual object moves around with the mouse. If this slot contains an object, however, then that object is used as an interim-feedback object, and it moves around with the mouse, and the actual object is moved or changed size only when the stop-when event happens (e.g., when the mouse button is released). *Don't forget to add the feedback object to a group or window* ***in addition*** *to adding it as the* Am_FEEDBACK_OBJECT. While the feedback object is moving around, the original object simply stays in its original position.

- Am_WHERE_ATTACH: This slot controls what part of the object is attached to the mouse as the object is manipulated. The options are defined by the enum type Am_Move_Grow_Where_Attach in inter.h. They are:
  - Am_ATTACH_WHERE_HIT: (This is the default.) The mouse is attached where the mouse is pressed down. If growing the object, then checks which edge the mouse is closest to, and grows from there.
  - Am_ATTACH_CENTER: The center of the object. This is illegal if growing the object.
  - Am_ATTACH_NW, Am_ATTACH_N, Am_ATTACH_NE, Am_ATTACH_E, Am_ATTACH_SE, Am_ATTACH_S, Am_ATTACH_SW, Am_ATTACH_W The mouse is attached at this corner or at the center of this side of the object.
  - Am_ATTACH_END_1, Am_ATTACH_END_2: Only available for lines. End_1 is the end defined by Am_X1 and Am_Y1.
- Am_MINIMUM_WIDTH, Am_MINIMUM_HEIGHT : When growing, these are the minimum legal size. Default is 0.
- Am_MINIMUM_LENGTH: Minimum length when growing lines. Default is 0.

5.3.5.3.2 Gridding

There are two ways to do gridding for Am_Move_Grow_Interactors and Am_New_Point_Interactors. The first is to provide a function, and the second is the provide the gridding origin and multiples:

- Am_GRID_PROC: (Default is NULL (0)). If supplied, this should be a function of the type:

```
typedef void Am_Custom_Gridding_Proc(Am_Object inter, int x, int y,
                                     int& out_x, int & out_y);
```

  which will be given the current x and y and should return the new x and y to use. This kind of gridding is also useful for snapping, "gravity," and keeping the object being dragged inside a region.
- Am_GRID_X, Am_GRID_Y: If Am_GRID_PROC is not supplied, then these slots can hold the number of pixels the mouse skips over. Default is 0.
- Am_GRID_ORIGIN_X, Am_GRID_ORIGIN_Y: These can hold the offset in pixels from the edge of the window for the origin of the gridding. Default is 0.

5.3.5.3.3 Standard operation of the Am_Move_Grow_Command

Am_Move_Grow_Command:

Slot	Default Value	Type
Am_START_ACTION	Am_Move_Grow_Command_Start	Am_Object_Proc*
Am_INTERIM_DO_ACTION	Am_Move_Grow_Command_Interim_Do	Am_Object_Proc*
Am_DO_ACTION	Am_Move_Grow_Command_Do	Am_Object_Proc*
Am_UNDO_ACTION	Am_Move_Grow_Command_Undo	Am_Object_Proc*
Am_UNDO_THE_UNDO_ACTION	Am_Move_Grow_Command_Undo_The_Undo	Am_Object_Proc*
Am_ABORT_ACTION	Am_Move_Grow_Command_Abort	Am_Object_Proc*
Am_LABEL	"Move_Grow interactor"	Am_String
Am_ACTIVE	true	bool
Am_OBJECT_MODIFIED	0	Am_Object
Am_INTERIM_VALUE	0	Am_Four_Ints
Am_OLD_VALUE	0	Am_Four_Ints
Am_VALUE	0	Am_Four_Ints

As the Move_Grow_Interactor is operating, it calls the various methods of the Command object stored in its Am_COMMAND slot. The default Command object, Am_Move_Grow_Command, uses the following mechanisms to show the operation. If this is not sufficient for your needs, then you can create a custom Command object, as described in Section 5.5.3.2

If there is a feedback object in the Am_FEEDBACK_OBJECT slot then its size is set to the size of the object being manipulated, and its Am_VISIBLE slot is set to true. Then it is moved or changed size with the mouse. Otherwise, the object itself is manipulated. At any time while the Interactor is running, the abort event can be hit (default is "control-g") to restore the object to its original position. When the stop event happens, then the feedback object is made invisible, and the object is moved or changed size to the final position.

The Undo_Action of the default Command object in the Am_Move_Grow_Interactor simply resets the object to its original size and position.

5.3.5.3.4 Simple Example

See the file testinter.cc for lots of additional examples of uses of Interactors and Command objects. The following Interactor will move any object in the window when the middle button is held down.

```
Am_Object move_inter = Am_Move_Grow_Interactor.Create("move_object")
    .Set (Am_START_WHERE_TEST, (Am_Object_Proc)&Am_Inter_In_Part)
    .Set (Am_START_WHEN, "MIDDLE_DOWN");
window.Add_Part (move_inter);
```

5.3.5.4 Am_New_Points_Interactor

The Am_New_Points_Interactor is used for creating new objects. The programmer can specify how many points are used to define the object (currently, only 1 or 2 points are

supported), and the Interactor lets the user rubber-band out the new points. It is generally required for the programmer to provide a feedback object for a Am_New_Points_Interactor so the user can see where the new object will be. If 1 point is desired, the feedback will still follow the mouse until the stop event, but the final point will be returned, rather than the initial point. Gridding can be used as with a Am_Move_Grow_Interactor. To create the actual new objects, in the default Command object for the Am_New_Points_Interactor, the programmer provides a call-back function in the Am_CREATE_NEW_OBJECT_ACTION slot of the Command object.

Am_New_Points_Interactor:

Slot	Default Value	Type
Am_START_WHEN	Am_Input_Char("LEFT_DOWN")	Am_Input_Char
Am_START_WHERE_TEST	Am_Inter_In	Am_Where_Function
Am_ABORT_WHEN	Am_Input_Char("CONTROL_g")	Am_Input_Char
Am_RUNNING_WHERE_OBJECT	true	Am_Object, bool
Am_RUNNING_WHERE_TEST	Am_Inter_In	Am_Where_Function
Am_AS_LINE	0	bool
Am_FEEDBACK_OBJECT	0	Am_Object
Am_HOW_MANY_POINTS	2	int
Am_FLIP_IF_CHANGE_SIDES	true	bool
Am_ABORT_IF_TOO_SMALL	false	bool
Am_STOP_WHEN	Am_Input_Char("ANY_MOUSE_UP")	Am_Input_Char
Am_GRID_X	0	int
Am_GRID_Y	0	int
Am_GRID_ORIGIN_X	0	int
Am_GRID_ORIGIN_Y	0	int
Am_GRID_PROC	0	Am_Custom_ Gridding_Proc
Am_MINIMUM_WIDTH	0	int
Am_MINIMUM_HEIGHT	0	int
Am_MINIMUM_LENGTH	0	int
Am_COMMAND	Am_New_Points_Command	Am_Command

5.3.5.4.1 Special Slots of New_Points Interactors

- Am_AS_LINE: If true, then create the new object as a line, and set the Am_X1, Am_Y1, etc. slots of the feedback object. If false, the default, then create the new object as a rectangle, and set the Am_LEFT, Am_TOP, etc. of the feedback object.
- Am_FEEDBACK_OBJECT: Object to rubber band to show where the new object will be.
- Am_HOW_MANY_POINTS: How many points are desired. Lines, rectangles, etc. are normally defined by 2 points, which is the default. Currently, the only supported values are 1 and 2.
- Am_MINIMUM_WIDTH, Am_MINIMUM_HEIGHT, Am_MINIMUM_LENGTH: Same as for Am_Move_Grow_Interactor (Section 5.3.5.3.1).
- Am_ABORT_IF_TOO_SMALL: If true, and if the size is less than the minimum, then no object will be created (if the stop event happens while the object is less than the minimum, then the Interactor aborts). If false (the default), then an object is created with the minimum size.

- Am_FLIP_IF_CHANGE_SIDES: If true, then if the cursor goes above and/or to the left of the original point, the object is flipped. If false, then the new object is pegged at its minimum size. This is only relevant if Am_AS_LINE is false.
- Am_GRID_X, Am_GRID_Y, Am_GRID_ORIGIN_X, Am_GRID_ORIGIN_Y, Am_GRID_PROC: Same as for Am_Move_Grow_Interactor (Section 5.3.5.3.2).

5.3.5.4.2 Standard operation of the Am_New_Point_Command

Am_New_Points_Command:

Slot	Default Value	Type
Am_START_ACTION	Am_New_Points_Command_Start	Am_Object_Proc*
Am_INTERIM_DO_ACTION	Am_New_Points_Command_ Interim_Do	Am_Object_Proc*
Am_DO_ACTION	Am_New_Points_Command_Do	Am_Object_Proc*
Am_UNDO_ACTION	Am_New_Points_Command_Undo	Am_Object_Proc*
Am_UNDO_THE_UNDO_ACTION	Am_New_Points_Command_Undo_ The_Undo	Am_Object_Proc*
Am_ABORT_ACTION	Am_New_Points_Command_Abort	Am_Object_Proc*
Am_LABEL	"New_Points interactor"	Am_String
Am_ACTIVE	true	bool
Am_TOO_SMALL	0	bool
Am_INTERIM_VALUE	0	Am_Four_Ints
Am_VALUE	0	any
Am_CREATE_NEW_OBJECT_ ACTION	0	Am_Create_New_ Object_Proc*

As the New Point Interactor is operating, it calls the various methods of the Command object stored in its Am_COMMAND slot. The default Command object, Am_New_Point_Command, uses the following mechanisms to show the operation. If this is not sufficient for your needs, then you need to create a custom Command object, as described in Section 5.5.3.3.

While the Interactor is operating, the appropriate slots of the feedback object are set, as controlled by the parameters described above. If the user hits the abort key while the Interactor is running ("control_g" by default), the feedback object is made invisible and the Interactor aborts. If the user performs the stop_when event (usually by releasing the mouse button), then the Command object calls the procedure in the Am_CREATE_NEW_OBJECT_ACTION slot of the Command object. *Note*: this procedure should be set into the Command object, *not* into the Interactor. (If there is no procedure, then nothing happens). The procedure is called as:

```
typedef Am_Object Am_Create_New_Object_Proc(Am_Object Command_obj,
            int a, int b, int c, int d);
```

The four integers will be the left, top, width and height, or the x1, y1, x2, y2 of the new object to be created depending on the Am_As_Line slot. (Note: this interface may change when we support more than 2 points). After creating the new object and adding it as a part to some group or window, the procedure should return the new object.

The default Undo_Action makes the object invisible (so the default Undo_The_Undo method will make it visible again). When Undo_The_Undo is no longer possible (determined by the type of undo handler in use--section 5.5.2), and the Command object is de-allocated by the undo handler, then the object is automatically destroyed.

5.3.5.5 Am_Text_Edit_Interactor

The Am_Text_Edit_Interactor is used for single-line, single-font editing of the text in Am_Text objects. (Support for multi-line, multi-font text editing will be in a future release.) The default Command object in the Am_Text_Edit_Interactor directly sets the Am_TEXT and Am_CURSOR_INDEX slots of the Am_Text object to reflect the user's changes. Most of the special operations and types used by the Am_Text_Edit_Interactor are defined in text_fns.h.

When the start_when event occurs, the Interactor puts the text object's cursor where the start event occurred, and initializes the Command object. Subsequent events are sent to the Command object to edit the text object. When the stop_when event happens, the cursor is turned off and the command object's DO_ACTION method is called. The stop_when event is *not* entered into the string.

The default Am_RUNNING_WHERE_OBJECT is true, meaning the Interactor will run no matter where the user moves the cursor. If this slot is set to be a particular object, leaving that object causes the Interactor to hide the text object's cursor until the user moves back into the object.

Notice that Am_Text_Edit_Interactor's Am_START_WHERE_TEST slot is set to the value Am_Inter_In_Text. Am_Text_Edit_Interactors only work properly on Am_Text objects, so one of the text tests should be used for the Am_START_WHERE_TEST slot.

Am_Text_Edit_Interactor:

Slot	Default Value	Type
Am_START_WHEN	Am_Input_Char("LEFT_DOWN")	Am_Input_Char
Am_START_WHERE_TEST	Am_Inter_In_Text	Am_Where_Function
Am_ABORT_WHEN	Am_Input_Char("CONTROL_g")	Am_Input_Char
Am_RUNNING_WHERE_OBJECT	true	Am_Object, bool
Am_RUNNING_WHERE_TEST	Am_Inter_In	Am_Where_Function
Am_STOP_WHEN	Am_Input_Char("RETURN")	Am_Input_Char
Am_TEXT_EDIT_FUNCTION	Am_Default_Text_Edit_Function	Am_Text_Edit_Function
Am_EDIT_TRANSLATION_TABLE	Am_Edit_Translation_Table::Default_Table()	Am_Edit_Translation_Table
Am_COMMAND	Am_Edit_Text_Command	Am_Command

5.3.5.5.1 Special Slots of Text Edit Interactors

- Am_TEXT_EDIT_FUNCTION: This is a function of the form:

```
typedef void Am_Text_Edit_Function (Am_Object text,
                                    Am_Input_Char ic,
                                    Am_Object inter)
```

  The text edit function should edit the text object's Am_TEXT field given the input character ic. It can also modify the Am_CURSOR_INDEX slot of the object, but shouldn't change other slots. The default function: Am_Default_Text_Edit_Function, uses the Am_Edit_Translation_Table specified in the Interactor's Am_EDIT_TRANSLATION_TABLE slot to provide the basic editing operations (see the description of the Am_EDIT_TRANSLATION_TABLE slot below). If the input character doesn't match any operation in the translation table, the default edit function does the following:

  - ASCII characters between ' ' (SPACE) and '~' are inserted into the string before the cursor.
  - All other keyboard events make the Interactor beep and do nothing.
  - Non-keyboard events are ignored.

- Am_EDIT_TRANSLATION_TABLE: This is an Am_Edit_Translation_Table (defined in the file text_fns.h), a table that maps input characters to Am_Text_Edit_Operations.

  Am_Edit_Translation_Table::Default_Table() defines the following mappings:

  - CONTROL_h, BACKSPACE, DELETE: delete character before cursor
  - CONTROL_w, CONTROL_BACKSPACE, CONTROL_DELETE: delete word before cursor
  - CONTROL_d: delete character after cursor
  - CONTROL_u: delete the entire string
  - CONTROL_k: delete string from cursor to end of line
  - CONTROL_b, LEFT_ARROW: move cursor one character to the left
  - CONTROL_f, RIGHT_ARROW: move cursor one character to the right
  - CONTROL_a: move cursor to beginning of line
  - CONTROL_e: move cursor to end of line
  - CONTROL_y: insert the contents of the X cut buffer at the cursor position
  - CONTROL_c: copy the current string into the X cut buffer
  - any mouse button inside the string: move the cursor
  - any mouse button outside the string: beep

5.3.5.5.2 Standard operation of the Am_Edit_Text_Command object:

Am_Edit_Text_Command:

Slot	Default Value	Type	
Am_START_ACTION	Am_Text_Command_Start	Am_Object_Proc*	
Am_INTERIM_DO_ACTION	Am_Text_Command_Interim_Do	Am_Object_Proc*	
Am_DO_ACTION	Am_Text_Command_Do	Am_Object_Proc*	
Am_UNDO_ACTION	Am_Text_Command_Undo	Am_Object_Proc*	
Am_UNDO_THE_UNDO_ACTION	Am_Text_Command_Undo_The_Undo	Am_Object_Proc*	
Am_ABORT_ACTION	Am_Text_Command_Abort	Am_Object_Proc*	
Am_LABEL	"text interactor"	Am_String	
Am_ACTIVE	true	bool	
Am_OBJECT_MODIFIED	0	Am_Object	*//object edited*
Am_INTERIM_VALUE	0	Am_Input_Event	*//each event set here*
Am_OLD_VALUE	0	Am_String	*//set to old string*
Am_VALUE	0	Am_String	*//new final string*

The text interactor's start action sets the Command object's initial values. It saves a copy of the text object's original Am_TEXT slot in the Am_OLD_VALUE slot of the text Command, sets the Command's Am_OBJECT_MODIFIED slot to be the object the Interactor is acting on (feedback objects are not supported yet), and it moves the cursor to the location specified by where the Interactor start event occurred. The Command object's start function does nothing.

The text Interactor sends events to the Command object by setting the Am_INTERIM_VALUE slot of the Command object before calling its interim do function. If the event is any mousedown inside the object, the Command object moves the cursor to the position clicked on. It then sends the event to the Am_Text_Edit_Function specified in the Interactor's Am_TEXT_EDIT_FUNCTION slot.

An abort event causes the Command object to restore the original text object's text from its Am_OLD_VALUE slot, and to set the object's Am_CURSOR_INDEX to Am_NO_CURSOR.

When the stop event occurs, the Command object's Am_VALUE slot is set to the new value of the text object's Am_TEXT slot, and the text object's Am_CURSOR_INDEX is set to Am_NO_CURSOR. The stop_when event is *not* entered into the string.

5.4 Advanced Features

5.4.1 Output Slots of Interactors

As they are operating, Interactors set a number of slots in themselves which you can access from the Command's DO procedure, or from constraints that determine slots of the object. The slots set by all Interactors are:

- Am_START_OBJECT - Set with the object returned by the Am_START_WHERE_TEST each time the Interactor starts. This might be useful, for example, if there are two

types of "handles" connected to objects that are to be modified: one for moving and one for growing, distinguished by the value of the IS_A_MOVING_HANDLE slot. Then you might have a formula in the Am_Growing slot of a Am_Move_Grow_Interactor as follows:

```
Am_Define_Formula (bool, grow_or_move) {
   Am_Object start_object;
   start_object = self.GV(Am_START_OBJECT);
   if ((bool)start_object.GV(IS_A_MOVING_HANDLE))
            return false;
   else return true;
}
```

- Am_START_CHAR - The initial Am_Input_Char that started the Interactor. This is most useful when the Am_START_WHEN slot is something like Am_ANY_KEYBOARD, or Am_ANY_MOUSE_DOWN. For example, you might put a constraint in the Am_As_Line slot that depends on which mouse button starts the Interactor. In the following, a line is created when the SHIFT key is held down, otherwise to a rectangle is created:

```
Am_Define_Formula (bool, as_line_if_shift)  {
  Am_Input_Char start_char =
          Am_Input_Char::Narrow(self.GV(Am_START_CHAR));
  if (start_char.shift) return true;
  else return false;
}
```

- Am_WINDOW - The window the Interactor is currently running in. Like graphical objects, this slot shows which window the Interactor is currently attached to. For Interactors that run over multiple windows (see Section 5.4.3), this slot is continuously updated with the current window.
- Am_CURRENT_OBJECT - The current object the Interactor is working on. This will be object returned by Am_START_WHERE_TEST when the Interactor starts running, and then by the Am_RUNNING_WHERE_TEST while the Interactor is running. This is most useful for Am_Choice_Interactors where the object the Interactor is running over changes as the Interactor runs.

The specific Interactors also set special slots in the Command objects which are used by the specific DO procedures. These are described below in Section 5.5.3 about each specific type of Interactor.

5.4.2 Priority Levels

When an input event occurs in a window, Amulet tests the Interactors attached to objects in that window in a particular order. Normally, the correct Interactor is executed. However, there are cases where the programmer needs more control over which Interactors are run, and this section discusses the two slots which control this:

- Am_PRIORITY: The priority of this Interactor. The default is 1.0.

- Am_RUN_ALSO: If true, then let other Interactors run after this one is completed. The default is false.

All the Interactors that can operate on a window are kept in a sorted list. The list is sorted first by the Interactor's priority number, and then by the display order of the graphical object the Interactor is attached to. The result is that for Interactors of the same priority, the one attached to the least covered (front-most) graphical object is handled first.

The priority of the Interactor is stored in the Am_PRIORITY slot and can be any positive or negative number. When an Interactor starts running, the priority level is increased by a fixed amount (defined by Am_INTER_PRIORITY_DIFF which is 100.0 and is defined in inter_advanced.h). This makes sure that Interactors that are running take priority over those that are just waiting. If you want to make sure that your Interactor runs before other default Interactors which may be running, then use a priority higher than 101.0. For example, the debugging Interactor which pops up the inspector (see section 5.6) uses a priority of 300.0.

For Interactors with the same priority, the Interactor attached to the front and leaf most graphical object will take precedence. This is implemented using the slots Am_OWNER_DEPTH and Am_RANK of the graphical objects which are maintained by Opal. What this means is that an Interactor attached to a part has priority over an Interactor attached to the group that the part is in, if they both have the same value in the Am_PRIORITY slot. Note that this determinations does *not* take into account which objects the Interactor actually affects, just what object the Interactor is a part of. Thus, if Interactor A is attached to group G and has a Am_START_WHERE_TEST of Am_Inter_In_Part, and Interactor B is attached to part P which is in G and has a Am_START_WHERE_TEST of Am_Inter_In, then the Interactor on B will take precedence by default, even though both A and B can affect P.

If the Interactor that accepts the event has its Am_RUN_ALSO slot set to true (the default is false), then other Interactors will also get to process the current event. Thus, the run-also Interactor operates without grabbing the input event. This might be useful for Interactors that just want to monitor the activity in a window, say to provide a "tele-pointer" in a multi-user application. In this case, you would want the Interactor with Am_RUN_ALSO set to true to have a high priority.

Furthermore, if an Interactor that has Am_RUN_ALSO slot set to false accepts the current event, the system will continue to search to find if there are any other Interactors with Am_RUN_ALSO slot set to true that have the *same* priority as the Interactor that is running. These are also allowed to process the current event.

5.4.3 Multiple Windows

A single Interactor can handle objects which are in multiple windows. Since an Interactor must be attached to a single window, graphical object or group, a special

mechanism is needed to have an Interactor operate across multiple windows. This is achieved by using the Am_OTHER_WINDOWS slot. The value of this slot can be:

- false or zero: which means that this slot is ignored, and the Interactor only works on the window of the object it is attached to. This is the default.
- true (or any non-zero integer): the Interactor operates on *all* windows created using Amulet.
- a single window: which means that the Interactor operates on this window in addition to the window the Interactor is attached to.
- a Am_Value_List containing a list of windows, which means that the Interactor works on all of these windows, plus the one it is attached to.

Of course, the function in the Am_START_WHERE_TEST slot must search for objects in all of the appropriate windows. It might use the value of the Am_WINDOW slot of the Interactor, which will contain the window of the current event.

5.5 Command Objects

Unlike other toolkits where the widgets call "call-back" procedures, the widgets and Interactors in Amulet allocate *Command Objects* and call their "Do" methods. Whereas so far this is pretty-much equivalent, Command Objects also have slots that handle undoing, enabling and disabling, and help. Command objects must be added as *parts* of the objects they are attached to, so every Interactor and widget has a part named Am_COMMAND which contains an Am_Command object.

The top-level definition of a Command object is:

```
Am_Command = Am_Root_Object.Create ("Am_Command")
   .Set (Am_DO_ACTION, &Am_Command_Do)
   .Set (Am_UNDO_ACTION, &Am_Command_Undo)
   .Set (Am_UNDO_THE_UNDO_ACTION, &Am_Command_Undo_The_Undo)
   .Set (Am_ACTIVE, true)
   .Set (Am_LABEL, "A Command")
   .Set (Am_VALUE, 0)
   .Set (Am_PARENT, 0)
//the next slots are only used for Commands in Interactors
   .Set (Am_INTERIM_VALUE, 0)
   .Set (Am_START_ACTION, &Am_Command_Start)
   .Set (Am_INTERIM_DO_ACTION, &Am_Command_Interim_Do)
   .Set (Am_ABORT_ACTION, &Am_Command_Abort);
```

Most Command objects supply a Am_DO_ACTION procedure which is used to actually execute the Command. It will typically also store some information in the Command object itself (often in the Am_VALUE and Am_OLD_VALUE slots) to be used in case the Command is undone. The Am_UNDO_ACTION procedure is called if the user wants to undo this Command, and usually swaps the object's current values with the stored old values. The Am_UNDO_THE_UNDO_ACTION procedure is used for "redo" which is when the user wants to undo the undo. Usually, it is the same procedure as the Am_UNDO_ACTION. The

Am_ACTIVE slot controls whether the Interactor or widget that owns this Command object should be active or not (see section 5.3.3.3). This works because all widgets and Interactors have a constraint in their active field that looks at the value of the Am_ACTIVE slot of their Command object. Often, the Am_ACTIVE will contain a constraint that depends on some state of the application, such as whether there is an object selected or not. The Am_LABEL slot is used for Command objects which are placed into buttons and menus to show what label should be shown for this Command. The Am_VALUE slot is set with the value for use by the Am_DO_ACTION. You have to look at the documentation for each Interactor or Widget to see what form the data in the Am_VALUE slot is. Command objects which are used in Interactors also support the Am_START_ACTION, Am_INTERIM_DO_ACTION, and Am_ABORT_ACTION methods (see Section 5.5.3).

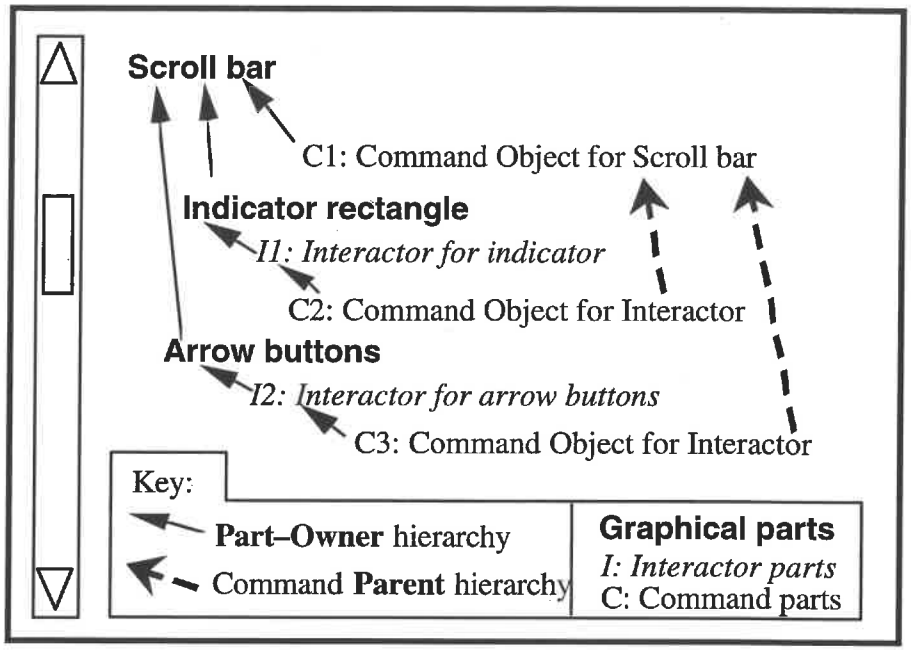
Note that you must set the Am_DO_ACTION and access the Am_VALUE slot of the Command object in the Interactor or widget; one *cannot* access those slots directly in the Interactor or widget themselves. Thus:

```
my_inter.Get_Part(Am_COMMAND).Get(Am_VALUE); //right
my_inter.Get_Part(Am_COMMAND)
   .Set(Am_DO_ACTION, my_do_action); //right
my_inter.Get(Am_VALUE); //WRONG
my_inter.Set(Am_DO_ACTION, my_do_action); //WRONG
```

5.5.1 Parent hierarchy

Normal objects are part of two hierarchies: the prototype-instance hierarchy and the part-owner hierarchy. The Command objects has an *additional* hierarchy defined by the Am_PARENT slot. Based on the Ph.D. research of David Kosbie[1], we allow lower-level Command objects to invoke higher-level Command objects. For example, the Command object attached to a move-grow Interactor which is allowing the user to move a scroll bar indicator calls the Command object attached to the scrollbar itself, as shown in the figure.

[1] David S. Kosbie and Brad A. Myers, "Extending Programming By Demonstration With Hierarchical Event Histories," *The 1994 East-West International Conference on Human-Computer Interaction*. St. Petersburg, Russia, August, 1994. pp. 147-157.



The default Am_DO_ACTION procedure supplied by the Am_Command object automatically calls the Am_DO_ACTION of the Command object in the parent slot. Thus, in the figure, the Command objects in the two Interactors (C2 for I2 and C3 for I3) would have the Command object of the scroll bar (C1) in their Am_PARENT slots, and the DO action of C1 would be called automatically after the DO action of C2 or C3. This will typically be the correct behavior since the DO action of C2 and C3 simply adjust the scroll-bar's indicator, but the DO action of the scroll bar's Command is responsible for updating the scrolling window, or whatever the scroll bar is attached to. The advantage of this design is that the low-level Command objects do not need to know how they are being used, and can just operate normally, and the higher-level Command objects will update whatever is necessary. Note that the parent hierarchy is not usually the same as the part-owner hierarchy. Unfortunately, it seems to be difficult or impossible for Amulet to deduce the parent hierarchy from the part-owner hierarchy, which is why the programmer must explicitly set the Am_PARENT slot when appropriate. Of course, the built-in widgets (like the scroll bar) have the internal Command objects set up appropriately.

You might use the Am_PARENT slot for the Command object in the "OK" button widget inside a dialog box, so the OK widgets's action will automatically call the dialog box's Command object. Another example is that for the button panel widget (see the Widgets chapter), you can have a Command object for each individual item and/or a Command object for the entire panel. If you want the individual item's Command to be called *and* the top-level Command to be called, then you would make the top-level Command be the Am_PARENT of each of the individual item Commands.

5.5.2 Undo

All of the Command objects built into the Interactors and widgets automatically support Undo. This means that the default Am_DO_ACTION procedures store the appropriate information so the default Am_UNDO_ACTION and Am_UNDO_THE_UNDO_ACTION will work correctly. Built-in "undo-handlers" know how to copy the command objects when they are executed, save them in a list of undoable actions, and execute the undo and undo_the_undo actions of the commands. Thus, to have an application support undo is generally a simple process. You need to create an undo-handler object and attach it to a window, and then have some button or menu item in your application call the undo-handler's method for undo and undo_the_undo.

5.5.2.1 Enabling and Disabling Undoing of Individual Commands

Throughout the manual, it has been mentioned that you should call the prototype's do action inside your custom Am_DO_ACTION routines. This is because the top-level Am_Command object's Am_DO_ACTION method automatically handles the queuing of the command object on the undo list, as described below.

If there are operations in the application that are *not* undoable, for example like File Save, then you will typically still call the prototype method from your Am_DO_ACTION routine, but you will have the Am_UNDO_ACTION and Am_UNDO_THE_UNDO_ACTION methods be NULL. This will insure that the menu item for undo will still notify the user that the previous operation is not undoable.

If there are operations that should not go on the undo list at all, for example like scrolling, there are different ways to achieve this. First, if all the operations in a particular window should not be queued (for example, in a pop-up dialog box), you can simply have the Am_UNDO_HANDLER slot of that window be NULL (see below). Second, if this is not the case, you can arrange for the prototype's Am_DO_ACTION to *not* be called by the Am_DO_ACTION in the commands that should not be queued. This done simply not having the Am_Call to the prototype's method in your do action code. The third way to arrange for the command to not be queued is to have the Am_PARENT slot of the command object be non-null. Normally, the parent's do action is called automatically, but it is always OK for a method to be missing. Thus, you could even put an instance of Am_Root_Object as the parent of a command object to make sure it is not queued for undoing.

5.5.2.2 Using the standard Undo Mechanisms

There are two styles of undo supplied by Amulet. These are described in this section. The next section discusses how programmers can provide other undo mechanisms.

Am_Single_Undo_Object:

Slot	Default Value	Type
Am_REGISTER_COMMAND	Am_Single_Undo_Register_Command	Am_Register_Command_Proc
Am_UNDO_THE_UNDO_ALLOWED	<formula>	Am_Object or 0
Am_UNDO_ALLOWED	<formula>	Am_Object or 0
Am_PERFORM_UNDO	Am_Single_Perform_Undo	Am_Object_Proc*
Am_PERFORM_UNDO_THE_UNDO	Am_Single_Perform_Undo_The_Undo	Am_Object_Proc*

Am_Multiple_Undo_Object:

Slot	Default Value	Type
Am_REGISTER_COMMAND	Am_Multiple_Undo_Register_Command	Am_Register_Command_Proc
Am_UNDO_ALLOWED	<formula>	Am_Object or 0
Am_UNDO_THE_UNDO_ALLOWED	<formula>	Am_Object or 0
Am_PERFORM_UNDO	Am_Multiple_Perform_Undo	Am_Object_Proc
Am_PERFORM_UNDO_THE_UNDO	Am_Multiple_Perform_Undo_The_Undo	Am_Object_Proc
Am_LAST_UNDONE_COMMAND	0	Am_Command

The two undo handlers supplied by Amulet are:

- Am_Single_Undo_Object: This handler supports undoing a single Command, like the Macintosh's undo. The last operation can be undone, and the last undone operation can be redone. As soon as another operation is performed, the previous Command is discarded so it can no longer be undone or redone.
- Am_Multiple_Undo_Object: This handler supports undoing an arbitrary number of Commands, all the way back to the first Command. This is implemented by saving all the Commands executed since the application is started, so the list can grow quite long. If a Command is undone, then it can be redone (undo-the-undo), but only the last undone Command is saved. Thus, after undoing a series of Commands, after undoing the last undo (redo), there is nothing available to redo. The Undo itself is not part of the command history.

To use either of these kinds of undo, the programmer simply attaches an instance of the appropriate handler object to the window in the Am_UNDO_HANDLER slot. The standard command DO_ACTION methods look for the Am_UNDO_HANDLER and if found, automatically call the Undo-handler's method to register this command for possible later undoing. You can put the same undo-handler object into the Am_UNDO_HANDLER slot of multiple windows, if you want a single list of undo-actions for multiple windows (for example, for applications which use multiple windows). Creating the undo-handler is simpler than it sounds:

```
my_win = Am_Window.Create("my_win")
  .Set (Am_LEFT, ...)
  ...
  .Set (Am_UNDO_HANDLER, Am_Multiple_Undo_Object.Create("undo"))
```

Now, all the Commands executed by any widgets or Interactors that are part of this window will be automatically registered for undoing.

Next, you need to have a widget that will allow the user to execute the undo and redo. Typically, this will be a widget that should also be active or inactive depending on whether the undo and redo are currently allowed. The undo_handler objects provide the Am_UNDO_ALLOWED slot to tell whether undo is allowed. This slot contains the command object that will be undone (in case you want to have the label of the Undo Command show what will be undone). The Am_UNDO_THE_UNDO_ALLOWED slot of the undo object tells whether redo is allowed and it also will contain a command object or NULL. To actually perform the undo or redo, you call the method in the Am_PERFORM_UNDO or Am_PERFORM_UNDO_THE_UNDO slots of the undo object. For example, the undo button in the demo_space demo is set up as follows (the redo code is similar): (see amulet/samples/space/space.cc for the complete code):

```
// First, define formulas to determine when the Undo command should be active
Am_Define_Formula(bool, undo_active) {
  Am_Object undo_handler, last_command;
  undo_handler = SP_Clip_Win.GV(Am_UNDO_HANDLER);
  last_command = undo_handler.GV(Am_UNDO_ALLOWED);
  //undo is allowed if there is a last command, and if it has an undo method.
  if (last_command.Valid()) {
    Am_Value undo_proc;
    last_command.Get(Am_UNDO_ACTION, undo_proc);
    if (undo_proc.Valid()) return true;
    else return false;
    }
  else return false;
}

// Actually perform the Undo by asking the Undo_handler to do it
void do_undo (Am_Object) {
  Am_Object undo_handler, last_Command;
  undo_handler = SP_Clip_Win.Get(Am_UNDO_HANDLER);
  //don't have to make sure that last_Command is valid here since
  //do_undo can only be called when undo_active formula returns true
  Am_Call(Am_Object_Proc, undo_handler, Am_PERFORM_UNDO,
              (undo_handler));
}
...
// Here is the widget that will allow the user to execute the Undo.  See the Widgets chapter for details
  SP_Button_Panel = Am_Button_Panel.Create()
    .Set (Am_ITEMS, Am_Value_List ()
      .Add (Am_Button_Command.Create("undo_Command")
          .Set(Am_LABEL, "Undo")
          .Set(Am_DO_ACTION, do_undo)
          .Set(Am_ACTIVE, Am_Formula::Create(undo_active))
          )
        ...
```

5.5.2.3 Building your own Undo Mechanisms

Usually, one of the two the supplied Undo objects will do what you want, but Amulet is designed to be easily extensible with new kinds of undo mechanisms. For example, you might want to support arbitrary numbers of redo's or put a limit on the number of Commands that can be undone, or allow the user to pick a specific past Command to explicitly undo. Do implement these, you would make your own undo handler object as an instance of `Am_Undo_Handler` and supply values in the following slots:

- `Am_REGISTER_COMMAND`, a method to be called to register a newly executed Command.
- `Am_PERFORM_UNDO`, a method to be called to execute the undo of the next command to be undone.
- `Am_PERFORM_UNDO_THE_UNDO`, a method to be called to execute the redo of the last undone Command.
- `Am_UNDO_ALLOWED`, supplies a command object or NULL, usually computed by a formula, to say whether undo is currently allowed.
- `Am_UNDO_THE_UNDO_ALLOWED`, supplies a command object or NULL, usually computed by a formula, to say whether redo (undo the undo) is currently allowed.

See the code in `amulet/src/command_basics.cc` to see how these methods and slots are implemented.

5.5.3 Building Custom Command Objects

In many cases, the built in Command objects will be sufficient for your needs. If not, this section will explain how to create your own Command objects. It also explains what you need to know to write your own DO methods for Commands in the built-in Interactors.

Creating Command objects for the built-in widgets is easier than creating new ones for the Interactors, because the widgets only use the `Am_DO_ACTION`, `Am_UNDO_ACTION` and `Am_UNDO_THE_UNDO_ACTION` methods, whereas the Interactors also need the `Am_START_ACTION`, `Am_INTERIM_DO_ACTION`, and `Am_ABORT_ACTION` methods. Furthermore, the Interactors are set up so that the various action methods perform the standard behavior of the Interactor (such as moving the object around) so that if you override one of the default methods, you will be responsible for performing the behavior yourself.

Since all of these methods only take a single parameter which is the Command object, the Interactor or widget stores the required values in the Command object itself before the method is called. Each type of widget or Interactor defines what slots are set in the Command object and with what types of values, and these are described below.

When re-implementing an action routine for a Command, it is very important that you always call the prototype method for the top level Am_Command object, *if you want the Command to be undoable*. The top-level Command object's Do and Undo actions call the appropriate routines to register the command with the appropriate undo-handler object, as discussed above. Typically, these calls will be at the end of the action procedure. For example, you might have the following in a do-action method:

```
void my_do_action (Am_Object command_obj) {
  //first, do my custom stuff
  ...
  //now call the prototype method to set up for Undo
  Am_Call(Am_Object_Proc, Am_Command, Am_DO_ACTION, (command_obj));
}
```

All of the default methods for the Commands in the Interactors contain code to report what is being set for using during tracing (see Section 5.6) which makes the built-in methods look more complicated than yours would have to be.

5.5.3.1 Command Objects for Am_Choice_Interactor and Am_One_Shot_Interactor

The Am_Choice_Interactor and Am_One_Shot_Interactor are implemented using the same command objects and procedures. The important slots of the Command object for these are:

- Am_OLD_INTERIM_VALUE which is set with an object or NULL which is the previously interim selected object.
- Am_INTERIM_VALUE which is set with the newly selected object.
- Am_VALUE which is set with the final result which may be NULL, an object (if only a single object can be selected) or a Am_Value_List of objects.
- Am_OLD_VALUE which is set with the previous value of Am_VALUE for use if the command is undone.

The default Am_Choice_Command_Do method first turns off the interim feedback by setting to false the Am_INTERIM_SELECTED slot of the object in the Am_INTERIM_VALUE slot of the Command object. Next, it uses that object, and the Am_HOW_SET value of the Interactor, to determine how to compute the final selected set. Thus, if you override the Am_INTERIM_DO_ACTION or Am_DO_ACTION of a choice Interactor, you might want to either call the prototype method from the Am_Choice_Command object or set the slots of the objects yourself. For example:

```
void my_choice_do_action (Am_Object command_obj) {
  //first, do my custom stuff
  ...
  //now call the prototype method to do the regular work of the Interactor
  Am_Call(Am_Object_Proc, Am_Choice_Command, Am_DO_ACTION,
                    (command_obj));
}
```

5.5.3.2 Command Objects for Am_Move_Grow_Interactors

The important slots of the Command object for a Move-Grow Interactor are:

- Am_OBJECT_MODIFIED which is the object being moved or changed size.
- Am_FEEDBACK_OBJECT which is set with a copy of that slot from the Interactor, and will either be a feedback object or NULL. If NULL, then the Am_OBJECT_MODIFIED is modified by the interim_do method.
- Am_AS_LINE which says whether to modify the object as a line or as a box. If the Am_AS_LINE slot of the Interactor is true, and if the object is being grown, then this slot will be true, otherwise it will be false (lines are moved the same way as rectangles: by setting their left and top). Thus this slot is not quite a copy of the corresponding Interactor slot.
- Am_INTERIM_VALUE which contains a list of four integers which are the current position and size of the object, stored as a Am_Four_Ints structure. These are either left, top, width and height, or x1, y1, x2, y2 depending on Am_AS_LINE.. The Am_Four_Ints also contains a reference object, so you can call Am_Translate_Coordinates to convert the coordinates in the Am_Four_Ints to whatever is required for your object.
- Am_OLD_VALUE which holds a copy of the old (original) value as an Am_Four_Ints, used in case the Interactor is aborted or later undone.
- Am_VALUE which contains the final Am_Four_Ints used to set the position and size of the object.

The Move-Grow Interactor and the Command's Am_START_ACTION set these up, and the default interim_do and do actions update the objects based on the values in the slots.

5.5.3.3 Command Objects for Am_New_Point_Interactors

The important slots of the Command object for a New_Point Interactor are:

- Am_FEEDBACK_OBJECT which is set with a copy of the Am_FEEDBACK_OBJECT slot from the Interactor, and should be a feedback object. If NULL, then there is no feedback while the Interactor is running.
- Am_AS_LINE which says whether to modify the object as a line or as a box.
- Am_TOO_SMALL which is set by the interactor if the size is currently smaller than the minimum allowed. The default method turns off the feedback if Am_TOO_SMALL is true.
- Am_INTERIM_VALUE which contains a list of four integers which are the current position and size of the feedback object, stored as a Am_Four_Ints structure. These are either left, top, width and height, or x1, y1, x2, y2 depending on Am_AS_LINE. The Am_Four_Ints also contains a reference object, so you can call Am_Translate_Coordinates to convert the coordinates in the Am_Four_Ints to whatever is required for your object.

- Am_CREATE_NEW_OBJECT_ACTION should be set by the programmer with a function to create the new objects.
- Am_VALUE which holds the object which was created as a result of this Command. The default Am_New_Points_Command_Do method calls the method in the Am_CREATE_NEW_OBJECT_ACTION slot which returns the created object, and this object is stored in the Am_VALUE slot for later use by undo.
- Am_HAS_BEEN_UNDONE which is initially false, and is set to true when the creating Command has been undone. That way, when the Command is destroyed (for example, because it can no longer be redone), then it knows that it is OK to destroy the object which was created as a result of this Command.

The default Am_New_Points_Command_Undo method just sets the created object to be invisible, and the Am_New_Points_Command_Undo_The_Undo method makes it visible again.

5.5.3.4 Command Objects for Am_Text_Interactors

The important slots of the Command object for a Text Interactor are:

- Am_OBJECT_MODIFIED - the object being edited, which will be an instance of Am_Text.
- Am_INTERIM_VALUE - set by the Interactor with the current Am_Input_Event.
- Am_OLD_VALUE - the original string for the object, in case the user aborts or calls Undo.
- Am_VALUE which is the new (final) string for the object.

5.6 Debugging

The Interactors and default Commands provide a number of mechanisms to help programmers debug programs. The primary one is a *tracing* mechanism that supports printing to standard output (cout) whenever an "interesting" Interactor or Command event happens. Amulet supplies many options for controlling when printout occurs, as described below. You can either set these parameters in your code and recompile, or they can be dynamically changed as your application is running. If you have a C++ interpreter like ObjectCenter you can set them from there, or else the interactive Inspector (see section 5.6) allows tracing of everything to be turned on and off.

```
typedef enum { Am_INTER_TRACE_NONE, Am_INTER_TRACE_ALL,
         Am_INTER_TRACE_EVENTS, Am_INTER_TRACE_SETTING,
         Am_INTER_TRACE_PRIORITIES, Am_INTER_TRACE_NEXT,
         Am_INTER_TRACE_SHORT } Am_Inter_Trace_Options;

void Am_Set_Inter_Trace(); //prints current status
void Am_Set_Inter_Trace(Am_Inter_Trace_Options trace_code);
void Am_Set_Inter_Trace(Am_Object inter_to_trace);
void Am_Clear_Inter_Trace();
```

By default, tracing is off. Each call to Am_Set_Inter_Trace *adds* tracing of the parameter to the set of things being traced (except for Am_INTER_TRACE_NONE which clears the entire trace set). The options for Am_Set_Inter_Trace are:

- no parameters: If Am_Set_Inter_Trace is called with no parameters, it prints out the current tracing status.
- Am_INTER_TRACE_NONE: If Am_Set_Inter_Trace is called with zero or Am_INTER_TRACE_NONE, then it sets there to be nothing be traced. This is the same as calling Am_Clear_Inter_Trace.
- Am_INTER_TRACE_ALL: Traces everything.
- Am_INTER_TRACE_EVENTS: Only prints out the incoming events, and not what happens as a result of these events. When you trace anything else, Amulet automatically also adds Am_INTER_TRACE_EVENTS to the set of things to trace, so you can tell the event which causes things to be updated.
- Am_INTER_TRACE_SETTING: This very useful option just shows which slots of which objects are being set by Interactors and Commands. It is very useful for determining why an object slot is being set.
- Am_INTER_TRACE_PRIORITIES: This prints out changes to the priority levels.
- Am_INTER_TRACE_NEXT: This turns on tracing of the next Interactor to be executed. This is very useful if you don't know the name of the Interactor to be traced.
- Am_INTER_TRACE_SHORT: This prints out only the name of the Interactors which are run.
- an Interactor: This prints lots of information about the execution of that one Interactor.

5.7 Building Custom Interactor Objects

We believe that in almost all cases, programmers will be able to create their applications by using the pre-defined types of Interactors that are listed above. However, there might be rare cases when an entirely new type of Interactor is required. For example, in Garnet which had a similar Interactor model, *none* of the applications created using Garnet needed to create their own Interactor types. However, when the Garnet group wanted to add support for Gesture recognition, this required writing a new Interactor. Since Amulet is designed to support investigation into new interactive styles and techniques, new kinds of Interactors may be needed to explore new types of interaction beyond the conventional direct manipulation styles supported by the built-in Interactors. In summary, we feel you should only need to create a new kind of Interactor when you are supporting a radically different interaction style.

This section gives an overview of the standard operation of the Interactor mechanism. You may need to look at the source code for one of the built-in Interactors to see how they operate in detail.

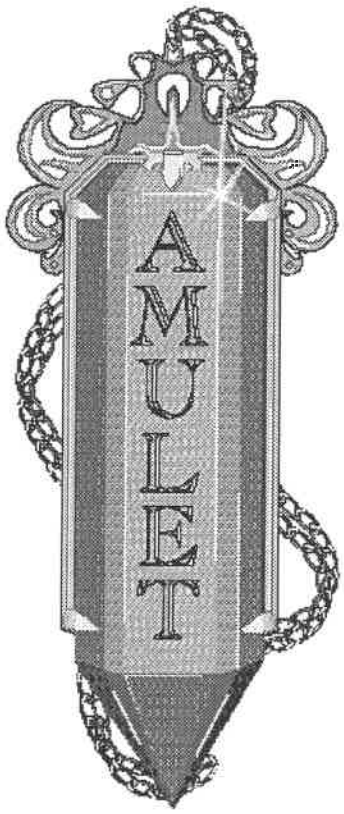
The main event loop in Amulet takes each input event and looks at the sorted list of Interactors with each window, and then asks each Interactor in turn if they want to handle the input event. This is done by sending the Interactor one of the messages listed below, based on the current state of the Interactor, held in the slot Am_CURRENT_STATE (see the state machine figure in Section 5.3.2). The different Interactors are distinguished by having different functions for these messages. All of the default Interactors are built by the Am_Initialize_Interactors function implemented in inter_basics.cc. All of the messages are of type Am_Action_Function (defined in inter_advanced.h). *It is very important that each action method call the top-level prototype's method to maintain the overall Interactor state machine.* For example, the move-grow start action starts off as:

```
void Am_Move_Grow_Start_Action (Am_Object inter, Am_Object object,
                                Am_Object event_window, int x, int y,
                                Am_Input_Char ic) {
  // first, call the prototype's method
  Am_Call(Am_Action_Function, Am_Interactor, Am_START_ACTION,
          (inter, object, event_window, x, y, ic));
...}
```

The specific action methods you need to write for a new type of Interactor are stored in the slots:

- Am_START_ACTION: This is called when the Interactor should change from state Am_INTER_WAITING to Am_INTER_RUNNING. Typically, the method would initialize various fields and then call the Am_START_ACTION of the Command object attached to the Interactor, and then call the Am_INTERIM_DO_ACTION of the Command object. Note that the start action should call the Command object's Am_INTERIM_DO_ACTION on the first input event.
- Am_RUNNING_ACTION: This is called for each incremental mouse movement or keyboard key while the Interactor is executing. Typically, it would set some slots of its Command object and then call the Am_INTERIM_DO_ACTION of the Command object.
- Am_OUTSIDE_ACTION: This is called if the mouse moves outside of the Am_RUNNING_WHERE_OBJECT while the Interactor is running. Typically it will call the Am_ABORT_ACTION of the Command object.
- Am_BACK_INSIDE_ACTION: This is called if the mouse moves back inside the Am_RUNNING_WHERE_OBJECT while the Interactor is running. Typically it will call the Am_START_ACTION followed by the Am_INTERIM_DO_ACTION of the Command object.
- Am_ABORT_ACTION: This is called when the user executes the abort key to cause the Interactor to abort while it is executing. Typically, this will call the Am_ABORT_ACTION of the Command object.
- Am_STOP_ACTION: This is called when the Interactor stops (finishes). Typically it will set some slots in the Command object and then call the Am_DO_ACTION of the Command object.

- `Am_OUTSIDE_STOP_ACTION`: This is called when the user executes the stop event while the Interactor is outside. For all the built-in Interactors, the `Am_OUTSIDE_STOP_ACTION` method is the same as the `Am_ABORT_ACTION` method.



6. Widgets

Abstract

Amulet provides a full set of widgets, including buttons, menus, scroll bars, and text input fields. Eventually, these will display themselves in different looks, corresponding to the various platforms. The built-in widgets have a large number of parameters to allow programmers to customize them, and the programmer can also create new kinds of widgets by writing new methods.

6.1 Introduction

Many user interfaces, spanning a wide variety of applications, have several elements in common. Menus and scroll bars, for example, are used so frequently that an interface designer would waste considerable time and effort recreating those objects each time they were required in an application.

The intent of the Amulet Widget set is to supply several frequently used objects that can be easily customized by the designer. By importing these pre-constructed objects into a larger Amulet interface, the designer is able to specify in detail the desired appearance and behavoir of the interface, while avoiding the programming that this specification would otherwise entail.

This document is a guide to using Amulet's Widgets. The objects were constructed using the complete Amulet system, and their descriptions assume that the reader has some knowledge of the components of this system: Opal, Interactors, and ORE.

6.1.1 Current Widgets

Amulet currently supports the following widgets:

- `Am_Border_Rectangle`: a rectangle with a raised (or lowered) edge, but no interaction.
- `Am_Button`: a single button
- `Am_Button_Panel`: a panel consisting of multiple buttons with the labels inside the buttons.
- `Am_Checkbox_Panel`: a panel of toggle checkboxes with the labels next to the checkboxes.
- `Am_Radio_Button_Panel`: a panel of mutually exclusively selectable radio buttons with the labels next to the radio buttons.
- `Am_Menu`: a menu panel
- `Am_Menu_Bar`: a menu bar used to select from several different menu panels
- `Am_Vertical_Scroll_Bar`: scroll bar for choosing a value from a range of values.
- `Am_Horizontal_Scroll_Bar`: scroll bar for choosing a value from a range of values.
- `Am_Scrolling_Group`: an Amulet group with (optional) vertical and horizontal scrollbars
- `Am_Text_Input_Widget`: a field to accept text input, like for a filename.

These widgets are described in this chapter in detail, and summarized in chapter 8.

6.1.2 Customization

We have tried to make the widgets flexible enough to meet any need. Each widget has a large number of slots which control various properties of its appearance and behavior, which you can set to customize the look and feel. The designer may choose to leave many of the default values unchanged, while modifying only those parameters that integrate the object into the larger user interface.

The visual appearance and the functionality of a widget is affected by values set in its slots. When instances of widgets are created, the instances inherit all of the slots and slot values from the prototype object. The designer can then change the values of these slots to customize the widget. Instances of the custom widget will inherit the customized values. The slot values in a widget prototype can be considered "default" values for the instances.

6.1.3 Using Widget Objects

Include files necessary to use Amulet widgets are `widgets.h` for the widget object definitions, and `standard_slots.h` for the widget slot definitions. These files are included in `amulet.h`, providing a simple way to make sure all needed files are included. Programmers who are designing their own custom widget objects will also need `widgets_advanced.h`.

Widgets are standard Amulet objects, and are created and modified in the same way as any other Amulet object. The following sample code creates an instance of `Am_Button`, and changes the values of a few of its slots.

```
Am_Object my_button = Am_Button.Create("My Button")
  .Set (Am_LEFT, 10)  // change the position of the button
  .Set (Am_TOP, 10)
  .Set (Am_COMMAND, "Push Me");
                      // a string in the Am_COMMAND slot specifies the button's label: see below
```

6.1.4 Application Interface

Like interactors, widgets interface to application code through *command objects* added as parts of the widgets. Thus, instead of executing a call-back procedure as in other toolkits, Amulet widgets call the `Am_DO_ACTION` method of the command object stored as the `Am_COMMAND` part of the widget.

In addition to the `Am_DO_ACTION` method, each command also contains the other typical slots of command objects (see the discussion of command objects in the Interactor's chapter). In particular, the `Am_VALUE` slot of the command object is normally set with the result of the widget. Of course, the type of this value is dependent on the type of the widget. For scroll bars, the value will be a number, and for a radio-button-panel, it will be a list of the selected items.

For all of the widgets, the Am_VALUE slot of the command objet can also be *set* to adjust the value displayed by the widget. Setting the Am_VALUE slot of the command object of a scrollbar will cause the indicator to move. It is also appropriate to put a constraint into the Am_VALUE slot if you want the value shown by the widget to track a slot of another object.

NOTE: Be sure to set the Am_VALUE and Am_DO_ACTION slots of the command object part of the widget, and *do not* set these slots directly in the widget, since this will have no effect. Thus:

```
my_widget.Get_Part(Am_COMMAND).Get(Am_VALUE);  // Correct
my_widget.Get_Part(Am_COMMAND).Set(Am_VALUE, 10);  // Correct
my_widget.Get_Part(Am_COMMAND).Set(Am_DO_ACTION, my_proc);  // Correct
my_widget.Get(Am_VALUE);  // WRONG
my_widget.Set(Am_VALUE, 10);  // WRONG
my_widget.Set(Am_DO_ACTION, my_proc);  // WRONG
```

In some situations, the programmer might want to have a constraint dependent on the Am_VALUE slot. This constraint can perform side effects like updating an external data base or even setting slots in Amulet objects or creating or destroying new objects. Other times, the programmer will need to write an Am_DO_ACTION method which will typically access the value in the command's Am_VALUE slot. An example of each of these methods can be found below. Of course, if you write your own Am_DO_ACTION method and you want the widget to be undo-able, you will also need to write a corresponding Am_UNDO_ACTION method, etc. The default Am_DO_ACTION just registers the command as undoable, if appropriate. If you write your own do action, you will usually want to call the prototype's do method as the final line of your code, or else the command will not be registered for Undo. For example, if you create a new command object for some kind of button, the final line of the Am_DO_ACTION might be:

```
Am_Call(Am_Object_Proc, Am_Command, Am_DO_ACTION, (command_obj));
```

All of the widgets are designed so that you can completely replace the DO method of the command and the widget will still operate correctly, but you still need to call the prototype method to register the command for undoing. However, the UNDO methods built into the commands for the various kinds of widgets take care of resetting the widget to show the new value. For example, the undo method of the radio button panel makes sure that the radio buttons show the previous selection. If you write a DO method that does something with the radio button's value, you would then want to write an UNDO method that undoes that action. If when the user executes UNDO, you want the radio buttons to go back to their previous state, you should be sure to call the prototype's undo method. There are different kinds of command objects for the different kinds of widgets, so you need to be sure to call the correct one. All the buttons, menus and panels use Am_Button_Command, the scroll bars use the Am_Scroll_Command, and the text input widget uses Am_Text_Input_Command. Thus, if you were setting the Am_UNDO_ACTION and Am_UNDO_THE_UNDO_ACTION methods of a scroll bar, the last line of your procedure should be something like:

```
Am_Call(Am_Object_Proc, Am_Scroll_Command, Am_UNDO_ACTION, (command_obj));
```

Similarly, for a button panel, you would use Am_Button_Command.

Note that the Am_Scrolling_Group does not support undoing the scrolling, but the individual scroll bars do.

Internally, each widget is implemented using graphical objects and interactors. Each internal interactor has its own associated command objects, but these are normally irrelevant to the programmer, since the internal command objects will call the top-level widget command object automatically. This is achieved by setting the Am_PARENT slot of the internal command objects to be the widget's command object, and then Amulet automatically does the right thing.

6.2 The Standard Widget Objects

Each of the objects in the Widget Set is an interface mechanism through which the designer obtains chosen values from the user. Buttons, panels, and menus allow the selection of one or more items from a list of possible alternatives. The scrolling group and scrollbars are used to obtain values in a specific range, between maximum and minimum allowed values.

This section describes the widgets in detail. Each object contains customizable slots, but the designer may choose to ignore many of them in any given application. Any slots not explicitly set by the application are inherited from the widget's prototype.

6.2.1 Slots Common to All Widgets

There are several slots the programmer can set which are used by all widgets in a similar way:

- Am_TOP, Am_LEFT: As with all graphical objects, these slots describe the location of the widget, in coordinates relative to the object's parent's location. Default values are 0 for both top and left.
- Am_VISIBLE: If this boolean is true, the object is visible; otherwise, it is not drawn on the screen. Default is true.
- Am_WIDGET_LOOK: The value of this slot tells Amulet how you want your widgets to look when drawn on the screen. Possible values are Am_MOTIF_LOOK, Am_WINDOWS_LOOK, or Am_MACINTOSH_LOOK. Any look is available on any platform, and Am_MOTIF_LOOK is the default. Currently only Am_MOTIF_LOOK is implemented.
- Am_FILL_STYLE: This slot determines the color of the widget. Amulet automatically figures out appropriate foreground, background, shadow, and highlight colors given a fill color. Acceptable values are any Am_Style, and the

default is Am_Amulet_Purple. The only part of the style used is the color of the style. On a black and white screen, a default set of stipples are used to make sure the widgets are visible.

- Am_ACTIVE_2: This slot turns off interaction with the widget without turning it grey. This is mainly aimed at interactive tools like Interface Builders that want to allow users to select and move widgets around. It might also be useful in a multi-user situation where users who do not have the "floor" should not have their widgets responding. For a widget to operate, both Am_ACTIVE_2 and Am_ACTIVE must be true. The default value is true.

The command objects in all widgets have the following standard slots:

- Am_VALUE: This slot is set to the current value of the widget.
- Am_DO_ACTION: The method to be called when the widget executes. This procedure takes one parameter: the command object. The default for all widgets is only to register the widget with the undo handler.
- Am_UNDO_ACTION: This method is called when the widget's actions are to be undone. The default procedure here resets the Am_VALUE slot of the widget to its previous value, and updates the widget's appearance appropriately.
- Am_UNDO_THE_UNDO_ACTION: This method is called when the widget's actions are to be re-done. The default action simply resets the Am_VALUE slot and updates the widget's appearance appropriately. The built-in widgets all use the same procedure for Am_UNDO_ACTION and Am_UNDO_THE_UNDO_ACTION.
- Am_ACTIVE: This slot in the command is used to determine whether the widget is enabled or not (greyed out). Often, this slot will contain a formula dependent on some system state. The default value is true. (Actually, the widget itself also contains an Am_ACTIVE slot, but this one should not normally be used. The widget-level slot contains a constraint that depends on the Am_ACTIVE slot of the command object part of the widget.)
- Am_PARENT: This slot should be set with the command object which the widget's command should invoke. For example, if the widget is the "OK" button of a dialog box, the Am_PARENT of the OK widget's command might be the command object for the entire dialog box. Then Amulet will correctly know how to handle Undo, and it will call the parent command automatically.

6.2.2 Border_Rectangle

Am_Border_Rectangle: *(a Motif-like rectangle with border)*

Slot	Default Value	Type
Am_SELECTED	false	bool
Am_WIDGET_LOOK	Am_MOTIF_LOOK	Am_Widget_Look
Am_WIDTH	50	int
Am_HEIGHT	50	int
Am_TOP	0	int
Am_LEFT	0	int
Am_VISIBLE	true	bool
Am_FILL_STYLE	Am_Amulet_Purple	Am_Style

The Am_Border_Rectangle has a raised or lowered edge of a lighter or darker shade of the Am_FILL_STYLE. It ignores the Am_LINE_STYLE. It looks pressed in if Am_SELECTED is true, and sticking out of the screen if Am_SELECTED is false. This widget has no interaction or response to the mouse.

6.2.3 Buttons and Menus

All of the buttons and menus operate fairly similarly. The label that is displayed in the widget is determined by the Am_LABEL slot of the Am_COMMAND part of the widget. Thus, for a single button, the Am_COMMAND part's label will contain the string to display.

The various panel objects (that display a set of buttons) and the menus (that display a set of buttons) all take an Am_ITEMS slot which must contain an Am_VALUE_LIST. The items in this value list can be:

- a C string (char*), in which case this string is displayed as the label,
- a graphical object, in which case this object (or an instance of this object if the object is already a part of another object) is displayed as the label. This object can of course be a group, so arbitrary pictures can be displayed as the value of a widget.
- a command object, in which case the value of the Am_LABEL field of the command object is used as the item's label. The Am_LABEL field itself can contain either a C string or a graphical object.

There are two basic ways to use the panel-type objects, including menus:

1) Each individual item has its own command object, and the Am_DO_ACTION of this command does the important work of the item. This would typically be how menus of operations like Cut, Copy, and Paste would be implemented.
2) The top-level panel itself has a command object and the individual items do *not* have a command object. For example, the Am_ITEMS slot of the widget contains an Am_VALUE_LIST of strings. In this case, the top-level command object will be called, and it typically will look in its Am_VALUE slot to determine which item was

selected. This method is most appropriate when the panel or menu is a list of values, like colors or fonts, and you do not want to create a command for each item.

Note that the top-level command object is *not* called if the individual item has a command object, unless you explicitly set the Am_PARENT of the item's command to be the widget's command. It would be unusual, but is perfectly legal, to have a Am_Value_List that contains some commands and some strings.

6.2.3.1 Am_Button_Command

Am_Button_Command:

Slot	Default Value	Type
Am_DO_ACTION	Am_Command_Do	Am_Object_Proc*
Am_UNDO_ACTION	Am_Button_Command_Undo	Am_Object_Proc*
Am_UNDO_THE_UNDO_ACTION	Am_Button_Command_Undo	Am_Object_Proc*
Am_LABEL	"Label"	Am_String or any graphical object
Am_ID	0	any
Am_ACTIVE	true	bool
Am_VALUE	0	any

Special slots of the command object associated with buttons are as follows. If you are replacing the command of a button, you will typically want to make an instance of a Am_Button_Command, rather than, say, of a top level Am_Command.

- Am_LABEL: This slot can contain a string or a graphical object, which will be drawn as the label for this item.
- Am_ID: Normally, buttons set the Am_VALUE slot to the Am_LABEL of the command. However, this typically requires doing string matching. Therefore, if the Am_ID field is non-zero, then the Am_VALUE slot is set with the value of the Am_ID slot instead of the Am_LABEL slot.
- Am_VALUE: This slot contains the label(s) or ID(s) of currently the selected button(s). In a single button, this contains 0 if the button is not selected, or the button's label or ID if it is selected. If multiple items can be selected, as in a check box panel or for a button panel if you set Am_HOW_SET to be Am_CHOICE_LIST_TOGGLE, then this slot will always contain an Am_Value_List with the labels or IDs of the selected items. If no items are selected, then the list will be empty. Note: the value of the Am_VALUE slot will not be 0; it will be a list that is empty. For panels where only a single item can be selected, such as a radio button panel or button panels with Am_HOW_SET set to be Am_CHOICE_SET, the Am_VALUE slot is set to the single button's label or ID, or 0 if nothing is selected. If you set the Am_VALUE slot, the widget will update its appearance on the screen. appropriately.
- Am_ACTIVE: This controls whether the widget is active or not (greyed out). If the

Am_ACTIVE slot of the top-level command in a panel or menu is set to false, then all the items are greyed out. More typically, the Am_ACTIVE slot of the command associated with a single item will be false, signaling that just that one item should be greyed out.

6.2.3.2 Am_Menu_Line_Command

Am_Menu_Line_Command is a special purpose type of command object provided by Amulet to draw horizontal dividing lines in menus. To add a horizontal line in a menu, simply include an instance of Am_Menu_Line_Command in the menu's Am_ITEMS list. An example of this can be found in section 6.2.3.7. Am_Menu_Line_Command has no customizable slots, and it is an inactive menu item.

Am_Menu_Line_Command:

Slot	Default Value	Type
Am_DO_ACTION	NULL	Am_Object_Proc*
Am_UNDO_ACTION	NULL	Am_Object_Proc*
Am_UNDO_THE_UNDO_ACTION	NULL	Am_Object_Proc*
Am_LABEL	"Menu_Line_Command"	Am_String
Am_ACTIVE	false	bool
Am_VALUE	NULL	any

6.2.3.3 Am_Button

The Am_Button object is a single stand-alone button. A button can have a text label, or can contain an arbitrary graphical object when drawn.

Am_Button:

Slot	Default Value	Type
Am_LEFT	0	int
Am_TOP	0	int
Am_WIDTH	<formula>	int
Am_HEIGHT	<formula>	int
Am_Y_OFFSET	0	int
Am_H_SPACING	0	int
Am_V_SPACING	0	int
Am_H_ALIGN	Am_CENTER_ALIGN	{Am_LEFT_ALIGN, Am_RIGHT_ALIGN, Am_CENTER_ALIGN}
Am_V_ALIGN	Am_CENTER_ALIGN	{Am_TOP_ALIGN, Am_BOTTOM_ALIGN, Am_CENTER_ALIGN}
Am_FIXED_WIDTH	Am_NOT_FIXED_SIZE	int
Am_FIXED_HEIGHT	Am_NOT_FIXED_SIZE	int
Am_INDENT	0	int
Am_MAX_RANK	false	bool
Am_MAX_SIZE	false	bool
Am_ITEM_OFFSET	5	int
Am_ACTIVE	<formula>	bool
Am_ACTIVE_2	true	bool
Am_WIDGET_LOOK	Am_MOTIF_LOOK	Am_Widget_Look
Am_KEY_SELECTED	false	bool
Am_FONT	Am_Default_Font	Am_Font
Am_FINAL_FEEDBACK_WANTED	false	bool
Am_FILL_STYLE	Am_Amulet_Purple	Am_Style
Am_COMMAND	Am_Button_Command	Am_Command

Special slots of Am_BUTTONS are:

- Am_WIDTH, Am_HEIGHT: By default, the width and height of the button are automatically calculated by formulas in these slots. A button is made big enough to contain its text label or graphical object, including borders, and offset pixels (see below). A user can replace the width and height formulas by setting these slots directly. Once the values are set with new values or formulas, the formulas will be removed.
- Am_ITEM_OFFSET: The string or object displayed inside the button is set away from the border of the button by Am_ITEM_OFFSET pixels, in both the horizontal and vertical directions. The default is 5.
- Am_FONT: The button's text label (if any) is drawn in this font. Acceptable values are any Am_Font, and the default is Am_Default_Font.
- Am_FINAL_FEEDBACK_WANTED: This determines if the button should be drawn as if

it is still selected, even after user interaction has stopped. This is useful if you want to use the button to show whether it is selected or not. The default is `false`.

- `Am_COMMAND`: This contains an instance of `Am_Button_Command`, as described in section 6.2.3.1.

6.2.3.4 Am_Button_Panel

An `Am_Button_Panel` is a panel of `Am_Buttons`, with a single interactor in charge of all the buttons. Since an `Am_Button_Panel's` prototype object is a `Am_Map`, all the slots that `Am_Map` uses are also used by `Am_Button_Panel`. See the Opal chapter for a description of `Am_Map`. Some `Am_Map` slots are described below along with slots specific to `Am_Button_Panel`.

Am_Button_Panel:

Slot	Default Value	Type
Am_LEFT	0	int
Am_TOP	0	int
Am_WIDTH	<formula>	int
Am_HEIGHT	<formula>	int
Am_HOW_SET	Am_CHOICE_SET	Am_How_Set
Am_ITEM_OFFSET	5	int
Am_ACTIVE	<formula>	bool
Am_ACTIVE_2	true	bool
Am_WIDGET_LOOK	Am_MOTIF_LOOK	Am_Widget_Look
Am_KEY_SELECTED	false	bool
Am_FONT	Am_Default_Font	Am_Font
Am_FILL_STYLE	Am_Amulet_Purple	Am_Style
Am_FINAL_ FEEDBACK_WANTED	false	bool
Am_WIDTH	Am_Width_Of_Parts	int
Am_HEIGHT	Am_Width_Of_Parts	int
Am_LAYOUT	Am_Vertical_Layout	{Am_Vertical_Layout, Am_Horizontal_Layout NULL}
Am_H_ALIGN	Am_LEFT_ALIGN	{Am_LEFT_ALIGN, Am_RIGHT_ALIGN, Am_CENTER_ALIGN}
Am_ITEMS	<special>	int, Am_Value_List of commands or strings, etc.

6.2.3.4.1 Slot Descriptions

- `Am_FONT`: The font used for the button labels.
- `Am_FINAL_FEEDBACK_WANTED`: Whether to show which item is selected or not. The default is `false`.

- Am_WIDTH, Am_HEIGHT: The width and height slots contain the standard Amulet formulas Am_Width_Of_Parts and Am_Height_Of_Parts, respectively. If these slots are set to specific values by the designer, those values replace the formulas, and the panel will no longer resize itself if its contents change.
- Am_FIXED_WIDTH: This slot determines how the buttons in a panel get their width. An integer value of 0, or a boolean value of false, means each button is as wide as its calculated width based on the contents. Thus, each button will be a different size. An integer value of 1, or a boolean value of true, means that all the buttons in the panel are set to be as wide as the calculated width of the widest button in the panel. An integer value greater than 1 sets the width of all buttons in the panel to that specific value. The default is true.
- Am_FIXED_HEIGHT: This slot determines the height of buttons in a panel. It acts the same way as Am_FIXED_WIDTH. The default is false.
- Am_HOW_SET: This slot determines whether single or multiple buttons can be selected. Its default value is Am_CHOICE_SET, which allows a single selection. Changing this to Am_CHOICE_TOGGLE will allow the selected item to be turned off by clicking it again. Am_CHOICE_LIST_TOGGLE allows there to be multiple selections. See the Interactors manual for a complete description of the legal values.
- Am_LAYOUT: This specifies a function determining how the button panel should be arranged. A more complete description of the slot can be found in section 4.7.2. Am_Vertical_Layout is the default, and Am_Horizontal_Layout is another good value.
- Am_H_ALIGN: In a vertically arranged button panel with variable width buttons, this determines how the buttons should be arranged in the panel. The default is Am_LEFT_ALIGN, and other possible values are Am_CENTER_ALIGN and Am_RIGHT_ALIGN.
- Am_V_ALIGN: This slot works like Am_H_ALIGN, except is only used in horizontally arranged panels width variable height buttons. Possible values are Am_TOP_ALIGN, Am_CENTER_ALIGN, and Am_BOTTOM_ALIGN.
- Am_ITEMS: This slot specifies the items which are to be put in the button panel. An Am_Value_List should be used to specify specific items to add to the panel. See section 6.2.3.1 for a complete description. In summary, elements of the value list can be either strings, graphical objects, or command objects. A string value is used as the label for the button in the panel. A graphical object is displayed in the button. A command object is used to specify a custom command for that particular button in the panel. For commands, the button's string label or graphical object is taken from the command object's Am_LABEL slot.
- Am_COMMAND: This slot contains an Am_Button_Command. See section 6.2.3.1 for a complete description.

6.2.3.4.2 Example of Using a Button Panel

Each button in the panel is drawn with a text label or a graphical object inside it. An Am_Value_List in the Am_ITEMS slot tells the button panel what to put inside each button. If a string is specified, it is used as the button's label. If a graphical object is specified, it is drawn in the button. If a command object is specified, that command object's Am_DO_ACTION method is called each time the button is pressed, and the button's label or graphical object is obtained from the command object's Am_LABEL slot. The following code specifies a button panel with three buttons in it.

```
// a graphical object and custom do action, defined elsewhere:
extern Am_Object My_Graphical_Object;
extern void My_Custom_Do_Action (Am_Object command_obj);
Am_Object my_command;
// my button panel:
Am_Object My_Button_Panel = Am_Button_Panel.Create ("My Button Panel")
  .Set (Am_ITEMS,
        Am_Value_List ()
          .Add ("Push me.")
          .Add (My_Graphical_Object)
          .Add (my_command = Am_Button_Command.Create()
                  .Set (Am_LABEL, "Push me too.")
                  .Set (Am_DO_ACTION, &My_Custom_Do_Action)));
```

The first button in the panel is drawn with the text label "Push me." and does not have its own command object. The second button in the panel is drawn containing My_Graphical_Object drawn inside it, and also does not have its own command. The third button in the panel is drawn with the text label "Push me too." and has its own command object associated with it.

When the third button is pressed, My_Custom_Do_Action is called, with the button's command object (my_command) as an argument. The command object's Am_VALUE slot will already have been set with either 0, if the button was not selected, or "Push me too." if the button was selected. We assume that this command is not undoable since there is no custom Undo action to go with My_Custom_Do_Action.

If any of the other buttons in the panel are pressed, the do action of My_Button_Panel's command object (in its Am_COMMAND slot) will be called, with the command object as an argument. The Am_VALUE of the command object is set with the labels or objects corresponding to the currently selected buttons.

If you wanted the button panel's command to be invoked when the third button was pressed, you would have to set the third button's command object's Am_PARENT slot to point to the button panel's command object. For example, after executing the following code, My_Other_Custom_Do_Action in the panel will be called when *any* of the buttons are selected.

```
Am_Object panel_command = My_Button_Panel.Get(Am_COMMAND);
panel_command.Set (Am_DO_ACTION, &My_Other_Custom_Do_Action);
My_Command.Set (Am_PARENT, panel_command);
```

You also need to make `My_Custom_Do_Action` call the do action of `Am_Command` to make sure its parent command is called. Custom do actions should always call `Am_Command`'s DO action, unless you know that you don't want to be able to undo the command, and you don't want to call the command's parent. The following code would be the last line of `My_Custom_Do_Action`:

```
Am_Call (Am_Object_Proc, Am_Command, Am_DO_ACTION, (command_obj));
```

where `command_obj` is the command object that your custom do action was passed as an argument.

6.2.3.5 Am_Radio_Button_Panel

`Am_Radio_Button_Panel:` `Am_Button_Panel`

Slot	Default Value	Type
all the slots of the button panel		
`Am_HOW_SET`	`Am_CHOICE_SET`	`Am_How_Set`
`Am_BOX_WIDTH`	`15`	`int`
`Am_BOX_HEIGHT`	`15`	`int`
`Am_FIXED_WIDTH`	`false`	`int, bool`
`Am_FINAL_FEEDBACK_WANTED`	`true`	`bool`
`Am_H_ALIGN`	`<formula>`	`{Am_LEFT_ALIGN, Am_RIGHT_ALIGN, Am_CENTER_ALIGN}`

A radio button panel is a set of small buttons with items appearing either to the right or left of each button. Exactly one button from the set should be selected at any particular time, and the button stays selected after the user stops interacting with it. The radio button panel is often used to present a user with several different options, only one of which can be in effect at any particular time.

An `Am_Radio_Button_Panel` is essentially the same as an `Am_Button_Panel`, with a few exceptions. There are a few new slots, and some of the defaults of the other slots are different. All other slots not listed below act the same way as in an `Am_Button_Panel`. Since radio buttons always only allow a single selection, the `Am_VALUE` slot of the top-level `Am_COMMAND` is always set with either 0 or the ID or label of the selected item.

- `Am_BOX_HEIGHT, Am_BOX_WIDTH`: These specify the size in pixels of the small radio button box that is drawn next to the item in the button. The defaults are 15 for each.
- `Am_BOX_ON_LEFT`: This boolean determines whether the radio box should be drawn to the left of the item, or to the right. If `true`, the box is drawn on the left, and if `false`, it is drawn on the right. The default is `true`.
- `Am_H_ALIGN`: This slot contains a formula which evaluates to `Am_LEFT_ALIGN` if the `Am_BOX_ON_LEFT` is `true`, or `Am_RIGHT_ALIGN` if it is `false`.
- `Am_FIXED_WIDTH`: The default is `false` for this slot in a radio button panel.
- `Am_FINAL_FEEDBACK_WANTED`: The default is `true` for this slot in a radio button

panel. This makes sure the user selected button stays selected after interaction is complete.

- Am_HOW_SET: Since radio buttons are only allowed to have a single selection, this slot defaults to Am_CHOICE_SET. However, if you want to allow the user to turn off the current selection by clicking on it again, you can set this slot to be Am_CHOICE_TOGGLE. It would be wrong to use Am_CHOICE_LIST_TOGGLE.

6.2.3.6 Am_Checkbox_Panel

Am_Checkbox_Panel: Am_Button_Panel

Slot	Default Value	Type
all the slots of the button panel		
Am_HOW_SET	Am_CHOICE_SET	Am_How_Set
Am_BOX_WIDTH	15	int
Am_BOX_HEIGHT	15	int
Am_FIXED_WIDTH	false	int, bool
Am_FINAL_FEEDBACK_WANTED	true	bool
Am_H_ALIGN	<formula>	{Am_LEFT_ALIGN, Am_RIGHT_ALIGN, Am_CENTER_ALIGN}

A checkbox panel is a set of small buttons with items appearing either to the right or left of each button. Zero or more buttons from the set can be selected at any particular time, and the buttons stay selected after the user stops interacting with the panel. The checkbox panel is often used to present a user with several different options that can be in effect at the same time.

An Am_Checkbox_Panel is essentially the same as an Am_Radio_Button_Panel. It is drawn slightly differently, and the following slot is different:

- Am_HOW_SET: The default for a checkbox panel is Am_CHOICE_LIST_TOGGLE, which allows multiple items to be selected at the same time.

Since multiple items can be selected in an Am_Checkbox_Panel, the Am_VALUE slot of the top-level Am_COMMAND contains an Am_VALUE_LIST of the labels or IDs of the selected items.

6.2.3.7 Am_Menu

An Am_Menu is a single menu panel, implemented as another form of Am_Button_Panel. A menu panel has a background rectangle behind it, and the items are drawn differently than in Am_Buttons.

Am_Menu: Am_Button_Panel

Slot	Default Value	Type
all the slots of the button panel		
Am_HOW_SET	Am_CHOICE_SET	Am_How_Set
Am_FINAL_FEEDBACK_WANTED	false	bool
Am_WIDTH	<formula>	int
Am_HEIGHT	<formula>	int
Am_X_OFFSET	2	int
Am_Y_OFFSET	2	int
Am_V_SPACING	-2	int

6.2.3.7.1 Am_Menu Slots

The following slots of an Am_Menu differ from those in an Am_Button_Panel:

- Am_WIDTH, Am_HEIGHT: The default formulas in these slots calculate the width and height of the menu's items, and add enough width and height to contain the menu's outer border.
- Am_HOW_SET: The default for a menu is Am_CHOICE_SET.
- Am_X_OFFSET, Am_Y_OFFSET: These slots cause the menu items to be offset from the upper left corner of the menu. The defaults are 2 for X and Y, to make room for the outer border around the menu.
- Am_V_SPACING: This creates extra space between the items in a menu. The default value is -2, which pushes the menu items vertically closer together in the menu.
- Am_TEXT_OFFSET: This offset is used only in the horizontal direction when a text label is being displayed in the menu item (as opposed to a graphical object). This allows greater horizontal spacing for text, while keeping the standard vertical spacing. The default value is 2 pixels.

6.2.3.7.2 Simple Example

Here is an example of creating an Am_Menu object.

```
Am_Object my_menu = Am_Menu.Create("my_menu")
   .Set (Am_LEFT, 150)
   .Set (Am_TOP, 200)
   .Set (Am_ITEMS, Am_Value_List()
            .Add ("Menu item")
            .Add (Am_Menu_Line_Command.Create("my menu line"))
            .Add (Am_Button_Command.Create("item2")
                   .Set(Am_ACTIVE, false)
                   .Set(Am_LABEL, "Not active"))
            .Add (Am_Button_Command.Create("item2")
                   .Set(Am_LABEL, ("Active item"))));
my_window.Add_Part(my_menu);
```

The menu has three menu items with a line between the first and second items. The first item appears as "Menu item" in the menu, and has no corresponding command object.

If that item is selected by the user, the do action of my_menu's command object will be called with "Menu item" in its Am_VALUE slot. Since there is no command object associated with the first menu item, there is no way to make it inactive without making the whole menu inactive.

The second menu item appears in the menu as "Not active". It will be grayed out, because the Am_ACTIVE slot of its corresponding button command object is set to false. This item cannot be chosen from the menu because it is inactive.

The third menu item appears in the menu as "Active item". It does have a command object associated with it, so if it is selected by the user, that command's do action will be executed, and the widget's top level command will not be executed. The widget's top level command object is not called unless you set the individual button command object's Am_PARENT slot to point to it. Since no do action is explicitly set, it defaults to Am_Command's do action, which just registers the command for undo, if appropriate.

6.2.3.8 Am_Menu_Bar

Am_Menu_Bar: Am_Menu

Slot	Default Value	Type
Am_LEFT	0	int
Am_TOP	0	int
Am_WIDTH	<formula>	int
Am_HEIGHT	<formula>	int
Am_ACTIVE	<formula>	bool
Am_ACTIVE_2	true	bool
Am_WIDGET_LOOK	Am_MOTIF_LOOK	Am_Widget_Look
Am_FONT	Am_Default_Font	Am_Font
Am_FILL_STYLE	Am_Amulet_Purple	Am_Style
Am_ITEMS	NULL	Am_Value_List
Am_COMMAND	Am_Button_Command	Am_Command

The Am_Menu_Bar is a menubar like you might find at the top of a window that has a horizontal row of items you can select, and each one pops down a menu of further options. Sometimes it is called a pull-down menu. Amulet's menu bar currently supports a single level of sub-menus (no pull-outs from the pull-downs), and it does not yet support accelerators (keyboard shortcuts to the commands). However, any menu item (either at the top level or a sublevel) can be an abitrary Amulet object, just like with other button-type objects.

The interface to menu bars is similar to other button widgets: the Am_ITEMS slot of the menu_bar object should contain an Am_Value_List. However, unlike other objects, the list *must* contain command objects. The label field of this command object serves as the top-level menubar item. In the command object should be an Am_ITEMS slot containing an Am_Value_List of the sub-menu items. This list can contain command objects, strings or Amulet objects, as with other menus and button panels. For example:

```
my_menu_bar = Am_Menu_Bar.Create()
  .Set(Am_ITEMS, Am_Value_List ()
        .Add (Am_Button_Command.Create("File_Command")
              .Set(Am_LABEL, "File")
              .Set(Am_DO_ACTION, &my_file_do)
              .Set(Am_ITEMS, Am_Value_List ()
                    .Add ("Open...")
                    .Add ("Save As...")
                    .Add (Am_Button_Command.Create("Quit_Command")
                          .Set(Am_DO_ACTION, &my_quit)
                          .Set(Am_LABEL, "Quit"))
                    )
              )
        .Add (Am_Button_Command.Create("Edit_Command")
              .Set(Am_LABEL, "Edit")
              .Set(Am_DO_ACTION, &my_edit_do)
              .Set(Am_ITEMS, Am_Value_List ()
                    .Add (undo_command.Create())
                    .Add ("Cut")
                    .Add ("Copy")
                    .Add ("Paste")
                    .Add (Am_Menu_Line_Command.Create("my menu line"))
                    .Add ("Find...")
                    )
              )
        )
```

If a sub-menu item has a command (like Quit or Undo above), then its Am_DO_ACTION is called when the item is chosen by the user. If it does *not* have a command object (like Cut and Paste above), then the command object of the main item is called (here, the do action called my_edit_do in the command object named Edit Command will be called for Cut and Paste, and the Am_VALUE slot of the Edit Command will be set to the string of the particular item selected). Note that because the first level value list must contain command objects, the command object stored in the menu_bar object itself will never be used unless the programmer explicitly sets the Am_PARENT slot of a command to the menu_bar's command object. The Am_VALUE of whatever command object is executed will be set to the label or ID of the selected item.

Am_Menu_Bars allow the top level item to be chosen (unlike, say the Macintosh), in which case its command object is called with its own label or ID as the Am_VALUE. The programmer should ignore this selection if, as usually is the case, pressing and releasing on a top-level item should do nothing.

Individual items can be made unselectable by simply setting the Am_ACTIVE field of the command object for that item to false. If the Am_ACTIVE field of a top-level command object is false, then the entire sub-menu is greyed out, although it will still pop up so the user can see what's in it.

Unlike regular menus and panels, the Am_Menu_Bar will not show the selected value after user lets up with the mouse. That is, you cannot have Am_FINAL_FEEDBACK_WANTED as true.

Slots that behave different for the Am_Menu_Bar are:

- Am_WIDTH, Am_HEIGHT: By default, these slots contain formulas that make the menubar be the width of its owner (usually the width of the window) and the height of the current font. However, you can override these defaults with constant values or other formulas.

6.2.4 Scroll Bars

`Am_Vertical_Scroll_Bar` and `Am_Horizontal_Scroll_Bar` are widget objects that allow the selection of a single value from a specified range of values, either as a `int` or a `float` (see section 6.2.4.1). You specify a minimum and maximum legal value, and the scroll bar allows the user to pick any number in between. The user can click on the indicator and drag it to set the value. As the indicator is dragged, the value is updated continuously. If the user clicks on the arrows, in the scroll bar, the scroll bar increments or decrements the current value by `Am_SMALL_INCREMENT`. If the user clicks above or below the scroll bar, the value jumps by `Am_LARGE_INCREMENT`. Unfortunately, auto-repeat (repeatedly incrementing while the mouse button is held down) is not implemented yet. You can also adjust the indicator's size to show what percent of the entire contents is visible.

Like all other widgets, the `Am_Vertical_Scroll_Bar` and `Am_Horizontal_Scroll_Bar` store the value in the `Am_VALUE` slot of the object in the `Am_COMMAND` slot. **Remember: *not* in the `Am_VALUE` slot of the scroll bar itself.** As the value is changed by the user, the `Am_DO_ACTION` is also continuously called. The `Am_VALUE` slot can also be set by a program to adjust the position of the scroll bar indicator.

The `Am_Scrolling_Group` provides a convenient interface for a scrollable area. It operates similarly to a regular `Am_Group` (see the Opal manual), except that it optionally displays two scroll bars which the user can use to see different parts.

6.2.4.1 Integers versus Floats

There are four slots that control the operation of the scroll bars: `Am_VALUE_1`, `Am_VALUE_2`, `Am_SMALL_INCREMENT`, and `Am_LARGE_INCREMENT`. If *all* of these slots hold values of type integer, then the result stored into the `Am_VALUE` slot will also be an integer. If any of these values is a float, however, then the result will be a float. The default values are 0, 100, 1 and 10, so the default result is an integer. Note that the inspector and `cout` display floats without decimals as integers, but the scroll bar still treats them as floats.

6.2.4.2 Am_Scroll_Bar_Command

Am_Scroll_Command:

Slot	Default Value	Type
Am_DO_ACTION	Am_Command_Do	Am_Object_Proc*
Am_UNDO_ACTION	Am_Scroll_Command_Undo	Am_Object_Proc*
Am_UNDO_THE_UNDO_ACTION	Am_Scroll_Command_Undo	Am_Object_Proc*
Am_LABEL	"Scrollbar"	Am_String
Am_ACTIVE	true	bool
Am_VALUE	50	int

The Am_Scroll_Bar_Command works similarly to other widget commands. The main difference is in the Am_VALUE slot.

- Am_ACTIVE: This determines whether the scroll bar is active or not. Inactive scroll bars do not respond to user input. However, in the default Motif look and feel, they are *not* drawn any differently.
- Am_VALUE: This holds the currently selected value on the scroll bar. As discussed above, it will either be an integer or float value.

The scroll bar command supplies an undo method which resets the Am_VALUE slot and the displayed value to the previous value. However, most applications do not allow scrolling operations to be undone, in which case, you should make sure that the scrolling command is not queued on the undo list (see the section on Undo in the Interactors manual).

6.2.4.3 Horizontal and vertical scroll bars

Am_Vertical_Scroll_Bar:

Slot	Default Value	Type	
Am_LEFT	0	int	
Am_TOP	0	int	
Am_WIDTH	20	int	
Am_HEIGHT	200	int	
Am_WIDGET_LOOK	Am_MOTIF_LOOK	Am_Widget_Look	
Am_FILL_STYLE	Am_Amulet_Purple	Am_Style	
Am_VALUE_1	0	int or float	*// Value at top*
Am_VALUE_2	100	int or float	*// Value at bottom*
Am_SMALL_INCREMENT	1	int or float	*// When click arrow*
Am_LARGE_INCREMENT	10	int or float	*// When click "page"*
Am_PERCENT_VISIBLE	0.2	float	*// Size of indicator*
Am_COMMAND	Am_Scroll_Command	Am_Command	

Am_Horizontal_Scroll_Bar:

Slot	Default Value	Type
Am_LEFT	0	int
Am_TOP	0	int
Am_WIDTH	200	int
Am_HEIGHT	20	int
Am_WIDGET_LOOK	Am_MOTIF_LOOK	Am_Widget_Look
Am_FILL_STYLE	Am_Amulet_Purple	Am_Style
Am_VALUE_1	0	int or float
Am_VALUE_2	100	int or float
Am_SMALL_INCREMENT	1	int or float
Am_LARGE_INCREMENT	10	int or float
Am_PERCENT_VISIBLE	0.2	float
Am_COMMAND	Am_Scroll_Command	Am_Command

Here are the customizable slots of a scroll bar:

- Am_WIDTH: This determines the width of the scroll bar. This includes the height of the arrows at the ends of horizontal scroll bars. The default is 20 for vertical bars, and 200 for horizontal bars.
- Am_HEIGHT: This determines the height of the scroll bar. This includes the height of the arrows at the ends of vertical scroll bars. The default is 200 for vertical bars, and 20 for horizontal bars.
- Am_VALUE_1: This is the value selected in the scroll bar when the indicator is at the top (for vertical scroll bars) or left (for horizontal) end of the scroll bar. The default type is an int, but it can also be a float. The default is 0. Note that Am_VALUE_1 is *not* required to be less than Am_VALUE_2, in case you want the bigger value to be at the top or left.
- Am_VALUE_2: This is the value selected in the scroll bar when the indicator is at the bottom (for vertical scroll bars) or right (for horizontal) end of the scroll bar. The defauls type is an int, but it can also hold a float. The default is 100. Note that Am_VALUE_2 is *not* required to be bigger than Am_VALUE_1.
- Am_SMALL_INCREMENT: This is the amount the value is changed when the user clicks on the arrows at the end of the scroll bar. The default value is 1 of type int. The slot can contain either an int or a float.
- Am_LARGE_INCREMENT: This is the amount the value is changed when the user clicks on the scroll area on either side of scroll handle. The default value is 10 of type int. The slot can contain either an int or a float.
- Am_PERCENT_VISIBLE: This slot specifies how large the indicator will be with respect to the region it is dragged back and forth in. The slot should hold a float between 0.0 and 1.0, and the default is 0.2. If this value determines a thumb smaller than 6 pixels long, a 6 pixel thumb is drawn instead.
- Am_COMMAND: Each scroll bar contains an instance of Am_Scroll_Command. See the section 6.2.4.2 for descriptions of its slots.

6.2.4.4 Am_Scrolling_Group

Am_Scrolling_Group: *(used like a group with scroll bars)*

Slot	Default Value	Type
Am_LEFT	0	int
Am_TOP	0	int
Am_WIDTH	0	int
Am_HEIGHT	0	int
Am_X_OFFSET	0	int
Am_Y_OFFSET	0	int
Am_WIDGET_LOOK	Am_MOTIF_LOOK	Am_Widget_Look
Am_FILL_STYLE	Am_Amulet_Purple	Am_Style
Am_INNER_FILL_STYLE	0	Am_Style or 0
Am_H_SCROLL_BAR	true	bool
Am_V_SCROLL_BAR	true	bool
Am_H_SCROLL_BAR_ON_TOP	false	bool
Am_V_SCROLL_BAR_ON_LEFT	false	bool
Am_H_SMALL_INCREMENT	10	int
Am_H_LARGE_INCREMENT	<formula>	int
Am_V_SMALL_INCREMENT	10	int
Am_V_LARGE_INCREMENT	<formula>	int
Am_INNER_WIDTH	400	int
Am_INNER_HEIGHT	400	int

An Amulet scrolling group is useful when you want to display something bigger than will fit into a window and allow the user to scroll around to see the contents. You can use the Am_Scrolling_Group just like a regular group, but the user will be able to scroll around using the optional vertical and horizontal scrollbars.

A scrolling group has two distinct rectangular regions. One is the region that is drawn on the screen, and contains scroll bars, and a rectangle with a visible portion of the group. This region is defined by the Am_LEFT, Am_TOP, Am_WIDTH and Am_HEIGHT of the Am_Scrolling_Group itself. The other region is called the inner region which is the size of all the objects, some of which might not be visible. This area is controlled by the Am_INNER_WIDTH and Am_INNER_HEIGHT slots.

By default, the Am_ACTIVE slots of the scroll bars are calculated based on whether the scroll bars are needed (whether any of the group is hidden in that direction). The percent-visible is also calculated based on the amount of the group that is visible. The Am_H_LARGE_INCREMENT and Am_V_LARGE_INCREMENT are also calculated based on the screen size.

6.2.4.4.1 Members of a Am_Scrolling_Group

You can add and remove members to a scrolling group using the regular Add_Part and Remove_Part methods (be sure to adjust the inner size of the group if the new members change it--you can arrange for this to happen automatically by putting an appropriate constraint into the Am_INNER_WIDTH and Am_INNER_HEIGHT slots, such as Am_Width_Of_Parts and Am_Height_Of_Parts.). However, when enumerating the parts of a Am_Scrolling_Group, do *not* use a Am_Part_Iterator, since this will also

list the scroll bars. Instead, use the Am_Value_List stored in the Am_GRAPHICAL_PARTS slot of the group, which will only contain the objects you added. The Am_GRAPHICAL_PARTS slot can also be used for normal groups (instances of Am_Group and Am_Map), so you can write code that will operate on either scrolling groups or regular groups.

6.2.4.4.2 Am_Scrolling_Group Slots

- Am_WIDTH, Am_HEIGHT: These default to 150 in a scrolling group. The width and height determine the size of the group's graphical appearance on the screen, including space for scroll bars.
- Am_X_OFFSET,
- Am_Y_OFFSET: These are the coordinates of the visual region, in relation to the origin of the inner region. The slots always contain a nonnegative integer, with 0 corresponding to no offset (meaning that the scrollable region's top and left are the same at the top, left of the visible area). The default is 0 for both X and Y offset. You may also Get or Set these slots, and the slots can even contain formulas. Getting the slot gives you the current scrollbar position, and setting the slot changes the current scrollbar position and scrolls the area.
- Am_FILL_STYLE: The filling style (color) used to draw the scroll bars (and the background of the window if Am_INNER_FILL_STYLE is 0).
- Am_INNER_FILL_STYLE: This determines what the background fill of the group will be. If it is an Am_Style, that style is used. If it contains 0, the Am_FILL_STYLE slot is used.
- Am_H_SCROLL_BAR,
- Am_V_SCROLL_BAR: These booleans determine whether the group will have vertical and/or horizontal scroll bars. These slots default to true.
- Am_H_SCROLL_BAR_ON_TOP, Am_V_SCROLL_BAR_ON_LEFT: These booleans determine which side of the group the scroll bars appear on. The defaults are false for both, which puts the horizontal scroll bar at the bottom of the group, and the vertical scroll bar at the right of the group as on most standard windows.
- Am_H_SMALL_INCREMENT, Am_V_SMALL_INCREMENT: This is the small increment in pixels of the horizontal and vertical scroll bars. The value determines how much the scrolling group is moved when the user clicks on the scroll arrows. The default is 10 pixels.
- Am_H_LARGE_INCREMENT, Am_V_LARGE_INCREMENT: This is the large increment, in pixels, of the horizontal and vertical scroll bars. The value determines how much the scrolling group is moved when the user clicks on the scroll areas beside the scroll indicators. The default is calculated by a formula to jump one visible screen full.

- Am_INNER_WIDTH, Am_INNER_HEIGHT: This is the size of the entire group, not just the visible portion. The defaults are 400 for both. This will usually be calculated by a formula based on the contents of the scrolling group (e.g., Am_Width_Of_Parts and Am_Height_Of_Parts). It is OK if these are smaller than the scrolling group's Am_WIDTH and Am_HEIGHT: this just means that the entire area is visible, and so the appropriate scroll bars will be disabled.

6.2.4.4.3 Using a Scrolling Group

To use an Am_Scrolling_Group, simply create an instance of it, customize the Am_TOP, Am_LEFT, Am_WIDTH, and Am_HEIGHT slots of the group to define its size, set the Am_INNER_WIDTH and Am_INNER_HEIGHT based on the contents, and add graphical parts to the group.

6.2.4.4.4 Simple Example

Here is a simple example of using a scrolling group.

```
my_scrolling_group = Am_Scrolling_Group.Create("scroll_group")
    .Set(Am_LEFT, 10)
    .Set(Am_TOP, 10)
    .Set(Am_WIDTH, 200)
    .Set(Am_HEIGHT, 300))
    .Add_Part(Am_Rectangle.Create()
        .Set(Am_LEFT, 0)
        .Set(Am_TOP, 0)
        .Set(Am_WIDTH, 15)
        .Set(Am_HEIGHT, 15)
        .Set(Am_FILL_STYLE, Am_Blue)
        );
my_window.Add_Part(my_scrolling_group);
```

This creates a scrolling group with an area of 200 by 300 pixels, and an internal region 400 by 400 pixels (the default values are inherited since none were specified). The scrolling group is displayed at location 10,10 in my_window. It contains a single object, a blue square 15 pixels on a side, in the upper left corner of the inner region. The scrolling group will have a vertical scroll bar on the right side of the group, and a horizontal scroll bar on the bottom of the group, as specified by the defaults.

6.2.5 Am_Text_Input_Widget

Am_Text_Input_Widget:

Slot	Default Value	Type
Am_LEFT	0	int
Am_TOP	0	int
Am_WIDTH	150	int
Am_HEIGHT	<formula>	int
Am_WIDGET_LOOK	Am_MOTIF_LOOK	Am_Widget_Look
Am_FONT	Am_Default_Font	Am_Font
Am_LABEL_FONT	bold_font	Am_Font
Am_FILL_STYLE	Am_Amulet_Purple	Am_Style
Am_ACTIVE_2	true	bool
Am_COMMAND	Am_Text_Input_Command	Am_Command

The Am_Text_Input_Widget is used to get text from the user, for example for filenames. The widget has an optional label to the left of a text type-in field. The label is the value of the Am_LABEL field of the command object (and can be a string or arbitrary Amulet graphical object). The user can click the mouse button in the field, and then type in a new value. The Am_VALUE of the command in the Am_COMMAND slot is set to the new string, and the command's Am_DO_ACTION is called. The command's default Am_UNDO_ACTION restores the Am_VALUE and the displayed string to its previous value (section 6.2.5.1 describes the Am_Text_Input_Command). As the user types, if the string gets too long to be displayed, it scrolls left and right as appropriate so the cursor is always visible. Currently, the user *must* end the typing by hitting return or ^G (to abort). Eventually, clicking outside the box will also end the editing, but currently this beeps.

The special slots of the Am_Text_Input_Widget are:

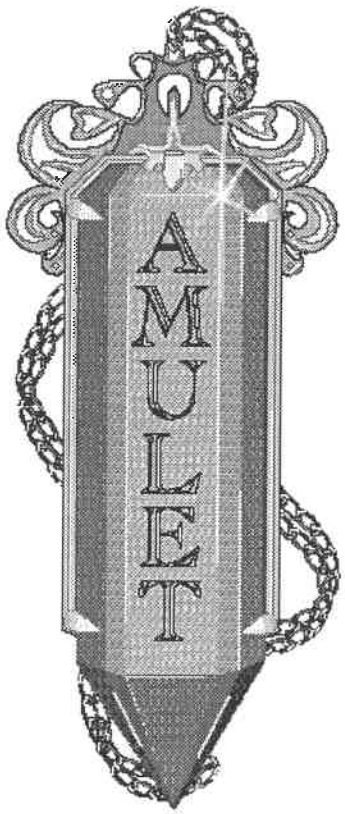
- Am_WIDTH - unlike most widgets, the default width is a constant (150) since the widget scrolls the text to fit. You will probably want to set the width to some other constant or formula.
- Am_HEIGHT - the default formula for the height uses the maximum of the height of the label and the height of the string.
- Am_FONT - this slot holds the font of the string that the user edits, and the default is the regular default font.
- Am_LABEL_FONT - this slot holds the font used for the label if the label is a string (the label comes from the Am_LABEL slot of the command object in the widget). The default is a bold font.
- Am_FILL_STYLE - the color used for the user type-in field.

6.2.5.1 Am_Text_Input_Command

Am_Text_Input_Command:

Slot	Default Value	Type
Am_DO_ACTION	Am_Command_Do	Am_Object_Proc*
Am_UNDO_ACTION	Am_Text_Input_Command_Undo	Am_Object_Proc*
Am_UNDO_THE_UNDO_ACTION	Am_Text_Input_Command_Undo	Am_Object_Proc*
Am_LABEL	"Label"	Am_String
Am_ACTIVE	true	bool
Am_VALUE	""	Am_String

The command object in the Am_Text_Input_Widget will usually be an instance of a Am_Text_Input_Command. The Am_LABEL of the command is used as the label of the input field, so if you do not want a label, make the slot be the null string "". The Am_VALUE of the widget is set with the value the user types, and the Am_DO_ACTION is called. The default Am_UNDO_ACTION resets the displayed string and the Am_VALUE to their previous values, so if you have your own UNDO action, you should also call the Am_Text_Input_Command's to reset the display.



7. Gem--Low-level Graphics Layer

Abstract

This manual describes "GEM", the low-level graphics system Amulet. Gem provides a machine-independent layer so the rest of Amulet can work on different window managers without changing the code. Most programmers will not use the Gem layer, but it is available for advanced programmers who need especially efficient code.

This research was sponsored by NCCOSC under Contract No. N66001-94-C-6037, Arpa Order No. B326. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NCCOSC or the U.S. Government.

7.1 Introduction

Gem is the low-level graphics and input interface in Amulet that provides a window-manager-independent layer so the rest of Amulet is independent of the platform. We expect that most Amulet programmers will not use the Gem layer since the Opal and Interactors layer provide all the functionality of the Gem layer. In fact, Opal, Interactors and the widgets are written using Gem. We provide the Gem interface for expert programmers who are particularly worried about efficiency.

Gem, which stands for the "Graphics and Events Manager", can be used independent of most of the rest of the system. Gem uses the wrapper mechanism (for styles and fonts), but not Amulet objects.

7.2 Include Files

The primary include file for Gem is `gem.h`. Gem also uses `types.h` for wrappers, `gdefs.h` for styles and fonts, and `idefs.h` for the input events.

7.3 Drawonables

The primary data structure in gem is the `Am_Drawonable`, which is a C++ object that corresponds to a window-manager window or off-screen buffer (for example in X/11 it corresponds to a "drawable"). We called it a "Drawonable" because it is something that you can draw on. We also wanted to reserve the word "Window" for the Opal level object that corresponds to the drawonable. In this manual, sometimes "window" is used for "drawonable" since drawonables are implemented as window-manager windows.

7.3.1 Creating Drawonables

Programmers create a "root" drawonable as the initialization, and then create drawonables as children of the root (or as a child of another drawonable). The typical initialization is:

```
Am_Drawonable *root = Am_Drawonable::Get_Root_Drawonable();
```

At the Opal level, this is called automatically by `Am_Initialize` to set up the exported `Am_Screen` object. For X/11, `Get_Root_Drawonable` takes an optional string parameter which is the name of the screen. You can call therefore call `Get_Root_Drawonable` multiple times to support multiple screens.

Creating subsequent drawonables uses the `Create` method. If you use `root.Create` you get a top-level window, and if you use another drawonable, then it creates a sub-window. All of the parameters of the create call are optional, and are:

- `int l = 0`: the left of the new window in the coordinates of its parent.

- `int t = 0`: the top of the window
- `unsigned int w = 100`: width of the window
- `unsigned int h = 100`: height
- `const char* tit = ""`: the title for the window
- `const char* icon_tit = ""`: the string to display with the icon for the window.
- `bool vis = true`: whether the window is initially visible on the screen on not.
- `bool initially_iconified = false`: whether the window starts out as an icon.
- `Am_Style back_color = Am_No_Style`: the initial color for the background of the window.
- `unsigned int border_w = 2`: the size of the border of the window. This is ignored by most window managers for the top-level windows.
- `bool save_under_flag = false`: save the bitmaps underneath the window (useful for pop-up menus).
- `int min_w = 1`: The minimum size allowed for this window (when the user or program resizes it). You can't have 0 size windows.
- `int min_h = 1`: Minimum height.
- `int max_w = 0`: The maximum width allowed for the window. 0 is illegal so means no maximum.
- `int max_h = 0`: Maximum height.
- `bool title_bar_flag = true`: Whether the title line is displayed or not. Under X/11 having no title line means the window is not managed by the window manager.
- `bool query_user_for_position = false`: If true, then the initial left and top are ignored and the user is queried instead.
- `bool query_user_for_size = false`: If true, then the initial width and height are ignored and the user is queried instead.
- `bool clip_by_children_flag = true`: If false, then graphics drawn on the window show through all children windows (drawonables) created as children of this window.
- `Am_Input_Event_Handlers *evh = NULL) = 0`: How input is handled for this window, see Section 7.5.1.

To create an off-screen drawonable, you can use the shorter form:

```
virtual Am_Drawonable* Create_Offscreen (
  int width = 0, int height = 0,
  Am_Style background_color = Am_No_Style) = 0;
```

7.3.2 Modifying and Querying Drawonables

There are a number of methods on drawonables that query and set the various properties:

- `Am_Drawonable* Am_Drawonable::Narrow (Am_Ptr ptr)`: given an arbitrary pointer, this casts it to be a `Am_Drawonable`. Because drawonables are not wrappers, no checking is done.
- `void Destroy ()`: Destroys the drawonable and all its children (including removing them from the screen).
- `void Reparent (Am_Drawonable *new_parent)`: Change a drawonable to have a different parent (owner).
- `bool Inquire_Window_Borders(int& left_border, int& top_border, int& right_border, int& bottom_border, int& outer_left, int& outer_top)`: return the current window border sizes. For X/11, this may be inaccurate for windows that are not yet visible.
- `void Raise_Window (Am_Drawonable *target_d)`: Move the window to the "top" of all its siblings. If `target_d` is NULL, then so it is not covered by any other windows, or if supplied, then on top of `target_d`.
- `void Lower_Window (Am_Drawonable *target_d)`: Move the window to the "bottom" of all its siblings (if `target_d` is NULL), or just until it is below `target_d`.
- `void Set_Iconify (bool iconified)`: Make the window be iconified or not iconified.
- `void Set_Title (const char* new_title)`: Change window title.
- `void Set_Icon_Title (const char* new_title)`: Change icon title.
- `void Set_Position (int new_left, int new_top)`: Move the drawonable.
- `void Set_Size (unsigned int new_width, unsigned int new_height)`: Change the size.
- `void Set_Max_Size (unsigned int new_width, unsigned int new_height)`: Change the maximum size.
- `void Set_Min_Size (unsigned int new_width, unsigned int new_height)` : Change the minimum size.
- `void Set_Visible (bool vis)`: Make the window visible or not.
- `void Set_Border(bool new_title_bar, unsigned int new_width)`: Set whether has a title bar and how thick the border is.
- `void Set_Background_Color (Am_Style new_color)`
- `bool Get_Iconify ()`
- `const char* Get_Title ()`
- `const char* Get_Icon_Title ()`

- `void Get_Position (int& l, int& t)`: Sets l and t with current left and top.
- `void Get_Size (int& w, int& h)`
- `void Get_Max_Size (int& w, int& h)`
- `void Get_Min_Size (int& w, int& h)`
- `bool Get_Visible ()`
- `void Get_Border (bool& title_bar_flag, unsigned int& width)`
- `Am_Style& Get_Background_Color ()`
- `int Get_Depth ()`: Returns the current pixel depth in bits (e.g. 8 for 8-bit color).
- `bool Is_Color ()`: Returns true if the window is color or `false` if not.
- `void Get_Values (int& l, int& t, int& w, int& h, const char*& tit, const char*& icon_tit, bool& vis, bool& iconified_now, Am_Style& back_color, unsigned int& border_w, bool& save_under_flag, int& min_w, int& min_h, int& max_w, int& max_h, bool& title_bar_flag, bool& clip_by_children_flag, int& bit_depth)`: returns all of the parameters at once.

7.4 Drawing objects

The normal operation to draw on drawonables is that you first set a clipping region, then you call one or more drawing operations, and then you flush the output. The flush is necessary on X/11 to make the graphics appear.

7.4.1 General drawing operations

- `void Beep()`: Cause a sound on the machine attached to this drawonable (under X/11, you might have multiple machines controlled by one process, so this method will beep on the machine that the drawonable is connected to. This is called by the Opal-level `Am_Beep()` routine.
- `void Set_Cursor(Am_Image_Array image, Am_Image_Array mask, Am_Style fg_color, Am_Style bg_color)`: set the cursor for the drawonable.
- `virtual void Bitblt (int d_left, int d_top, int width, int height, Am_Drawonable* source, int s_left, int s_top, Am_Draw_Function df = Am_DRAW_COPY)`: Bitblt is the standard rectangular area copy routine. The destination for bitblt is the `Am_Drawonable` this message is sent to.
- `void Clear_Area (int left, int top, int width, int height)`: Set the area to the drawonable's background color.
- `void Flush_Output()`: Cause all pending output to be actually displayed on the screen. Under X/11, you usually will not see any graphics until this is called.
- `void Translate_Coordinates (int src_x, int src_y, Am_Drawonable *src_d, int& dest_x_return, int& dest_y_return)`: Convert the

coordinates in one window to be coordinates in another window. To translate from screen coordinates, pass your root drawonable as src_d.

- `void Translate_From_Virtual_Source (int src_x, int src_y, bool title_bar, int border_width, int& dest_x_return, int& dest_y_return)`: Translates a point from drawonable that hasn't neccessarily been created to screen coordinates. This function only works on root drawonables.

7.4.2 Image arrays and fonts

Under X/11, you cannot get the size of image arrays and fonts without having a window, since they can take on different bit sizes depending on the particular screen resolution. Therefore, the following are messages to drawonables rather than on image array and font objects.

- `void Get_Image_Size (Am_Image_Array& image, int& ret_width, int& ret_height)`: sets `ret_width` and `ret_height` with the width and height of the image array on this drawonable.
- `int Get_Char_Width (Am_Font Am_font, char c)`: returns the width of the character in the font.
- `int Get_String_Width (Am_Font Am_font, const char* the_string, int the_string_length)`
- `void Get_String_Extents (Am_Font Am_font, const char* the_string, int the_string_length, int& width, int& ascent, int& descent, int& left_bearing, int& right_bearing)`: The total height of the bounding rectangle for this string, or any string in this font, is ascent + descent. The left_bearing is the distance from the origin of the text to the first "inked" pixel. The right_bearing is the distance from the origin of the text to the last "inked" pixel.
- `void Get_Font_Properties (Am_Font Am_font, int& max_char_width, int& min_char_width, int& max_char_ascent, int& max_char_descent)`: The max ascent and descent include vertical spacing between rows of text. The min ascent and descent are computed on a per-char basis.

7.4.3 Clipping Operations

Gem supports clipping of all graphic operations to a specified clip region. Once a clip region is specified, all subsequent drawing operations are clipped so only parts inside the current clip region will show. To change the clip region of a drawonable, you invoke one of the member functions listed below on the drawonable.

Note that there is only one clip mask that is shared by all drawonables on a single screen (on one root drawonable). So as you set the clip mask for a window, you are actually setting it for all windows on the same screen as that window.

There are two ways to specify how a drawonable's clip region should be changed: by describing the change with integers (left, top, width, and height), or by providing an actual region. Regions can be of any shape, not necessarily rectangular, and are discussed in Section 7.4.4. Furthermore, the `Push_Clip()` and `Pop_Clip()` routines allow you to iteratively "nest" regions so that the current clip region is the intersection of all the previous clip regions that have been "pushed".

- `void Clear_Clip()`
- `void Set_Clip (Am_Region* region)`
- `void Set_Clip (short left, short top, unsigned short width, unsigned short height)`
- `void Push_Clip (Am_Region* region)`
- `void Push_Clip (short left, short top, unsigned short width, unsigned short height)`
- `void Pop_Clip ()`

The `In_Clip()` routines provide a way to ask a drawonable if a point is inside of its clip region. When asking if a given region is inside the drawonable's clip region, you can use the `total` parameter to determine whether the given region is completely inside the clip region, or whether it just intersects it.

- `bool In_Clip (short x, short y)`
- `bool In_Clip (short left, short top, unsigned short width, unsigned short height, bool &total)`
- `bool In_Clip (Am_Region *rgn, bool &total)`

7.4.4 Regions

Instances of the `Am_Region` class describe arbitrarily-shaped areas. `Am_Region` is a generalization of a drawonable's clip region, discussed in Section 7.4.3. By using the member functions listed below, you can build a region of arbitrary shape, and ultimately install it as the clip region of a drawonable.

```
class Am_Region {
public:

static  Am_Region* Create ();
  virtual void Destroy () = 0;
  virtual void Clear () = 0;
  virtual void Set (short left, short top, unsigned short width,
                    unsigned short height) = 0;
  virtual void Push (Am_Region* region) = 0;
  virtual void Push (short left, short top, unsigned short width,
                     unsigned short height) = 0;
  virtual void Pop () = 0;
  virtual void Union (short left, short top, unsigned short width,
                      unsigned short height) = 0;
  virtual void Intersect (short left, short top, unsigned short width,
                          unsigned short height) = 0;
  virtual bool In (short x, short y) = 0;
  virtual bool In (short left, short top, unsigned short width,
                   unsigned short height, bool &total) = 0;
  virtual bool In (Am_Region *rgn, bool &total) = 0;
};
```

7.4.5 Specific Drawing Functions

All of the drawing functions take a `Am_Draw_Function` which controls how the pixels of the drawn shape affect the screen. Since most programmers will use color screens, draw functions are not particularly useful. The supported values for `Am_Draw_Function` are `Am_DRAW_COPY`, `Am_DRAW_OR` or `Am_DRAW_XOR`.

The various options for styles and fonts (from `gdefs.h`) are explained in detail in the Opal manual, so they are not repeated there. Also, the details of most of the parameters to the operations correspond to similarly-named slots of the associated Opal objects, so the manual on Opal can be used to understand these operations more fully.

- `void Draw_Arc (Am_Style ls, Am_Style fs, int left, int top, unsigned int width, unsigned int height, int angle1 = 0, int angle2 = 360, Am_Draw_Function f = Am_DRAW_COPY, Am_Arc_Style_Flag asf = Am_ARC_PIE_SLICE )`: draw an arc. This can also be used for circles.
- `void Draw_Image (int left, int top, int width, int height, Am_Image_Array image, int i_left = 0, int i_top = 0, Am_Style ls = Am_No_Style, Am_Style fs = Am_No_Style, Am_Draw_Function f = Am_DRAW_COPY)`: The `ls` parameter is used to control the color of 'on' bits, and the `fs` parameter is for the background behind the image.
- `void Get_Polygon_Bounding_Box (Am_Point_List pl, Am_Style ls, int& out_left, int& out_top int& width, int& height)`: calculates the bounding box of the polygon.
- `void Draw_Line (Am_Style ls, int x1, int y1, int x2, int y2, Am_Draw_Function f = Am_DRAW_COPY)`: Draw a single line.
- `void Draw_Lines (Am_Style ls, Am_Style fs, Am_Point_List pl, Am_Draw_Function f = Am_DRAW_COPY)`: draw a polygon.
- `void Draw_2_Lines (Am_Style ls, Am_Style fs, int x1, int y1, int x2, int y2, int x3, int y3, Am_Draw_Function f = Am_DRAW_COPY)`:

Draw a polygon with just two lines.

- `void Draw_3_Lines (Am_Style ls, Am_Style fs, int x1, int y1, int x2, int y2, int x3, int y3, int x4, int y4, Am_Draw_Function f = Am_DRAW_COPY)`: Draw a polygon with just three lines. (This is used by arrowheads so a point-list does not have to be allocated).
- `void Draw_Rectangle (Am_Style ls, Am_Style fs, int left, int top, int width, int height, Am_Draw_Function f = Am_DRAW_COPY )`:
- `void Draw_Roundtangle (Am_Style ls, Am_Style fs, int left, int top, int width, int height, unsigned short x_radius, unsigned short y_radius, Am_Draw_Function f = Am_DRAW_COPY )`: a rectangle with rounded corners.
- `void Draw_Text (Am_Style ls, const char *s, int str_len, Am_Font Am_font, int left, int top, Am_Draw_Function f = Am_DRAW_COPY, Am_Style fs = Am_No_Style, bool invert = false)`: The `fs` is for the background behind the text.

7.5 Input Handling

7.5.1 Am_Input_Event_Handlers

The general way that input is handled for drawonables is that the underlying window manager generates various events. These are turned into messages sent to an instance of the C++ class `Am_Input_Event_Handlers` which is stored with the drawonable. For example, each time the user hits a keyboard key or a mouse button, the `Input_Event_Notify` member function is called of the `Am_Input_Event_Handlers` stored with the drawonable. Opal and Interactors define standard handlers for all these functions, so normally programmers do not need to deal with these messages, but instead will create new interactors. Most of these handlers simply update the associated object's slots. If you want to make a subclass of the standard event handlers and override some of the methods, `opal_advanced.h` exports `Am_Standard_Opal_Handlers` which are a subclass of `Am_Input_Event_Handlers`. `Am_Input_Event_Handlers` is defined as:

```
class Am_Input_Event_Handlers {
public:
    virtual void Iconify_Notify (Am_Drawonable* draw, bool iconified) = 0;
    virtual void Frame_Resize_Notify (Am_Drawonable* draw, int left,
                                      int top, int right, int bottom) = 0;
    virtual void Destroy_Notify (Am_Drawonable *draw) = 0;
    virtual void Configure_Notify (Am_Drawonable *draw, int left, int top,
                                   int width, int height) = 0;
    virtual void Exposure_Notify (Am_Drawonable *draw,
                                  int left, int top,
                                  int width, int height) = 0;
    virtual void Input_Event_Notify (Am_Drawonable *draw,
                                     Am_Input_Event *ev)=0;
};
```

The functions are:

- `Iconify_Notify`: Called when the window is being iconified or de-iconified.
- `Frame_Resize_Notify`: Called whenever the window's border size changes.
- `Destroy_Notify`: Called when the user requests that the window be destroyed. Note that window managers usually don't actually destroy the windows, but rather call this routine to tell the programs to destroy the window.
- `Configure_Notify`: Called whenever the user changes the window's size or position.
- `Exposure_Notify`: Called when the window becomes uncovered and part of it needs to be redrawn.
- `Input_Event_Notify`: Called for all input events from the keyboard and mouse. The `Am_Input_Event` is described below.

You can set and get the handlers in a drawonable using the following functions. If the event handlers are not set for a drawonable, they are inherited from the drawonable it was created from.

```
void Set_Input_Dispatch_Functions (Am_Input_Event_Handlers* evh)
void Get_Input_Dispatch_Functions (Am_Input_Event_Handlers*& evh)
```

7.5.2 Input Events

The input event passed to the `Input_Event_Notify` method is a C++ class containing the x and y of the mouse when the event occured, the drawonable of the event, a timestamp, and an `Am_Input_Char` describing the event. `Am_Input_Chars` are described in the Interactors manual.

```
class Am_Input_Event {
public:
    Am_Input_Char input_char; //the char and modifier bits; see idefs.h
    int x;
    int y;
    Am_Drawonable *draw; // Drawonable this event happened in
    unsigned long time_stamp;
};
```

You can control which input events are generated for a drawonable using the following member functions of drawonables:

- `void Set_Enter_Leave (bool want_enter_leave_events)`: Whether you want events generated when the mouse enters and leaves the drawonable. The default is that they are not.
- `void Set_Want_Move (bool want_move_events)`: Whether events are generated while the mouse is moving around inside the drawonable. The default is that they are not.
- `void Set_Multi_Window (bool want_multi_window)`: When an interactor should run over multiple windows, this method should be called on each window.

Otherwise, the cursor is "reserved" for the original window the mouse is clicked in.

- `void Get_Window_Mask (bool& want_enter_leave_events, bool& want_move_events, bool& want_multi_window)`

You can control the time interval for multiple clicks. The exported global variable `Am_Double_Click_Time` is the inter-click wait time in milleseconds. The default value is 250. If 0, then no double-click processing is done.

7.5.3 Main Loop

.The normal Amulet program calls `Am_Initialize` (which among other things, calls `Get_Root_Drawonable`), then sets up a number of objects, and then calls `Am_Main_Event_Loop`. This routine then calls a Gem level level routine where the events are actually processed. This routine is `Am_Drawonable::Process_Event()`. An Gem-level programmer who wants to process events, but *not* use any of the higher-level Amulet operations like demons and Opal might use `Am_Drawonable::Main_Loop`. This just repeatedly calls `Am_Drawonable::Process_Event ()`. To stop any of the main loops, you can set the exported `bool` called `Am_Main_Loop_Go` to false

The difference between `Process_event` and `Process_Immediate_Event` is that `Process_Event` waits for the next event, and processes exactly one input event and all non-input events (like refresh and configure_notify events) before and after that input event before returning. For example:

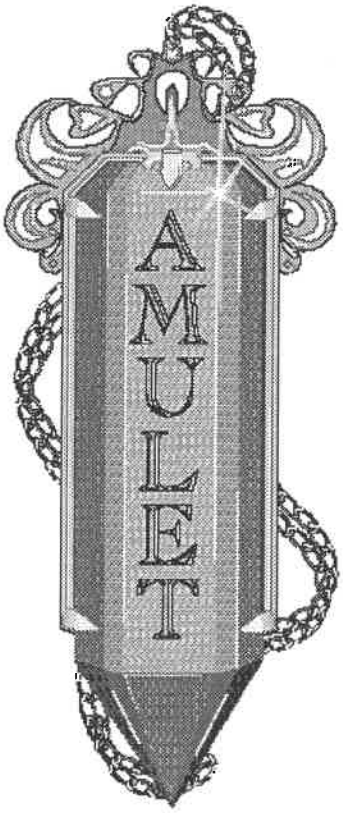
before		after
xxxIyyyIzzz	--->	Izzz

`Process_Event` returns when it encounters a second input event or when the queue is empty.

`Process_Immediate_Event` does not wait for an event, but processes the first event in the queue and all non-input events after it until an input event is seen. `Process_Immediate_Event` returns when it encounters an input event (excluding the case where the first event is an input event) or when the queue is empty.

```
// Should Am_Drawonable::Main_Loop and Am_Main_Event_Loop keep running?
extern bool Am_Main_Loop_Go;

class Am_Drawonable {
public:
    ...
    static void Main_Loop ();
    static void Process_Event ();
    static void Process_Immediate_Event ();
    ...
```



8. Summary of Exported Objects and Slots

Abstract

This chapter provides a summary of all the objects and slots exported by Amulet that the normal Amulet programmer will use. The specifics of the operation of the objects are discussed in other chapters of this manual.

This research was sponsored by NCCOSC under Contract No. N66001-94-C-6037, Arpa Order No. B326. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NCCOSC or the U.S. Government.

8.1 Am_Style:

Constructors:

```
Am_Style (float red, float green, float blue,          //color part
          short thickness = 0,
          Am_Line_Cap_Style_Flag cap_flag = Am_CAP_BUTT,
          Am_Join_Style_Flag join_flag = Am_JOIN_MITER,
          Am_Line_Solid_Flag line_flag = Am_LINE_SOLID,
          const char* dash_list = Am_DEFAULT_DASH_LIST,
          int dash_list_length = Am_DEFAULT_DASH_LIST_LENGTH,
          Am_Fill_Solid_Flag fill_flag = Am_FILL_SOLID,
          Am_Fill_Poly_Flag poly_flag = Am_FILL_POLY_EVEN_ODD,
          Am_Image_Array stipple = Am_No_Image)

Am_Style (const char* color_name,                      //color part
          short thickness = 0,
          Am_Line_Cap_Style_Flag cap_flag = Am_CAP_BUTT,
          Am_Join_Style_Flag join_flag = Am_JOIN_MITER,
          Am_Line_Solid_Flag line_flag = Am_LINE_SOLID,
          const char *dash_list = Am_DEFAULT_DASH_LIST,
          int dash_list_length = Am_DEFAULT_DASH_LIST_LENGTH,
          Am_Fill_Solid_Flag fill_flag = Am_FILL_SOLID,
          Am_Fill_Poly_Flag poly_flag = Am_FILL_POLY_EVEN_ODD,
          Am_Image_Array stipple = Am_No_Image)
```

Color styles: all have thickness zero (which really means 1--explained in the manual)

Am_Red	Am_Cyan	Am_Motif_Gray	Am_Motif_Light_Gray
Am_Green	Am_Orange	Am_Motif_Blue	Am_Motif_Light_Blue
Am_Blue	Am_Black	Am_Motif_Green	Am_Motif_Light_Green
Am_Yellow	Am_White	Am_Motif_Orange	Am_Motif_Light_Orange
Am_Purple		Am_Amulet_Purple	

Thick and dashed line styles: all are black

Am_Thin_Line	Am_Line_1	Am_Line_4	Am_Dashed_Line
Am_Line_0	Am_Line_2	Am_Line_8	Am_Dotted_Line

Stippled styles: all are black and white

Am_Gray_Stipple	Am_Opaque_Gray_Stipple
Am_Light_Gray_Stipple	Am_Diamond_Stipple
Am_Dark_Gray_Stipple	Am_Opaque_Diamond_Stipple

Special:

Am_No_Style - can be used in place of NULL

8.2 Am_Font:

Constructors:

```
Am_Font (Am_Font_Family_Flag family = Am_FONT_FIXED,
         bool is_bold = false,
         bool is_italic = false,
         bool is_underline = false,
         Am_Font_Size_Flag size = Am_FONT_MEDIUM)

Am_Font (const char* the_name)
```

Pre-Defined Fonts:

`Am_Default_Font` - a fixed, medium-sized font

8.3 Predefined formula constraints:

`Am_Width_Of_Parts` - Useful for computing the width of a group: returns the distance between the group's left and the right of its rightmost part. You might put this into a group's `Am_WIDTH` slot.

`Am_Height_Of_Parts` - Analogous to `Am_Width_Of_Parts`, but for the `Am_HEIGHT`.

`Am_Right_Is_Right_Of_Owner` - Useful for keeping a part at the right of its owner. Put this formula in the `Am_LEFT` slot of the part.

`Am_Bottom_Is_Bottom_Of_Owner` - Useful for keeping a part at the bottom of its owner. Put this formula in the `Am_TOP` slot of the part.

`Am_Center_X_Is_Center_Of_Owner` - Useful for centering a part horizontally within its owner. Put this formula in the `Am_LEFT` slot of the part.

`Am_Center_Y_Is_Center_Of_Owner` - Useful for centering a part vertically within its owner. Put this formula in the `Am_TOP` slot of the part.

`Am_Center_X_Is_Center_Of` - Useful for horizontally centering `obj1` inside `obj2`. Put this formula in the `Am_LEFT` slot of `obj1`, and put `obj2` in the `Am_CENTER_X_OBJ` slot of `obj1`.

`Am_Center_Y_Is_Center_Of` - Useful for vertically centering `obj1` inside `obj2`. Put this formula in the `Am_TOP` slot of `obj1`, and put `obj2` in the `Am_CENTER_Y_OBJ` slot of `obj1`.

`Am_Horizontal_Layout` - Constraint which lays out the parts of a group horizontally in one or more rows. Put this into the `Am_LAYOUT` slot of a group.

`Am_Vertical_Layout` - Constraint which lays out the parts of a group vertically in one or more columns. Put this into the `Am_LAYOUT` slot of a group.

8.4 Opal Graphical Objects

Am_Graphical_Object:

Slot	Default Value	Type
Am_LEFT	0	int
Am_TOP	0	int
Am_WIDTH	10	int
Am_HEIGHT	10	int
Am_VISIBLE	true	bool

Am_Line:

Slot	Default Value	Type
Am_LINE_STYLE	Am_Black	Am_Style
Am_X1	0	int
Am_Y1	0	int
Am_X2	0	int
Am_Y2	0	int
Am_LEFT	0	int
Am_TOP	0	int
Am_WIDTH	1	int
Am_HEIGHT	1	int
Am_VISIBLE	true	bool
Am_HIT_THRESHOLD	0	int

// Am_X1, Am_Y1, Am_X2, Am_Y2,
// Am_LEFT, Am_TOP, Am_WIDTH, and
// Am_HEIGHT are constrained in such a
// way that if any one of them changes,
// the rest will automatically be updated
// to reflect that change.

Am_Rectangle:

Slot	Default Value	Type	
Am_VISIBLE	true	bool	
Am_LEFT	0	int	
Am_TOP	0	int	
Am_WIDTH	10	int	
Am_HEIGHT	10	int	
Am_FILL_STYLE	Am_Black	Am_Style	*// Inside of rectangle*
Am_LINE_STYLE	Am_Black	Am_Style	*// Edge of rectangle*

Am_Arc: *(useful for circles, ovals, and arcs)*

Slot	Default Value	Type	
Am_VISIBLE	true	bool	
Am_LEFT	0	int	
Am_TOP	0	int	
Am_WIDTH	10	int	
Am_HEIGHT	10	int	
Am_ANGLE1	0	0..360	*// Origin, degrees from 3:00*
Am_ANGLE2	360	0..360	*// Terminus, distance from origin*
Am_FILL_STYLE	Am_Black	Am_Style	*// Inside of arc*
Am_LINE_STYLE	Am_Black	Am_Style	*// Edge of arc*

Am_Roundtangle: *(rectangle with rounded corners)*

Slot	Default Value	Type	
Am_VISIBLE	true	bool	
Am_LEFT	0	int	
Am_TOP	0	int	
Am_WIDTH	10	int	
Am_HEIGHT	10	int	
Am_RADIUS	Am_SMALL_RADIUS	Am_Radius_Flag or int	*// { Am_SMALL_RADIUS,* *// Am_MEDIUM_RADIUS,* *// Am_LARGE_RADIUS }*
Am_FILL_STYLE	Am_Black	Am_Style	*// Inside of roundtangle*
Am_LINE_STYLE	Am_Black	Am_Style	*// Edge of roundtangle*

Am_Polygon:

Slot	Default Value	Type
Am_VISIBLE	true	bool
Am_LEFT	<formula>	int
Am_TOP	<formula>	int
Am_WIDTH	<formula>	int
Am_HEIGHT	<formula>	int
Am_POINT_LIST	empty Am_Point_List	Am_Point_List

Am_Text:

Slot	Default Value	Type	
Am_VISIBLE	true	bool	
Am_LEFT	0	int	
Am_TOP	0	int	
Am_WIDTH	<formula>	int	
Am_HEIGHT	<formula>	int	
Am_TEXT	""	Am_String	*// String to display*
Am_FONT	Am_Default_Font	Am_Font	
Am_CURSOR_INDEX	Am_NO_CURSOR	int	*// Position of cursor in string*
Am_LINE_STYLE	Am_Line_2	Am_Style	*// Color of text*
Am_FILL_STYLE	Am_No_Style	Am_Style	*// Background behind text*
Am_X_OFFSET	<formula>	int	
Am_INVERT	false	bool	*// Whether to exchange line and fill style*

Am_Bitmap:

Slot	Default Value	Type	
Am_VISIBLE	true	bool	
Am_LEFT	0	int	
Am_TOP	0	int	
Am_WIDTH	<formula>	int	
Am_HEIGHT	<formula>	int	
Am_LINE_STYLE	Am_Black	Am_Style	*// Color of on pixels*
Am_FILL_STYLE	Am_No_Style	Am_Style	*// Color of off pixels* *// if opaque stipple*
Am_IMAGE	Am_No_Image *(solid)*	Am_Image_Array	*// Stipple pattern*

Am_Group:

Slot	Default Value	Type	
Am_VISIBLE	true	bool	
Am_LEFT	0	int	
Am_TOP	0	int	
Am_WIDTH	10	int	
Am_HEIGHT	10	int	
Am_GRAPHICAL_PARTS	empty Am_Value_List	Am_Value_List	*// Read-only list*
Am_LAYOUT	NULL	<formula>	
Am_X_OFFSET	0	int	
Am_Y_OFFSET	0	int	
Am_H_SPACING	0	int	
Am_V_SPACING	0	int	
Am_H_ALIGN	Am_CENTER_ALIGN	{Am_LEFT_ALIGN, Am_RIGHT_ALIGN, Am_CENTER_ALIGN}	
Am_V_ALIGN	Am_CENTER_ALIGN	{Am_TOP_ALIGN, Am_BOTTOM_ALIGN, Am_CENTER_ALIGN}	
Am_FIXED_WIDTH	Am_NOT_FIXED_SIZE	int	
Am_FIXED_HEIGHT	Am_NOT_FIXED_SIZE	int	
Am_INDENT	0	int	
Am_MAX_RANK	false	int, bool	
Am_MAX_SIZE	false	int, bool	

Am_Map:

Slot	Default Value	Type
Am_VISIBLE	true	bool
Am_LEFT	0	int
Am_TOP	0	int
Am_WIDTH	Am_Width_Of_Parts	int
Am_HEIGHT	Am_Height_Of_Parts	int
Am_GRAPHICAL_PARTS	<formula>	Am_Value_List
Am_ITEMS	0	int, Am_Value_List
Am_ITEM_PROTOTYPE	Am_No_Object	Am_Object
Am_LAYOUT	NULL	<formula>
Am_X_OFFSET	0	int
Am_Y_OFFSET	0	int
Am_H_SPACING	0	int
Am_V_SPACING	0	int
Am_H_ALIGN	Am_CENTER_ALIGN	{Am_LEFT_ALIGN, Am_RIGHT_ALIGN, Am_CENTER_ALIGN}
Am_V_ALIGN	Am_CENTER_ALIGN	{Am_TOP_ALIGN, Am_BOTTOM_ALIGN, Am_CENTER_ALIGN}
Am_FIXED_WIDTH	Am_NOT_FIXED_SIZE	int
Am_FIXED_HEIGHT	Am_NOT_FIXED_SIZE	int
Am_INDENT	0	int
Am_MAX_RANK	false	int, bool
Am_MAX_SIZE	false	int, bool

Am_Window:

Slot	Default Value	Type
Am_VISIBLE	true	bool
Am_LEFT	0	int
Am_TOP	0	int
Am_WIDTH	100	int
Am_HEIGHT	100	int
Am_GRAPHICAL_PARTS	empty Am_Value_List	Am_Value_List
Am_FILL_STYLE	Am_White	Am_Style
Am_MAX_WIDTH	0	int
Am_MAX_HEIGHT	0	int
Am_MIN_WIDTH	1	int
Am_MIN_HEIGHT	1	int
Am_TITLE	"Amulet"	char*
Am_ICON_TITLE	"Amulet"	char*
Am_ICONIFIED	false	bool
Am_USE_MIN_WIDTH	false	bool
Am_USE_MIN_HEIGHT	false	bool
Am_USE_MAX_WIDTH	false	bool
Am_USE_MIN_HEIGHT	false	bool
Am_QUERY_POSITION	false	bool
Am_QUERY_SIZE	false	bool
Am_LEFT_BORDER_WIDTH	0	int
Am_TOP_BORDER_WIDTH	0	int
Am_RIGHT_BORDER_WIDTH	0	int
Am_BOTTOM_BORDER_WIDTH	0	int
Am_CURSOR	NULL	Am_Cursor
Am_OMIT_TITLE_BAR	false	bool
Am_CLIP_CHILDREN	false	bool
Am_SAVE_UNDER	false	false

8.5 Interactors

Am_Interactor:

Slot	Default Value	Type	
Am_START_WHEN	Am_Input_Char ("LEFT_DOWN")	Am_Input_Char	
Am_START_WHERE_TEST	Am_Inter_In	Am_Where_Function	
Am_ABORT_WHEN	Am_Input_Char ("CONTROL_g")	Am_Input_Char	
Am_RUNNING_WHERE_OBJECT	true	Am_Object, bool	
Am_RUNNING_WHERE_TEST	Am_Inter_In	Am_Where_Function	
Am_STOP_WHEN	Am_Input_Char ("ANY_MOUSE_UP")	Am_Input_Char	
Am_ACTIVE	true	bool	*// Section 5.3.3.3*
Am_START_OBJECT	0	Am_Object	*// Section 5.4.1*
Am_START_CHAR	0	Am_Input_Char	*// Section 5.4.1*
Am_CURRENT_OBJECT	0	Am_Object	*// Section 5.4.1*
Am_RUN_ALSO	false	bool	*// Section5.4.2*
Am_PRIORITY	1.0	float	*// Section 5.4.2*
Am_OTHER_WINDOWS	NULL	Am_Value_List or Am_Window	*// Section 5.4.3*
Am_WINDOW	NULL	Am_Window	*Set with current window*
Am_COMMAND	Am_Command	Am_Command	

Am_Choice_Interactor:

Slot	Default Value	Type	
Am_START_WHEN	Am_Input_Char ("LEFT_DOWN")	Am_Input_Char	
Am_START_WHERE_TEST	Am_Inter_In	Am_Where_Function	
Am_ABORT_WHEN	Am_Input_Char ("CONTROL_g")	Am_Input_Char	
Am_RUNNING_WHERE_OBJECT	<formula>	Am_Object, bool	*// computes owner*
Am_RUNNING_WHERE_TEST	<formula>	Am_Where_Function	*// same as start*
Am_STOP_WHEN	Am_Input_Char ("ANY_MOUSE_UP")	Am_Input_Char	
Am_HOW_SET	Am_CHOICE_TOGGLE	Am_Choice_How_Set	
Am_FIRST_ONE_ONLY	false	bool	*// whether menu-* *// or button-like*
Am_COMMAND	Am_Choice_Command	Am_Command	

Am_One_Shot_Interactor:

Slot	Default Value	Type	
Am_START_WHEN	Am_Input_Char ("LEFT_DOWN")	Am_Input_Char	
Am_START_WHERE_TEST	Am_Inter_In	Am_Where_Functio	
Am_ABORT_WHEN	Am_Input_Char ("CONTROL_g")	Am_Input_Char	
Am_RUNNING_WHERE_OBJECT	*owner*	Am_Object, bool	
Am_RUNNING_WHERE_TEST	Am_Inter_In	Am_Where_Functio	*// same as start*
Am_STOP_WHEN	Am_Input_Char ("ANY_MOUSE_UP")	Am_Input_Char	
Am_RUN_ALSO	false	bool	
Am_ACTIVE	true	bool	
Am_HOW_SET	Am_CHOICE_TOGGLE	Am_Choice_How_Se	
Am_FIRST_ONE_ONLY	true	bool	*// whether menu-* *// or button-like*
Am_COMMAND	Am_Choice_Command	Am_Command	

Am_Text_Edit_Interactor:

Slot	Default Value	Type
Am_START_WHEN	Am_Input_Char("LEFT_DOWN")	Am_Input_Char
Am_START_WHERE_TEST	Am_Inter_In_Text	Am_Where_Function
Am_ABORT_WHEN	Am_Input_Char("CONTROL_g")	Am_Input_Char
Am_RUNNING_WHERE_OBJECT	true	Am_Object, bool
Am_RUNNING_WHERE_TEST	Am_Inter_In	Am_Where_Function
Am_STOP_WHEN	Am_Input_Char("RETURN")	Am_Input_Char
Am_TEXT_EDIT_FUNCTION	Am_Default_Text_Edit_ Function	Am_Text_Edit_ Function
Am_EDIT_TRANSLATION_TABLE	Am_Edit_Translation_ Table::Default_Table()	Am_Edit_Trans- lation_Table
Am_COMMAND	Am_Edit_Text_Command	Am_Command

Am_Move_Grow_Interactor:

Slot	Default Value	Type	
Am_START_WHEN	Am_Input_Char ("LEFT_DOWN")	Am_Input_Char	
Am_START_WHERE_TEST	Am_Inter_In	Am_Where_Function	
Am_ABORT_WHEN	Am_Input_Char ("CONTROL_g")	Am_Input_Char	
Am_RUNNING_WHERE_ OBJECT	true	Am_Object, bool	
Am_RUNNING_WHERE_ TEST	Am_Inter_In	Am_Where_Function	
Am_GROWING	false	bool	
Am_AS_LINE	<formula>	bool	
Am_FEEDBACK_OBJECT	0	Am_Object	*// interim feedback*
Am_GRID_X	0	int	
Am_GRID_Y	0	int	
Am_GRID_ORIGIN_X	0	int	
Am_GRID_ORIGIN_Y	0	int	
Am_GRID_PROC	0	Am_Custom_ Gridding_Proc	
Am_WHERE_ATTACH	Am_ATTACH_ WHERE_HIT	Am_Move_Grow_ Where_Attach	Am_ATTACH_ .. {WHERE_HIT, NW, N, NE, E, SE, S, SW, W, END_1, END_2, CENTER}
Am_MINIMUM_WIDTH	0	int	
Am_MINIMUM_HEIGHT	0	int	
Am_MINIMUM_LENGTH	0	int	
Am_COMMAND	Am_Move_Grow_ Command	Am_Command	

Am_New_Points_Interactor:

Slot	Default Value	Type
Am_START_WHEN	Am_Input_Char("LEFT_DOWN")	Am_Input_Char
Am_START_WHERE_TEST	Am_Inter_In	Am_Where_Function
Am_ABORT_WHEN	Am_Input_Char("CONTROL_g")	Am_Input_Char
Am_RUNNING_WHERE_OBJECT	true	Am_Object, bool
Am_RUNNING_WHERE_TEST	Am_Inter_In	Am_Where_Function
Am_AS_LINE	0	bool
Am_FEEDBACK_OBJECT	0	Am_Object
Am_HOW_MANY_POINTS	2	int
Am_FLIP_IF_CHANGE_SIDES	true	bool
Am_ABORT_IF_TOO_SMALL	false	bool
Am_STOP_WHEN	Am_Input_Char("ANY_MOUSE_UP")	Am_Input_Char
Am_GRID_X	0	int
Am_GRID_Y	0	int
Am_GRID_ORIGIN_X	0	int
Am_GRID_ORIGIN_Y	0	int
Am_GRID_PROC	0	Am_Custom_ Gridding_Proc
Am_MINIMUM_WIDTH	0	int
Am_MINIMUM_HEIGHT	0	int
Am_MINIMUM_LENGTH	0	int
Am_COMMAND	Am_New_Points_Command	Am_Command

8.6 Interactor Command Objects

Am_Command:

Slot	Default Value	Type
Am_INTERIM_DO_ACTION	Am_Command_Interim_Do	Am_Object_Proc*
Am_DO_ACTION	Am_Command_Do	Am_Object_Proc*
Am_UNDO_ACTION	Am_Command_Undo	Am_Object_Proc*
Am_UNDO_THE_UNDO_ACTION	Am_Command_Undo_The_Undo	Am_Object_Proc*
Am_ABORT_ACTION	Am_Command_Abort	Am_Object_Proc*
Am_LABEL	"A command"	Am_String
Am_ACTIVE	true	bool
Am_VALUE	0	any

Am_Choice_Command:

Slot	Default Value	Type
Am_START_ACTION	Am_Choice_Command_Start	
Am_INTERIM_DO_ACTION	Am_Choice_Command_Interim_Do	Am_Object_Proc*
Am_DO_ACTION	Am_Choice_Command_Do	Am_Object_Proc*
Am_UNDO_ACTION	Am_Choice_Command_Undo	Am_Object_Proc*
Am_UNDO_THE_UNDO_ACTION	Am_Choice_Command_Undo_ The_Undo	Am_Object_Proc*
Am_ABORT_ACTION	Am_Choice_Command_Abort	Am_Object_Proc*
Am_LABEL	"choice interactor"	Am_String
Am_ACTIVE	true	bool
Am_INTERIM_VALUE	0	Am_Object
Am_OLD_INTERIM_VALUE	0	Am_Object
Am_OLD_VALUE	0	any
Am_VALUE	0	any

Am_Move_Grow_Command:

Slot	Default Value	Type
Am_START_ACTION	Am_Move_Grow_Command_Start	Am_Object_Proc*
Am_INTERIM_DO_ACTION	Am_Move_Grow_Command_Interim_ Do	Am_Object_Proc*
Am_DO_ACTION	Am_Move_Grow_Command_Do	Am_Object_Proc*
Am_UNDO_ACTION	Am_Move_Grow_Command_Undo	Am_Object_Proc*
Am_UNDO_THE_UNDO_ ACTION	Am_Move_Grow_Command_Undo_ The_Undo	Am_Object_Proc*
Am_ABORT_ACTION	Am_Move_Grow_Command_Abort	Am_Object_Proc*
Am_LABEL	"Move_Grow interactor"	Am_String
Am_ACTIVE	true	bool
Am_OBJECT_MODIFIED	0	Am_Object
Am_INTERIM_VALUE	0	Am_Four_Ints
Am_OLD_VALUE	0	Am_Four_Ints
Am_VALUE	0	Am_Four_Ints

Am_New_Points_Command:

Slot	Default Value	Type
Am_START_ACTION	Am_New_Points_Command_Start	Am_Object_Proc*
Am_INTERIM_DO_ACTION	Am_New_Points_Command_ Interim_Do	Am_Object_Proc*
Am_DO_ACTION	Am_New_Points_Command_Do	Am_Object_Proc*
Am_UNDO_ACTION	Am_New_Points_Command_Undo	Am_Object_Proc*
Am_UNDO_THE_UNDO_ACTION	Am_New_Points_Command_Undo_ The_Undo	Am_Object_Proc*
Am_ABORT_ACTION	Am_New_Points_Command_Abort	Am_Object_Proc*
Am_LABEL	"New_Points interactor"	Am_String
Am_ACTIVE	true	bool
Am_TOO_SMALL	0	bool
Am_INTERIM_VALUE	0	Am_Four_Ints
Am_VALUE	0	any
Am_CREATE_NEW_OBJECT_ ACTION	0	Am_Create_New_ Object_Proc*

Am_Edit_Text_Command:

Slot	Default Value	Type	
Am_START_ACTION	Am_Text_Command_Start	Am_Object_Proc*	
Am_INTERIM_DO_ ACTION	Am_Text_Command_ Interim_Do	Am_Object_Proc*	
Am_DO_ACTION	Am_Text_Command_Do	Am_Object_Proc*	
Am_UNDO_ACTION	Am_Text_Command_Undo	Am_Object_Proc*	
Am_UNDO_THE_UNDO_ ACTION	Am_Text_Command_Undo_ The_Undo	Am_Object_Proc*	
Am_ABORT_ACTION	Am_Text_Command_Abort	Am_Object_Proc*	
Am_LABEL	"text interactor"	Am_String	
Am_ACTIVE	true	bool	
Am_OBJECT_MODIFIED	0	Am_Object	*//object edited*
Am_INTERIM_VALUE	0	Am_Input_Event	*//each event set here*
Am_OLD_VALUE	0	Am_String	*//set to old string*
Am_VALUE	0	Am_String	*//new final string*

8.7 Undo objects

Am_Undo_Handler:

Slot	Default Value	Type
Am_REGISTER_COMMAND	0	Am_Register_Command_Proc
Am_UNDO_THE_UNDO_ALLOWED	0	Am_Object or 0
Am_UNDO_ALLOWED	0	Am_Object or 0
Am_PERFORM_UNDO	0	Am_Object_Proc*
Am_PERFORM_UNDO_THE_UNDO	0	Am_Object_Proc*

Am_Single_Undo_Object:

Slot	Default Value	Type
Am_REGISTER_COMMAND	Am_Single_Undo_Register_ Command	Am_Register_ Command_Proc
Am_UNDO_THE_UNDO_ALLOWED	<formula>	Am_Object or 0
Am_UNDO_ALLOWED	<formula>	Am_Object or 0
Am_PERFORM_UNDO	Am_Single_Perform_Undo	Am_Object_Proc*
Am_PERFORM_UNDO_THE_UNDO	Am_Single_Perform_Undo_ The_Undo	Am_Object_Proc*

Am_Multiple_Undo_Object:

Slot	Default Value	Type
Am_REGISTER_COMMAND	Am_Multiple_Undo_Register_ Command	Am_Register_ Command_Proc
Am_UNDO_ALLOWED	<formula>	Am_Object or 0
Am_UNDO_THE_UNDO_ALLOWED	<formula>	Am_Object or 0
Am_PERFORM_UNDO	Am_Multiple_Perform_Undo	Am_Object_Proc*
Am_PERFORM_UNDO_THE_UNDO	Am_Multiple_Perform_Undo_ The_Undo	Am_Object_Proc*
Am_LAST_UNDONE_COMMAND	0	Am_Command

8.8 Widget objects

Am_Border_Rectangle: *(a Motif-like rectangle with border)*

Slot	Default Value	Type	
Am_SELECTED	false	bool	
Am_WIDGET_LOOK	Am_MOTIF_LOOK	Am_Widget_Look	*// { Am_MOTIF_LOOK,* *// Am_MACINTOSH_LOOK,* *// Am_WINDOWS_LOOK }*
Am_WIDTH	50	int	
Am_HEIGHT	50	int	
Am_TOP	0	int	
Am_LEFT	0	int	
Am_VISIBLE	true	bool	
Am_FILL_STYLE	Am_Amulet_Purple	Am_Style	

Am_Button:

Slot	Default Value	Type	
Am_LEFT	0	int	
Am_TOP	0	int	
Am_WIDTH	<formula>	int	
Am_HEIGHT	<formula>	int	
Am_Y_OFFSET	0	int	
Am_H_SPACING	0	int	
Am_V_SPACING	0	int	
Am_H_ALIGN	Am_CENTER_ALIGN	{Am_LEFT_ALIGN, Am_RIGHT_ALIGN, Am_CENTER_ALIGN}	
Am_V_ALIGN	Am_CENTER_ALIGN	{Am_TOP_ALIGN, Am_BOTTOM_ALIGN, Am_CENTER_ALIGN}	
Am_FIXED_WIDTH	Am_NOT_FIXED_SIZE	int	
Am_FIXED_HEIGHT	Am_NOT_FIXED_SIZE	int	
Am_INDENT	0	int	
Am_MAX_RANK	false	bool	
Am_MAX_SIZE	false	bool	
Am_ITEM_OFFSET	5	int	
Am_ACTIVE	<formula>	bool	
Am_ACTIVE_2	true	bool	
Am_WIDGET_LOOK	Am_MOTIF_LOOK	Am_Widget_Look	*// { Am_MOTIF_LOOK, // Am_MACINTOSH_LOOK, // Am_WINDOWS_LOOK }*
Am_KEY_SELECTED	false	bool	
Am_FONT	Am_Default_Font	Am_Font	
Am_FINAL_FEEDBACK_WANTED	false	bool	
Am_FILL_STYLE	Am_Amulet_Purple	Am_Style	
Am_COMMAND	Am_Button_Command	Am_Command	

Am_Button_Panel:

Slot	Default Value	Type	
Am_LEFT	0	int	
Am_TOP	0	int	
Am_WIDTH	<formula>	int	*// Read-only*
Am_HEIGHT	<formula>	int	*// Read-only*
Am_HOW_SET	Am_CHOICE_SET	Am_How_Set	
Am_ITEM_OFFSET	5	int	
Am_ACTIVE	<formula>	bool	
Am_ACTIVE_2	true	bool	
Am_WIDGET_LOOK	Am_MOTIF_LOOK	Am_Widget_Look	*// { Am_MOTIF_LOOK,* *// Am_MACINTOSH_LOOK,* *// Am_WINDOWS_LOOK }*
Am_KEY_SELECTED	false	bool	
Am_FONT	Am_Default_Font	Am_Font	
Am_FILL_STYLE	Am_Amulet_Purple	Am_Style	
Am_FINAL_FEEDBACK_WANTED	false	bool	
Am_WIDTH	Am_Width_Of_Parts	int	
Am_HEIGHT	Am_Width_Of_Parts	int	
Am_LAYOUT	Am_Vertical_Layout	{Am_Vertical_Layout, Am_Horizontal_Layou NULL}	
Am_H_ALIGN	Am_LEFT_ALIGN	{Am_LEFT_ALIGN, Am_RIGHT_ALIGN, Am_CENTER_ALIGN}	
Am_ITEMS	<special>	int, Am_Value_List of commands or strings, etc.	

Am_Radio_Button_Panel: Am_Button_Panel

Slot	Default Value	Type
all the slots of the button panel		
Am_HOW_SET	Am_CHOICE_SET	Am_How_Set
Am_BOX_WIDTH	15	int
Am_BOX_HEIGHT	15	int
Am_FIXED_WIDTH	false	int, bool
Am_FINAL_FEEDBACK_WANTED	true	bool
Am_H_ALIGN	<formula>	{Am_LEFT_ALIGN, Am_RIGHT_ALIGN, Am_CENTER_ALIGN}

Am_Checkbox_Panel: Am_Button_Panel

Slot	Default Value	Type
all the slots of the button panel		
Am_HOW_SET	Am_CHOICE_SET	Am_How_Set
Am_BOX_WIDTH	15	int
Am_BOX_HEIGHT	15	int
Am_FIXED_WIDTH	false	int, bool
Am_FINAL_FEEDBACK_WANTED	true	bool
Am_H_ALIGN	<formula>	{Am_LEFT_ALIGN, Am_RIGHT_ALIGN, Am_CENTER_ALIGN}

Am_Menu: Am_Button_Panel

Slot	Default Value	Type
all the slots of the button panel		
Am_HOW_SET	Am_CHOICE_SET	Am_How_Set
Am_FINAL_FEEDBACK_WANTED	false	bool
Am_WIDTH	<formula>	int
Am_HEIGHT	<formula>	int
Am_X_OFFSET	2	int
Am_Y_OFFSET	2	int
Am_V_SPACING	-2	int

Am_Menu_Bar: Am_Menu

Slot	Default Value	Type
Am_LEFT	0	int
Am_TOP	0	int
Am_WIDTH	<formula>	int
Am_HEIGHT	<formula>	int
Am_ACTIVE	<formula>	bool
Am_ACTIVE_2	true	bool
Am_WIDGET_LOOK	Am_MOTIF_LOOK	Am_Widget_Look
Am_FONT	Am_Default_Font	Am_Font
Am_FILL_STYLE	Am_Amulet_Purple	Am_Style
Am_ITEMS	NULL	Am_Value_List
Am_COMMAND	Am_Button_Command	Am_Command

Am_Vertical_Scroll_Bar:

Slot	Default Value	Type	
Am_LEFT	0	int	
Am_TOP	0	int	
Am_WIDTH	20	int	
Am_HEIGHT	200	int	
Am_WIDGET_LOOK	Am_MOTIF_LOOK	Am_Widget_Look	
Am_FILL_STYLE	Am_Amulet_Purple	Am_Style	
Am_VALUE_1	0	int or float	*// Value at top*
Am_VALUE_2	100	int or float	*// Value at bottom*
Am_SMALL_INCREMENT	1	int or float	*// When click arrow*
Am_LARGE_INCREMENT	10	int or float	*// When click "page"*
Am_PERCENT_VISIBLE	0.2	float	*// Size of indicator*
Am_COMMAND	Am_Scroll_Command	Am_Command	

Am_Horizontal_Scroll_Bar:

Slot	Default Value	Type	
Am_LEFT	0	int	
Am_TOP	0	int	
Am_WIDTH	200	int	
Am_HEIGHT	20	int	
Am_WIDGET_LOOK	Am_MOTIF_LOOK	Am_Widget_Look	
Am_FILL_STYLE	Am_Amulet_Purple	Am_Style	
Am_VALUE_1	0	int or float	*// Value at top*
Am_VALUE_2	100	int or float	*// Value at bottom*
Am_SMALL_INCREMENT	1	int or float	*// When click arrow*
Am_LARGE_INCREMENT	10	int or float	*// When click "page"*
Am_PERCENT_VISIBLE	0.2	float	*// Size of indicator*
Am_COMMAND	Am_Scroll_Command	Am_Command	

Am_Scrolling_Group: *(used like a group with scroll bars)*

Slot	Default Value	Type	
Am_LEFT	0	int	
Am_TOP	0	int	
Am_WIDTH	0	int	
Am_HEIGHT	0	int	
Am_X_OFFSET	0	int	*// Where scrolled to*
Am_Y_OFFSET	0	int	*// " " " "*
Am_WIDGET_LOOK	Am_MOTIF_LOOK	Am_Widget_Look	
Am_FILL_STYLE	Am_Amulet_Purple	Am_Style	
Am_INNER_FILL_STYLE	0	Am_Style or 0	*// If 0 uses FILL_STYLE*
Am_H_SCROLL_BAR	true	bool	*// Whether show horiz bar*
Am_V_SCROLL_BAR	true	bool	*// " " vertical "*
Am_H_SCROLL_BAR_ON_TOP	false	bool	
Am_V_SCROLL_BAR_ON_LEFT	false	bool	
Am_H_SMALL_INCREMENT	10	int	
Am_H_LARGE_INCREMENT	<formula>	int	*// Computed from page size*
Am_V_SMALL_INCREMENT	10	int	
Am_V_LARGE_INCREMENT	<formula>	int	*// Computed from page size*
Am_INNER_WIDTH	400	int	*// Size of scrollable area*
Am_INNER_HEIGHT	400	int	*// " " " "*

Am_Text_Input_Widget:

Slot	Default Value	Type
Am_LEFT	0	int
Am_TOP	0	int
Am_WIDTH	150	int
Am_HEIGHT	<formula>	int
Am_WIDGET_LOOK	Am_MOTIF_LOOK	Am_Widget_Look
Am_FONT	Am_Default_Font	Am_Font
Am_LABEL_FONT	bold_font	Am_Font
Am_FILL_STYLE	Am_Amulet_Purple	Am_Style
Am_ACTIVE_2	true	bool
Am_COMMAND	Am_Text_Input_Command	Am_Command

8.9 Widget command objects

Am_Button_Command:

Slot	Default Value	Type
Am_DO_ACTION	Am_Command_Do	Am_Object_Proc*
Am_UNDO_ACTION	Am_Button_Command_Undo	Am_Object_Proc*
Am_UNDO_THE_UNDO_ACTION	Am_Button_Command_Undo	Am_Object_Proc*
Am_LABEL	"Label"	Am_String or any graphical object
Am_ID	0	any
Am_ACTIVE	true	bool
Am_VALUE	0	any

Am_Menu_Line_Command:

Slot	Default Value	Type
Am_DO_ACTION	NULL	Am_Object_Proc*
Am_UNDO_ACTION	NULL	Am_Object_Proc*
Am_UNDO_THE_UNDO_ACTION	NULL	Am_Object_Proc*
Am_LABEL	"Menu_Line_Command"	Am_String
Am_ACTIVE	false	bool
Am_VALUE	NULL	any

Am_Scroll_Command:

Slot	Default Value	Type
Am_DO_ACTION	Am_Command_Do	Am_Object_Proc*
Am_UNDO_ACTION	Am_Scroll_Command_Undo	Am_Object_Proc*
Am_UNDO_THE_UNDO_ACTION	Am_Scroll_Command_Undo	Am_Object_Proc*
Am_LABEL	"Scrollbar"	Am_String
Am_ACTIVE	true	bool
Am_VALUE	50	int

Am_Text_Input_Command:

Slot	Default Value	Type
Am_DO_ACTION	Am_Command_Do	Am_Object_Proc*
Am_UNDO_ACTION	Am_Text_Input_Command_Undo	Am_Object_Proc*
Am_UNDO_THE_UNDO_ACTION	Am_Text_Input_Command_Undo	Am_Object_Proc*
Am_LABEL	"Label"	Am_String
Am_ACTIVE	true	bool
Am_VALUE	""	Am_String

9. Index

—B—

—C—

—D—

—E—

—R—

—S—

—T—

—U—

—V—

—W—