New Planet Software

The JX Development Environment

User Guide

For:
JX Application Framework 2.0
Code Crusader 3.0
Code Medic 1.1

Glenn W. Bach

John Lindal

Revision 2.0, January 2001

New Planet Software
P.O. Box 7634
Los Angeles, CA 91346
info@newplanetsoftware.com
http://www.newplanetsoftware.com

Contents

Part I

Programs

Chapter 1

Installation

The cdrom included with the package includes an install program that will install the programs and, if you choose, will also install the JX Application Framework for writing X applications.

1.1 Mounting the CDROM

The first step is to mount the cdrom. Typically, when you install Linux, a device is created called `/dev/cdrom`. This device points to your actual cdrom device. In addition, the typical install will set up the system to be able to easily mount this device (by adding an entry to your systems `/etc/fstab` file). If you have such a system, mounting the cdrom is quite easy. As root, you enter the following command:

```
> mount /mnt/cdrom
```

If this doesn't work, you will have to do a little bit more. First create a directory that will act as the mount directory for the cdrom:

```
> mkdir /mnt/cdrom
```

Then figure out which is the appropriate device for your cdrom. If there is a `/dev/cdrom` device, then this is probably it, otherwise, you will have to find the actual device. If you have a typical system, this device will be `/dev/hdx` where `x` is either `a, b, c`... You can find out for sure by running:

```
> dmesg | less
```

on a command line. You can search for `hdx` lines in the output. If you have a SCSI cdrom, the device will be in the form `/dev/sdx` where `x` is the appropriate letter. Once you have found the appropriate device, you enter the following on a command line:

```
> mount /dev/hda /mnt/cdrom
```

Where `/dev/hda` should be replaced by the actual device. You should now be able to view the contents of the cdrom by typing:

```
> ls /mnt/cdrom
```

1.2 Running the install program

Now that the cdrom is mounted, you need to move to the cdrom's directory. Do this by typing:

```
    > cd /mnt/cdrom
```

At this point, if you type `ls` you should be able to see the files included on the cdrom.

Run the install script by typing:

```
    > ./install
```

If you run the install program as the `root` user, the programs will be installed in the appropriate system directories. If you install as another user, you will be asked where you would like to install the programs, with the default being the `bin` directory within your home directory. If this is ok, simply press the return key. Otherwise, specify the directory where you want the programs to be placed. Make sure that this directory is on your execution path.

The binaries installed if you are not the `root` user are statically linked. If you would like to use binaries that use shared libraries, you can install these yourself from the cdrom. All of the programs and libraries are included in tar files.

After installing the programs you will be asked if you would like to install the JX libraries. These libraries will allow you to write your own graphical programs in X. If you do want to install them, you will be asked for the directory where JX should be placed.

1.3 Installing by hand

You can install everything on the cdrom without using the install program . Binaries of the programs for Linux-Intel systems are in the `programs` directory. The `RPMS` directory also contains rpms of the binaries if you are running a system that supports them (like RedHat). These rpms can be installed by typing:

```
> rpm -i <program-name>.rpm
```

where `<program-name>` is the program that you wish to install. Finally, the JX libraries can be found in the source directory. These are tar files, and can be untarred wherever you want them to be. This is done by typing:

```
> tar xzf <tar-file-name>.tgz
```

where `<tar-file-name>` is the name of the source you want to untar. After untarring the source, the contents must be compiled before they can be used. Do this by running `make` in the `JX` directory and following the instructions that are printed.

1.4 Setting environment variables

There are times when you will need to set your systems environment variables. The method for doing this depends on the shell that you use. At the command line type:

```
echo $SHELL
```

This will tell you if you are using `/bin/tcsh` or `/bin/bash` as your shell.

If you are using `/bin/tcsh`:

```
setenv VARIABLE VALUE
```

If you append this to the file `~/.cshrc`, the variable will be set automatically every time you log in.

If you are using `/bin/bash`:

```
export VARIABLE=VALUE
```

If you append this to the file `~/.bashrc`, the variable will be set automatically every time you log in.

Chapter 2

Using Code Crusader

2.1 Getting started with Code Crusader

This tutorial will walk you through the steps required to create and run the famous Hello World program in C++ using the GNU C++ compiler. The basic steps are the same for other compilers and languages.

2.1.1 Creating the project

- Select "`New project...`" from the `File` menu to get the `Save File` dialog.
- Go to the directory where you want to save the project files. (usually a new, empty directory)
- Specify the name of the project as `hello_world.jcc`

- Click the `Save` button.

This creates the project file `hello_world.jcc` along with the text files `Make.header` and `Make.files`. You will learn how to edit `Make.header` later in this tutorial. `Make.files` will be automatically rewritten for you based on the contents of the Project window. These two files will be used by makemake to create `Makefile` which will in turn be used to compile your program.

2.1.2 Configuring the project

Since this is a C++ project, you need to link the standard C++ library to your program:

- Click the Config button in the upper right corner of the Project window to open `Make.header`.
- Add `-lstdc++` in front of `-lm` on the line that begins with `LOADLIBES`.

You need to learn at least the basics of GNU Make (you can learn more about this at http://www.cl.cam.ac.uk/texinfodoc/make_toc.html) in order to fully understand the contents of `Make.header`. You do not have to do this in order to complete this tutorial, however.

2.1.3 Creating your program

- Select "`New text file`" from the `File` menu to get an editor window.
- Type the following text into the editor window:

```
#include <iostream.h>

int main()
{
    cout << \"Hello world!\" << endl;
}
```

- Select "`Save`" from the `File` menu and save the file as `hello_world.cc`

Notice that context-sensitive highlighting became active once you saved it. This is because the suffix "`.cc`" tagged it as a source file.

2.1.4 Building the project

- Select "`Add to hello_world.jcc`" from the `Project` menu in the Editor window.
- Select "`Run (using hello_world.jcc)`" from the `Project` menu in the editor window.

The first window shows the result of compiling and linking your program. The second window shows the result of running your program. The program's name is by default `a.out`. This can be changed by selecting "`Edit build configuration...`" from the project window's `Project` menu. The `Build Configuation` dialog box lets you set the program's name. If you do change the name, make sure to change the name in the `Run Settings` dialog box as well (see Section 2.7.3).

2.1.5 Templates

Project templates The "Save as template..." item on the File menu in the Project window lets you create a template from the current project. It will save all the project settings, including Make.files, Make.header, the search paths, the Make and Run settings, and the C preprocessor configuration. It will also save the contents of all the files in the project that are in the project directory or in directories below the project directory. (i.e. files that are specified as `./something`) These files will be recreated automatically when the project template is used. When creating a template, you can add other files to the project window as well, but it usually only makes sense to include files relative to the root directory (`/`) since these will always be there.

Project templates must be stored in `~/.jxcb/project_templates/`. When you create a template, the Save File dialog will automatically display this directory. When creating a project, after you specify where to create the project, Code Crusader will automatically let you choose a template from this directory, if the directory exists.

Project wizards There are actually two different types of files that can be placed in the Code Crusader project template directory. The first is the type discussed above. The second type is a wizard interface. A wizard is a program that will create a customized project after asking you for the relevant information. The format of a wizard interface file is:

```
jx_browser_project_wizard
0
command to run the wizard program
```

The zero on the second line is the version number. The command on the third line can run an arbitrary program, but it must be a single command, not a sequence of commands separated by semicolons. (For this case, use a separate shell script.) In the command, `$path` will be replaced by the directory in which the user wants to create the project, and `$name` will be replaced by the name of the project.

The wizard must notify Code Crusader when it is finished by invoking Code Crusader (jcc) with the full path and name of the project file that has been created. (i.e. just as if the user had requested to open the project via the command line) If an error occurs in the wizard, it is the wizard's responsibility to report it.

Text file templates The "Save as template..." item on the File menu in the Text Editor window lets you create a template from the text in the window. You can use the template by selecting the "New text file from template..." item from the File menu in any window. This creates a new Text Editor window containing the template's text. You can then use Search & Replace and macros to fill in the template.

Text file templates must be stored in `~/.jxcb/text_templates/`. As with project templates, Code Crusader automatically displays this directory when you create or wish to use a text file template.

2.1.6 Creating a JX project

The easiest way to get started with a JX project, is to start with a basic program, like one of the tutorials. We'll start with the first tutorial, so we'll create a directory in the JX `programs` directory named `hello`. Inside this directory we'll create a directory named `code`. All of the source from the first tutorial should be copied into the `code` directory that we just created.

We then need to copy a JX `Make.header` template into our `hello` directory. This template can be found in JX's `doc` directory. This template sets up the make variables needed by a JX program.

So far, here's what we've done:

- `cd JX-x.x.x/programs` (where `x.x.x` is the current JX version number)
- `mkdir hello`
- `cd hello`
- `mkdir code`
- `cp ../../doc/Make.header_template Make.header`
- `cp ../../tutorial/01-HelloWorld/* code/`

Now we're ready to create a JX project. Start `Code Crusader` and select `New project` from the `File` menu. In the save dialog box, change the directory to your `hello` directory if it isn't already there, and save the project as `hello.jcc`.

The tutorial source now needs to be added to the project. Do this by selecting `Add files` from the `Source` menu. In the choose file dialog, change directories to your `code` directory, and by holding down the shift key, select all of the ".cc" files by clicking on them and then clicking on the `Open` button. After the dialog closes, the two source files should be visible in the project window under a group titled `New group`.

The project now needs to be configured. First select `Edit project configuration...` from the `Project` menu. In the `Project Configuration` dialog box, select the check box to generate the `Make.files`, change the target name to `./hello`, and add `${LIB_DEPS}` to the dependencies input field. Make sure to include the "./" before the name, or your system may not know where to find the program once it is built. The "./" specifies that it is in the current directory, and not somewhere on the path.

Project setup steps:

- Select `New Project` from the `File` menu
- In the Save dialog cd to your `hello` directory and save the project as `hello.jcc`
- Select `Add files` from the `Source` menu
- Select all of the "*.cc" files and click the `Open` button
- Select `Edit project configuration...` from the `Project` menu
- In the dialog, select the generate Make.files check box, change the target name to `./hello` and add `${LIB_DEPS}` to the dependencies input field.

The final step in setting up the project, is to adjust the `Make.header` file, by clicking on the `Config` button in the upper right corner of the project window. In the Make.header file, change all instances of the word myprog to hello, and, if you desire, change all the instances of the word MYPROG to HELLO. Both of these changes can be made by the Find dialog box, but you will need to make sure to unselect the Ignore case check box.

You are now ready to compile and run the program. You can do this by clicking on the `Run` button in the project window's tool bar.

2.2 Viewing differences between files

The File Differences dialog lets you specify the two files to compare and the styles used to indicate the differences between them. Text that is only in the first file is displayed using the top Style menu, and similarly for text that is only in the second file. Text that is common to both files is displayed using the "Shared text style" menu. The Choose buttons have the shortcuts Meta-1 and Meta-2, respectively.

Once the text is displayed, you can edit it and then save it. You can quickly find the differences by using the Differences menu. If you change one of the files, you can recalculate the differences via the "Redo diff" button.

You can get the File Differences dialog via the "Compare files..." item on the Search menu in the Text Editor window and the Project menu in the Project and Symbol windows and the C++ and Java Class Tree windows.

A command line option is provided so you can alias "`diff`" to "`jcc --diff`" whenever you are running X. This normally requires two files. If you only specify one file, it assumes you want to compare file˜ and file, unless the file name begins with #, in which case it assumes you want to compare it with the original. If you specify a file and a directory, it assumes that you want to use the file with the same name in the second directory.

2.3 Editing text

Code Crusader is designed to resemble most full feature development environments. All the commands are in the menus, and most are the standard commands that all text editors support.

By default, Code Crusader follows the Macintosh clipboard model. This means that text is not automatically copied when it is selected. Instead, it is only copied when you use the "`Copy`" item on the `Edit` menu or `Meta-C`. The main advantage of this model is that paste can then replace the selected text. If you prefer the X clipboard model, the `Editor Preferences` dialog lets you use this instead. Either way, the clipboard is persistent, so if you copy text in an editor window and then close the window, you can still paste the text later on.

When a file name in the `Files` menu is red, it means that the file has been modified and needs to be saved. This is also shown by drawing three asterisks to the left of the window title.

Code Crusader can read UNIX, Macintosh, and DOS/Windows text files. Each file's format is indicated on the "`File format`" submenu on the `File` menu. To convert a file to a different format, simply select the desired format from this menu and then save the file.

2.3.1 Searching for text

Enter the text for which to search in the "Search for" input area. The right "Find" button searches for the next match and selects it, while the left "Find" button searches for the previous match.

If "Ignore case" is checked, the search will not be case-sensitive. If "Wrap search" is checked, the search will wrap around if it hits the beginning or end of the text and will continue from the end or the beginning, respectively. If "Entire words only" is checked and the search text is also entire words (e.g. "hello" or "hi there", but not "this?"), the search will ignore matches that are part of other words. As an example, "and" will not match "sand".

If "Regex search (egrep)" is checked, the search text will be interpreted as a regular expression. The "Ref" button displays a short reference summary of regular expression syntax. If "Treat as single line" is checked, the regex will be allowed to match across multiple lines. Otherwise, each match will only be allowed to extend until it reaches a newline character.

Clicking on the down arrow next to the "Search for" input area displays a menu of the last 20 search patterns. Selecting one copies it to the input area.

The right "Find" button has the shortcuts Return and Meta-G. The left "Find" button has the shortcuts Shift-Return and Meta-Shift-G. The latter shortcuts are the same as in the Text Editor window.

2.3.2 Replacing text

Enter the text with which to replace the currently selected match in the "Replace with" input area. The "Replace" button replaces the selected text if it matches the current search text. The "Replace & Find" buttons do the same and then search for and select the next or previous match. If "Wrap search" is checked, the "Replace all" buttons search for all matches in the specified direction starting from the beginning or end of the text and replace each one. Otherwise, they search for all matches starting from the current caret position or selection and replace each one, stopping at the beginning or end of the text.

If "Regex replace" is checked, the replace pattern will be interpreted as a regular expression. Note that the shortcut for toggling this checkbox is Meta-Shift-X, because Meta-X is the shortcut for "Regex search."

If "Preserve case" is checked, the case of the replace text will be adjusted to match that of the matched search text, when possible. This is primarily useful when performing a case-insensitive search. The rules for when the case is adjusted are as follows:

1. If the texts have the same length, the case of each character is adjusted separately. For example, matching "aBc" and replacing with "xyz" produces "xYz".

2. Otherwise, the first letter's case is adjusted, and then the rest of the letters' cases are adjusted if all but the first letter in the matched text have the same case. For example, matching "aBC" and replacing with "wxyz" produces "wXYZ", while matching "AbC" produces "Wxyz".

Clicking on the down arrow next to the "Replace with" input area displays a menu of the last 20 replace patterns. Selecting one copies it to the input area.

The "Replace" button has the shortcut Meta-=. The right "Replace & Find" button has the shortcut Meta-L. The left "Replace & Find" button has the shortcut Meta-Shift-L. These are the same shortcuts as in the Text Editor window.

2.3.3 Searching for text in multiple files

If "Search directory" is checked, the right "Find" button will search each text file that matches the wildcard filter in the specified directory. If you specify a relative path, it will be relative to the directory containing the active project. If "Recurse" is checked, subdirectories will also be searched. (Symbolic links are not followed to avoid infinite loops.)

If "Search files" is checked, the right "Find" button will search each text file in the list.

These two "Search ..." checkboxes can be turned on at the same time. The checkboxes on the Options menu apply to both.

The output window will display each line that contains a match, underlining the text that matches. (If a file in the list is binary, it will not be searched.) The search is performed in the background, so you can continue to work. The Search Results window displays a progress indicator to show how much has been done.

As soon as each result is added to the Search window, you can open the file to see the match by Control-double-clicking on the file name or by using the Matches menu.

If you only want a list of the files that contain matches, turn off the "Show text of each match" option on the Files menu. If you want a list of the files that do not contain matches, turn on the "List files that do not match" option.

The list of files acts just like all other lists in Code Crusader. You can search for a file by clicking on the list and then typing the first few letters in the name. A separate help section explains how to select more than one file at a time. Since the Return key triggers the "Find" buttons, you can open the selected files with Meta-Return. (Double-clicking also works, of course, though only for a single file.) The Backspace key removes the selected files from the list.

You can add files to the list by:

- using the "Add files..." item on the Files menu
- dragging them from any XDND compliant file manager (You can also drag files from this list to such file managers.)
- dragging files from the File List window (shown via the "Show file list" menu item in the Project, Symbol, and class tree windows)
- dragging classes from the class trees (This adds both the source and header file for C++.)
- use the –search option on the command line (e.g. jcc –search source/*.cc headers/*.h misc/README.*)

Since it can be quite a bit of work to specify exactly which files to search, "Save file set..." on the Files menu lets you save the list so you can use it again later by loading it via the "Load file set..." item.

2.3.4 Replacing text in multiple files

If "Search files" is checked, the right "Replace all" button will search each file in the list, open the ones with matches, and replace all the matches. This provides a second way to undo the changes, in addition to the "Cancel" button in the progress window, because you can simply close a file without saving it.

This option is only available if you are using the built-in editor.

2.3.5 Sharing search parameters between programs

Since many programs allow you to search through text, Code Crusader has been designed to allow the search parameters to be automatically shared with other programs via the XSearch protocol . The advantage of this is that you can use the "Enter search string" menu item in one program, the "Enter replace string" menu item in a second program, and then search and replace in a third program without ever having to use the clipboard to transfer any of the text. All the other options in the Search Text dialog are also shared, so you can configure a search in one program and then use it in any other program that supports the protocol.

2.3.6 Drag-And-Drop

When Drag-and-drop is enabled in the `Editor Preferences` dialog window, you can

- move text around inside an editor window by dragging it
- copy text inside an editor window by holding down the `Meta` key and dragging the text
- copy text to another editor window by dragging it
- move text to another editor window by holding down the `Meta` key and dragging the text
- display a dialog window asking what to do with the text by holding down the `Control` key and dragging the text

- copy text to any window that interprets middle click to mean paste (Unfortunately, you may lose the current contents of the clipboard. To get it to work with xterm, add "`XTerm.VT100.allowSendEvents: True`" to your `.Xdefaults` file.)

If the text will be copied when dropped, a small plus (`+`) will be displayed in the cursor. If the dialog window will be displayed when the text is dropped, a small question mark (`?`) will be displayed in the cursor.

Drag-and-drop has the disadvantage that that you cannot click and drag in the selected text to make a new selection. Some people find that this slows them down while editing. Holding down the `Meta` key toggles the Drag-And-Drop setting. This allows you to get the other behavior when you need it. (Once you begin dragging, you can release the Meta key so that you can move the text. This trick also works in all text input fields.)

Drag-And-Drop can be cancelled by pressing the `Escape` key before releasing the mouse button.

The underlying X protocol is XDND
(`http://www.newplanetsoftware.com/xdnd/`).

2.3.7 Automatic indentation

When this feature is turned on in the Text Editor Preferences and you press `Return`, the Clean Paragraph Margin Rules discussed below are used to determine the current line's prefix. This prefix is then used to compute the prefix of the newly created line.

To copy only the whitespace, hold down the `Shift` key while pressing `Return`.

2.3.8 Cleaning paragraph margins

The "`Clean paragraph margins`" item on the `Edit` menu inserts newlines into the paragraph that the caret is in so that no line exceeds 75 characters. A paragraph is defined as a sequence of non-empty lines that have the same prefix. A line is empty if it has no characters or only prefix characters. If there is a selection, the item reads "`Clean margins for selection`" instead and performs the above operation on each paragraph touched by the selection. The "`Coerce paragraph margins`" item defines a paragraph as a sequence of non-empty lines, even if the prefixes are different. This is useful if you want to force a change in the prefix, because you only have to change the prefix of the first line by hand.

2.3.9 Defining paragraph prefixes

The `"Clean paragraph margins rules..."` item on the Preferences menu in the Text Editor windows lets you define what a paragraph prefix is. You can define as many different sets of rules as you wish, and you can share them with others by using the "Load" and "Save" buttons. The File Type dialog lets you choose which set of rules to use for each file.

Each rule has three parts:

First line prefix	regular expression that matches the prefix of the first line of the paragraph
Rest of lines prefix	regular expression that matches the prefix of the rest of the lines of

	the paragraph
Replace prefix with	replace expression from which the prefix of the rest of the lines of the paragraph is calculated

The rule for treating only whitespace as a prefix is built in, so you only have to specify rules for other patterns. You also do not need to use the up caret (ˆ) anchor to specify that the match must occur at the beginning of the line because Code Crusader does this automatically.

A simple example is the single rule required for email:

1. `[[:space:]]*(>+[[:space:]]*)+`
2. `[[:space:]]*(>+[[:space:]]*)+`
3. `$0`

The First and Rest patterns are the same because all lines begin with some mix of brackets (>) and spaces. The Replace pattern simply says to use the first line's prefix on all the following lines. "`Clean paragraph margins`" will therefore collect consecutive lines that begin with exactly the same pattern of brackets and spaces and rearrange the words to fit within the line width constraint set in the Text Editor Preferences. "`Coerce paragraph margins`" will first collect consecutive lines that begin with any pattern of brackets and spaces and will then change the prefix of each line to match the prefix of the first line before rearranging the words to fit within the line width constraint. Both will stop collecting if they find a line that contains nothing but brackets and spaces, because this is recognized as an empty line.

Here is an example:

Original text:

```
|---------------|
> This sentence does not
> fit on one
>>line.
```

Clean paragraph margins result:

```
|---------------|
> This sentence
> does not fit
> on one
>>line.
```

Coerce paragraph margins result:

```
|---------------|
> This sentence
> does not fit
> on one line.
```

A more complicated example is the two rules required for multiline C comments:

1. `([[:space:]]*)/((\*+[[:space:]]*)+)`
2. `[[:space:]]*(\*+/?[[:space:]]*)+`
3. `$1 $2`

and

1. `[[:space:]]*(\*+[[:space:]]*)+`

2. `[[:space:]]*(\*+/?[[:space:]]*)+`

3. `$0`

The first rule specifies that the prefix of the first line is `/*` followed by more asterisks and spaces. Following lines start with spaces and asterisks, like this:

```
/* This C comment
* spans two lines.
*/
```

The Rest pattern includes the possibility of a slash following the last asterisk in order to treat the line with `*/` as empty and force the collecting to stop.

The second rule specifies that one can also have a paragraph where all the prefixes are spaces and asterisks, like this:

```
/* This is the first line.
*
* This is a paragraph
* within a C comment.
*/
```

Note the Rest pattern again includes the possibility of `*/`, for the same reason.

Whenever you specify a rule where the first line's prefix is different from the rest, you must include a second rule which matches only the first rule's Rest pattern.

This not only allows sub-paragraphs, as in the C comment above, but it is also required by the algorithm that Code Crusader uses to find paragraph boundaries.

2.3.10 Balancing grouping symbols

The "`Balance closest grouping symbols`" item on the `Search` menu searches for the closest pair of grouping symbols (`()`, `{}`, `[]`) that enclose either the caret or the current selection and selects the text between this pair of grouping symbols. Repeated use of this menu item expands the selection outward to the next pair. The one special case is that, if you select a single grouping symbol, this menu item will attempt to balance that symbol instead of treating it like a normal selection. This feature is very useful when trying to check a complicated expression containing many nested pairs of parentheses. It is also useful for block indenting since it can instantly select all the text between a pair of braces.

The algorithm does not use a lexer to balance the grouping symbols. Because of this it will incorrectly beep when grouping symbols are used inside character and string constants in C source code. The reason for doing it this way is that it allows one to balance grouping symbols inside all other files, including other languages such as perl, sh, and FORTRAN. It makes a few mistakes there, too, but the ability to get it right 99% of the time in all files is better than getting it right 100% of the time only in C/C++ files.

2.3.11 Editing source code

If you have Exuberant ctags version 5.0 or later on your execution path, Code Crusader will automatically display an additional menu in the menu bar listing all the functions defined in the file. When you select a function from this menu, the text scrolls to display that function's definition. In the Preferences , you can choose whether to display the function names alphabetically or in the order defined in the file. With the latter configuration, a horizontal line is drawn on the menu to indicate the location of the caret in the file. If you want to momentarily display the functions in the opposite order, hold down the Meta key while opening the menu.

ctags version 5,0 works with Assembly, ASP, AWK, Bourne Shell, C, C++, Cobol, Eiffel, FORTRAN, Java, Lisp, make, Perl, PHP, Python, Scheme, TCL, and Vim. Newer versions may support additional languages. If you need support for another language, please contact Darren Hiebert.

2.3.12 Editing HTML

The `Web Browser Preferences` dialog window lets you specify the commands for viewing files and URL's in the browser of your choice. The default is to use Netscape:

- `netscape -remote "openFile(%f)"` - Used when you press the `View` button in the upper right corner of the window and when you `Ctrl-douhble-click` on a file that matches the "`Files to view in HTML browser`" preference described below.
- `netscape -remote "openURL(%u)"` - Used when you `Ctrl-double-click` on a URL.

You can use any other browser, if you can figure out the commands to execute. "`$f`" is replaced by the full path and file name when it is used in the command to view a file, while

"`$u`" is replaced by the URL name when it is used in the command to view a URL. If a character is preceded by a backslash, the backslash is removed and any special meaning is ignored. Thus, use `\$` to get a dollar character and `\\` to get a backslash character.

2.3.13 Mouse shortcuts

Left clicking obviously positions the insertion caret. Left clicking and dragging selects the characters that you drag across. Left double clicking selects the entire word under the cursor. If you then drag (click, release, click, drag), the selection will extend one word at a time. Left double clicking while holding down the partial word modifier (settable in the `Editor Preferences` dialog) selects the partial word under the cursor, and dragging extends the selection by one partial word at a time. Left triple clicking selects the entire line, and dragging extends the selection by a line at a time.

Holding down the `Shift` key while left clicking or simply right clicking extends the selection in the same way that the original selection was performed. i.e. If the original selection was one word at a time, the extended selection will also be one word at a time. Middle clicking pastes the current contents of the clipboard at the position where you click. If drag-and-drop is turned on in the `Editor Preferences`, left clicking on the current selection and dragging lets you drag the selected text instead of selecting text.

2.3.14 Really cool mouse shortcuts

- Button in upper right corner of window containing source or header file to open the header or source file, respectively (or press `Meta-Tab`)

- Button in upper right corner of window containing HTML file to view it in the browser of your choice (see above)
- `Ctrl-left-double-click` on a file name to automatically find and open it.
- `Ctrl-left-double-click` on a URL to open it in the browser of your choice. (see above)
- `Meta-double-click` on a symbol name to find it in the symbol database and class trees.

The double click shortcuts activate when you release the mouse button, so you can select more than a single word. This is useful for strings like "std::ios".

2.3.15 Key shortcuts

The arrow keys obviously move the caret around inside the text. Holding down the Control key makes the up and down arrows move the caret to the top and bottom of the visible text, respectively, while the left and right arrows move the caret by one word at a time. Control-up/down is useful for scrolling the text by one line without having to use the scrollbar: simply press Control-up/down-arrow and then up/down. It is also useful for bringing the cursor to the visible text after scrolling.

Holding down the partial word modifier key (settable in the Preferences dialog) makes the left and right arrows move the caret by one partial word at a time. As an example, this lets you move around inside both `GetFileName` and `get_file_name`.

Holding down the Meta key makes the up and down arrows move the caret to the beginning and end of the file, respectively, while the left and right arrows move the caret to the beginning and end of the current line, respectively. In addition, Meta-left-arrow is smart enough to move the caret to the beginning of the text on the line. Pressing it a second time moves the caret to the beginning of the line. (This is configurable in the preferences.)

Holding down the Shift key while using the arrow keys selects text. This works even in conjunction with the Meta, Control, and partial word modifier keys.

The backspace key on the main keyboard deletes the character to the left of the caret. The delete key on the keypad deletes the character to the right of the caret.

If automatic indentation is turned on, Shift-Return maintains the current indentation but ignores the line prefix.

Meta-comma activates the line number display so you can jump to a different line. Meta-Shift-comma activates the column number display so you can jump to a particular column on the current line. Pressing return takes you to the line or column that you have entered and returns you to the text. Pressing Shift-return in the line number display takes you to the line and activates the column number display.

2.3.16 External scripts

The "`Run script`" item on the `Edit` menu allows you to paste the output of an arbitrary script into an editor window at the caret position. If you select text before running the script, this text is fed to the script via `stdin` and is replaced by the script's output.

The following strings are replaced when they are used in the command:

- $f - full path and file name
- $p - path
- $n - file name
- $l - line number

If a character is preceded by a backslash, the backslash is removed and any special meaning is ignored. Thus, use `\$` to get a dollar character and `\\` to get a backslash character.

2.3.17 Preferences

The following options can be changed in the Editor Preferences dialog window:

- `Font & Size` - These menus lets you change the font used in the editor windows. The `Size` menu is only active if you choose a font from the second section of the `Font` menu.
- `Change color` (5 buttons) - These buttons let you change the color of the text, the background, the insertion caret, the background of text that is selected, and the frame that outlines the selected text when the window does not have focus. The sample text above the buttons is displayed using the colors that you select. Note that you have to edit the context-sensitive highlighting colors separately to make them visible against the background color that you choose.
- `Spaces per tab` - Adjusts the spacing of the tab stop.

- `Clean right margin line width` - Adjusts the width enforced by the "`Clean right margin`" item on the `Edit` menu.
- `Undo depth` - Determines the number of actions that can by undone in each editor window. Reducing the depth reduces the memory used by each editor window.
- `Allow Drag-And-Drop` - If this is checked, you can drag text around inside each editor window and also drag text from one editor window to another, as described above.
- `Automatically indent new lines` - If this is checked, the caret will be placed below the beginning of the text on the previous line after you press return.
- `Tab inserts spaces` - If this is checked, pressing the tab key will insert spaces to match the tab width set in the "Spaces per tab" input field. This has the advantage that the text will look the same in other programs that use a different tab width, but has the disadvantage that the position will not be updated if you later change the value in the "Spaces per tab" input field. To remove all the spaces inserted by a tab, use Undo instead of Backspace.
- `Tab completes words and activates macro` - If this is checked, pressing the `Tab` key after typing a partial keyword will complete the keyword, and pressing the `Tab` key after typing the name of a macro will activate it. If this is not checked, you can still use `Ctrl-Space` to perform the same actions.
- `Meta-left-arrow moves to front of text` - If this is checked, `Meta-left-arrow` will move the caret to the front of the text on the line. Using

it again will move the caret to the front of the line. Otherwise, `Meta-left-arrow` will move the caret to the front of the line the first time it is used.

- `Automatically copy text when it is selected` - If this is checked, any text that you select will automatically be copied to the clipboard. This applies to input fields in dialogs as well as editor windows.

- `Balance grouping symbols while typing` - If this is checked and you type `)`, `}`, or `]` in a source window, the matching `(`, `{`, or `[` will flash to show you where it is.

- `Scroll if outside view` - If this is checked, the text will scroll if the balancing grouping symbol is not visible in the window. After flashing the symbol, the text will scroll back to where it was so you can keep typing.

- `Beep if symbol not balanced` - If this is checked and you type an unbalanced `)`, `}`, or `]`, Code Crusader will beep.

- `Alphabetize function menu in source windows` - If this is checked, the function names displayed in the Functions menu in source windows will be alphabetized. Otherwise, they will be displayed in the order defined in the file.

- `Use small font in Functions menu` - If this is checked, the function names displayed in the Functions menu in source windows will be displayed in a smaller font so that more items fit on the screen.

- `Create backup file` - Determines whether or not the original contents of the file will be copied to a backup file (e.g. foo.cc goes to foo.cc˜) before the new contents are saved.

- `Only create backup file if it doesn’t already exist` - If this is

  checked, a backup file will only be created if one does not already exist. Otherwise, a backup file will be created every time after you open a file and save the first changes. This checkbox is disabled if "`Create backup file`" is not checked.

- `Open complement file on top of existing window` - Normally, when one opens a source or header file, the preferences are used to place the window. When this is checked, however, the window is placed on top of its complement file, if this is already open.

- `Allocate space for *** in window title` - The fvwm window manager decides the size of an iconified window when one iconifies it for the first time and does not change it after that, even if the window title changes. This can be annoying since Code Crusader prepends `***` to the window title after you modify the text. If this option is checked Code Crusader will prepend spaces to the initial window title so that the `***` won’t affect the window title’s width. This option should not be used with `fvwm95` because it allocates space in the task bar in a different way.

- `Modifier to move by partial word` - This lets you choose which modifier key should activate "`move by partial word`." Section 2.10.10 explains how to make a key map to `Mod2` to `Mod5`. This is configurable because hardward constraints force some of `Mod2` to `Mod5` to be used for modifiers like `NumLock`, and this varies from system to system.

- `Home/End move to beginning/end of line` - If this is checked, the Home / End keys move the caret to the beginning/end of the line instead of scrolling to the top/bottom of the text. To compensate, Control-Home/End scrolls to the top/bottom of the text. (Please read the FAQ to understand why I hate this option.) All the other shortcuts still work.
- `Caret follows when you scroll` - If this is checked, the caret follows along when you scroll the text. You will also lose the selection when it scrolls out of view. If you use scrolltabs, then turning this option off makes it easier to return to the position where you are editing.

2.3.18 Safety features

Code Crusader has several features to protect you from accidental crashes of both the X Server and Code Crusader itself:

- As mentioned in the `Editor Preferences` section above, Code Crusader can create a backup of the previous contents whenever you save a file. (This is turned on by default.)
- Each file that has unsaved changes is safety saved every 30 seconds in a # file. (e.g. "`hello_world.c`" is safety saved in "`#hello_world.c#`")

- If the X Server crashes, every unsaved file is safety saved again to catch the very latest changes.
- If Code Crusader itself crashes on an `assert()`, every file is safety saved in a ## file. (e.g. "`hello_world.c`" is safety saved in "`##hello_world.c##`") The original # file is not modified because, when the program itself crashes, data may have become corrupted, so the # file may be better than the ## file.
- When a file is opened, Code Crusader automatically checks to see if a newer safety saved version exists, and asks if you would like to open it.

2.3.19 Specifying File Types

The `"File types..."` item on the `Preferences` menu lets you define how you want to deal with each text file that you open. A file can be recognized either by the file name suffix (e.g. .txt) or a regular expression that matches the beginning of the file's contents. Once the file is recognized, the attributes that you specified in the table are attached to the associated Editor window.

If the text in the `"Suffix / Regex"` column begins with an up caret (ˆ), it is interpreted as a regular expression. Otherwise, it is assumed to be a file suffix. The file's type is recalculated every time you save it, so a type based on the content will not be recognized in a brand new file until the file is saved for the first time.

Clicking in the "Type" column displays a fixed menu of choices that are linked to built-in features:

- Assembly

  - Label menu
  - Included in symbol browser
  - Included in Make.files
  - Compilable via the Project menu

- ASP (http://support.microsoft.com/support/default.asp?PR=asp)
  (Microsoft's Active Server Page scripting language)

  - Functions menu
  - Included in symbol browser

- AWK

  - Functions menu
  - Included in symbol browser
  - Bourne Shell
  - Functions menu
  - Keyword completion
  - Included in symbol browser

- C/C++ source

  - Context-sensitive highlighting
  - `Function` menu
  - Included in `Make.files`
  - Compilable via the `Project` menu
  - Provides "`Header`" button in the editor window

- C/C++ header
  - Context-sensitive highlighting
  - `Function` menu
  - Provides "`Source`" button in the editor window
  - Parsed by the C++ class tree if it is a file suffix type
- Cobol
  - Paragraph menu
  - Included in symbol browser
  - Included in Make.files
  - Compilable via the Project menu
- Eiffel
  - `Function` menu
  - Included in `Make.files`
  - Compilable via the `Project` menu
- FORTRAN
  - `Function` menu
  - Included in `Make.files`
  - Compilable via the `Project` menu
- HTML

  - Context-sensitive highlighting
  - Provides "`View`" button in the editor window

- Java source

  - `Function` menu
  - Keyword completion
  - Included in symbol browser
  - Included in `Make.files`
  - Compilable via the `Project` menu

- Java archive

  - Included in Make.files

- Lisp (http://www.lisp.org/)

  - Functions menu
  - Included in symbol browser

- Modula-2 module

  - Included in Make.files
  - Compilable via the Project menu
  - Provides "Interface" button in the Text Editor window

- Modula-2 interface

  - Provides "Module" button in the Text Editor window

- Modula-3 module
  - Included in Make.files
  - Compilable via the Project menu
  - Provides "Interface" button in the Text Editor window
- Modula-3 interface
  - Provides "Module" button in the Text Editor window
- Perl (http://www.perl.com/)
  - Subroutine menu
  - Keyword completion
  - Included in symbol browser
- PHP (http://php.net/)
  - Functions menu
  - Keyword completion
  - Included in symbol browser
- Python (http://www.python.org/)
  - Functions menu

  - Keyword completion
  - Included in symbol browser

- Ratfor
  - Functions menu
  - Keyword completion
  - Included in symbol browser
  - Included in Make.files
  - Compilable via the Project menu
- Scheme
  - Functions menu
  - Included in symbol browser
- TCL (http://www.scriptics.com/)
  - Functions menu
  - Included in symbol browser
- Vim (http://www.vim.org/)
  - Functions menu
  - Included in symbol browser
- Other source

- Included in `Make.files`
- Compilable via the `Project` menu

* Static libraries
  - Included in `Make.files`
* Shared libraries
  - Included in `Make.files`
* Documentation
  - Automatically opened if the equivalent C/C++ source file cannot be found
* Other text
  - Catch-all to allow you to set the other attributes
* Binary
  - Opens the file in the Binary Editor instead of the Text Editor.
* External
  - Opens the file by running the associated Open Command instead of creating a Text Editor window. This is especially useful for graphical window layouts and images.

Clicking in the "`Macro set`" column displays a menu of all the macro sets defined in the associated dialog. Similarly, clicking in the "`CPM rules`" column displays a menu of all the `Clean Paragraph Margins` rule sets defined in the associated dialog. Clicking in the "`Word wrap`" column toggles the value between `Yes` and `No`. This controls whether or not lines are wrapped if they extend beyond the right edge of the editor window. Holding down the `Meta` key while clicking the mouse toggles the value and sets the entire column to the new value.

If the file type is `External`, the "`Open command`" column will be editable. When you use `$f` in this command, it will be replaced by the full path and name of the file to be opened.

2.3.20 Character Actions

Character Actions provide a way to automatically type a sequence of characters after you type a particular character. As an example, when writing C code, you could have the editor automatically type return after you type a semicolon.

2.3.21 Macros

Macros provide shortcuts for typing sequences of characters. A macro is activated by typing the shortcut and then pressing `Tab` or `Control-Space`. As an example, when writing C code, you could type `#"` and then `Tab` could activate the macro that converts this into `#include ""`.

Keyword completion provides the complementary functionality of completing words regardless of how much you have already typed. This is checked after macros so that macros can end with ordinary characters.

You can turn off the special behavior of Tab in the Text Editor Preferences dialog. To insert a Tab character without triggering the macro, hold down the Shift key.

2.3.22 Specifying actions and macros

You can create sets of actions and macros in the dialog obtained from the "`Macros...`" item on the `Preferences` menu in each editor window.

In this dialog, the left columns contain the action or macro, while the right columns contain the keystrokes to perform. In the right column, all characters except for backslash (`\`) and dollar (`$`) translate directly to keystrokes. Backslash indicates a special character as follows:

- \n - newline (return)
- \t - tab
- \b - backspace
- \f - forward delete
- \l - left arrow
- \r - right arrow
- \u - up arrow

- \d - down arrow
- \\ - backslash

If any other character is preceded by a backslash, the backslash is simply removed.

Dollar is used to indicate a variable or a command to be executed:

- $(...) - execute command inside parentheses The output of the command is treated as a sequence of keystrokes. This construct is nestable.
- $f - full path and file name
- $p - path
- $n - file name
- $l - line number of caret or top of selected text

To get a right parenthesis inside a command, use \). Similarly, to get a dollar symbol, use \$.

The characters typed by actions and macros do not trigger other actions or macros because it is very hard to avoid infinite loops and because it is much clearer if each keystroke script is self-contained.

Examples

For C/C++, the following actions can be useful:

; -> \n	Starts a new line every time you type a semicolon.

{ - >\n}\u\n	This creates the closing brace and then puts the caret on a blank line between the braces.

For C/C++, the following macros can be useful:

/* -> */\l\l\l	Closes the comment (note the two spaces in front of */) and puts the caret inside so you can type the text.
#" -> \binclude ""\l	Produces `#include ""` and puts the caret between the quotes so you can type the file name.
#< -> \binclude <>\l	Produces `#include <>` and puts the caret between the brackets so you can type the file name.

For HTML, the following macros can be useful:

<a -> href="">\l\l	Fills in the anchor tag (note the space in front of `href`) and puts the caret between the quotes so you can type the URL.
<b -> ></b>\l\l\l\l	Creates the closing boldface tag and puts the caret between the tags so you can type the text.
<ul -> >\n<li>\n</ul>\u	Creates a single item and the closing list tag and puts the caret after the item tag so you can type the text.

2.3.23 Keyword completion

Keyword completion allows you to type the beginning of a word and then press `Tab`, `Control-Space`, or `Control-Shift-Space` to complete the word automatically.

As an example, when writing C code, you could type cont and then `Tab` would complete it to give continue. (Macros provide the complementary functionality of automating specific, repetitive typing tasks. These are checked before keyword completion so that they can end with ordinary characters. `Control-Shift-Space` is provided as a trigger for keyword completion so macros can be ignored if necessary.)

You can turn off the special behavior of `Tab` in the Text Editor Preferences dialog. To insert a `Tab` character without triggering the completion mechanism, hold down the Shift key.

If the text matches more than one keyword, completion will fill in as many characters as possible. As an example, in `C`, `r` matches `register`, `return`, and `reinter-pret_cast`, but they all start with re, so pressing `Tab` after `r` will add `e`. You can then type another character and press `Tab` again. (e.g. typing `t` would complete the word to return)

If you press `Tab` and nothing is added, you can press `Tab` again to display a menu of all the matching completions. The first ten choices have the shortcuts 1, 2, ..., 9, 0, and the next twenty-size choices have the shortcuts a, b, ..., z, so you can avoid using the mouse most of the time. To close the menu without selecting anything, press `Tab` again. To insert a tab character via the menu, select the first item, which has the shortcut ‘ (single back quote).

The words that can be completed are the reserved keywords for the language (see below), any words that you enter into the bottom list in the corresponding dialog for context-sensitive highlighting, and all class names and methods in all open projects.

Bourne Shell	All the reserved keywords are included by default.
C/C++	All the reserved keywords from the latest draft standard are included by default.
Eiffel	All the reserved keywords are included by default.
FORTRAN	All the reserved keywords and intrinsic function names are included by default.
HTML	Only the names of special characters are included by default because it is a feature of HTML that there is no fixed list of tags, and the common tags are all very short. If you frequently use any long tags, simply enter them into the context-sensitive highlighting dialog. Usually, it is more convenient to create a macro for each tag because this can initialize the attributes at the same time.
Java	All the reserved keywords and special method names are included by default.
Perl	Only the basic reserved keywords are included by default since the list of available functions is nearly infinite.
PHP	Only the basic reserved keywords are included by default since the list of available functions is nearly infinite.
Python	All the reserved keywords are included by default.

2.3.24 Context-sensitive Highlighting

Code Crusader provides context-sensitive highlighting for the following file types:

- C/C++
- HTML

The "... styles" items on the Preferences menu in the Text Editor windows allow you to turn highlighting on or off and to set the styles to be displayed.

The top list specifies the styles for the different types of text. As an example, the "Comment" item for C/C++ specifies how comments will be displayed in source and header files.

The bottom list specifies the styles for specific words that override the top list. As an example, if you created an item "#pragma" in the C/C++ window, then all #pragma directives would use this style instead of the style for "Preprocessor directive" in the top list. (The words in this list are also used for keyword completion.)

To change the style of an item, hold down the Meta key, click on the item, and select from the menu that appears. To change the style of several items, select them and then Meta-click on any one of them. (There are separate sections devoted to selecting multiple items and the Color Chooser dialog.) Note that fonts like 6x13 cannot be displayed in italic because there is no italic version of these fonts installed on most systems. Courier can be displayed in italic, however.

You cannot change the items in the top list, only their styles.

To add an item to the bottom list, click the "New" button. To remove items from the bottom list, select them and click the "Remove" button. To change the text of an item in the bottom list, double click on it.

Notes for C/C++ The items in the bottom list must be one of the following types:

- Identifier
- Reserved keyword
- Numerical or character constant
- Preprocessor directive
- Operator (e.g. +=) Delimiter (parentheses, brackets, braces, comma, semicolon)
- Trigraph
- Respelling

Preprocessor directives are specified by their name (e.g. #include) even though the style is applied to both the name and the arguments.

The last item in the top list is "Detectable error" because only a few of the simplest errors can be found by a lexer:

- Illegal character and octal constants
- Unterminated string constants
- Unterminated C comments

- Illegal characters (e.g. lone #)

Notes for HTML The items in the bottom list must be one of the following types:

- Tag
- Name of scripting language
- Special character

Tags are specified by their name (e.g. font) even though the style is applied to both the name and the arguments.

Special characters are specified by their name, excluding the leading &. (e.g. lt instead of &lt)

The last item in the top list is "Detectable error" because only empty tags (i.e. <>) can be found by a lexer.

Supporting other languages The context-senstitive highlighting API is very simple, so it is easy to add support for other languages once a lexer is written. This cannot be linked in dynamically, however, because it does require minor changes to other parts of Code Crusader, such as the preferences file and the Preferences menus. If you are interested in adding support for another language, please read the discussion on the Bazaar Projects (http://www.newplanetsoftware.com/jcc/projects.html) web page or contact jcc@newplanetsoftware.com.

Note that, in order to be compatible with Code Crusader, the code will have to be written in C++ and will have to use the JCore library (http://www.newplanetsoftware.com/jx/jcore.html).

Also, before suggesting support for FORTRAN, please realize that it is impossible to write a lexer for it. FORTRAN requires a full parser, which, while not impossible, is a lot more work. One ought to be able to write a lexer if one requires whitespace between symbols, however.

2.4 The Project Window

The project window is a convenient place to store all the files associated with a project and provides a foundation for the interface to UNIX make. The tutorial above (Section 2.1) provides an introduction to this interface.

Any file can be added to the project, not only a source file. To do this, use "`Add files`..." on the `Project` menu. When you select the files, you can also choose the type of path information to store. The default is relative to the project file's path, which makes it easy to transfer the project's directory structure to a different location or even a different machine. Once a file has been added to the project, its path can be changed by holding down the `Meta` key and double clicking on the file with the left button. You can also add a file that is open in an editor window to a project by using the "`Add to ...`" menu on the `Project` menu in the editor window. If you select a file in the project before using "`Add files...`", the new files will be inserted after the selection. If you select a group, the new files will be appended to the group.

To open a file from the project window, double click with the left button. To open the complement file of a C/C++ source or header file, double click with the middle button. If you have selected multiple files, you can either use the items on the `Project` menu or the shortcuts `Return` and `Meta-Tab`. These are the same bindings as in the class tree which is discussed.

If you add a library to the project and then try to open it, you will be asked to find the project file corresponding to the library. This makes it easy to open sub-projects from the main project window. These sub-projects are automatically built before building the main project. You can change the project file for the library and other options via the "Edit sub-project configuration" item on the Source menu.

As with all lists in Code Crusader, to find a group or a file that is visible, simply type the first few letters of its name.

Groups can be used to collect related files and hide them when you are not working on them. To rearrange files or groups, simply select them and drag them to where you want them to be. You can also drag them to other windows that support the XDND protocol, like the multi-file search list in the `Search Text` dialog. If you drag a group, it is like dragging all the files in that group. To change the name of a group, simply double click on it. To open or close a group, click on the triangle to the left of the name. You can also use the following key shortcuts:

- Left arrow - close selected groups
- Right arrow - open selected groups
- Meta-left - arrow close all groups
- Meta-right - arrow open all groups

2.4.1 Interface to GNU Make

The basic idea of the project interface is the following sequence of steps:

1. Add the source files to the project window.
2. Configure the project.
3. Configure the compiler and linker by editing `Make.header`.
4. Ask Code Crusader to build the project.
5. It builds the project associated with each library in the project, writes Make.files, and then invokes first the "Recalculate dependencies" command and then the "Make project" command in the Compile dialog.

The tutorial above (Section 2.1) provides a step-by-step introduction to this procedure. Once you understand this, you should study the `Compile` and `Run` dialogs (see Sections 2.7.2 and 2.7.3). These specify the commands used to compile, run, and debug your program.

The `Config` button in the upper right corner of the project window opens the `Make.header` file. The shortcut for this button is `Control-Tab`, similar to `Meta-Tab` for opening a header file.

As mentioned above, you can add libraries to the project. (static or shared, as specified in the `Edit File Types` dialog) These will be included in `Make.files` as dependencies. Unfortunately, make is not powerful enough to recognize `.so.<vers>`, so Code

Crusader doesn't recognize it either. To get a pure .so file, either create a symbolic link or place the version number in front of the .so suffix.

Note that the directories that you specify in the Search Paths dialog are not passed to the compiler because every compiler has a different way of handling this. You have to edit Make.header.

2.4.2 Configuring the project

The following options can be changed in the Project Configuration dialog:

- `Generate Make.files from files in Project window` - If this is checked, Code Crusader will generate Make.files for use with makemake.

- `Target name` - This is the name of the program or library to be built. It can also be a list of names separated by commas. All names will be built using the same set of source files, so this is mainly useful for generating both static and shared versions of a library.

- `Expression to append to list of dependencies` - This expression can do anything that make allows: evaluate variables, invoke functions, etc. This is useful for conditional or variable dependencies.

2.4.3 Preferences

The following options can be changed in the Project Preferences dialog:

- `Reopen text files when project is opened` - If this is checked, the files that were open when the project was closed will be reopened the next time you open the project. This does not apply to sub-projects opened during a build because these are always opened as silently as possible.

- `Update symbol database while compiling` - If this is checked, Code Crusader will update all affected inheritance trees while you build a project or compile a source file.

- `Beep after Make is finished` - If this is checked, Code Crusader will beep after the Make process is finished.

- `Beep after Compile is finished` - If this is checked, Code Crusader will beep after the Compile process is finished.

- `Path type to use when adding files via Drag-And-Drop` - If "Ask when files are dropped" is selected, the other options will be presented in a dialog when you drop files on a project. Otherwise, the selected path type will be used.

2.4.4 Choosing your text editor and binary editors

You can use the "`Choose external editors...`" item on the `Preferences` menu to set which text and binary editors to use when displaying files. For the text editor, the common choices are:

- built-in editor

- emacs:

  ```
  emacsclient %f
  emacsclient +%l %f
  (you must also run "M-x server-
  start" from within emacs)
  ```

- vi:

  ```
  xterm -title "%f" -e vi %f
  xterm -title "%f" -e vi +%l %f
  ```

For the binary editor, the choices are:

- emacs:

  ```
  If you install the full remote control package
  (not just emac-
  sclient), you can tell emacs to open a file
  via the hexl-find-file command.
  ```

If you install the full remote control package (not just `emacsclient`), you can tell emacs to open a file via the hexl-find-file command.

- vi:

  ```
  xterm -title "$f" -e vi $f
  ```

If a character is preceded by a backslash, the backslash is removed and any special meaning is ignored. Thus, use `\$` to get a dollar character and `\\` to get a backslash character.

The advantages of using the built-in editor are explained in the discussion above on editing text (Section 2.3).

2.5 The C++ Class Tree

The tree displays the inheritance relationships between the C++ classes that are found by parsing header files. Green indicates public inheritance, yellow is protected inheritance, and red is private inheritance. For the primary parent, black is used to indicate public inheritance to avoid too many distracting colors and make the protected and private inheritances stand out more.

Classes that are declared with `class` are drawn with black text. Classes that are declared as `struct` or `enum` are drawn with gray text. If the parent of a class is not found in any of the files that are parsed, a `ghost` class is created to indicate this. Ghost classes are drawn with a gray background.

To add classes to the tree, use the "`Add classes...`" item on the `Tree` menu. To update the current tree without quitting, use the `"Update"` item. The "`Edit file types...`" item lets you set the suffixes for source and header files. It is a good idea to add all the directories that contain source or header files, not just those that contain classes. This way, when Code Crusader searches for a file (e.g. when you use "`Open selection`" in an editor window), it is more likely to find it. (Note that `/usr/include` is always searched.)

To locate a particular class, press the space bar and then type the first few letters in its name. This search is case-insensitive and ignores the namespaces. To then open the source file, press Return. To open the header file, press `Meta-Tab`. You do not have to de-iconify the window before typing.

To locate all classes that implement a particular function, select "`Find function`" from the `Tree` menu and then enter the name of the function to search for. This is case sensitive. If a single function is found, its definition will automatically be displayed.

If you click on a class in the class tree window:

- Any button once (or type the first few characters of the name) - select the class
- `Shift` key + any button once - select/deselect the class without deselecting other classes
- Left button twice (or `Return`) - open source file for the class (opens documentation file if source can't be found)
- Middle button twice (or `Meta-Tab`) - open header file for the class
- Right button twice - open list of functions implemented by the class

There are also several shortcuts in the built-in editor that connect back to the tree.

2.5.1 Displaying the functions implemented by a class

To get a list of the functions that a class implements, either click on it and hold for a moment to display a menu or `right-double-click` to create a new window.

The function names are colored to indicate whether they are public (black), protected (brown, since yellow text on a white background is very difficult to read), or private (red).

Italic indicates a pure virtual function. Qt signals are indicated by an icon showing an arrow leaving an object. Qt slots are indicated by an icon showing an arrow entering an object.

The setting in the `Tree Preferences` dialog controls whether or not the menu and each new window also displays the functions inherited from base classes. If this is turned on, functions implemented by the class itself are boldfaced to make them easy to find. Changing the setting in the `Tree Preferences` dialog does not affect existing windows. In these windows, there is a menu item on the `Options` menu that controls the display.

To locate a particular function, press the space bar and then type the first few letters in its name. This search is case-insensitive. To then open the file containing the function's code, press `Return`. To open the header file, press `Meta-Tab`. You do not have to de-iconify the window before typing.

If you click on a function name in a function list window:

- Any button once (or type the first few characters of the name) - select the function
- Left button twice (or `Return`) - display source for the function
- Middle button twice (or `Meta-Tab`) - display prototype for the function
- Right button twice - search the tree for all classes that implement the function

When using the menu, the mouse button that was pressed determines the action, just like when double clicking in the window.

2.5.2 Preferences

The following options can be changed in the `Tree Preferences` dialog window:

- `Font size` - This menus lets you change the font used in the class tree and function list windows.
- `Show inherited functions in lists` - If this is checked, the popup menu and all newly created windows will display both the functions implemented by the class and the functions inherited from all base classes. The functions implemented by the class will be boldfaced to make them easy to find.
- `Raise window if search matches single class` - If there is only one match when you Meta-dbl-click in an editor window, it is usually less disruptive if only the source file containing what you are looking for is displayed. This will be the case if this item is not checked.
- `Automatically minimize MI link lengths` - If this is checked, the multiple inheritance link lengths will be minimized every time the class tree is changed. If this is not checked, you can use the "`Minimize MI link lengths now`" item on the `Tree` menu to do it when needed. This option is provided because the optimal algorithm is exponential so it can take a long time if you make heavy use of multiple inheritance.
- `Draw MI links` - Multiple inheritance links are much easier to follow if they are drawn on top of everything else. However, sometimes it is better to drawn them underneath to minimize the clutter.

2.5.3 Defining preprocessor symbols

Code Crusader can never automatically know which preprocessor symbols that are defined at compile time. In order to parse header files correctly, however, it does need to know the definitions of the symbols used in your class definitions. You can specify these using the "Preprocessor macros..." item on the Preferences menu.

As an example, the ACE networking library defines classes with

```
class ACE_Export class_name {...}
```

On Windows, ACE_Export evaluates to the compiler dependent stuff required to build a DLL. On UNIX, it evaluates to nothing, so it simply disappears. Code Crusader requires that you define ACE_Export to evaluate to nothing, because otherwise the parser will not recognize it as a class definition.

This implementation is not a true preprocessor. It can handle only simple names, not full macros with arguments. This is generally sufficient because only macros in the class declaration need to be defined, and as above, they usually only need to be removed. If this is not sufficient, please contact New Planet Software.

Code Crusader can, however, replace a macro that takes arguments with a fixed string. As an example, the `Qt` widget library defines the macro `Q_PROPERTY(...)`. This confuses Code Crusader's parser, but defining `Q_PROPERTY()` to be replaced by nothing (i.e. removed) is sufficient to allow Code Crusader to parse the rest of the file.

2.6 The Java Class Tree

The tree displays the inheritance relationships between the Java classes that are found by parsing header files. The names of interfaces, classes that are declared abstract, and classes that declare abstract functions are italicized. If an interface or class is not declared public, its name will be gray. If an interface or class is declared final, its name will be boldfaced. If the parent of a class is not found in any of the files that are parsed, a "ghost" class is created to indicate this. Ghost classes are drawn with a gray background.

To add classes to the tree, use the "Add classes..." item on the Tree menu. To update the current tree without quitting, use the "Update" item. The "Edit file types..." item lets you set the suffixes for source and header files. It is a good idea to add all the directories that contain source or header files, not just those that contain classes. This way, when Code Crusader searches for a file (e.g. when you use "Open selection" in a Text Editor window), it is more likely to find it.

To locate a particular class, press the space bar and then type the first few letters in its name. This search is case-insensitive and ignores the namespaces. To then open the source file, press Return. You do not have to de-iconify the window before typing.

To locate all classes that implement a particular function, select "Find function" from the Tree menu and then enter the name of the function to search for. This is case sensitive. If a single function is found, its definition will automatically be displayed.

If you click on a class in the Tree window:

- Any button once (or type the first few characters of the name) - select the class

- Shift key + any button once - select/deselect the class without deselecting other classes
- Left button twice (or Return) - open source file for the class (opens documentation file if source can't be found)
- Right button twice - open list of functions implemented by the class

There are also several shortcuts in the built-in editor that connect back to the tree.

2.6.1 Displaying the functions implemented by a class

To get a list of the functions that a class implements, either click on it and hold for a moment to display a menu or right-double-click to create a new window.

The function names are colored to indicate whether they are public (black), default (dark green), protected (brown, since yellow text on a white background is very difficult to read), or private (red). Italic indicates an abstract function.

The setting in the Tree Preferences dialog controls whether or not the menu and each new window also displays the functions inherited from base classes. If this is turned on, functions implemented by the class itself are boldfaced to make them easy to find. Changing the setting in the Tree Preferences dialog does not affect existing windows. In these windows, there is a menu item on the Options menu that controls the display.

To locate a particular function, press the space bar and then type the first few letters in its name. This search is case-insensitive. To then open the file containing the function's code, press Return. You do not have to de-iconify the window before typing.

If you click on a function name in a Function List window:

- Any button once (or type the first few characters of the name) -select the function
- Left button twice (or Return) - display source for the function
- Right button twice - search the tree for all classes that implement the function

When using the menu, the mouse button that was pressed determines the action, just like when double clicking in the window.

2.7 Compiling your program

2.7.1 Introduction to Compiling under UNIX

Development environments on the Macintosh (e.g. Metrowerks) and on Windows (e.g. Visual C++) hide the process of compiling the source files and building the finished program. They are able to do this because the project manager, compiler, and linker are all integrated into one system.

On UNIX, however, the compiler and linker are separate programs, and different people and different platforms (e.g. Linux vs Solaris) prefer to use different compilers and linkers. Thus, while Code Crusader and `makemake` can hide most of the work done by a project manager, it cannot hide the details of dealing with the compiler and linker.

If a C program is contained entirely within one source file and does not require any header files other than those in `/usr/include`, one can compile and link it with the following command:

```
gcc -o program_name source_file.c
```

A C++ program that is contained within one source file can be compiled and linked with:

```
g++ -o program_name source_file.cc
```

If you have GNU's version of `make`, you can use:

```
make source_file
```

in place of either of the above commands.

For a program that is spread across more than one source file, you want to learn how to use `makemake` and GNU's version of `make`. `makemake` is included as part of the Code Crusader package .

2.7.2 Compiling with Code Crusader

In order to compile anything, you must first create or open a project. You can create a new project by using the "`New project...`" item on the File menu

.The `Make Settings` window lets you specify three commands:

- recalculating which header files each source file depends on
- building the project
- compiling a single source file

The first is usually either `makemake` (which is included in the Code Crusader distribution) or `make depend`. The other two are normally calls to make with different arguments. You must also specify which directory to run the commands from.

The `Make` button runs the command to build the project. If the "`Save all before make`" checkbox is on, then all the editors will be saved first. If the "`Recalculate dependencies before make`" checkbox is on, then the command to recalculate dependencies will also be run first. The output of the command is displayed in an editor window, so you can use `Ctrl-double-click` to open a file (line numbers are also parsed, e.g. `/usr/include/stdio.h:35`) and also edit and save the output.

If the files `Make.headers` or `Make.files` (for use by makemake) exist in the directory in which to run the "`Make project`" command and they have changed since the last build, the "`Rebuild dependencies`" command will be run before the "`Make project`" command regardless of the setting of the "`Recalculate dependencies before make`" checkbox.

The command to compile a single source file is run by selecting "`Compile`" from the `Project` menu in an editor window that contains source code. This is only available if the file is recognized as a source file and always saves the file and its complement file if this is open. Since multiple projects can be open simultaneously, the name of the project from which the command will be taken is displayed in parentheses in the menu. (You can choose the active project by clicking in the corresponding project window or by holding down the `Meta` key and selecting the project from the Files menu.) The following strings are replaced when they are used in the command:

- $f – full path and file name
- $p – path
- $n – file name
- $r – file name root (e.g. file.c -> file)

If a character is preceded by a backslash, the backslash is removed and any special meaning is ignored. Thus, use `\$` to get a dollar character and `\\` to get a backslash character. The output of this command is also displayed in an editor window.

While `make` is running, you can pause it by pressing the `Pause` button in the upper right corner or by pressing `Ctrl-Z` (as on the command line) or the `Escape` key. You can stop it by pressing the `Stop` button in the upper right corner or by pressing `Ctrl-`C (as on the command line) or `Meta-period` (for Macintosh fans). You can use the key shortcuts even when the window is iconified. (Note that, while the window is iconified, the `***` in the window title tells you that the compile is running. If errors are found, this changes to `!!!`.) The Make button in the `Make Settings` window also becomes the `Stop` button, so you can stop the compile from there, too.

If errors occur while compiling, the new `Errors` menu in the `Compiler Output` window becomes active. This lets you jump from one error to the next. For convenience, if you haven't already started looking at the errors, the first error is hilighted when make finishes. You can examine the errors before the compile finishes because the `Errors` menu becomes active as soon as the first error is detected. To open the file containing an error, you can either `Ctrl-double-click` on the file name or use "`Open selection to` `error`" on the `Errors` menu.

When the compile is finished and errors have occurred, the window will be de-iconized if it is iconized but will not be raised otherwise. This allows users who prefer a single virtual screen to be notified by having the window de-iconize and allows users who prefer multiple virtual screens to avoid having the window appear on their current screen by leaving it de-iconized on another virtual screen.

The history menus in the `Make Settings` window can be used to store multiple directories and commands. For example, you could store various make commands with different targets in the menu for the "`Make project`" field.

For security reasons, you cannot use semicolons to include multiple commands. Write a script instead.

2.7.3 Running and Debugging Your Program

The Run Settings window lets you specify two commands:

- running your program
- displaying a source file in your debugger

You must also specify which directory to run the commands from.

The `Run` button runs your program. If the "`Make before run`" checkbox is on, then the "`Make project`" command from the `Make` window will be invoked first. If this succeeds, your program will then be run. If the "`Create window to display output`" checkbox is on, the output of the program (if any) is displayed in an editor window, so you can edit and save it. If the program terminates abnormally, Code Crusader will beep and display the reason in the editor window. The disadvantage of using a window to display the output is that you will not be able to quit Code Crusader until you quit the program. If you do not use a window, the program's output will appear in the xterm from which you started Code Crusader.

While the program is running, you can pause it by pressing the `Pause` button in the upper right corner or by pressing `Ctrl-Z` (as on the command line) or the `Escape` key. You can stop it by pressing the `Stop` button in the upper right corner or by pressing `Ctrl-C` (as on the command line) or `Meta-period` (for Macintosh fans). You can use the key shortcuts even when the window is iconified. (Note that, while the window is iconified, the `***` in the window title tells you whether or not the program is running.)

The input field at the bottom of the window lets you send text to the running process. Simply type the text and press Return. The `EOF` button sends `end-of-file` to the process. You can also use `Control-D`, as on the command line.

The `Run` button does not become a `Stop` button because you can run as many programs as you wish at the same time. Each time you run a new program, it either displays the output in an existing, inactive window, or it creates a new one. Each window title contains the name of the program that it is running so you can identify each one.

The command to display a file in the debugger is run by selecting `"Debug"` from the `File` menu in an editor window that contains source code or a header file. Since multiple projects can be open simultaneously, the name of the project from which the command will be taken is displayed in parentheses in the menu. (You can choose the active project by clicking in the corresponding project window or by holding down the Meta key and selecting the project from the Files menu.) The following strings are replaced when they are used in the command:

- $f - full path and file name
- $p - path
- $n - file name
- $r - file name root (e.g. file.c -> file)
- $l - line number

If a character is preceded by a backslash, the backslash is removed and any special meaning is ignored. Thus, use `\$` to get a dollar character and `\\` to get a backslash character.

This command blocks until finished because it should only contact a Multiple Document Interface (MDI) server and then exit.

If you choose to use Code Medic, you might find it useful to modify the default command by inserting the name of your executable in front of the `-f` option. This will start Code Medic with the correct binary if it isn't already running and will have no effect otherwise.

The history menus can be used to store multiple directories and commands. For example, you could store the command to invoke the debugger and various commands to run your program with different arguments in the menu for the "`Run program`" field.

For security reasons, you cannot use semicolons to include multiple commands. Write a script instead.

2.7.4 Viewing UNIX Man Pages

The `Man Pages` window lets you specify what function to look up and which section to look in. If you leave the section blank, the system will search all sections. To search the man pages for all occurrences of the function, check the `Apropos` checkbox. You can get the `Man Pages` window via the "`Look up man page...`" menu item on the `Search` menu in the Text Editor window and the Project menu in the Project and Symbol windows and the C++ and Java Class Tree windows. Man pages are initially read-only.

In the editor window, `Meta-double-click` checks the UNIX man pages if it can't find the function name in the class tree. `Meta-double-click` also understands the apropos format "`name (#)`" and automatically looks in the correct section.

The following command line options are provided so you can alias "`man`" to "`jcc --man`" and "`apropos`" to "`jcc --apropos`" whenever you are running X:

- –man page_name - displays man page.
- –man section_name page_name - displays man page from given section.
- –man -k search_string
  –apropos search_string - displays list of relevant man pages.

2.8 The Symbol Browser

If you have Exuberant ctags (http://ctags.sourceforge.net/) version 4.1 or later on your execution path, the symbol browser will display a list of all the symbols in all the source files on the project's search paths.

ctags version 4.1 works with Assembly, AWK, Bourne Shell, C, C++, Cobol, Eiffel, FORTRAN, Java, Lisp, Perl, PHP, Python, Scheme, TCL, and Vim. Newer versions may support additional languages. If you need support for another language, please contact Darren Hiebert (darren@hiebert.com).

2.8.1 Purpose of the Symbol Browser

The Symbol Browser is primarily intended to work behind the scenes. Unlike the C++ and Java class inheritance trees, the display itself does not provide useful information because it is simply an alphabetized list of all symbols that have been defined. The real power of the Symbol Browser lies in its connection to the Text Editor. When you Meta-double-click on a symbol in a Text Editor window, Code Crusader searches the list of symbols:

- If a single class or function name is matched, and you used the left mouse button, the file containing the definition will be opened.
- If a single class or function name is matched, and you used the middle mouse button, the file containing the declaration will be opened.
- If multiple symbols are matched, or you used the right mouse button, a list of the fully qualified symbol names will be displayed so you can choose the one you want. To automatically close this window after you have found the symbol you want, hold down Meta and Control while double clicking on the symbol name. Alternatively, you can use the arrow keys to select the symbol and then press Meta-Ctrl-Return.
- If nothing matches, Code Crusader automatically checks the UNIX man pages.

Code Crusader also attempts to narrow down the appropriate choices by using context. If the name of the source file displayed in the Text Editor window matches a known class name, only symbols defined by this class and its ancestors will be considered. Since this is sometimes inappropriate, you can use Meta-Shift-double-click to avoid context sensitivity.

2.8.2 Using the Symbol Browser

To locate a particular symbol, press the space bar and then type the first few letters in its name. This search is case-insensitive. To open the file to show the symbol's definition, press Return. You do not have to de-iconify the window before typing.

Since both a function's declaration and its definition are included in the list of symbols, the declaration is italicized so you can tell the difference.

If a symbol is defined within a namespace (i.e. not at global scope), both the fully qualified version and the unqualified version will be displayed in the list, though at very different positions since the symbols are alphabetized. If a symbol is defined in more than one namespace, incremental search by typing will find the unqualified version (unless you type the entire namespace, too). To display a list of the fully qualified versions so you can choose the one you want, hold down the Meta key and double click on the unqualified symbol name, just like in the Text Editor window (see above).

2.8.3 Preferences

The following options can be changed in the Symbol Preferences dialog:

- `Raise class inheritance trees when searching with right mouse` `button` - If this is checked and you right-double-click on a symbol in a Text Editor or the Symbol Browser, the class inheritance trees that find a match will automatically show themselves. If this is not checked, you can show the trees by using the items on the Actions menu in the window that displays the search results.

2.9 The File List Window

This window is available via the "Show file list" item on the Project menu in all the windows related to a project. It displays every file on the project's search paths that matches a suffix type from the list of types in the File Types dialog.

To find a particular file in the list, simply type the first few characters of its name. To display a subset of the files, you can use either a wildcard filter, which works just like file globbing on the command line (i.e., ? for a single unknown character, and * for more than one unknown character), or a regular expression.

2.10 Miscellaneous notes

2.10.1 Design Philosophy

Code Crusader faithfully implements the Macintosh paradigm of allowing you to safely explore features by always providing the ability to undo the result.

If a menu item ends with ellipsis (`...`), this means that it opens a dialog that asks for more information and allows you to cancel if it wasn't what you were looking for.

2.10.2 The Clipboard

Code Crusader also follows the Macintosh clipboard model. This means that text is not automatically copied when it is selected. Use the "`Copy`" item on the `Edit` menu or `Meta-C`. `Meta-C` works in any input area, regardless of whether or not the window contains an Edit menu. The main advantage of this model is that paste can then replace the selected text. If you prefer the X clipboard model, the Text Editor Preferences dialog lets you use this instead.

2.10.3 Useful Shortcuts

- On PC's, `Meta` is normally bound to the `Alt` key on the keyboard.
- Drag-and-drop can be cancelled by pressing the `Escape` key before releasing the mouse button.
- Tab shifts focus to the next input area. `Shift-Tab` shifts focus to the previous input area.
- To use a shortcut for a button or checkbox, hold down the `Meta` key and press the underlined character.

2.10.4 Scrolling

All scrollbars display increment/decrement arrow pairs if there is space. Coupled with the `Shift` key which makes the arrows scroll a half page at a time, this conveniently packages all the scrolling functions in one place.

Clicking the middle button in a scrollbar forces the thumb to jump to the place where you clicked. The arrow keys scroll the tree, read-only text, and help by small steps the way the scrollbar arrows do. To save your current position so you can easily jump back to it later, hold down the `Meta` key and click the left mouse button on the scrollbar thumb. This creates a scrolltab. Clicking on this scrolltab scrolls to the saved location. You can place as many scrolltabs on each scrollbar as you wish.

- Meta-left-click on scrollbar thumb - Place a scrolltab
- Left-click on scrolltab - Jump to the saved position
- Meta-left-click on scrolltab - Remove it
- Meta-Shift-left-click on scrolltab - Remove all scrolltabs

These functions are also accessible via the menu that pops up when you right-click on the thumb or any scrolltab.

In the project, tree and text editor windows, `Ctrl-1` through `Ctrl-9` scroll to the first through ninth vertical scrolltab, respectively.

If you have a wheel mouse, the wheel will scroll the widget that you point to:

- No modifiers - Scroll five steps
- `Shift` - Scroll one step
- `Control` - Scroll one page
- `Meta/Alt` - Scroll horizontally instead of vertically

2.10.5 The File Chooser

Name completion in the `Path` and `File` input fields is not supported because `Tab` is already reserved for switching between input areas. You don't actually need it, either, because typing in the file list table performs an incremental search, after which you can use the following keys:

- `Up-arrow` - Moves up one item
- `Down-arrow` - Moves down one item
- `Left-arrow`
  `Meta-up-arrow` - Goes up one directory (just like clicking on the Up button)
- `Right-arrow`
  Meta-down-arrow - Enters the selected directory
- `Return` - Enters the selected directory or chooses the selected file

Note that `Meta-up-arrow` and `Meta-down-arrow` work even if the file list table does not have focus.

Other useful features in the file chooser:

- Double clicking a file in the `Save File` dialog copies the name to the input field. This is useful if you want to replace a file or create a new file with a similar name.
- The window is resizable, and always remembers its size and position.

2.10.6 Selecting more than one item in a list

If selecting more than one item at a time is possible in a particular list, you can do so with the following actions:

- Left-click and drag - Selects files between where you click and where you release the button
- `Shift-left-click` and drag or Right-click and drag - Selects files between where you last clicked and where you release the button
- `Shift-up/down-arrow` - Selects files just like `Shift-left-click`
- `Control-left-click` and `drag` - Selects/Deselects each file that you drag over

2.10.7 Editing items in a list or table

If you are editing an item in a list or a table, you can switch to a different item by using the following shortcuts:

- `Meta-up-arrow`
  `Meta-8`
  `Shift-Return` - Moves up one row
- `Meta-down-arrow`
  `Meta-2 Return` - Moves down one row

- `Meta-left-arrow`

  `Meta-4`

  `Shift-Tab` - Moves left one column

- `Meta-right-arrow`

  `Meta-6`

  `Tab` - Moves right one column

- The digits correspond to the appropriate positions on a numeric keypad.

2.10.8 Color Chooser

This dialog lets you create an arbitrary color via either the HSV or RGB color model. HSV is implemented as a set of sliders because the values are meaningless. `Hue` scans the color spectrum, `Saturation` can be set to the far left to obtain gray, and `Value` controls the brightness of the color. RGB is provided as a set of values because it is primarily useful when requesting a known color, e.g. one obtained from a drawing program, `xv`, `xmag`, or `xcmap`. For convenience, the input fields accept hex values prefixed by `0x`. (e.g. `0xA4D3`)

2.10.9 How to avoid fiddling with windows

Code Crusader's has several features that minimize the amount of effort required to deal with window placement and focus:

- Dialogs and messages are centered on the screen and automatically grab keyboard focus so you don't have to move the mouse.

- "`Save window size`" item on the `Preferences` menu lets you save the default size of the many window types listed below.
  - Class tree
  - C/C++ source file, other source file, documentation file
  - C/C++ header file
  - Other files
  - Compile and Run output
  - Search output
- Any dialog window that can be resized automatically remembers its position and size.
- The shortcuts `Meta-0` through `Meta-9` work in all the non-modal dialog windows, such as the `Search Text`, `Find File`, and `Man Pages` dialogs.
- `Meta-W` will close any non-modal window.
- `Escape` cancels and closes any dialog.

Most window managers have an option that defeats Code Crusader's automatic window placement. To avoid this, you can do the following:

- fvwm - Comment out "`NoPPosition`" from `~/.fvwmrc`.
- fvwm2 - Use "`UsePPosition`" instead of "`NoPPosition`" in `~/.fvwm2rc`.

- twm - Set "`UsePPosition on`" in `~/.twmrc`.

Window managers like fvwm2 and fvwm95 that provide a taskbar listing all the currently open windows allow you to filter this list based on the window's title or `WM_CLASS` attribute. Code Crusader sets `WM_CLASS` as follows:

- Project window - `Code_Crusader_Project`
- Symbol List window - `Code_Crusader_Symbol_List`
- Tree window - `Code_Crusader_Tree`
- File List window - `Code_Crusader_File_List`
- Editor window - `Code_Crusader_Editor`
- Compiter output window - `Code_Crusader_Editor_Compile_Output`
- Search Output window - `Code_Crusader_Editor_Search_Output`

The setting for the Compiler Output window allows you to either filter out all text windows with `Code_Crusader_Editor*` or filter out all other text windows with simply `Code_Crusader_Editor`. It is often convenient to display the Compiler Output window on the taskbar because the `***` disappears when the compiler is finished.

2.10.10 Remapping modifier keys

You can choose which keys on your keyboard should map to each modifier by using `xmodmap` in your `~/.xinitrc` file. As an example, if you have a PC keyboard with the new `Windows` key between `Ctrl` and `Alt`, you can change the left `Windows` key to produce the `Mod4` modifier as follows:

```
xmodmap -e "clear mod4"
xmodmap -e "keycode 115 = Super_L"
xmodmap -e "add mod4 = Super_L"
```

To check which keys produce each modifier, use `xmodmap -pm`.

You can use `xev` to get the keycode for a particular key on your keyboard. Run it from an `xterm`, move the cursor into the resulting window, press the key, and check the keycode value that is printed.

Chapter 3

Using Code Medic

3.1 Getting started with Code Medic

Code Medic is a graphical front-end to the gdb debugger. It was designed to work with Code Crusader (http://www.newplanetsoftware.com/jcc/) but can also be used alone. In addition to the infomation listed in this chapter, you may also find useful both the GDB manual (http://www.gnu.org/manual/gdb/) and a discussion of general debugging techniques (http://www.newplanetsoftware.com/medic/techniques.php).

3.1.1 Running Code Medic

When invoking Code Medic, you can include as arguments the program to be debugged, the core file (if any) to debug, and any source files that you want to display, in that order.

The formats for specifying source files are the following:

- -f file_name
- -f +line_number file_name
- -f file_name:line_number

3.1.2 Selecting the program to debug

If you did not specify a program to debug as an argument to Code Medic, you must select one by using the "Choose binary..." item on the Debug menu in either the Command Line window or any Source View window. After loading the program, Code Medic will fill the Files window with all the source files used by the program. It will then display the function main() in the Current Source View window.

3.1.3 Examining the state of the program

To display the value of a variable or expression when your program is paused at a breakpoint, select the variable name or expression in a Source View window and then use the "Display variable" item on the Debug menu. This will open the Variables window and display the result.

To display the current execution stack when your program is paused at a breakpoint, open the Stack Trace window. (You can also pause your program by using the "Pause execution" item on the Debug menu.)

3.1.4 Debugging a core file

Once a program is loaded, you may load a core file by using the "Choose core..." item on the Debug menu in either the Command Line window or any Source View window. Once the core file has been loaded, the Stack Trace window will automatically be opened to show you the state of the program when it crashed and to allow you to navigate through the execution stack to perform the post mortem.

3.1.5 Debugging a running program

Once a program is loaded, you may debug a running copy of the program by using the "Choose process..." item on the Debug menu in either the Command Line window or any Source View window. Code Medic will pause the process and display the Stack Trace window to show you the current state of the program.

3.2 Viewing source files

When the program is paused, the Current Source window displays the source file corresponding to the stack frame that is currently being debugged. If a cyan colored arrow is visible to the left of the line numbers, this indicates the line at which the program is paused.

All other Source View windows display the source files that you have opened either via the Files window or the "Open source file..." item on the File menu.

These windows only display the program's source files. They do not allow you to edit the files because then it would no longer correspond to the debugging information stored in the program. When the files and the debugging information do not correspond, the result is pure confusion.

Since one cannot edit the files, all keypresses (except menu shortcuts) are sent to the Command Line.

3.2.1 Opening source files

The Files window displays a list of all the source files that are used by the program being debugged. (Unfortunately, gdb does not report files used by shared libraries until after the program has been started.)

To open a file, either select it and press Return or double click on it. You can also select several files and press Return to open them all at once.

To locate a file, simply type the first few letters in its name, and the first file name that matches will be selected. You can search for groups of files by using the wildcard or the regular expression filter on the Actions menu.

3.2.2 Setting Breakpoints

To set a breakpoint, click on the line number in any Source View window. The presence of a breakpoint is indicated by a red circle to the left of the line number. To remove a breakpoint, click on the red circle or the corresponding line number. To remove all breakpoints, use "Clear all breakpoints" on the Debug menu.

3.3 Displaying variables

The Variables window displays the values of variables and expressions and the contents of structures and objects. Structures, objects, and pointers have an arrow to the left of their names. Clicking on this arrow displays the variable's contents below its name.

Whenever the program pauses at a breakpoint, the values are updated automatically. Values that have changed since the last pause are displayed in blue. Variables and expressions that cannot be evaluated in the current context are displayed in gray.

To display a variable or expression, either use the "New expression" item on the Actions menu and then type in the expression in the input field that is created, select it in a Source View window and use the "Display variable" item on the Debug menu, or double click with the middle mouse button on the text in the Source View window. You can display the value of any arbitrary expression, not just a simple variable name.

Here is the complete list of all the ways to view the contents of structures, objects, and pointers:

- `Right-arrow`

  `Click on triangle when item is closed` - Open the item to display its contents.

- `Left-arrow`

  `Click on triangle when item is open` - Close the item.

- `Meta-Right-arrow`

  `Meta-click on triangle when item is closed` - Open the item to display its contents and the contents of all the items that are contained within it.

- `Meta-Left-arrow`

  `Meta-click on triangle when item is open` - Close the item and all the items contained within it. (If you only close the item itself, the contained items will remain open and be visible next time you open the item.)

- `Shift-Right-arrow`

  `Shift-click on triangle when item is closed` - Open all items to display their contents.

- `Shift-Left-arrow`

  `Shift-click on triangle when item is open` - Close all items.

The keyboard shortcuts obviously only work when one or more items are selected (GDB-MiscHelp#MultiSelect).

3.3.1 Examining an array

To examine the contents of an array, select its name and then use the "View as array" item on the Actions menu. If it is an array of pointers rather than an array of values or objects, use the "View as array of addresses" item instead.

An Array window displays the range of elements specified at the top of the window. When you tab out of either input field, the display is automatically adjusted to match the new range.

If the elements in the array are structures, object, or pointers, they will each have an arrow to the left of the name, and you can use the above techniques to display the contents of each item.

3.4 Examining the execution stack

When the program is paused, the Stack Trace window displays the current execution stack. You can navigate between the frames by clicking on the function name or by using the up and down arrow keys to move by one frame at a time. The source code for the currently selected frame is displayed in the Current Source View window.

The arguments that were passed to each function are displayed below the corresponding function name when you click on the arrow to the left of the function name. To show the arguments to all functions, hold down the Shift key while clicking on any of the arrows.

To display an argument in the Variables window, simply click on it.

3.5 The command line interface

The Command Line window is used to interact directly with the underlying gdb process. The main area of the window displays the results of this interaction. You can save this text with the "Save history" command on the File menu.

Below the interaction history is the command line input area where you enter the commands to be set to gdb. Simply type in the command and press Return. Pressing the Tab key performs command and symbol name completion, just like when using plain gdb. The menu below the (gdb) prompt lists the last one hundred commands sent to gdb. You can also access these commands by using the up and down arrow keys, just like when using plain gdb.

When your program is running or gdb is otherwise busy, the (gdb) prompt will be gray. This indicates that your commands will be buffered.

If your program reads from the standard input, you can send it text by typing in the command line input area when your program is running.

The button in the lower left corner displays the name of the program that is being debugged. You can choose which program to debug by clicking on this button.

The input field next to this button contains the arguments that will be passed to the program when it is started. You can enter anything you want in this input field.

The "Restart gdb" item on the File menu is provided in case gdb should crash. It restarts gdb while maintaining the current breakpoints.

3.5.1 Using a custom version of gdb

By default, Code Medic uses the version of gdb that is on your execution path. You can run a different version of gdb by using the "Change gdb binary" item on the File menu.

Chapter 4

Code and Makefile Generation Utilities

4.1 Makemake

When compared with the scriptable, integrated development environments available on other platforms, UNIX make is a painful mess. makemake was written to alleviate some of the pain by generating a Makefile from a list of source files.

More specifically, given the two files `Make.header` and `Make.files,` makemake parses the files specified in `Make.files` to calculate their dependency lists and then generates a Makefile compatible with GNU's version of make.

`Make.header` is simply copied to become the first part of the Makefile. It should contain all the variable definitions and special targets. (makemake automatically generates the targets all, tidy (double colon), clean (double colon), checksyntax (see makecheck below), touch (see maketouch below), and jdepend (internal use only).) In particular, `LINKER` must be defined to be the name of the program to use during linking, and `DEPENDFLAGS` must be set to contain all the compiler directives so that makedepend will work correctly. (`LINKER` is usually just the same as the name of the compiler, but is needs to be `mcc` in order to use Mathematica's MathLink package.)

Here is a simple example of a `Make.header` file:

```
# Useful directories

MYCODEDIR := .

# Directories to search for header files

SEARCHDIRS := -I- -I${MYCODEDIR}

# makemake variables
LINKER := gcc
DEPENDFLAGS := -g -Wall -Werror ${SEARCHDIRS}
TOUCHHEADERS := ${MYCODEDIR}/*.h
# make variables
CC := gcc
CXX := g++
CCC := g++
CPPFLAGS = ${DEPENDFLAGS}
```

`Make.files` contains a simple list of all the files required to build each target as follows:

```
@<name of first target>
<first object file>
<second object file>
...
```

```
@<name of second target>
...
```

The object files should not contain a suffix, because makemake appends the appropriate suffixes automatically. (e.g. `.c`, `.cc`, `.o`, etc.) The source file suffix is `.c` by default, but can be set with the `-suffix` option. The current suffix can be overridden by placing the desired suffix in front of the source name as follows:

```
-suffix .cc
@myprog
.c myprog
JString
```

The first line sets the default suffix to `.cc`. The next line specifies the name of the make target. The third line tells makemake that the object file myprog.o is required in order to build myprog, and that the source file is called `myprog.c`. The last line tells makemake that the object file `JString.o` is also required in order to build `myprog`, and that the source file is called `JString.cc`.

You can also specify the suffix for the output file by including it between the source suffix and the file name:

```
@myprog
.java .class myprog
```

Note that this is not usually necessary since makemake is reasonably intelligent about guessing the correct output suffix.

Makemake correctly calculates dependencies for lex, flex, yacc, and bison files. Simply specify the `.l` or `.y` suffix. Dependencies on libraries can be included by specifying the `.a` suffix. Libraries must also be listed in the standard make variable `LOADLIBES` so the linker will pick the correct type. (`.a` or `.so`) Dependencies on precompiled files can also be included by specifying the `.o` suffix. This is useful if some files have to be compiled in a completely different way, e.g. using a different Makefile. Makemake recognizes both `.a` and `.o` as special suffixes and does not try to calculate their dependencies.

The `-prefix` option sets the prefix of all subsequent source files. This is useful if a group of files is in a different directory from the Makefile. For convenience, the prefix is cleared when a new target is encountered. As usual, comments can be included in `Make.files` by starting the line with a hash (#).

If no objects files are listed for a target, it is assumed that they are the same as those for the next target in the file. This is especially convenient for building several different versions of the same library. (e.g. static `.a` and shared `.so` versions)

The command line options for makemake are as follows:

```
-h   prints help
-hf  <header file name>    - default Make.header
-if  <project file name>   - default Make.files
-of  <output file name>    - default Makefile
--obj-dir  <variable name> -
specifies directory for all .o files
--no-std-
inc  exclude dependencies on files in /usr/include
--check       only rebuild output file if in-
put files are newer
--choose      interactively choose the targets
```

The `--choose` option prints a list of the targets found in `Make.files`, and lets you choose which ones to include in the final Makefile. Another way to specify a particular subset of targets is to include their names after the other options.

As mentioned above, your own special targets should be included in `Make.header`. Some targets (e.g. `TAGS` used for the etags program) require a list of all the source files. makemake provides this in the variable called `MM_ALL_SOURCES`. Thus, to define the `TAGS` target, simply include the following in your Make.header:

```
.PHONY : TAGS
TAGS: ${MM_ALL_SOURCES}
    etags ${MM_ALL_SOURCES} ${TOUCHHEADERS}
```

4.2 Makecheck

Sometimes it is helpful to be able to check a single source file after modifying it. This provides instant feedback on typos and such. Of course, one would prefer not to have to run make and wait while other files are recompiled first. This can be done by running make directly, but one has to include the path exactly the way it is specified in the Makefile. makecheck was written to solve this problem. Once the Makefile is built, simply type "`makecheck MyClass.cc`" to recompile only `MyClass.cc`. makecheck finds the correct path for you.

4.3 Maketouch

Sometimes it is necessary to use compile-time flags to include or exclude certain pieces of code. (One should obviously try to minimize the need for this, but `NDEBUG`, which turns off `assert()`, will always be there.) In such cases, one would prefer not to have to recompile everything (i.e. make clean) when a flag that only affects a few source files is changed. maketouch was written to alleviate this problem. It works in conjunction with makemake's special touch target to insure that only the files that use the flag are actually recompiled.

Since makemake does most of the work automatically, maketouch is very easy to use. Simply type "maketouch compile_flag" to touch all the source and header files that include "compile_flag". If the program depends on libraries that use the same "compile_flag", simply add a rule as follows:

```
.PHONY : touch
touch::
  cd ${LIB_1_DIR}; ${MAKE} TOUCH-
STRING=${TOUCHSTRING} touch
  cd ${LIB_2_DIR}; ${MAKE} TOUCH-
STRING=${TOUCHSTRING} touch
  ...
```

4.4 JXLayout

`jxlayout` was written to generate code for `JX` from an `fdesign` file. It is a simple, text-based program that generates C++ code to construct the Widgets and arrange them in the Window. The best way to understand how `jxlayout` works is to study `testjx.fd` and the associated code.

Since each Window should have its own custom WindowDirector, Document, or DialogDirector, the creation and layout code is only a small part of the total code required. The rest of the code must be written by hand. As an example, a class MyDialog derived from DialogDirector requires at least a constructor, a destructor, and methods for retrieving the information when the Window is closed. In order to retrieve this information, the relevant objects in the window must be declared as instance variables. This, in turn, requires that the header file for MyDialog be directly modified by jxlayout. In addition, it is most convenient if the source code can also be placed directly in the source file for MyDialog. This is accomplished by delimiting a region of the source and header files for explicit use by `jxlayout`. The region begins with "`// begin JXLayout`" and ends with "`// end JXLayout`". `jxlayout` replaces everything inside this region with code generated from the .fd file. This code can, of course, be part of a larger routine. To create additional Widgets (e.g. menus) or adjust the initial values of Widgets constructed by jxlayout, simply append this code to the routine after the "`// end JXLayout`" delimiter. Note that jxlayout makes a backup (`file` -> `file~`) of each file that it modifies.

Unnamed objects in the `.fd` file are assigned unique, arbitrary names in the resulting C++ code. Named objects are considered to be instance variables and are declared in the header file. If one needs to refer to an object inside the routine, but nowhere else, one can enclose the name in parentheses (e.g. `(okButton)`). Thus, unlike unnamed objects, the name will not be arbitrary, but it will be declared locally in the routine instead of at class scope in the header file. If one needs to use a local variable that has already been declared, one can enclose the name in brackets. This is primarily useful for arrays (e.g. `<radioButton[2]>`).

`jxlayout.class.map` defines how objects that `fdesign` understands map to `JX` classes. Classes that are not listed in this file and custom classes derived from `JXWidget` can be included in the `.fd` file by creating an `fdesign` box object of type `NO_BOX` and setting the label to be the name of the `JX` class. The most common examples are `JXMenuBar` and `JXScrollbarSet`. Custom classes usually require extra arguments in the constructor. This can be handled by placing these arguments first, and then using "`MyWidget(arg1,arg2,`" as the label. `jxlayout` will append the standard arguments in order to complete the constructor. (Such code will obviously not compile unless `arg1` and `arg2` are defined before the "`// begin JXLayout`" delimiter.)

A Widget's container is calculated by finding the smallest Widget that has already been encountered in the `.fd` file and that contains the new Widget. `jxlayout` translates the `XForms` gravity specifications into `JX` resizing options. `NoGravity` translates to elastic resizing. To fix an edge, add the corresponding compass direction to `NWGravity`.

The code generated in the region delimited by "`// begin JXLayout`" and "`// end JXLayout`" always starts with the creation of a `JXWindow` object. If you want to create a layout for the inside of a `JXWidget` (e.g. each page of a `JXCardFile`), you can create a new form in `fdesign`, give it the same name, and then specify a different tag to look for in the source and header files by appending "`--tag--encl`" to the name of the form. As an example, the form called `"MyDirector"` will generate code in `MyDirector.cc` and `MyDirector.h` between "`// begin JXLayout`" and "`// end JXLayout`", and all Widgets will be constructed with window as their enclosure. The form called "`MyDirector--sub1--MyWidget`" will generate code in `MyDirector.cc` and `MyDirector.h` between "`// begin sub1`" and "`// end sub1`", and all Widgets will be constructed with `MyWidget` as their enclosure.

You can also specify extra constructor arguments for the `JXWindow` objects that are created with the `JXLayout` tag by appending "`JXLayout<"title">`" or "`JXLayout<"title", ownsColormap, colormap>`" to the name of the form. The default is to construct the window with:

```
new JXWindow(this, w,h, "");
With "JXLayout<"title">", this becomes:
new JXWindow(this, w,h, "title");
With "JXLayout<"title", ownsColormap, col-
ormap>", this becomes:
new JXWindow(this, w,h, "title", ownsColormap, col-
ormap);
```

The first option allows you to specify the window title without having to call `SetTitle()`. The second option allows you to construct a `JXWindow` with a private colormap. (Refer to the `TestDirector` class in the `testjx` application for an example of this last option.)

The `callback` argument field in `fdesign` is used to set the ID of RadioButtons.

4.4.1 fdesign

`fdesign` is the widget layout editor written for use with the `XForms` library. The documentation for `fdesign` is available on the web at

```
http://www.westworld.com/~dau/xforms/node4.html#SECTION04000000000000000000
```

and via ftp from

```
ftp://bragg.phys.uwm.edu/pub/xforms/
```

There are two important points to keep in mind when using `fdesign`. First, the widgets that `fdesign` provides are those that `XForms` provides. These are not identical to the `JX` widgets. The file `jxlayout.class.map` shows which `XForms` widgets map to `JX` widgets. The first column is the type selected in the fdesign control window. The second column is the type selected in the `Options` dialog window obtained by right-double-clicking on the widget.

The second point to remember is that you should not use the big arrow button in the upper right corner of the `Options` dialog window because jxlayout cannot parse what it produces.

4.4.2 jfdesign

This script file runs `fdesign` with the options that are appropriate for generating an `.fd` file for use with `jxlayout`. The script assumes that `fdesign` is on your search path.

4.5 JXWizard

`jxwizard` is a tool for generating a skeleton `JX` program. The program generated will contain all the major pieces of a complete program, including an application object, unlimited window director objects, a preferences manager, on-line help, configurable toolbars and `MDI` support.

When you run the program (which at this point is a command line utility like jxlayout), you will be asked for the project prefix, the program name and version number, the template directory (which is included with `jxwizard`), the directory where the new project will be placed (which must *not* yet exist), and author, email, url, and ftp. The project prefix will be appended to every class and file name.

After running `jxwizard`, run `makemake` and `make` in the new project directory, and run the new binary (which should be named according to what you specified to `jxwizard`).

Part II

JX Tutorials

Chapter 5

Getting Started with JX and the JX Tutorials

JX was created to satisfy two requirements. First, it provides all the standard functionality required in a modern, visual program in powerful yet simple modules. Second, every module can easily be enhanced to handle features specific to a particular application.

The fundamental example is the visual interface element itself, the widget. JX provides a large number of finished widgets, such as buttons, menus, input fields, and scrollbars. At the same time, it is very easy to create new widgets for use in a particular program.

As another example, the PostScript printer module provides all the standard options like paper size, orientation (landscape or portrait), page range, and the option to print to a file, so most programs will be able to use it as is. Should a program need to provide more options, however, it is very easy to add extra items to the dialogs and then create a derived class of the printer to use the values.

5.1 The Object Hierarchy

The objects in a program can be separated into four layers:

1. Application + Displays
2. Directors
3. Windows
4. Widgets

The application manages the event loop and owns all of the other global objects. There is exactly one `JXApplication` object in every program. Each `JXDisplay` represents a separate connection to an X server. Each `JXWindowDirector` manages a `JXWindow` on a particular `JXDisplay`. Each `JXWindow` contains `JXWidgets`. Widgets can contain other widgets. Widget is the generic term for all visual elements like buttons, menus, and input fields.

5.2 Types of Directors

JX provides three types of directors. Window directors are the simplest. They only manage a window. Dialog directors add the concept of modality. If a dialog is modal, it suspends the

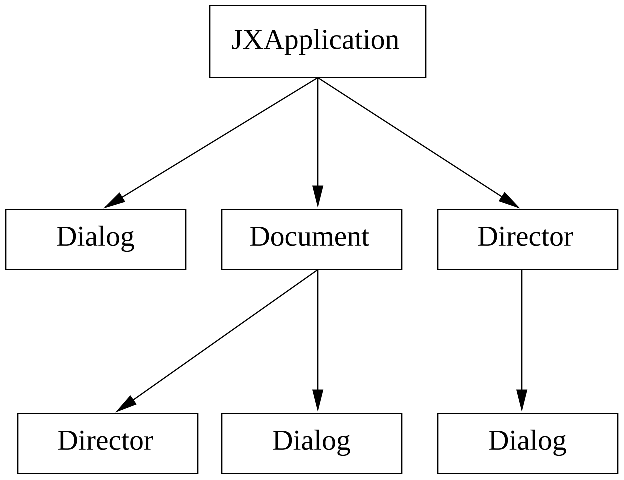


Figure 5.1: Example Application/director hierarchy

director that owns it while it is active. Documents extend window directors to also manage the data displayed in the window. Documents are usually all owned by the application rather than by other directors.

5.3 The Ownership Hierarchy

Every JX object except the application is owned by some other object, and the owner is responsible for deleting it. The application is special because it is the root of the ownership tree. It owns all the displays, all the documents, and the other main directors. Each director owns its window and can also own subsidiary directors. Each window owns the main widgets that it contains. Each widget owns the subsidiary widgets that it contains.

Figure 5.1 shows an example hierarchy showing the ownership of directors. Figure 5.2

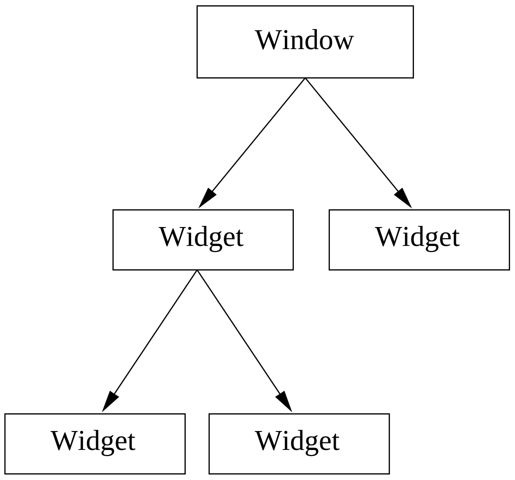


Figure 5.2: Example Window/widget hierarchy

displays an example hierarchy showing the ownership of objects within a window.

5.4 Typical Usage

Programs that require a visual user interface can be divided into two groups based on whether or not they use documents.

Programs that use documents should be designed to handle multiple documents at one time. The services required for this are provided by `JXDocumentMenu` and `JXMDIServer`. Typically, each document displays its data in the window that it manages. Auxiliary information can be displayed in window directors and dialogs owned by the document.

The feature shared by all programs that do not use documents (e.g. games and system utilities) is that they don't need to save anything other than user preferences. When the user tells the program to quit, there is no need to ask the user if anything should be saved or discarded. Such programs typically use one or more window directors owned by the application. User preferences can be saved by the destructor of each window director or by the destructor of the application, as appropriate. (cfr. `JPrefsManager`)

5.5 Model - View - Controller

Code is easier to understand and maintain when it is written as loosely coupled, reusable modules. Model-View-Controller provides a paradigm for achieving this which works very well when combined with the object oriented programming (OOP). The basic idea is to decouple your code into three parts:

Model	This stores the data.
View	This displays the data in a particular way.
Controller	This lets the user interact with and modify the data.

The best example in JX is the `JTable` suite. A derived class of `JTableData` stores the data, `JTable` draws the data, and a derived class decides what to do in response to mouse clicks.

By separating the Model from the rest, one can make it system independent and therefore reusable in other programs. In practice, the View and the Controller often end up tightly coupled because a particular view is often designed to allow a particular kind of interaction.

One only achieves true separation between the Model and the rest by using messages rather than function calls. In JX, this is handled by `JBroadcaster`. The messages are broadcast from the Model to all objects that are listening, not just to a particular object. Thus, when the Model sends "this piece of me changed," the View receives it and redraws itself.

Messages also solve the problem of displaying multiple Views of a single Model. With direct function calls, the Model has to know about every View. With messages, all the Views just listen to the Model via `JBroadcaster`.

5.6 Compiling JX

5.6.1 Preparation

You need `gcc 2.7.2` (or later) and GNU's version of `make`.

5.6.2 Compiling

If you want the binaries installed somewhere other than `${HOME}/bin`, set the environment variable `JX_INSTALL_ROOT` to the desired directory. This path (either `~/bin` or `$JX_INSTALL_ROOT`), must be on your execution path. The libraries will always be placed in lib, which should be added to your `LD_LIBRARY_PATH`.

Note that the compile will build everything in the `JX` directory, including any 3rd party libraries and programs that you unpack after the main distribution.

Compile everything by typing `make` in the top level `JX` directory and following the instructions. You can ignore any errors that make ignores.

If you cannot use the version of `libXpm` already installed on your system (e.g. because it is too old) or if you do not have `libXpm` at all, then uncomment the definition of `JX_INCLUDE_LIBXPM` near the top of the appropriate file in `include/make/sys/`

Since `ACE` is huge, only one version of the library is built. If you want the other version as well, go to the `ACE` directory and type `make shared` or `make static` after you have built everything else.

It is also important to realize that we do not maintain `ACE`. You are welcome to contact us if you have problems compiling it, but the `ACE` developers are more likely to be able to help, and patches should be sent to them, not to us. JX uses `ACE` because networking is horribly UNIX-variant-dependent. The `ACE` web page is:

```
http://www.cs.wustl.edu/~schmidt/ACE.html
```

If your system is not already supported by `JX`, create another entry in the `Makefile` by copying one of the existing ones and modifying it as follows:

- Start by assuming that `jMissingProto_empty.h` is appropriate. If you get compile errors, create a new one and fill it with whatever is necessary.

- Create a new file in `include/make/sys/` by copying `template_g++` and editing the `Adjustable Parameters` section.

- If you cannot find a suitable pair of `ACE` configuration files, contact the developers. (`http://www.cs.wustl.edu/~schmidt/ACE.html`)

- Once you get the entire distribution to compile, contact us so we can add your patches to the distribution so you won't have to do it again.

For further information, consult the FAQ at:

```
http://www.newplanetsoftware.com/jx/faq.html
```

5.6.3 Additional configuration

Get the `xforms` distribution from

```
ftp://bragg.phys.uwm.edu/pub/xforms/
ftp://einstein.phys.uwm.edu/pub/xforms/
```

and extract the `fdesign` binary. This should be placed in `~/bin`.
(or `$JX_INSTALL_ROOT`, if you set it) `fdesign` is the graphical layout tool.

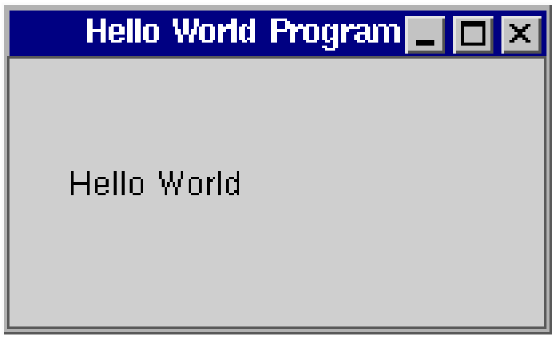
It is also a good idea to set the following environment variables:

```
setenv JMM_INITIALIZE "default"
setenv JMM_SHRED "default"
```

Check `libjcore/code/JMemoryManager.cc` for more info.

Chapter 6

Hello World



The hello world program, while simplistic, is useful for introducing the basics of a JX program. The structure is consistent with typical, more sophisticated JX programs.

The sequence is simple: create the application object, create the `HelloWorldDir` window director object, activate the director, then call `JXApplication::Run()`. The `HelloWorldDir` window director object is responsible for creating the window and its contents, which in this case consist of a single `JXStaticText` object that contains the words “`Hello World!`". The file `helloworld.cc` contains the `main()` function, and the files `HelloWorldDir.h` and `HelloWorldDir.cc` contain the interface and implementation of the `HelloWorldDir` class.

6.1 The function main

The following is a listing of the function `mainO`:

```
#include "HelloWorldDir.h"
#include <JXApplication.h>
#include "../TutorialStringData.h"
#include <JAssert.h>

static const JCharacter* kAppSignature = "tut1";

int
main
  (
  int    argc,
  char*  argv[]
  )
{
  JXApplication* app =
    new JXApplication(&argc, argv, kAppSignature,
        kTutorialStringData);
  assert( app != NULL );

  HelloWorldDir* mainDir = new HelloWorldDir(app);
  assert( mainDir != NULL );
```

```
      mainDir->Activate();

      app->Run();
      return 0;
    }
```

We begin by creating the `JXApplication` object. In this case, our program is basic enough that we don't need a custom application object. Once this is created, we create the window director, and activate it. Activating it displays it on the screen. We then begin the event loop by calling the `JXApplication::Run()` function. For our program, the real work is done in the `HelloWorldDir` object.

6.2 The HelloWorldDir class

The `HelloWorldDir` window director class creates and controls the window that contains our program. The essential portion of the director's code is contained in the constructor.

```
    #include "HelloWorldDir.h"
    #include <JXWindow.h>
    #include <JXStaticText.h>
    #include <jAssert.h>

    HelloWorldDir::HelloWorldDir
      (
      JXDirector* supervisor
      )
      :
      JXWindowDirector(supervisor)
    {
      JXWindow* window =
        new JXWindow(this, 200,100, "Hello World Pro-
    gram");
      assert( window != NULL );

      SetWindow(window);
      window->SetMinSize(200,100);
      window->SetMaxSize(200,100);

      JXStaticText* text =
        new JXStaticText("Hello World", window,
          JXWidget::kFixedLeft, JXWidget::kFixedTop,
          20, 40, 160, 20);
      assert( text != NULL );
    }
```

The very first step for the window director is to create the `JXWindow` that will contain all of the widgets. The window is created with a width of 200, a height of 100, and a title of "`Hello World Program`". The function `SetWindow()` tells the director that this is the window that it will be directing. A window director can control one and only one window. The `JXWindow` class has quite a number of features, two of which are demonstrated here, `SetMinSize()` and `SetMaxSize()`.

Once the window is created, it can be populated with the widgets that it will contain. In our case, this entails a single `JXStaticText` object that displays the text "`Hello World`". Our text object is created with the window that is its enclosure, an x and y top-left position, a width and height, and two sizing parameters. These parameters tell the object how to resize when the window resizes.

6.2.1 JXWidget resizing parameters

When a widget is created, it must know how to resize itself when its enclosure resizes. `JXWidget` has two parameters `HSizingOption`, and `VSizing0ption`, that control this behaviour. The valid values of `HSizingOption` are `kFixedLeft`, `kFixedRight`, and `kHElastic`. These correspond to a widget that remains a constant distant from the left side of the window, a constant distant from the right side of the window, and one that expands horizontally as the window expands horizontally. Similarly, the valid values of `VSizing0ption` are `kFixedTop`, `kFixedBottom`, and `kVElastic`, which correspond to a widget that remains a constant distant from the top of the window, a constant distant from the bottom of the window, and one that expands vertically as the window expands vertically.

6.2.2 JStringManager - Dealing with other languages

There is a problem with the above code that is obvious to programmers from non-English speaking countries. The window and window title text is in English, and there is no way to change it without editing the source code and re-compiling it. This is far from ideal. The `JStringManager` class solves this problem by allowing the programmer to specify a text ID instead of the text itself. This ID can point to the default text compiled into the program, or to translated text in a file. A few small changes to the `HelloWorldDir` code allows us to take advantage of this functionality.

```
#include "HelloWorldDir.h"
#include <JXWindow.h>
#include <JXStaticText.h>
#include <jGlobals.h>
#include <jAssert.h>
static const JCharacter* kWindowTitleID =
    "WindowTitle::HelloWorldDir";
static const JCharacter* kWindowTextID  =
    "WindowText::HelloWorldDir";
HelloWorldDir::HelloWorldDir
  (
  JXDirector* supervisor
  )
  :
  JXWindowDirector(supervisor)
{
  JXWindow* window =
    new JXWindow(this, 200,100,   JGet-
String(kWindowTitleID));
  assert( window != NULL );
  SetWindow(window);
  window->SetMinSize(200,100);
  window->SetMaxSize(200,100);
  JXStaticText* text =
    new JXStaticText(JGetString(kWindowTextID),
      window, JXWidget::kFixedLeft, JXWid-
get::kFixedTop,
      20, 40, 160, 20);
  assert( text != NULL );
}
```

The first change you see is the addition of two character constants, `kWindowTitleID`, and `kWindowTextID`. The values for these IDs are defined in the strings file located in the `01-HelloWorld` directory.

```
0
WindowTitle::HelloWorldDir "Hello World Program"
WindowText::HelloWorldDir "Hello World"
```

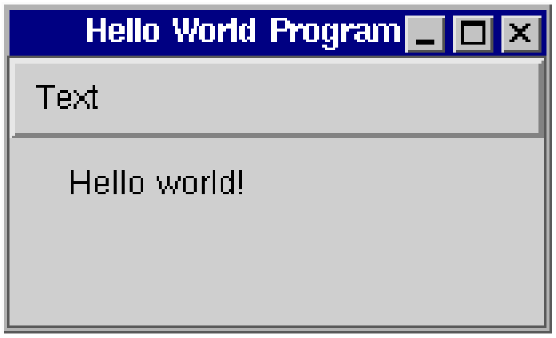
The `0` on the top line refers to the strings version number which for this version on `JX` is 0. The two following lines define the two IDs.

When the tutorials are compiled, these strings are compiled into the file `TutorialStringData.h` that is in the top level tutorial directory. This file is included in the `hello_world.cc` program and the character string it defines is passed to the `JXApplication` constructor above. In order to access the strings defined in this file, the function `JGetString` is used. In the code above, instead of passing the raw text to the `JXWindow` constructor, we passed `JGetString(kWindowTitleID)`, and `JGetString(kWindowTextID)` to the `JXStaticText` constructor.

In the `JXApplication` constructor above we passed in a character constant called `kAppSignature` that had the value "`tut1`". This defines the file name in the `.jx/string_data/` directory in the user's home directory that `JX` will look for translations. If you have a `LANG` environment variable, it will look for a file named `tut1_LANG` where `LANG` is the appropriate value. Otherwise it will look for a file simply named `tut1`. To test it, you can copy the strings file to `~/.jx/string_data/`, and change the text between the quotes (making sure not to change the format of the file). Without recompiling, run the `hello_world` program and you will see the new text in place of the compiled in text.

Chapter 7

Menus



```
Hello world!
```

One particular weakness of the "`HelloWorld`" program was that it only displayed a single message. We need a way for the user to select a different message. One way of accomplishing this is to have a menu with alternate messages listed, allowing the user to select the appropriate one. This leads us to our second example program "`MenuHello`".

This program will add a menubar to the "`HelloWorld`" program, and will require a discussion of `JBroadcaster`, the class that provides inter-object communication to JX applications. This functionality is necessary to provide a way for the menu to inform the `MenuWorldDir` object when a user selects a different message.

7.1 The MenuHelloDir class

To keep things a little more organized, a function is added to the `MenuHelloDir` object called `BuildWindow()`. This function creates the window and its contents.

```
void
MenuHelloDir::BuildWindow()
{
  JXWindow* window = new JXWindow(this, 200,100,
    "Hello World Program");
  assert( window != NULL );
  SetWindow(window);
  window->LockCurrentSize();

  JXMenuBar* menuBar =
    new JXMenuBar(window,
      JXWidget::kHElastic, JXWidget::kFixedTop,
      0,0, 200,kJXStdMenuBarHeight);
  assert( menuBar != NULL );

  itsTextMenu = menuBar-
>AppendTextMenu(kTextMenuTitleStr);
  itsTextMenu->SetMenuItems(kTextMenuStr);
  itsTextMenu-
>SetUpdateAction(JXMenu::kDisableNone);
  ListenTo(itsTextMenu);
```

```
      itsText =
        new JXStaticText("Hello world!", window,
          JXWidget::kFixedLeft, JXWidget::kFixedTop,
          20, 40, 160, 20);
      assert( itsText != NULL );
    }
```

Where the following is defined:

```
    static const JCharacter* kTextMenuTi-
    tleStr = "Text";
    static const JCharacter* kTextMenuStr =
      "Hello world! | Goodbye cruel world!";

    enum
    {
      kHello = 1,
      kGoodbye
    };
```

The window was created as before, except that the member function that prevents the window from resizing, `LockCurrentSize()`, was called. We also create a `JXStat-icText` object as before, but this time we use a member variable, `itsText`, so we can change it later.

7.1.1 JXMenuBar and JXTextMenu

What's really new in this program is the `JXMenuBar` object and the `JXTextMenu` object. The menubar is created to stick to the top of the window and resize horizontally as the window resizes (though in this case the window won't be resizing because we called `Lock-CurrentSize()` on the window). The `JXTextMenu` object that comes next is actually created by the menubar with the title passed to it in the `AppendTextMenu()` function. After the menu is created, it is populated with items with the `SetMenuItems()` function. This is a useful way to add a number of menu items given a string that specifies them, in this case `kTextMenuStr`. After a `JXTextMenu` is created, items can also be added with the `AppendItem()` and `InsertItem()` function calls. The function `SetUpdate-Action()` specifies how the individual menu items are disabled before opening. In our case, we don't want anything disabled, but some programs need to update the menu each time it is opened based on the current program state. The next function, `ListenTo()`, is the heart of the functionality of menus and of much of the power of JX. To understand this function, we need to discuss the base class that defines it, `JBroadcaster`.

7.1.2 JBroadcaster

A huge proportion of the classes in JX are derived from the `JBroadcaster` class. As a result, all of these classes can send messages via the `JBroadcaster::Broadcast()` function and receive messages by overriding the `JBroadcaster::Receive()` function. The messages that are sent via `Broadcast()` are objects themselves, derived from the `JBroadcaster::Message` class.

When a `JBroadcaster` derived object needs to receive a message from another broadcasting object, it must first call `ListenTo()` with a pointer to the broadcasting object passed as an argument. For example, if we create an object that needs to listen to another object that it creates, the code in the listening object would be the following:

```
ListeningObject:ListeningObject()
{
  BroadcastingObject* obj = new BroadcastingOb-
ject();
  ListenTo(obj);
)
```

When the broadcasting object broadcasts a message, the `Receive()` function of the listening object is called, with a pointer to the broadcasting object and a reference to the message being broadcast both passed as arguments to `Receive()`. An example implementation of the `Receive()` function might look like the following:

```
Listeningobject::Receive
  (
  J Broadcaster* sender,
  const Message& message
  )
{
  if (sender == myBroadcastingObject &&
      message. ls(BroadcastingObject: :kSomething-
Happened))
    (
    RespondToBroadcast();
    )
}
```

When a `JBroadcaster` derived object needs to send a message, it must call `Broadcast()`with a reference to the appropriate `JBroadcaster::Message` derived object as its argument. Often this message is a member class of the broadcasting object.

When the object broadcasts, all of the `JBroadcaster` derived objects that have called `ListenTo()` with a pointer to the broadcasting object as the argument will have their `Receive()` function called with the appropriate sender and message as arguments. The order in which they receive the message is arbitrary.

7.1.3 The MenuHelloDir: :Receive() function

In the case of menus, a `JXMenu::ItemSelected` message is broadcast whenever a menu item is selected. In our case, the `MenuHelloDir` object creates the `JXMenuBar` and attaches a `JXTextMenu` to it. It then calls `ListenTo()` with a pointer to the `JX-TextMenu` as the argument. In the `MenuHelloDir::Receive()` function, this particular message is checked and `HandleMessageMenu()` is called when the message is received.

```
void
MenuHelloDir::Receive
  (
  JBroadcaster*   sender,
  const Message&  message
  )
{
  if (sender == itsTextMenu &&
      message.Is(JXMenu::kItemSelected))
    {
    const JXMenu::ItemSelected* selection =
      dynamic_cast(const JX-
Menu::ItemSelected*, &message);
    assert( selection != NULL );
    HandleTextMenu(selection->GetIndex());
    }
  else
```

```
        {
        JXWindowDirector::Receive(sender,message);
        }
}
```

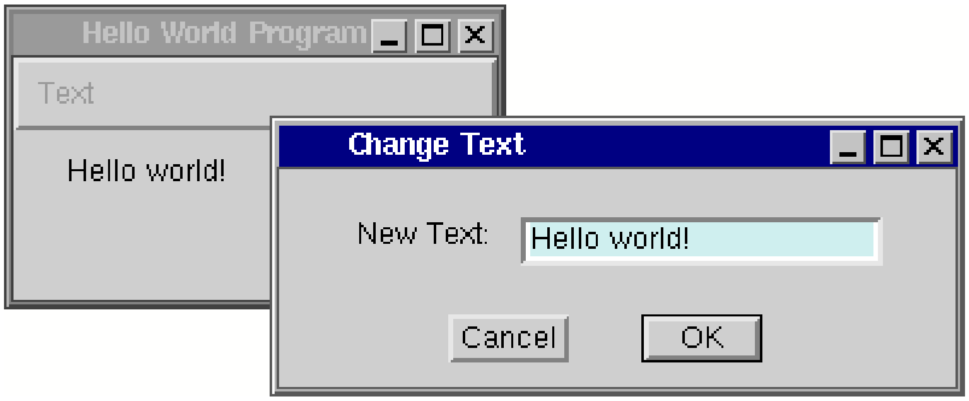
When the `JXTextMenu`, `itsTextMenu`, has an item selected, it broadcasts the `JXMenu ::ItemSelected` message. In the `MenuHelloDir::Receive()` function, we check for this message by checking its type, which in this case is `kltemSelected`. Since the message we received was a pointer to a `JBroadcaster::Message` object, we need to cast it to a `JXMenu::ItemSelected` object. This is done with `dynamic_cast`. Once we have cast the message properly, we can extract any information that it contains, which in this case is an index into the menu. We call a function `HandleTextMenu()` with the index of the menu that was called.

```
void
MenuHelloDir::HandleTextMenu
  (
  const JIndex index
  )
{
  if (index == kHello)
    {
    itsText->SetText("Hello world!");
    }
  else if (index == kGoodbye)
    {
    itsText->SetText("Goodbye cruel world!");
    }
}
```

It is in this small function that the text of the window is changed. The values we check the index against are typically part of an enum defined at the time the menu items were defined.

Chapter 8

Dialog Boxes



The MenuHello program improved upon the HelloWorld program by allowing the user to change the displayed message. The drawback with the menu based approach is that the user had a limited number of messages to choose from. Ideally, the user could display any message, as long as it fit in the space provided. For this to work, we need an input field for the user to type in their desired message. This input field could be in the main window, above or below the message field, but this would appear awkward. A better approach would be to pop up a dialog box containing an input field whenver the user wished to change the message. In this chapter, we will review the sample program DialogHello which does just that.

Like menus, dialog boxes rely on `JBroadcaster` messages for transferring their information. At some point, typically after selecting an appropriate menu item, a dialog box is created, filled with relevant information, and then activated with the `BeginDialog()` function. At this point, the window will be suspended if the dialog is a blocking dialog, or it will simply continue on as before. The user will then perform whatever actions are needed in the dialog box, which in our case involves simply typing in a new message. When the user is finished with the dialog box, he has the option of selecting the `Ok` or `Cancel` buttons or their equivalent. Regardless of how the dialog box is closed, the calling object receives a `JXDialogDirector::Deactivated()` message (assuming that the calling object called `ListenTo()`). This message object has a member function called `Successful()` that returns whether the dialog box was closed with `Ok` or `Cancel`.

8.1 The DHStringInputDialog class

The `DHStringInputDialog` class is derived from the `JXDialogDirector` class, which, like the other directors we have discussed, is derived from the `JXWindowDirec-`

`tor` class.For this reason, the constructor code is quite similar to our previous director's constructors. As in the previous example, most of the initialization code is contained in the `BuildWindow()` function.

```
void
DHStringInputDialog::BuildWindow()
{
  JXWindow* window =
    new JXWindow(this, 280,90, "Change Text");
  assert( window != NULL );
  SetWindow(window);

  JXTextButton* cancelButton =
    new JXTextButton("Cancel", window,
      JXWidget::kFixedLeft, JXWid-
get::kFixedBottom,
      70,60, 50,20);
  assert( cancelButton != NULL );

  JXTextButton* okButton =
    new JXTextButton("OK", window,
      JXWidget::kFixedRight, JXWid-
get::kFixedBottom,
      150,60, 50,20);
  assert( okButton != NULL );
  okButton->SetShortcuts("^M");

  itsText =
    new JXInputField(window,
      JXWidget::kHElastic, JXWidget::kFixedTop,
      100,20, 150,20);
  assert( itsText != NULL );

  JXStaticText* obj1 =
    new JXStaticText("New Text:", window,
      JXWidget::kFixedLeft, JXWidget::kFixedTop,
      30,20, 65,20);
  assert( obj1 != NULL );

  SetButtons(okButton, cancelButton);
  itsText->SetMaxLength(30);
}
```

Since this class is still a `JXWindowDirector`, all of the corresponding functions still apply, so, as before, a `JXWindow` is created and passed in via `SetWindow()`. The `JX-DialogDirector` class then requires at least a `Cancel` button, but in this case, both an `Ok` and `Cancel` button are created and passed in via the `SetButtons()` function. Typically, dialog boxes are designed such that the `Return` key accepts the input and dismisses the dialog. This is accomplished by calling `SetShortcuts()` on the `Ok` button with "`^M`" as the argument.

Having set up the basic dialog, we are now able to add the controls required for our specific application, which in this case is a `JXInputField` for entering the text that we wish to display in the main window. A `JXStaticText` object is also added as a label for the input field. Since our window is of a limitted size, we need to restrict the size of the text. This is done by calling `SetMaxLength()` on the input field object with the appropriate length.

Typically, data needs to be passed into a dialog box so it can reflect the current state of the program. This is done in the `DHStringInputDialog`'s constructor.

```
  DHStringInputDialog::DHStringInputDialog
    (
    JXWindowDirector* supervisor,
    const JCharacter* str
    )
    :
    JXDialogDirector(supervisor, kTrue)
  {
    BuildWindow();
    itsText->SetText(str);
  }
```

After calling the `BuildWindow()` code listed above, the current text in the main window is passed into the input field with the `SetText()` function.

After the dialog is dismissed, we need to access the text that was entered. This is done with our `GetString()` function.

```
const JString&
DHStringInputDialog::GetString()
  const
{
  return itsText->GetText();
}
```

This is called by the main window after hearing of the dismissal or deactivation of the dialog window. Our main window, which in this case is a `DialogHelloDir`, contains code to create and listen to the dialog window, and extract the pertinant information when the dialog is deactivated.

8.2 DialogHelloDir

The constructor of the `DialogHelloDir` class is the same as that of the `MenuHelloDir` class discussed above except for one line.

```
itsDialog = NULL;
```

Which is the last line in the constructor. The dialog box that the `DialogHelloDir` class owns is only non-`NULL` when it is active. Since we need to check for messages from the dialog box in the `Receive()` function, the code will fail and crash if our variable is not `NULL` and is not valid. We therefore set `itsDialog` to be `NULL` everytime the dialog box becomes invalid.

The dialog specific code comes into play after selecting a menu item.

```
void
DialogHelloDir::HandleTextMenu
  (
  const JIndex index
  )
{
  if (index == kChangeText)
    {
    CreateInputDialog();
    }
  else if (index == kQuit)
    {
    (JXGetApplication())->Quit();
    }
}
```

When the `Change text` menu item is selected, a new function, `CreateInputDialog()`, is called.

```
void
DialogHelloDir::CreateInputDialog()
{
        assert ( itsDialog == NULL );
        itsDialog =
          new DHStringInputDialog(this, itsText-
>GetText());
        assert ( itsDialog != NULL );

        ListenTo(itsDialog);

        itsDialog->BeginDialog();
}
```

Since we also make sure to set `itsDialog` to `NULL` whenever the dialog is invalid, we `assert` that it is `NULL` now to prevent somehow trying to create a dialog when another was already created. After verifying that we don't yet have an active dialog, we create a new one. We set the dialog's initial text to be the text contained in the main text field. We next call `ListenTo()` with the dialog so that we will hear when it is deactivated. Finally we call `BeginDialog()` which activates the dialog. The dialog is now created and active, and we need only to listen for its deactivation message.

As stated above, when the user presses the `Ok` or `Cancel` buttons, the dialog is deactivated, and a message is broadcast which the `DialogHelloDir` listens for in its `Receive()` function.

```
  else if (sender == itsDialog &&
           mes-
sage.Is(JXDialogDirector::kDeactivated))
    {
    const JXDialogDirector::Deactivated* info =
      dynamic_cast(const JXDialogDirec-
tor::Deactivated*,
                   &message);
```

```
        assert( info != NULL );

        if (info->Successful())
          {
          GetNewTextFromDialog();
          }

        itsDialog = NULL;
        }
```

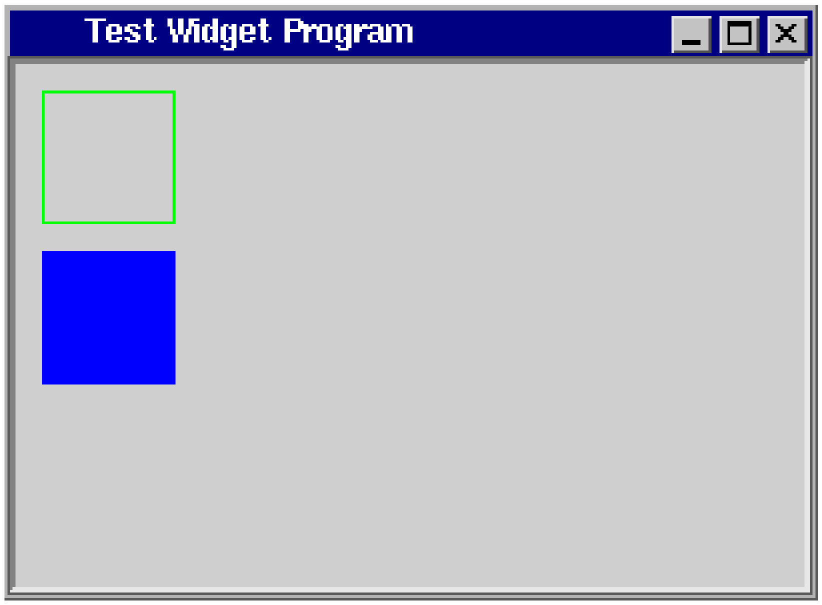
The first thing that must be done after verifying that the dialog box's deactivated message was received, is to convert the `Message` object into a `JXDialogDirector::Deactivated` object with a dynamic cast. The resulting object contains a member function called `Successful()` that returns whether the dialog box was closed with the `Ok` button or the `Cancel` button. In the above code, if the `Ok` button was pressed, `Successful()` returns kTrue, and the function `GetNewTextFromDialog()` is called. The member variable `itsDialog` is set to `NULL` when we are finished with the dialog box, because it will be automatically deleted. The `GetNewTextFromDialog()` function is responsible for extracting the edited text from the dialog box's input field.

```
    void
    DialogHelloDir::GetNewTextFromDialog()
    {
      assert ( itsDialog != NULL );

      const JString& str = itsDialog->GetString();
      itsText->SetText(str);
    }
```

The dialog box's text is extracted with its `GetString()` member function and passed into the `JXStaticText` object that contains the main window's text.

Chapter 9

Widgets



We now shift gears to discuss the `JXWidget` class and its descendants, in order to demonstrate how to write custom classes derived from them. The `widget` tutorial includes a `JXWidget` derived class called simply `Widget` that implements a very simple widget.

The constructor of `Widget` is quite simple.

```
Widget::Widget
  (
  JXContainer* enclosure,
  const HSizingOption hSizing,
  const VSizingOption vSizing,
  const JCoordinate x,
  const JCoordinate y,
  const JCoordinate w,
  const JCoordinate h
  )
  :
  JXWidget(enclosure, hSizing, vSizing, x, y, w, h)
{
  SetBorderWidth(kJXDefaultBorderWidth);
}
```

The only code we call is `SetBorderWidth()` which sets the width in pixels of the border that surrounds the widget. The default for `JXWidget` is to have a border width of zero, so we need to specify a non-zero width for our widget. It is worth noting that the parameters passed into our `Widget` class are typical of `JXWidget` derived classes. This is generally the core set of parameters, since this is what `JXWidget` itself requres.

9.1 The Draw and DrawBorder Functions

Beyond this simple initialization, there are two virtual functions that must be present, `Draw`(), and `DrawBorder()`. The function `Draw()` is called by the event loop whenever any portion of the widget needs to be redrawn.

```
void
Widget::Draw
  (
  JXWindowPainter& p,
  const JRect&     rect
  )
{
  JXColormap* cmap = GetColormap();

  p.SetPenColor(cmap->GetGreenColor());
  p.Rect(10, 10, 50, 50);

  p.SetFilling(kTrue);
  p.SetPenColor(cmap->GetBlueColor());
  p.Rect(10, 70, 50, 50);
}
```

The `Draw()` function is passed a `JXWindowPainter` and a `JRect`. The painter is used to hide the system dependant details of drawing, and to provide a uniform interface for drawing to the screen, to an image, or to a printer. The rectangle is the area that needs to be redrawn.

There are a number of painting functions available which are be listed in the file `JPainter.h`. We have chosen for our example just a sample to demonstrate the use of these functions. Our first step is to get the pointer to the colormap which will be used to set the colors for drawing. We then use this pointer to access the color green for the `SetPenColor()` function. The first rectangle drawn is only an outline, but the second rectangle is filled since we called `SetFilling()` with `kTrue`.

We are now left with only the `DrawBorder()` function to implement.

```
void
Widget::DrawBorder
  (
  JXWindowPainter& p,
  const JRect&     frame
  )
{
  JXDrawDownFrame(p, frame, kJXDefaultBorderWidth);
}
```

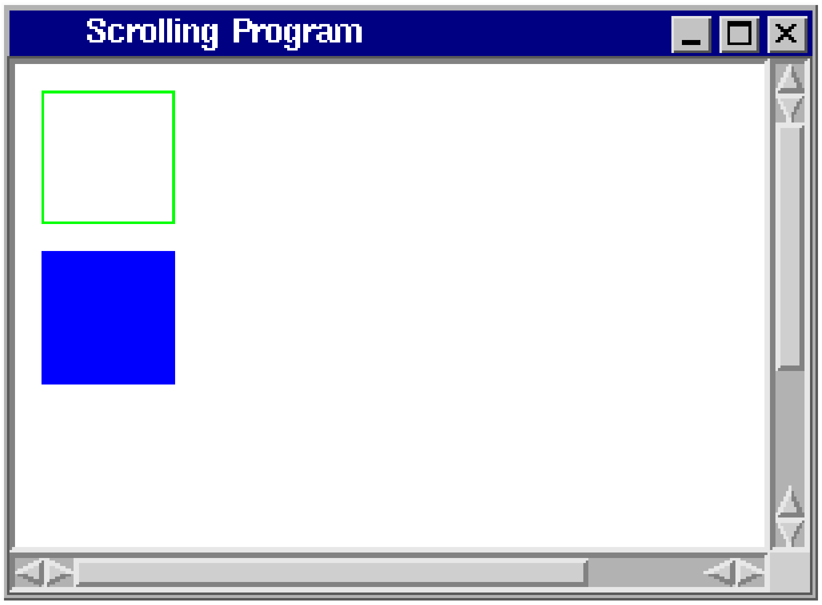
We implement a typical border here by calling `JXDrawDownframe()`.

To use our new `Widget` class we just need to instantiate it in our window director's `BuildWindow()` function.

```
void
WidgetDir::BuildWindow()
{
  JXWindow* window =
    new JXWindow(this, 300,200, "Test Widget Pro-
gram");
  assert( window != NULL );

  SetWindow(window);

  window->SetMinSize(300,200);
  window->SetMaxSize(800,600);

  Widget* widget =
    new Widget(window, JXWidget::kHElastic, JXWid-
get::kVElastic,
      0, 0, 300, 200);
  assert( widget != NULL );
}
```

Chapter 10

Scrolling Widgets



The widget in the previous example illustrated the minimum reuirements for a `JXWidget` derived object. Since a majority of the functionality of a program is is `JXWidget` derived objects, both standard and custom, one must be able to do a great deal more with widgets than just draw them. This example looks at wigdets that need to be bigger than the window that contains them. As is typical for this situation, we place scrollbars to the right and bottom of the widget (as needed) such that the user can scroll over the entire widget.

Before we can proceed with a discussion of the code, we must define three new terms, `bounds`, `aperture`, and `frame`. The `frame` and `aperture` differ only in the width of the border. If the border width is zero, they are the same, otherwise the `frame` includes the border while the `aperture` does not. For the case of the non-scrolling widget in the previous example, the `aperture` and `bounds` are the same. When the widget is scrollable, the `aperture` is the visible portion of the widget, while the `bounds` is the actual size of the widget. Drawing is always done in the bounds coordinate system. These coordinates are independant of scrolling, so the `Draw()` function can always do the same thing regardless of the position of the scrollbars.

The actual difference in code between our scrolling widget and the non-scrolling widget in the previous example are quite small. The widget must first be derived from `JXScrollableWidget` rather than `JXWidget`. It also no longer needs to define the `DrawBorder()` function, because `JXScrollableWidget` defines it for us. Finally, we must set the bounds of the widget. The `ScrollingWidget`'s constructor does all that needs to be done.

```
ScrollingWidget::ScrollingWidget
  (
  JXScrollbarSet* scrollbarSet,
  JXContainer* enclosure,
  const HSizingOption hSizing,
  const VSizingOption vSizing,
  const JCoordinate x,
  const JCoordinate y,
  const JCoordinate w,
  const JCoordinate h
  )
  :
  JXScrollableWidget(scrollbarSet, enclosure, hSiz-
ing, vSizing,
x, y, w, h)
{
  SetBounds(500, 400);
}
```

The `ScrollingWidget` class is derived from `JXScrollableWidget` and includes an additional parameter, the `JXScrollbarSet` which we will encounter in our discussion of the `JXScrollingWidgetDir` class. The bounds of our new widet are set via the `SetBounds()` function. The `Draw()` function is left exactly the same as that in the previous example. One difference in the drawing, however, is seen when the programs are run. The `ScrollingWidget`'s background is automatically set to white, because it has the current focus (ie. all of the key presses will be passed automatically to it).

10.1 Scrolling keys

One feature of JXScrollingWidget derived classes is their automatic handling of navigation keys. The Page Up/Down keys scroll the widget by a page, the Up/Down arrow keys scroll the widget by a single increment (the increment is dependant upon the application), and the Home/End keys scroll to the beginning and ending of the document respectively. Holding down the Alt key alters the behaviour of some of these keys. Page Up/Down and Home/End, when pressed with the Alt key, scroll the widget horizontally, rather than vertically. JXScrollingWidgets can also be scrolled using a wheel mouse if available.

The only discussion that now remains is how to set up the widget with scrollbars, which is done in the `ScrollingWidgetDir` class.

10.2 The ScrollingWidgetDir class

Most of the code in the `ScrollingWidgetDir` class is identical to that in the director for the non-scrolling widget in the previous chapter. The only addition is the code to deal with scrollbars. This can be found in the director's `BuildWindow()` function.

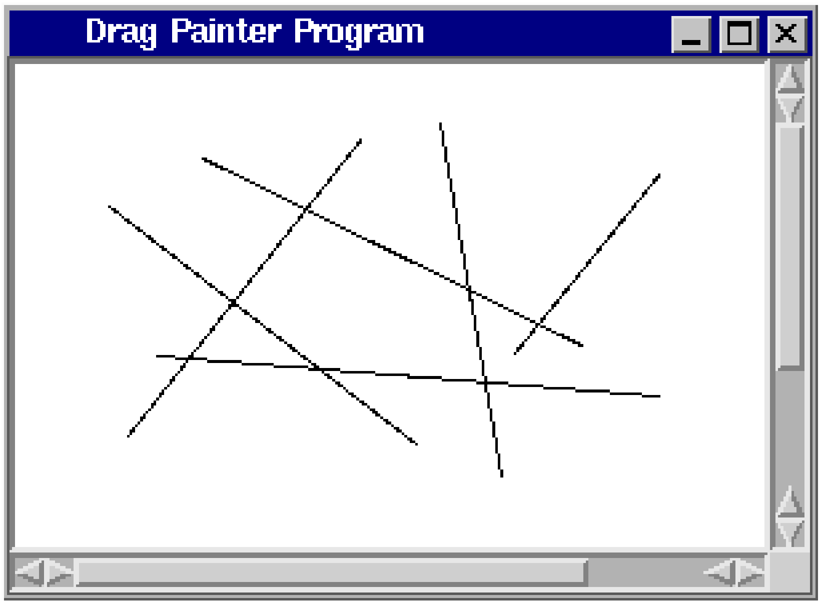
```
void
ScrollingWidgetDir::BuildWindow()
{
  JXWindow* window =
    new JXWindow(this, 300,200, "Scrolling Pro-
gram");
  assert( window != NULL );
  SetWindow(window);

  window->SetMinSize(300,200);
  window->SetMaxSize(800,600);

  JXScrollbarSet* scrollbarSet =
    new JXScrollbarSet(window,
      JXWidget::kHElastic, JXWid-
get::kVElastic, 0,0, 300,200);
```

```
      assert( scrollbarSet != NULL );

      ScrollingWidget* widget =
        new ScrollingWidget(scrollbarSet,
          scrollbarSet->GetScrollEnclosure(),
          JXWidget::kHElastic, JXWidget::kVElastic,
          0, 0, 10, 10);
      assert( widget != NULL );

      widget->FitToEnclosure(kTrue, kTrue);
    }
```

In this version, before we create the widget, we must first create a `JXScrollbarSet`. The scrollbar set is itself derived from `JXWidget`, so it takes the same arguments as the widget in last chapter's example. The new code comes into play when we create our new widget. As discussed above, the `ScrollingWidget,` like all `JXScrollableWidget` derived objects, requires a `JXScrollbarSet` to be passed as the first parameter. In the past, the second parameter has always been the `JXWindow` that we created earlier. In this case, the scrollbar set itself must control the enclosure. We therefore call the member function `GetScrollEnclosure()` to access the enclosure that will contain the widget. If the widget grows as a result, for example, of the window being resized, then the scrollbarset will adjust the scrollbars such that they reflect the correct size of the `aperture` relative to the `bounds`. If after resizing the `aperture` becomes bigger than the `bounds` in either dimension, then the scrollbar responsible for that dimension is hidden, and re-appears only when the `aperture` again becomes smaller than the `bounds`.

Chapter 11

Drag Painter



Up to this point we have considered only widgets that can not be changed, and that do not resond to mouse or keyboard events. To take full advantage of the power and flexibility of `JXWidget`, we must incorporate this functionality into our discussion. The `Drag-Widget` class captures both mouse and keyboard events in the functions `HandleMouse-Down()`, `HandleMouseDrag()`, `HandleMouseUp()` and `HandleKeyPress()`.

To demonstrate these functions, we will build a program that allows the user to draw random lines on the window. A new line will be created when the user clicks the mouse. The start of the new line will be the point where the mouse was first clicked. The end of the line will follow the mouse while it is dragged with its button still down. The end of the line will be fixed when the user releases the mouse button. The user can clear all of the lines by pressing the letter `c`, and can quit by pressing the letter `q`. One powerful feature that will be introduced in this chapter is the `JXDragPainter`. This provides a way to interactively draw to the screen in reponse to a mouse drag.

The window director in this case, `DragWidgetDir`, is basically the same as the director in the previous example. All of the functionality that we are adding to this program comes from the widget itself, so we will limit our discussion to the `DragWidget` class.

To set up our widget, we need to set its bounds as in the previous example, but we also need to initialize a data structure to represent the lines that the user will draw on the screen.

```
DragWidget::DragWidget
  (
  JXScrollbarSet*     scrollbarSet,
  JXContainer*        enclosure,
  const HSizingOption hSizing,
  const VSizingOption vSizing,
  const JCoordinate   x,
  const JCoordinate   y,
  const JCoordinate   w,
  const JCoordinate   h
  )
  :
  JXScrollableWidget(scrollbarSet, enclosure, hSiz-
ing, vSizing,
x, y, w, h)
{
  SetBounds(500, 400);

  itsPoints = new JArray<JPoint>;
  assert( itsPoints != NULL );
}
```

The constructor arguments are just the basic arguments for any `JXScrollableWidget` object. Also, as with our previous example, we need to set the bounds with the `SetBounds()` function. We then create a `JArray` of `JPoint`s. `JArray` is a template class, so you can instantiate an array of any struct. For making arrays of objects, use the `JPtrArray` class. There are many ways we could store the information needed to represent the points, but for the sake of simplicity, a simple array of `JPoint`s is used here. A `JPoint` is a struct containing an x and a y member. The odd `JPoint`s of this array hold the starting points, while the even `JPoint`s hold the ending points of the lines.

The `Draw()` function in this widget is a little different from our previous widgets.

```
  void
  DragWidget::Draw
    (
    JXWindowPainter& p,
    const JRect&     rect
    )
  {
    JSize count = itsPoints->GetElementCount();

    for (JSize i = 1; i <= count; i += 2)
      {
      p.JPainter::Line(itsPoints->GetElement(i),
                       itsPoints->GetElement(i+1));
      }
  }
```

Our first step is to set the drawing color, in this case to the color black which we access from the global colormap object. We then need to iterate through the points in our array and draw the lines defined by them. We draw lines between the `ith` and `(i+1)th` points as we iterate by twos.

As mentioned before, a new line is created in our widget whenver the user clicks the left mouse button. When a button click event is received the the application, our widget's `HandleMouseDown()` function is called.

```
void
DragWidget::HandleMouseDown
  (
  const JPoint&         pt,
  const JXMouseButton   button,
  const JSize           clickCount,
  const JXButtonStates& buttonStates,
  const JXKeyModifiers& modifiers
  )
{
  if (button == kJXLeftButton)
    {
    JPainter* p = CreateDragInsidePainter();

    p->Line(pt, pt);
    }
  else
    {
    ScrollForWheel(button, modifiers);
    }
  itsStartPt = itsPrevPt = pt;
}
```

Several arguments are passed to the `HandleMouseDown()` function that describe the state of the mouse and its buttons. The `pt` argument gives the coordinates where the mouse was clicked. The `button` argument contains the button that was pressed, either `kJXLeftButton`, `kJXRightButton`, or `kJXMiddleButton`, or the numbered buttons 4 and 5 (typically activated by the scrolling of a wheel mouse). The `clickCount` is the number of times that the mouse button has been pressed and released in rapid succession. The `buttonStates` struct contains the state of all of the mouse buttons, and finally, the `modifiers` struct contains the state of the modifier keys like `Shift`, `Ctrl`, and `Meta`.

In the code above, we create a `JXDragPainter` with the `CreateDragInsidePainter()` function if the left button was pressed. We then draw a line to this painter. We store the current point for future reference in the `HandleMouse-Drag()` function.

The `HandleMouseDrag()` function is called whenver the mouse button is left down.

```
void
DragWidget::HandleMouseDrag
```

```
      (
      const JPoint&         pt,
      const JXButtonStates& buttonStates,
      const JXKeyModifiers& modifiers
      )
    {
      const JBoolean scrolled = ScrollForDrag(pt);

      JPainter* p = GetDragPainter();

      if (buttonStates.left() && pt != itsPre-
    vPt && p != NULL)
        {

        if (!scrolled)
          {
          p->Line(itsStartPt, itsPrevPt);
          }
        p->Line(itsStartPt, pt);
        }

      itsPrevPt = pt;
    }
```

When dragging the mouse, there are times when the mouse leaves the widget's apurture. The appropriate response in this case would be to scroll the widget. This allows the user to drag to a place in the scrolling widget that is not currently visible. `JX` takes care of scrolling the widget appropriately with the function `ScrollForDrag()`. This function returns a boolean indicating whether or not it scrolled the window. It is called first in the `HandleMouseDrag()` function to scroll if necessary.

Having created the `JXDragPainter` in the `HandleMouseDown()` function, we simply need to access it here with the `GetDragPainter()` function. If for some reason we did not create the drag painter in the `HandleMouseDown()` function, then the painter returned will be `NULL`. Before drawing the new line, we verify that we should be drawing it by checking that the left button is pressed, that the mouse has moved, and that the drag painter is not equal to `NULL`.

Drag painters use the process called `rubber-banding`, where the line we are drawing expands and contracts like a rubber band as we move the mouse relative to the starting point. This works by erasing the old line and drawing a new line for the new end-point. The problem is that we don't want to erase whatever is drawn beneath the line. We solve this problem by drawing with the `XOR` mode. In the `XOR` mode, the line drawn is not all black, but black when the line is over white pixels and white when the line is over black pixels. When the line is over a colored pixel, it is drawn with whatever color is the bitwise flipped color corresponding to the original pixel's color. Using this technique, a line is erased by drawing it twice. The first time it shows the line, the second time it flips the pixels back to their original color.

It is worth mentioning at this point that it is not always necessary to use a drag painter in order to show a responce to a mouse drag. Sometimes an adequate solution is to change the internal state of an object while dragging and repeatedly call Redraw() to reflect that state change.

With this in mind, we see that the code above first draws the old line before drawing the new one. This is to erase the old one. The one exception is if the widget was scrolled. If it was, the line was already erased when the widget was redrawn. Drawing the old line in this case wouldn't erase it, but would display it. We would then be left with a series of lines instead of the one that we want.

After erasing the old line and drawing the new one, we save the current point so we will know what to erase later. This continual drawing and erasing will continue until the mouse is released in the `HandleMouseUp()` function.

```
void
DragWidget::HandleMouseUp
  (
  const JPoint&         pt,
  const JXMouseButton   button,
  const JXButtonStates& buttonStates,
  const JXKeyModifiers& modifiers
  )
{
  JPainter* p = GetDragPainter();

  if (button == kJXLeftButton && p != NULL)
    {
    p->Line(itsStartPt, itsPrevPt);

    DeleteDragPainter();

    itsPoints->AppendElement(itsStartPt);
    itsPoints->AppendElement(itsPrevPt);

    Refresh();
    }
}
```

The user has now adjusted the line the way they want it, so we want to clean things up and save the points. As before, we extract the drag painter with the `GetDragPainter()` function and check to see if we are drawing a line by checking for the left button and that the drag painter isn't equal to `NULL`. Once we have verified that we are finishing a line, we erase the current line as before by drawing over it in the `XOR` mode, and we delete the drag painter with `DeleteDragPainter()`. All we have left to do is save the points by appending, in order, the starting mouse point and the last drag point to our `JArray` with the `AppendElement()` function. When we then refresh the screen with `Refresh()`, the new line is drawn with the previously drawn lines.

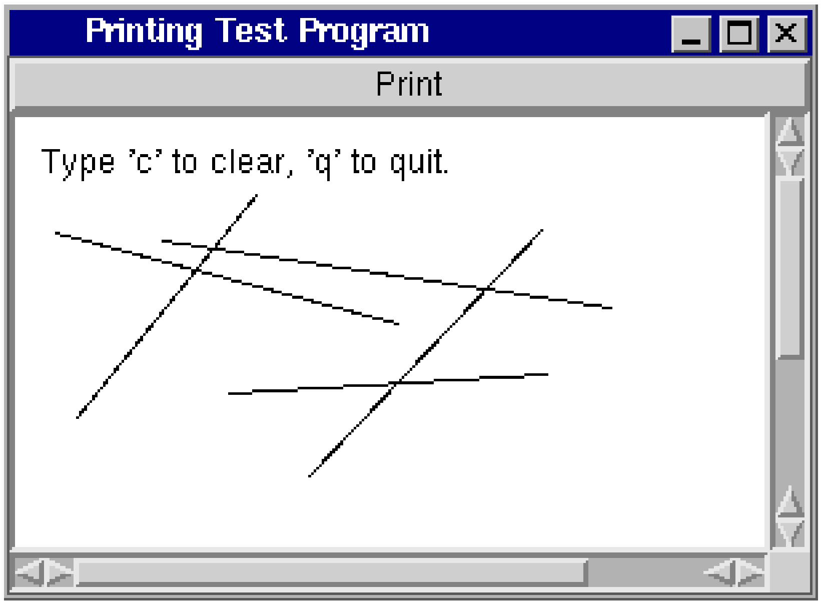
We have now covered the typical mouse functions found in most programs. The other major widget functionality that we need to discuss is the use of key press events. These events are received by the widget in the `HandleKeyPress()` function.

```
void
DragWidget::HandleKeyPress
  (
  const int             key,
  const JXKeyModifiers& modifiers
  )
{
  if (key == ’c’)
    {
    itsPoints->RemoveAll();
    Refresh();
    }

  else if (key == ’q’)
    {
    (JXGetApplication())->Quit();
    }

  else
    {
    JXScrollableWid-
get::HandleKeyPress(key,modifiers);
    }
}
```

In this example program, the only key presses that we are interested in are the `'c'` and `'q'` keys which are used to clear all of the lines and to quit the program respectively. If we receive any keys other than these, we need to pass them down to our base class, which in this case is `JXScrollableWidget`. If we don't pass them down, we may lose some of the functionality of our widget since one of its base classes may require key presses (see Section 10.1). The samecan also be true with the mouse functions we discussed above.

Chapter 12

Printing



Type 'c' to clear, 'q' to quit.

Now that we’ve discussed how to display widgets and interact with them, we need a way to print out our widget. As discussed in the first widget example, drawing functions are hardware independent. In other words, we use the same code to draw to a window, an image, or a printer. We may, however, need to draw different things when printing than when drawing to a window. To handle this, we’ve added a few new functions to our new widget, the `PrintWidget`. First, we’ve added a function named `DrawStuff()` that draws both to the window and to the printer. This function draws all of the lines as before.

```
void
PrintWidget::DrawStuff
  (
  JPainter& p
  )
{
  JSize count = itsPoints->GetElementCount();

  for (JSize i = 1; i <= count; i += 2)
    {
    p.Line(itsPoints->GetElement(i), itsPoints-
>GetElement(i+1));
    }
}
```

This is simply the same code that was in the `Draw()` function in the drag painter example. We needed to separate this code, because our `Draw()` function will only be called when the window needs to be redrawn. We now have a `Print()` function that also needs to use the same code. Rather than duplicate the code, we created this new function that both the `Draw()` and `Print()` functions could call. Now if there are differences between what we draw to the screen and the printer we have a way to separate it. For example, in our

`Draw()` function, we now write a few instuctions to the screen that we don’t want printed out.

```
void
PrintWidget::Draw
  (
  JXWindowPainter& p,
  const JRect& rect
  )
{
  p.String(10, 10, "Type ’c’ to clear, ’q’ to quit.");

  DrawStuff(p);
}
```

To draw the lines that the user created, we simply pass the `JXWindowPainter` to the `DrawStuff()` function. Before we call the function, however, we do the window specific drawing first, which in this case is a string.

The `Print()` function is similar to the `Draw()` function, in that it also calls the `DrawStuff()` function, but it must, in addition, prepare the page for printing.

```
void
PrintWidget::Print
  (
  JPagePrinter& p
  )
{
  const JCoordinate header-
Height = p.JPainter::GetLineHeight();

  p.OpenDocument();

  if (p.NewPage())
```

```
        {

        JRect pageRect = p.GetPageRect();
        p.JPainter::String(pageRect, "Printing Test",
                           JPainter::kHAlignCenter);
        p.LockHeader(headerHeight);

        DrawStuff(p);

        p.CloseDocument();
        }
    }
```

The `Print()` function is passed a `JPrinter` rather than a `JXWindowPainter`, since we need this to set up the page. For reference, we then save a copy of the line height. Also notice, because of a peculiarity of C++'s desing, we must specify that both the GetLineHeight and String functions are found in JPainter. The reason we have to do this is this particular case, is that derived classes of JPainter overload these functions with different arguments.

The first step in preparing a postscript page for printing is to call `OpenDocument()`. Then, for every page, we need to call `NewPage()`, which returns a boolean telling whether or not the print job has been cancelled. This is useful for mutli-page documents that take a long time to print. Once we have successfully created the new page, we can start drawing to it.

We can use the `GetPageRect()` function to get the size of the page, which is useful for drawing headers and footers. As an example, the header is drawn above with the `String()` function. The `LockHeader()` function is then called to tell the `JPrinter` that no more drawing should be done in the header. This shifts the origin so the rest of the drawing doesn't have to worry about it. We then draw the lines with the `DrawStuff()` function and close the document with `CloseDocument()`. After closing the document, it is sent to the printer by `JX`.

In addition to the code in the widget, its director must create a printer object and set it up, typically allowing the user to adjust the parameters with one or more dialog boxes.

12.1 Preparing a director for printing

In the last section we reviewed the code required for a widget to print itself out to a printer, but we need a way for the user to control and initiate printing. In this section we will add functionality to the print widget's director to provide printing access to the user.

This program was designed such that the user can adjust printing parameters with a dialog box, and initiate printing with a button. It would probably be more typical to initiate printing with a menu item, but this just gives us an opportunity to discuss the use of `JXButton`'s.

We get things started in the director's constructor by creating a `JXPSPrinter` object.

```
PrintWidgetDir::PrintWidgetDir
  (
  JXDirector* supervisor
  )
  :
  JXWindowDirector(supervisor)
{
  BuildWindow();
  itsPrinter =
    new JXPSPrinter(GetDisplay(), (GetWindow())-
>GetColormap());
  assert( itsPrinter != NULL );

  ListenTo(itsPrinter);
}
```

Here we've just chosen to call the `BuildWindow()` function first before setting up the printer object. We also set our dialog box pointer to `NULL` to prevent the program from crashing in the `Receive()` function (see Section 8.2). The `BuildWindow()` function demonstrates the use of buttons.

```
void
PrintWidgetDir::BuildWindow()
{
  JXWindow* window =
    new JXWindow(this, 300,200, "Printing Test Pro-

gram");
  assert( window != NULL );

  SetWindow(window);

  window->SetMinSize(300,200);
  window->SetMaxSize(800,600);

  itsPrintButton =
    new JXTextButton("Print", window,
      JXWidget::kHElastic, JXWidget::kFixedTop,
      0, 0, 300, 20);

  ListenTo(itsPrintButton);

  JXScrollbarSet* scrollbarSet =
    new JXScrollbarSet(window,
      JXWidget::kHElastic, JXWid-
get::kVElastic, 0,20, 300,180);
  assert( scrollbarSet != NULL );

  itsWidget =
    new PrintWidget(scrollbarSet,
      scrollbarSet->GetScrollEnclosure(),
      JXWidget::kHElastic, JXWidget::kVElastic,
      0, 0, 10, 10);
  assert( itsWidget != NULL );

  itsWidget->FitToEnclosure(kTrue, kTrue);
}
```

The only thing here that we haven't seen before is the creation of a `JXTextButton`. Since the `JXTextButton` is descended from `JXWidget`, it includes all of the standard widget arguments plus a label argument that is prepended to the list. We also call `ListenTo()` on the button, because just like menus and dialog boxes, buttons communicate by broadcasting messages. The print widget is member data, so that we can call its `Print()` member function.

Since both the button and the dialog box use broadcasting to communicate, the rest of the code for the director is in the `Receive()` function.

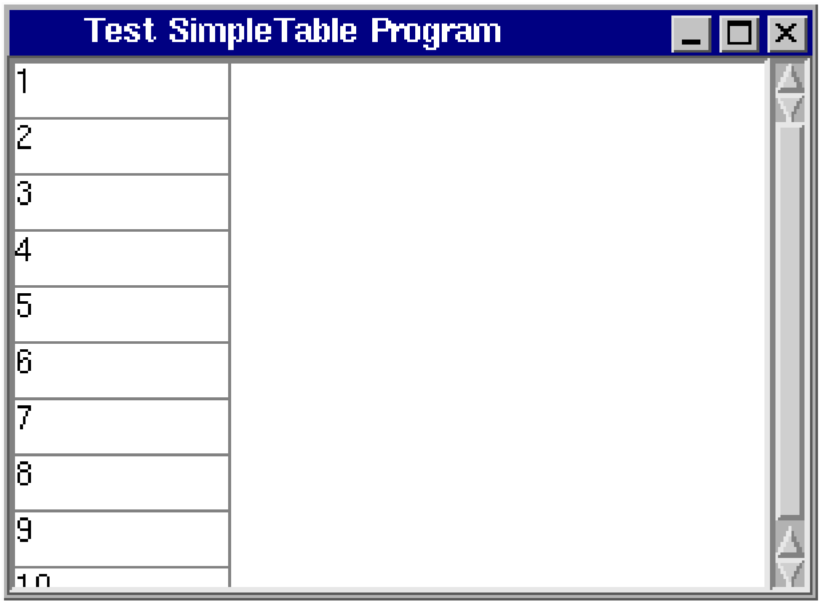
```
void
PrintWidgetDir::Receive
  (
  JBroadcaster*  sender,
  const Message&  message
  )
{
  if (sender == itsPrintButton && mes-
sage.Is(JXButton::kPushed))
    {
    itsPrinter->BeginUserPrintSetup(this);
    }

  else if (sender == itsPrinter &&
       message.Is(JPrinter::kPrintSetupFinished))
    {
    const JPrinter::PrintSetupFinished* info =
      dy-
namic_cast(const JPrinter::PrintSetupFinished*,
                   &message);
    assert( info != NULL );
    if (info->Successful())
      {
      itsWidget->Print(*itsPrinter); }
      }
    }

  else
    {
    JXWindowDirector::Receive(sender, message);
    }
}
```

When the print button is pressed, it broadcasts a `kPushed` message. Since there is no information in the message, we don't need to cast it from a `Message` object, we can just respond to the button press. We've chosen to have the button trigger the print dialog box. In this case, the printer object handles the creation and activation of the print dialog box, so all we have to do is call `BeginUserPrintSetup()` and everything is taken care of for us. All we are left to do is listen to the printer for PrintSetupFinished messages.

When we receive the deactivation PrintSetupFinished message from our printer, we need to call the widget's `Print()` function only if the user hit the `Ok` button on the dialog box. Again, the check for success is handled by the printer message object in the Succesful() function. If this function returns true, we call our widget's `Print()` function. A similar procedure is used when using the Page Seup dialog box.

Chapter 13

Simple Table



We’re now going to shift gears again and discuss a higher level set of class, those derived form `JTable`. Strictly speaking, a table is any object whose data is contained in a grid. They can contain numbers or text or images or any combination. Some tables are also editable.

In this chapter we’ll begin our discussion of the table functionality available in `JX`. This particular example contains no real data, and only demonstrates the procedure for setting up and drawing a table. All of the code is contained within the widget istelf. The window director treats a table as any scrollable widget.

The table widget’s constructor sets up the table.

```
SimpleTable::SimpleTable
  (
  JXScrollbarSet*     scrollbarSet,
  JXContainer*        enclosure,
  const HSizingOption hSizing,
  const VSizingOption vSizing,
  const JCoordinate   x,
  const JCoordinate   y,
  const JCoordinate   w,
  const JCoordinate   h
  )
  :
  JXTable(kDefRowHeight, kDefColWidth,
          scrollbarSet, enclosure, hSizing, vSiz-
ing, x, y, w, h)
{
  AppendCol(kDefColWidth);
  for (JIndex i = 1; i <= 10; i++)
    {
    AppendRow(kDefRowHeight);
    }
}
```

with the following values defined:

```
const JCoordinate kDefRowHeight  = 20;
const JCoordinate kDefColWidth   = 80;
```

The first step in preparing a table is to tell the base class, `JTable`, the number of rows and columns that it will contain. `JTable` will use these values to determine what it needs to draw. Table widgets use the `JTable` functions `AppendCol()` and `AppendRow()` to add columns and rows respectively. To draw the cells that these rows and columns define, `JTable` uses the `TableDrawCell()` function instead of the standard widget `Draw()` function. In most circumstances, a custom table should not need to define the `Draw()` function at all, but if it must, it must also call the base class's `Draw()` function.

This table's `TableDrawCell()` function is very basic:

```
void
SimpleTable::TableDrawCell
  (
  JPainter&      p,
  const JPoint&  cell,
  const JRect&   rect
  )
{
  const JString cellNumber(cell.y);
  p.JPainter::String(rect, cellNumber);
}
```

This gets called by `JTable` every time the cell passed to the function needs to be redrawn.

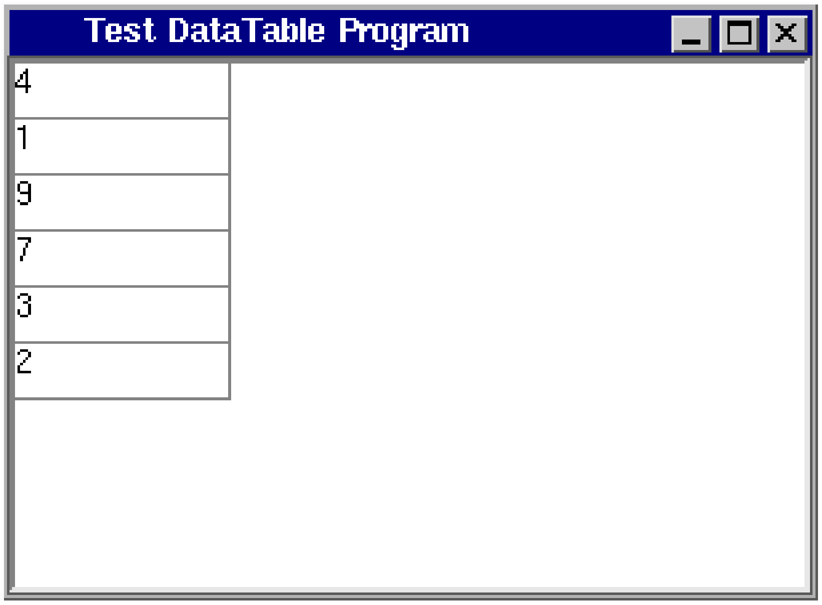
The `p` argument is the `JPainter` that you use to draw to the screen. It is the painter that was passed to the base class's `Draw()` function, so it could be drawing to a window, image, or printer. The `rect` argument is the boundary of the cell. The clipping rectangle has been set to this rectangle so anything that you draw outside this rectangle will not be displayed. The final argument, `cell`, gives the row and column of the cell that needs to be redrawn, where

```
    cell.x = column
    cell.y = row
```

In this function, arbitrary `JPainter` functions can be used. In this case, we've chosen to display numbers corresponding to the row number. We take advantage of ability of a `JString` to convert numbers to strings, and we print this resulting string with the `String` painter function.

Chapter 14

Data Table



```
4
1
9
7
3
2
```

The table in the previous chapter was rather contrived in that it contained no real data. Most actual tables have a data object that contains the data and broadcasts various messages as the data changes. The `JTable` class has built-in support for a data class called `JTableData`. `JX` contains a few table classes that use a `JTableData` data object, like `JXFloatTable`, and `JXStringTable`. Tables do not need to use a `JTableData` object, however. In the example, we'll use a `JArray` to contain the data. This array will be created in the window director and passed to the table. As the array changes, and the elements in that array change, the table will adjust to match the array.

We'll first consider the table before looking at the window director. As before, the constructor sets up the table by adding the appropriate rows and columns. In this example, this is done to synchronize the table with the data.

```
DataTable::DataTable
  (
  JArray<JIndex>*     data,
  JXScrollbarSet*     scrollbarSet,
  JXContainer*        enclosure,
  const HSizingOption hSizing,
  const VSizingOption vSizing,
  const JCoordinate   x,
  const JCoordinate   y,
  const JCoordinate   w,
  const JCoordinate   h
  )
  :
  JXTable(kDefRowHeight, kDefColWidth,
          scrollbarSet, enclosure, hSizing, vSiz-
ing, x, y, w, h)
{
  itsData = data;
  AppendCol(kDefColWidth);

  for (JIndex i = 1; i <= itsData-
>GetElementCount(); i++)
    {
    AppendRow(kDefRowHeight);
    }

  ListenTo(itsData);
}
```

This table, like the previous, has one column which is just added with the `Append-Col()` function. The row count, however, needs to match the number of elements in the `JArray` that is passed in to the constructor as its first argument. This is accomplished by adding rows with the `AppendRow()` function in a loop from 1 to `itsData->GetElementCount()`. Finally we call `ListenTo()` on the array so that we can receive the messages broadcast by the array as it changes.

The `TableDrawCell()` function is almost exactly the same as that in the previous example with a small but important exception.

```
void
DataTable::TableDrawCell
  (
  JPainter&     p,
  const JPoint&   cell,
  const JRect&   rect
  )
{
  const JString cellNumber(itsData-
>GetElement(cell.y));
  p.String(rect, cellNumber, JPainter::kHAlignLeft,
           JPainter::kVAlignTop);
}
```

As before, we create a `JString` and draw it to the screen with the `String` function. The difference here is how the `JString` is created. In this case, instead of using the row number as the data, we extract the appropriate value from the `JArray` using the `GetElement()` function with the row number as its argument.

The table now only needs to adjust itself as its data changes. Since the data array broadcasts when it changes, we perform the adjustments in the `Receive()` function.

```
void
DataTable::Receive
  (
  JBroadcaster*  sender,
  const Message&  message
  )
{

  if (sender == itsData)
    {

    if (mes-
sage.Is(JOrderedSetT::kElementsInserted))
      {
      JOrderedSetT::ElementsInserted* info =
        dy-
namic_cast(JOrderedSetT::ElementsInserted*, &mes-
sage);
      assert(info != NULL);

      for (JIndex i = 1; i <= info-
>GetCount(); i++)
        {
        InsertRow(info->GetFirstIndex());
```

```
          }
        }

      else if (mes-
  sage.Is(JOrderedSetT::kElementsRemoved))
        {
        JOrderedSetT::ElementsRemoved* info =
          dy-
  namic_cast(JOrderedSetT::ElementsRemoved*, &mes-
  sage);
        assert(info != NULL);

        for (JIndex i = info->GetLastIndex();
             i >= info->GetFirstIndex() ; i--)
          {
          RemoveRow(i);
          }
        }

      else if (mes-
  sage.Is(JOrderedSetT::kElementChanged))
        {
        JOrderedSetT::ElementChanged* info =
          dy-
  namic_cast(JOrderedSetT::ElementChanged*, &mes-
  sage);
        assert(info != NULL);

        TableRefreshCell(info->GetFirstIndex(), 1);
        }
      }

    else
      {
      JXTable::Receive(sender, message);
      }
  }
```

In the `Receive()` function, we are looking for messages fromour array pointer, `itsData`. At this point, we are interested in the messages `ElementsInserted`, `ElementsRemoved`, and `ElementChanged`. For both `ElementsInserted` and `ElementsRemoved`, we cast them to the appropriate message class and then iterate over the indices specified by the member functions `GetLastIndex()` and `GetFirstIndex()`, calling `InsertRow()` for each index in the former messsage, and `RemoveRow()` for each index in the latter message. These functions, like `AppendRow()` and `AppendCol()`, adjust the internal size of the table. When we receive the `ElementChanged` message, we just call `TableRefreshCell()` on the row specified by the message's `GetFirstIndex()`.

As mentioned earlier, we've chosen here to have the window director create the data in its constructor.

```
DataTableDir::DataTableDir
  (
  JXDirector* supervisor
  )
  :
  JXWindowDirector(supervisor)
{
  itsData = new JArray<JIndex>;
  assert(itsData != NULL);

  itsData->AppendElement(4);
  itsData->AppendElement(1);
  itsData->AppendElement(9);

  BuildWindow();

  itsData->AppendElement(7);
```

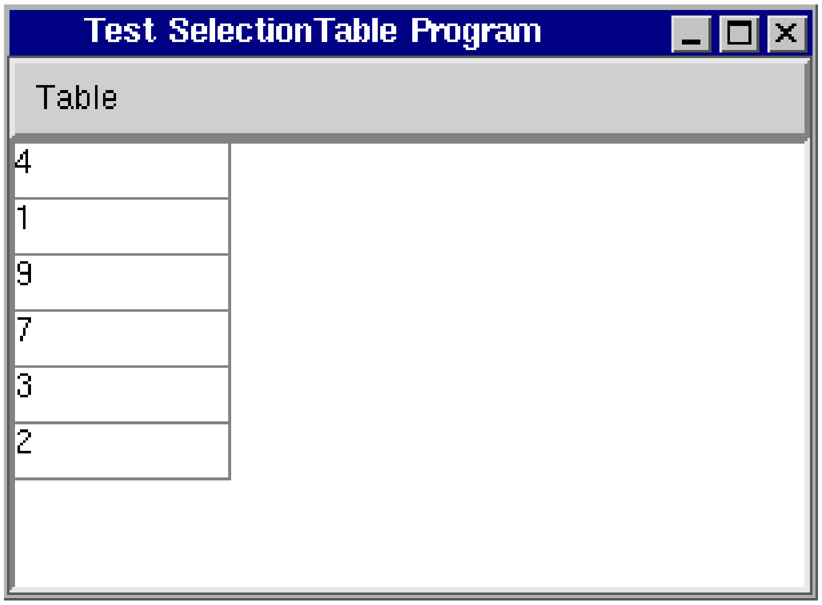
```
      itsData->AppendElement(8);
      itsData->AppendElement(2);
      itsData->SetElement(5,3);
    }
```

Before calling `BuildWindow()`, we need to first create our `JArray`, because we need to pass it to our table when we create it. `JArray` is a template, so we need to specify the type when we create it. In this case we want an array of the type `JIndex`. The file `jTypes.h` contains many typedefs including `JIndex`.

After creating the array, we add a few values with the function `AppendElement()`. In the `BuildWindow()` function the table is created, and in the table's constructor, it is sychronized with the data. Any changes made to the data made after the table is created is must be communicated to the table with broadcasts. To test the `Receive()` function of our table, we add some more elements to our data and change one of the elements with the `SetElement()` function. When the program is run, we can see that the table has adjusted itself to match the data.

Chapter 15

Selection Table



```
4
1
9
7
3
2
```

In this chapter we'll look at another feature of tables, selection. We've also added a menubar so we can insert new rows at the selection and remove selected rows. The table will manage the only menu on the menubar, so the menubar will have to be passed into the table's constructor.

```
SelectionTable::SelectionTable
  (
  JXMenuBar*          menuBar,
  JArray<JIndex>*     data,
  JXScrollbarSet*     scrollbarSet,
  JXContainer*        enclosure,
  const HSizingOption hSizing,
  const VSizingOption vSizing,
  const JCoordinate   x,
  const JCoordinate   y,
  const JCoordinate   w,
  const JCoordinate   h
  )
  :
  JXTable(kDefRowHeight, kDefColWidth,
          scrollbarSet, enclosure, hSizing, vSiz-
ing, x, y, w, h)
{
  itsData      = data;
  itsTableMenu = NULL;

  AppendCol(kDefColWidth);

  for (JSize i = 1; i <= itsData-
>GetElementCount(); i++)
    {
    AppendRow(kDefRowHeight);
    }

  ListenTo(itsData);
  itsTableMenu = menuBar-
>AppendTextMenu(kTableMenuTitleStr);
  itsTableMenu->SetMenuItems(kTableMenuStr);
  itsTableMenu-
>SetUpdateAction(JXMenu::kDisableNone);
  ListenTo(itsTableMenu);
}
```

The only difference in this constructor relative to that in the previous example is the addition of a menubar argument and the creation of a menu, `itsTableMenu`. The menu title and the items in the menu are defined by the strings:

```
static const JCharacter* kTableMenuTitleStr = "Ta-
ble";
static const JCharacter* kTableMenuStr =
  "Insert %k Meta-I | Remove %k Meta-
R| Quit %k Meta-Q";

enum
{
  kInsertCmd = 1,
  kRemoveCmd,
  kQuitCmd
};
```

These will be used to insert rows, remove rows, and quit the program.

Our discussion of selection will begin in the `HandleMouseDown()` function, where the selection starts.

15.1 JTableSelection

The way selection works depends on the application. For this program, clicking on a cell selects it, and clicking on the area outside the cells deselects all of the cells.

```
void
SelectionTable::HandleMouseDown
  (
  const JPoint&           pt,
  const JXMouseButton     button,
  const JSize             clickCount,
  const JXButtonStates&   buttonStates,
  const JXKeyModifiers&   modifiers
  )
{
  if (button == kJXLeftButton)
    {
    JPoint cell;
    if (GetCell(pt, &cell))
      {
      JTableSelection& selection = GetTableSelec-
tion();
      selection.SelectCell(cell);
      }
    else
      {
      JTableSelection& selection = GetTableSelec-
tion();
      selection.ClearSelection();
      }
    }
  else
    {
    ScrollForWheel(button, modifers);
    }
}
```

Once the left mouse button is pressed we need to determine whether to select a cell or clear the cells. The function `GetCells()` returns a boolean indicating whether or not the mouse click was in a cell. If it was, the cell passed into it is set to the cell containing the click point. So, if `GetCell()` returns true, we know we need to select the cell that it returns, otherwise we clear the selection. Either way, we need to have access to the `JTableSelection` object that `JTable` instantiates. We get this object with the `JTable` member function `GetTableSelection()`. We use two of the selection object's functions here, `SelectCell()` and `ClearSelection()`, which select the specified cell and clear the entire selection respectively.

In the `Draw()` function, we need to indicate the selection.

```
void
SelectionTable::TableDrawCell
  (
  JPainter&     p,
  const JPoint&   cell,
  const JRect&   rect
  )
{
  HilightIfSelected(p, cell, rect);
  JString cellNumber(itsData->GetElement(cell.y));
  p.String(rect, cellNumber, JPainter::kHAlignLeft,
           JPainter::kVAlignTop);
}
```

The last two lines of this function are the same as the table in the previous example. All we've added is a call to `HilightIfSelected()`, which is a `JTable` member function, that performs the default action, drawing the background of the cell in the default selection color. If we want to designate the selection in some other way, we can check the cell's selection ourselves and do whatever we need to do based on the results.

The way we make use of selection in this program is in inserting and removing cells. These are handled with the menu, so we'll first look at our `UpdateTableMenu()` function.

```
void
SelectionTable::UpdateTableMenu()
{
  JTableSelection& selection = GetTableSelection();

  if (selection.GetSelectedCellCount() == 1)
    {
    itsTableMenu->EnableItem(kInsertCmd);
    }
  else
    {
    itsTableMenu->DisableItem(kInsertCmd);
    }
}
```

In the constructor, we told the menu to leave the items enabled by calling `SetUpdateAction()` with `kDisableNone`. So all we need to do in our update function is disabled those items that should be disabled. We havechosen in this case to only allow insertion if exactly one cell is selected. To check this, we first get the selection object with the `GetTableSelection()` function. The `JTableSelection` class has a member function `GetSelectedCellCount()` that tells us the number of selected cells in the table. If this function indicates that there is only one selected cell, we enable the insert menu item, otherwise, we disable it.

We are now ready to look at the `HandleTableMenu()` function.

```
void
SelectionTable::HandleTableMenu
  (
  const JIndex index
  )
{
  JTableSelection& selection = GetTableSelection();
  JPoint cell;
  if ((index == kInsertCmd) &&
      selection.GetFirstSelectedCell(&cell))
    {
    itsData-
>InsertElementAtIndex(cell.y, kDefInsertValue);
    }

  else if (index == kRemoveCmd)
    {
    JTableSelectionIterator iter(&selection);

    JPoint cell;
    while (iter.Next(&cell))
      {
      itsData->RemoveElement(cell.y);
      }
    }

  else if (index == kQuitCmd)
    {
    JXGetApplication()->Quit();
    }
}
```

When we are inserting a cell, we just need to insert a value in to the `JArray` that contains our data, because as it is added, the table will automatically adjust to the change in the `Receive()` function. We get the `JTableSelection` object first with the `Get-`

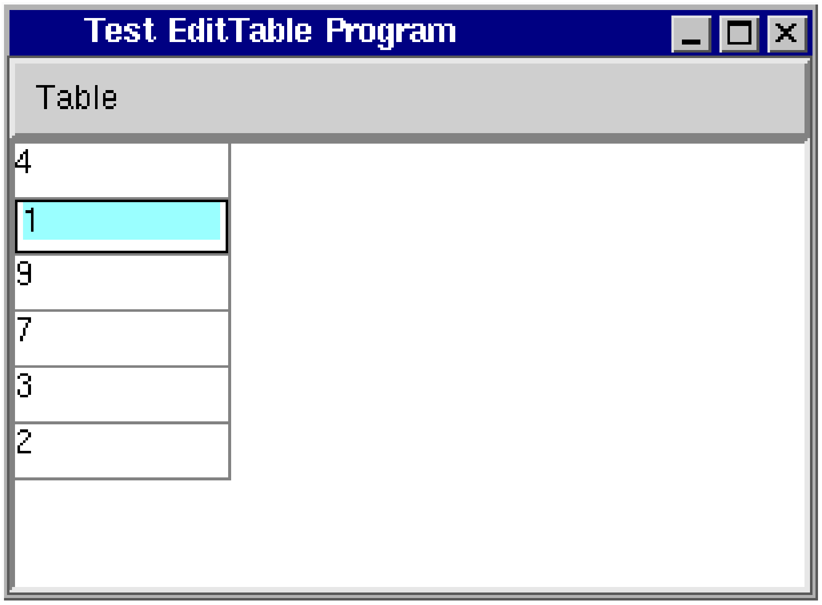
`TableSelection()` access function. We then use the `JTableSelection` access function `GetFirstSelectedCell()` to get the cell that is selected. Finally, we insert the default value into our data with the `InsertElementAtIndex()` function. The default value was defined at the top of the source file as:

```
const JIndex     kDefInsertValue  = 12;
```

When the user selects the remove menu item, we need to iterate over all of the selected cells and remove them all. Fortunately, `JTableSelection` has an iterator, so that the iteration is done for us automatically with the `JTableSelectionIterator` class. This iterator object is created with our selection object. We can then iterate over each selected cell with the iterator's `Next()` member function. For each selected cell, we simply remove the array element, because the table will adjust itself in the `Receive()` function as it does for insertions. If we didn't use an iterator, we would have to be very careful when deleting elements, so that we would not try to access elements outside the array.

Chapter 16

Edit Table



Our table program has been improving slowly, with new features being added with each new tutorial. The biggest weakness of our previous table programs was the inablity to change the values listed in the table. The `JTable` class has an editing facility built in, and this chapter will discuss the use of the functions required to make a table editable.

The first step in making a table editable is to derive it from the `JXEditTable` class which is what we do in this table's constructor below.

```
EditTable::EditTable
  (
  JXMenuBar*           menuBar,
  JArray<JIndex>*      data,
  JXScrollbarSet*      scrollbarSet,
  JXContainer*         enclosure,
  const HSizingOption hSizing,
  const VSizingOption vSizing,
  const JCoordinate   x,
  const JCoordinate   y,
  const JCoordinate   w,
  const JCoordinate   h
  )
  :
  JXEditTable(kDefRowHeight, kDefColWidth, scroll-
barSet,
    enclosure, hSizing, vSizing, x, y, w, h)
{
  itsData              = data;
  itsTableMenu         = NULL;
  itsIntegerInputField = NULL;

  AppendCol(kDefColWidth);
  for (JSize i = 1; i <= itsData-
>GetElementCount(); i++)
    {
    AppendRow(kDefRowHeight);
    }
  ListenTo(itsData);

  itsTableMenu = menuBar-
>AppendTextMenu(kTableMenuTitleStr);
  itsTableMenu->SetMenuItems(kTableMenuStr);
  itsTableMenu-
>SetUpdateAction(JXMenu::kDisableNone);
  ListenTo(itsTableMenu);
}
```

Other than that one change, there is only one difference between this constructor and the constructor in the previous example, the addition of an input field pointer that we need to set to `NULL` to make sure that we only have one created at a time. This is the input field that we will use to edit a cell.

Typically, we begin editing when the mouse double clicks on a cell. This would be handled in the `HandleMouseDown`() function.

```
void
EditTable::HandleMouseDown
  (
  const JPoint&       pt,
  const JXMouseButton   button,
  const JSize       clickCount,
  const JXButtonStates&   buttonStates,
  const JXKeyModifiers&   modifiers
  )
{
  if (button == kJXLeftButton)
    {
    JPoint cell;
    if (GetCell(pt, &cell))
      {
      if (clickCount == 2)
        {
        JTableSelection& selection = GetTableSelec-
tion();
        selection.ClearSelection();
        BeginEditing(cell);
        }
      else
        {
        JTableSelection& selection = GetTableSelec-
tion();
        selection.SelectCell(cell);
        }
      }
    else
      {
      JTableSelection& selection = GetTableSelec-
tion();
      selection.ClearSelection();
      }
    }
  else
    {
    ScrollForWheel(button, modifers);
    }
}
```

This is very similar to the `HandleMouseDown()` function in the previous program except for an extra test. After we have determined that the mouse was clicked inside a cell we check whether it was clicked once or twice. If the mouse was only clicked once, we select the cell as before, but if it was clicked twice, we need to begin editing.

Before editing, it is customary to clear the selection. We do this the same way as we would if the user had clicked outside of the cell. We can then begin editing by calling the

`JTable` function, `BeginEditing()`. This function is called with the cell where editing is to begin.

When the `BeginEditing()` function is called, the table automatically sets up everything necessary, and calls the virtual `CreateXInputField()` function. This function returns an input field that the table will use to edit the value contained in the cell.

```
JXInputField*
EditTable::CreateXInputField
  (
  const JPoint&        cell,
  const JCoordinate       x,
  const JCoordinate       y,
  const JCoordinate       w,
  const JCoordinate       h
  )
{
  assert( itsIntegerInputField == NULL );

  itsIntegerInputField =
    new JXIntegerInput(this, kFixedLeft, kFixed-

Top, x,y,w,h);
  assert( itsIntegerInputField != NULL );

  itsIntegerInputField->SetValue(itsData-
>GetElement(cell.y));
  itsIntegerInputField->SetLowerLimit(0);

  return itsIntegerInputField;
}
```

In this function, we first make sure we haven't yet created an input field by asserting that our input field is `NULL`. This is why it was set to `NULL` in the constructor. We then create the input field with the arguments passed in. We can then use our data array to fill the input field with the current value. The `JArray` function `GetElement()` is used to access the value of the cell being edited. This value is placed into the input field as the current value with the function `SetValue()`. If we need to, we can also further set up the input field with other specification, like a lower limit. For our example, the lower limit is set to `0` with the `SetLowerLimit()` function. The function finally returns the input field that it created.

By default, when the user presses the `Return` key while editing, the table will automatically end editing and call the `ExtractInputData()` virtual function.

```
JBoolean
EditTable::ExtractInputData
  (
  const JPoint& cell
  )
{
  assert( itsIntegerInputField != NULL );

  if (!itsIntegerInputField->InputValid())
    {
    return kFalse;
    }

  JInteger number;
  const JBoolean ok = itsIntegerInputField-
>GetValue(&number);
  assert( ok );

  itsData->SetElement(cell.y, number);
  return kTrue;
}
```

When this function is called, we know that the user has edited the old value and accepted the new value that they have entered. We now need to extract the value from the input field and place that value into the appropriate index in the data array.

The `ExtractInputData()` function returns a boolean indicating whether or not the value that the user typed into the field was valid. We can check that by calling the `InputValid()` function, and returning `kFalse` if it returns false. If the value is valid, we extract the value from the input field with the `GetValue()` function. One important result of calling the InputValid() function is the message displayed to the user if there is an error in their input. This allows the user to understand the problem and thereby correct it. Once we have validated the new value and extracted it, we place this value into our array in the appropriate place with the `SetElement()` function.

After calling `ExtractInputData()` the table calls `DeleteXInputField()` so the derived class can delete its input field.

```
void
EditTable::DeleteXInputField()
{
  delete itsIntegerInputField;
  itsIntegerInputField = NULL;
}
```

This function is quite simple in our case, simply deleting the input field if it is non-`NULL`, and setting our pointer to `NULL`.

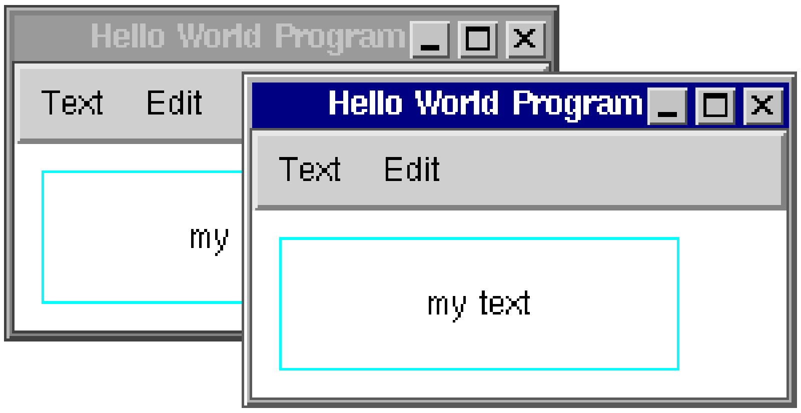
After the user presses the Return key, and editing has finished as discussed above, the default JTable behavior is to begin editing on the cell below it. This behavior can be altered be derived classes. Other ways that JTable adjusts to input as a cell is being edited include:

Return key	End editing and begin editing the cell below.
Shift-return	End editing and begin editing the cell above.
Tab key	End editing and begin editing the cell to the right.
Shift-tab	End editing and begin editing the cell to the left.
Meta-Left-Arrow	End editing and begin editing the cell to the left.
Meta-Right-Arrow	End editing and begin editing the cell to the right.
Meta-Up-Arrow	End editing and begin editing the cell above.
Meta-Down-Arrow	End editing and begin editing the cell below.
Meta-keypad	Using the 2, 4, 6, and 8 on the numeric keypad are equivalent to using the down, left, right, and up arrow keys respectively.

All of these behaviours can be altered by derived classes by overriding the HandleKeyPress function.

Chapter 17

Clipboard



We are now going to return to the code that is similar to the dialog box example in chapter 8, in order to discuss a feature that most programs require, the clipboard. Instead of using a `JXStaticText` object like we did in chapter 8, we'll create our own object that draws text and handles the clipboard. The clipboard is the standard technique for transfering information from one running program to another. The clipboard can be used to transfer virtually anything, from text and images, to sound and video. In this example, we'll use the clipboard to transfer the text of one window to another window.

To use the clipboard, we need to set up a few things in order to act as a source of a copy operation.

```
ClipboardWidget::ClipboardWidget
  (
  const JCharacter*   text,
  JXMenuBar*       menuBar,
  JXContainer*     enclosure,
  const HSizingOption hSizing,
  const VSizingOption vSizing,
  const JCoordinate   x,
  const JCoordinate   y,
  const JCoordinate   w,
  const JCoordinate   h
  )
  :
  JXWidget(enclosure, hSizing, vSiz-
ing, x, y, w, h),
  itsText(text)
{
  JXColormap* cmap = GetColormap();
  SetBackColor(cmap->GetWhiteColor());

  itsEditMenu = menuBar-
>AppendTextMenu(kEditMenuTitleStr);
  itsEditMenu->SetMenuItems(kEditMenuStr);
  itsEditMenu-
>SetUpdateAction(JXMenu::kDisableNone);
  ListenTo(itsEditMenu);

}
```

Typically, a program's interface to the clipboard is `Copy` and `Paste` menu items. This is why we have added a menu to the menubar that was passed in. The menu is defined by the following strings:

```
static const JCharacter* kEditMenuTi-
tleStr = "Edit";
static const JCharacter* kEditMenuStr =
  "Copy %k Meta-C | Paste %k Meta-V";

enum
{
  kCopyCmd = 1,
  kPasteCmd
};
```

Before we can act as a source of a copy operation, we need to add what are called selection targets. These are the types of data this widget will support. These targets are added wth the `GetDisplay()->RegisterXAtom(OurAtomString)` function. Since we are only supporting text, we don't need to register any atoms, because text is standard, so is already registered. We can get the selection manager at any time with the `GetSelec-tionManager()` function.

Once the widget has prepared itself for using the clipboard, it can make use of it with the edit menu. The use of this menu will triger the `HandleEditMenu()` function.

```
void
ClipboardWidget::HandleEditMenu
  (
  const JIndex index
  )
{
  if (index == kCopyCmd)
```

```
        {
        JXTextSelection* data =
           new JXTextSelection(GetDisplay(), its-
    Text);
        assert(data != NULL);
        if (!(GetSelectionManager())-
    >SetData(kJXClipboardName, data))
          {
          (JGetUserNotification())->
            ReportError("Unable to copy to the X Clip-
    board.");
          }
        }
      else if (index == kPasteCmd)
        {
        Paste();
        }
    }
```

The act of copying is actually not much of an operation. We simply create the appropriate selection object (derived from `JXSelectionManager::Data`) In the case of text, `JX` has a built-in class called `JXTextSelection`. Once our selection object is created, we pass this to the `JXSelectionManager`. If it is unable to become the selection owner, the selection manager will return `kJFalse` from this function, so we report an error. The selection data object handles all of the data transfer.

Since this program can act as both a source and target, we also support pasting, which takes place when the user selects the `Paste` menu item. When this is selected, we call the `Paste()` function.

```
    void
    ClipboardWidget::Paste()
```

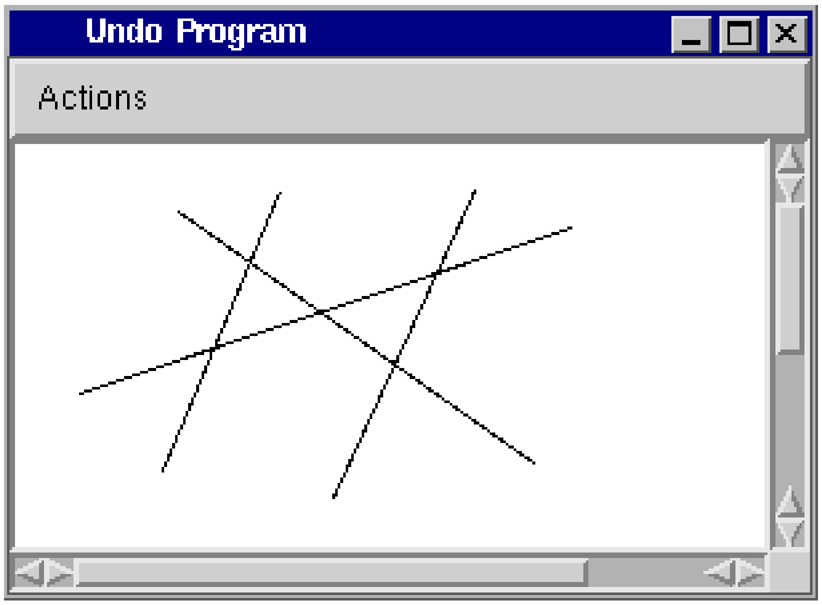
```
{
  JXWindow* window           = GetWindow();
  JXSelectionManager* selMgr = GetSelectionMan-
ager();

  JArray<Atom> typeList;
  if (selMgr->GetAvailableTypes(kJXClipboardName,
        CurrentTime, &typeList))
    {
    const JSize typeCount = type-
List.GetElementCount();
    for (JIndex i=1; i<=typeCount; i++)
      {
      const Atom atom = typeList.GetElement(i);

      if (atom == XA_STRING || atom == selMgr-
>GetTextXAtom())
        {
        unsigned char* data = NULL;
        JSize dataLength;
        Atom returnType;
        JXSelectionManager::DeleteMethod dMethod;
        if (selMgr->GetData(
            kJXClipboardName, CurrentTime,
            atom, &returnType, &data, &dataLength,
            &dMethod))
          {
          if (returnType == XA_STRING)
            {
            its-
Text.Set(reinterpret_cast<JCharacter*>(data),
                        dataLength);
            Refresh();
            }
          selMgr->DeleteData(&data, dMethod);
          if (returnType == XA_STRING)
            {
            return;
            }
          }
        else
          {
          (JGetUserNotification())->
           ReportError("Unable to retrieve text
                         from the clipboard.");
          }
        }
      }

    (JGetUserNotification())->
        ReportError("Unable to paste from clip-
board.");
    }
  else
    {
    (JGetUserNotification())-
>ReportError("Clipboard is empty.");
    }
}
```

When we paste, we find a selection type that we understand, and read from its data into our internal data.

The first step is to determine the available types. We can access this in the form of a `JArray` with the function `GetAvailableTypes()`. We then iterate through the types until we find one we can understand. If we don't find any, we display an error. If we do find a type that we understand, we call the selection manager function, `GetData()` to get the data from the selection source. Now the data must be converted to a form that our program can understand. This is completely dependent on the application. For our case, we simply need to cast the data with `reinterpret_cast()`, form an `unsigned char*` to `char*`. We use the result of this cast for our text, which we set with the `Set()` function. Whether we were successful or not, if we have called `GetData()`, then we must call the selection manager function, `DeleteData()`.

Chapter 18

Undo



We are once again going to visit the line drawing program. This program, like most, involves the editing of user data. One thing that was sorely lacking in the previous versions of this program, and is a requirement for any program that involves the editing of user data is the ability to undo changes. A single mis-typed key could destroy all of the user's data, so there must be a way to reverse the mistake. This danger is one example of why this is actually a badly designed program. It is just a useful demonstration of the need for the user to be able to reverse their changes. It is important to remember that users will often change their minds as well as make mistakes. In this chapter, we'll discuss the use of undo in a `JX` program.

This code involved in providing comprehensive undo and redo is quite involved, so we'll have to discuss it in several stages. The first involves a few changes in the constructor of the line drawing program.

```
UndoWidget::UndoWidget
  (
  JXScrollbarSet* scrollbarSet,
  JXContainer* enclosure,
  const HSizingOption hSizing,
  const VSizingOption vSizing,
  const JCoordinate x,
  const JCoordinate y,
  const JCoordinate w,
  const JCoordinate h
  )
  :
  JXScrollableWidget(scrollbarSet, enclosure, hSiz-
ing, vSizing,
x, y, w, h)
{
  itsFirstRedoIndex   = 1;
```

```
      itsUndoState        = kIdle;

      SetBounds(500, 400);

      itsPoints = new JArray<JPoint>;
      assert( itsPoints != NULL );

      itsUndoList = new JPtrArray<JUndo>;
      assert(itsUndoList != NULL);
    }
```

The way undo/redo works is through the use of an array of undo/redo objects. We save the index of the current redo object and use it to locate the appropriate undo/redo object when the user next calls for either an undo or a redo. When the user calls undo, the current undo object is replaced by a redo object, and the redo index is decreased by one. When the user then calls redo, the current redo object is replaced by an undo object and the redo index is increased by one. For internal use, we also maintain an undo state flag, telling us whether the current state, either idle, undo, or redo.

We introduce a new type of array in this constructor, the `JPtrArray`. This, as its name implies, is an array of pointers. This is the appropriate array type to use if we are storing objects. Like `JArray`, this pointer array is a template, so can work with any type we configure it to work with.

Since we are creating some dynamic objects, we'll need to delete them in the destructor.

```
    UndoWidget::~UndoWidget()
    {
      delete itsPoints;

      itsUndoList->DeleteAll();
      delete itsUndoList;
    }
```

A `JArray` contains a continuous range of memory, so deleting the array deletes its contents. This is not the case with an array of pointers. Deleting a `JPtrArray` simply deletes the pointers, not the objects that the pointers point to. To delete all of the objects, we first call `DeleteAll()` on the array before deleting it. This will safely delete all of the objects that are contained in the array.

We'll now start our discussion of undo with a function called `NewUndo()`.

```
    void
    UndoWidget::NewUndo
      (
      JUndo* undo
      )
    {
      if (itsUndoList != NULL && itsUndoState == kIdle)
        {
        const JSize undoCount = itsUndoList-
    >GetElementCount();
        for (JIn-
    dex i=undoCount; i>=itsFirstRedoIndex; i--)
          {
          itsUndoList->DeleteElement(i);
          }

        itsUndoList->Append(undo);
        itsFirstRedoIndex++;
        }

      else if (itsUndoList != NULL && itsUn-
    doState == kUndo)
        {
```

```
        assert( itsFirstRedoIndex > 1 );

        itsFirstRedoIndex--;
        JUndo* oldUndo = itsUndoList-
    >NthElement(itsFirstRedoIndex);
        delete oldUndo;
        itsUndoList-
    >SetElement(itsFirstRedoIndex, undo);

        undo->SetRedo(kTrue);
        undo->Deactivate();
        }

      else if (itsUndoList != NULL && itsUn-
    doState == kRedo)
        {
        assert(itsFirstRedoIndex <= itsUndoList-
    >GetElementCount());

        JUndo* oldRedo = itsUndoList-
    >NthElement(itsFirstRedoIndex);
        delete oldRedo;
        itsUndoList-
    >SetElement(itsFirstRedoIndex, undo);
        itsFirstRedoIndex++;

        undo->SetRedo(kFalse);
        undo->Deactivate();
        }

    }
```

This function is the heart of this process. It is called in three places, in the three different states. It is first called after a user draws a new line. After the line is created, an undo object is created and passed to this function. At this point, the undo state is still idle.

The only real complication at this stage is that the user may have performed several undos before this line was created. To keep things synchronized, all of the redos will have to be removed. This means that all of the objects in the undo array above the redo index are removed. After the redos have been removed, the undo object is appended to the array, and the redo index is incremented such that it is located at the end of the list.

This function is also called during both an undo or a redo operation. When called during a nundo, a redo object is passed in that redoes what is currently being undone. The current object is then deleted and replaced with this redo object. Finally, the redo index is set to point to this new redo object.

When this function is called during a redo operation, it does essentially the same this as during an undo object, except that the redo object is replaced with a new undo object and the redo index is placed after this new undo object.

The undo object is created when the line is first created. This occurs, as before, in the `HandleMouseUp()` function.

```
void
UndoWidget::HandleMouseUp
  (
  const JPoint&          pt,
  const JXMouseButton    button,
  const JXButtonStates&  buttonStates,
  const JXKeyModifiers&  modifiers
  )
{
  JPainter* p = GetDragPainter();

  if (button == kJXLeftButton && p != NULL)
     // p is NULL for multiple click
    {
    p->Line(itsStartPt, itsPrevPt);

    DeleteDragPainter();

    itsPoints->AppendElement(itsStartPt);
    itsPoints->AppendElement(itsPrevPt);

    UndoLine* undo = new UndoLine(this);
    assert(undo != NULL);
    NewUndo(undo);

    Refresh();
    }
}
```

The only addition to this function are those lines dealing with undo. An `UndoLine` object is created and passed to the `NewUndo()` object discussed above. These undo objects are added to the undo array as the user adds lines. Once the user has added a line, the user should be allowed to undo the line. The availablity of undo objects is given by the `HasUndo()` function.

```
JBoolean
UndoWidget::HasUndo()
  const
{
  return JConvertToBoolean(itsFirstRedoIndex > 1);
}
```

Since the redo index always points to the end of the undo objects and the beginning of the redo objects, it must be greater than `1` if there are undo objects present. Similarly, there is also a `HasRedo()` function available().

```
JBoolean
UndoWidget::HasRedo()
  const
{
  return JConvertToBoolean(itsFirstRedoIndex <=
                           itsUndoList-

>GetElementCount());
}
```

If there are redo objects available, then the redo index must be less than or equal to the total undo object count, since this index will be used to locate the current redo object. If the value is greater than the total undo count, than there are no available redo objects.

When the user initiates an undo operation, the current undo object is accessed with the `GetCurrentUndo()` function.

```
JBoolean
UndoWidget::GetCurrentUndo
  (
  JUndo** undo
  )
  const
{
  if (HasUndo())
    {
    *undo = itsUndoList-
>NthElement(itsFirstRedoIndex - 1);
    return kTrue;
    }
  else
    {
    return kFalse;
    }
}
```

This function first verifies that there are undo objects available, with the `HasUndo()` function, and then returns the undo object located below the redo index. This function is called by the `Undo()` function.

```
void
UndoWidget::Undo()
{
  assert( itsUndoState == kIdle );

  JUndo* undo;
  const JBoolean hasUndo = GetCurrentUndo(&undo);

  if (hasUndo)
    {
    itsUndoState = kUndo;
    undo->Deactivate();
    undo->Undo();
    itsUndoState = kIdle;
    }
}
```

This function first tries to access the current undo object. If it is successful, it calls that undo objects `Undo()` function which will, in our case, remove the last line created. The undo object will then be replaced by a corresponding redo object. Similar to these last two functions, are the related redo functions, `GetCurrentRedo()` and `Redo()`.

```
JBoolean
UndoWidget::GetCurrentRedo
  (
  JUndo** redo
  )
  const
{
  if (HasRedo())
    {
    *redo = itsUndoList-
>NthElement(itsFirstRedoIndex);
    return kTrue;
    }
  else
    {
    return kFalse;
    }
}
```

If there is a redo object available, this function accesses it at the current redo index. This will be called by the `Redo()` function.

```
void
UndoWidget::Redo()
{
  assert( itsUndoState == kIdle );

  JUndo* undo;
  const JBoolean hasUndo = GetCurrentRedo(&undo);

  if (hasUndo)
    {
    itsUndoState = kRedo;
    undo->Deactivate();
    undo->Undo();
    itsUndoState = kIdle;
    }
}
```

If this function is successful at acquiring the current redo object, it calls that object's `Undo()` function, which redraws the last line removed. The redo object is then replaced by the corresponding undo object.

The actual function called when a line needs to be removed is the `RemoveLastLine()` function.

```
void
UndoWidget::RemoveLastLine()
{
  const JSize count = itsPoints->GetElementCount();
  assert(count >= 2);
  JPoint start = itsPoints->GetElement(count - 1);
  JPoint end = itsPoints->GetElement(count);
  itsPoints->RemoveElement(count);
  itsPoints->RemoveElement(count - 1);
  Refresh();

  RedoLine* redo = new RedoLine(this, start, end);
  assert(redo != NULL);

  NewUndo(redo);
}
```

Since a line is made up of two points, this function removes the last two points in the point array and passes them to a `RedoLine` object, which is then installed with the `NewUndo()` function discussed above. This is the function called by the `UndoLine` object when its `Undo()` function is called.

```
void
UndoLine::Undo()
{
  itsWidget->RemoveLastLine();
}
```

A pointer to the widget is kept by the undo object so that it can access the `RemoveLastLine()` function. Before this function returns, the undo object itself has been deleted. This `Undo()` function must therefore be careful about not trying to access private data after it is called.

The function called when a line needs to be restore is the `AddLine()` function.

```
void
UndoWidget::AddLine
  (
  const JPoint& start,
  const JPoint& end
  )
{
  itsPoints->AppendElement(start);
  itsPoints->AppendElement(end);

  Refresh();

  UndoLine* undo = new UndoLine(this);
  assert(undo != NULL);

  NewUndo(undo);
}
```

This function simply adds the two points passed to it to the point array and creates a corresponding undo object which is passed to the `NewUndo()` function. This is the function called by the `RedoLine` object when its `Undo()` function is called.

```
void
RedoLine::Undo()
{
  itsWidget->AddLine(itsStartPt, itsEndPt);
}
```

This object has to maintain not only a pointer to the widget, but also the two points that make up a line. As with the `UndoLine` object, the `AddLine()` function deletes this object, so care must be taken not to access private data after it is called.

We are now ready to layout the menu items that are the user's only interface to the undo system. In the `UpdateActionsMenu()` function, which updates the menu that contains the undo and redo items, the items are enabled or disabled depending on whether or not it is relevant.

```
void
UndoWidgetDir::UpdateActionsMenu()
{
  if (itsWidget->HasUndo())
    {
    itsActionsMenu->EnableItem(kUndo);
    }
  else
    {
    itsActionsMenu->DisableItem(kUndo);
    }
  if (itsWidget->HasRedo())
    {
    itsActionsMenu->EnableItem(kRedo);
    }
  else
    {
    itsActionsMenu->DisableItem(kRedo);
    }
}
```

If the widget's `HasUndo()` function returns true, then the undo menu item is enabled, otherwise it is disabled. The process of handling the user's menu selection is then quite simple.

```
void
UndoWidgetDir::HandleActionsMenu
        (
        const JIndex index
        )
{
  if (index == kUndo)
    {
    itsWidget->Undo();
    }
  else if (index == kRedo)
    {
    itsWidget->Redo();
    }
  else if (index == kQuit)
    {
    (JXGetApplication())->Quit();
    }
}
```

If the user selects the undo menu item, the widget's `Undo()` function is called. If the user selects the redo menu item, the widget's `Redo()` function is called. The user is now able to safely edit their data, knowing that any changes can be removed, and lost data can be recovered.

Part III

Technical Overview

Chapter 19

JX Overview

Refer to Chapter 5 for an overview of the way the main JX objects work together.

19.1 Documents, Directors, and Dialogs

Each Document is responsible for a collection of data and a Window to display it in. When the user asks to close a Document, the OKToClose() method is invoked. Derived classes must override this to let the user decide whether to save the data or toss it.

A WindowDirector is only responsible for a Window and is usually owned by a Document or another WindowDirector. (It can be owned by the Application, too.) When the user asks to close a WindowDirector, the data displayed in the window is summarily tossed. However, if the user only asks to deactivate the WindowDirector, the data that is displayed is verified on the assumption that hidden data should not be invalid because the user can't get to it and fix it.

A DialogDirector is also responsible only for a Window. If it is modal, it suspends its owner while its Window is active. Closing a DialogDirector also summarily tosses the data it contains. Since one usually wants to at least look at the information that was entered before tossing it, deactivating (via EndDialog()) is the correct way to finish a dialog. DialogDirectors always call Close() after successfully deactivating. DialogDirectors automatically support OK and Cancel buttons (via SetButtons()–modal dialogs must have an OK button) and deactivate themselves when either one is pressed. (The OK button only succeeds if the information that was entered is valid.) In order to notify the dialog's owner about this, DialogDirectors always broadcast a message (JXDialogDirector::Deactivated) when they are deactivated. After activating a dialog, the owner simply has to listen for this deactivate message in Receive() and then extract any information it wants. To provide custom buttons in the dialog, the class derived from JXDialogDirector merely has to create them and ListenTo() them. If part of the action for such a button should be to close the dialog, simply call EndDialog(). Since this generates the deactivate message, extracting the data can still be done in the deactivate message handler. JXInputField objects automatically provide for simple checks on the information being entered. JXStringInput objects can be set to require a non-empty string and/or have a maximum allowable length. JXFloatInput and JXIntegerInput objects require numbers and integers, respectively, and can also constrain the value entered to be within specified limits. Derived classes can be created to implement more complex restrictions. For restrictions involving more than one value, the class derived from JXDialogDirector should override OKToDeactivate(), check that the Dialog wasn't cancelled, then check whatever conditions are required, and finally, notify the user if there is a problem. If the routine returns kFalse, the dialog window will not close. (Of course, the best approach is always to redesign the interface so that the graphical display itself enforces the restrictions. Popup menus, radio buttons, and checkboxes are the most familiar examples.) Dependencies between items in a dialog window should be handled in the Receive() method of the class derived from JXDialogDirector. For example, if an InputField is associated with a particular choice in a RadioGroup and should only be active when this choice is selected, then the DialogDirector can listen for changes in the RadioGroup selection and activate or deactive the InputField as appropriate.

The constructor for classes derived from JXDocument, JXWindowDirector, and JX-DialogDirector must create a Window and then call SetWindow(). (Code generated by jxlayout does this automatically.)

19.2 Windows

Each window must be owned by a class derived from `JXWindowDirector`. Most of the methods are self-explanatory, but one deserves some comment.

Calling `Close()`, or selecting `Quit` from the window menu provided by the X Window Manager can perform one of four actions:

1. Deactivate its WindowDirector (i.e. hide the window)
2. Close its WindowDirector
3. Close the X Displa
4. Quit the Application

The action is selectable via `SetCloseAction()`.

19.3 Widgets

Widgets provide the functionality required by all objects displayed in a window. All widgets must have an enclosure. This can be either the window or another widget. The enclosure determines a lot about how the widget acts. Showing, hiding, activating, deactivating, moving, and resizing messages are all passed down from the enclosure. This makes it possible to group a set of widgets together inside an enclosure (usually derived from `JXWidgetSet`) and have them all stay together and act as a unit. (The classic example is a set of radio buttons.) In addition, the image of a widget is always clipped to the enclosure's aperture.

There are three rectangles associated with every widget: bounds, frame, and aperture. The bounds is the actual size of the widget. The frame is the bounding rectangle that the user sees in the window. The aperture is the frame inset by the border width. This is the visible part of the bounds and the area to which enclosed widgets will be clipped when they are drawn.

The bounds can be larger or smaller than the aperture. If it is smaller, it is stuck in the upper left corner of the aperture. If it is larger, it can scroll, but the aperture must always remain entirely inside the bounds. The default is for the bounds to be locked equal to the aperture. Derive from `JXScrollableWidget` (See Section 19.8) to change this, and then use `SetBounds()` to change the size of the bounds.

Global coordinates are defined by the upper left corner of the window. Local coordinates are defined by the upper left corner of the bounds. Enclosure coordinates are the local coordinates of the enclosing widget. Moving the frame (`Place()`, `Move()`) is done in enclosure coordinates. Changing the frame size (`SetSize()`, `AdjustSize()`) and incremental scrolling (`Scroll()`) are obviously independent of the coordinate system. Absolute scrolling (`ScrollTo()`, `ScrollToRect()`) uses local coordinates. When a widget's enclosure's bounds is resized, the widget will adjust itself according to its resizing options. If an edge is fixed, it always remains the same distance from the corresponding edge of its enclosure. If a dimension is elastic, both edges remain the same distance from the corresponding edges of its enclosure.

To redraw a widget at the next convenient updating point, call `Refresh()`. To redraw a widget immediately, call `Redraw()`. `ScrollTo()` scrolls the widget so that the given point (in local coordinates) is at the upper left corner of the aperture. `ScrollToRect()` scrolls the widget as little as possible to make the given rectangle (in local coordinates) visible in the aperture.

Never call `KillFocus()` because it removes the focus from a widget regardless of the value returned by `OKToUnfocus()`. This is dangerous because information entered by the user should always be verified. It is provided only because `JXDialogDirector` objects need to be able to respond correctly to the `Cancel` button.

19.4 Displays

Unlike typical desktop windowing systems, X has the ability to display a programs windows on a monitor other than the one attached to the computer running the program. In fact, X can display a program on any system running X that it has permissions to access. The displaying system does not even have to be running on the same hardware as the running system. In X, a `Display` is any of these places that can display the running program. Every window in X requires a `Display`.

Display objects often need to be passed as parameters to functions, but application code will rarely need to use Display objects directly. The exception is when creating cursors.

To create a Window on a particular display, call the `JXApplication` member function `SetCurrentDisplay()` and then create the `JXWindow` object. `JXDisplay-Menu` provides an automated way to let the user open new displays and select which display to put new windows on.

19.5 Drawing Widgets

Unlike all other windowing systems, the X Window System does not support printing. `JPainter` was created to fill this gap so that the same code can be used to draw to the screen, an offscreen image, and to a Postscript page. `JPainter` encapsulates all the functions that can be supported by both X and Postscript. Thus, code written to use only `JPainter` methods will work with `JXWindowPainter`, `JXImagePainter`, and `JXPrinter` objects.

`JXWindowPainter` objects are always created by JX and passed to the appropriate `Draw()` function. `JXPrinter` objects are usually created by documents. `JXImage-Painter` objects can be created at any time. If you need to move the drawing origin at any time, use `ShiftOrigin()` rather than `SetOrigin()`. It is also a good idea to always put the origin back where it was after you are finished. If you need to change the clipping rectangle at any time, use `SetClipRect()` rather than `SetDefaultClipRect()`. (The latter requires global coordinates, which your code shouldn't use.) It is also a good idea to always use `ResetClipRect()` after you are finished.

After creating a JXPrinter object, the correct procedure is to enter a loop as follows:

```
JBoolean done = kFalse;
JBoolean cancelled = kFalse;
printer->OpenDocument();
while (!done)
  {
  if (!printer->NewPage())
    {
    cancelled = kTrue;
    break;
    }
  // your code -- calls printer->LockHeader()
  DrawHeader(printer);
  // your code -- calls printer->LockFooter()
  DrawFooter(printer);
  // your code -- gets page number from
  printer->GetPageInfo()
  done = DrawPage(printer);
  }
if (!cancelled)
  {
  printer->CloseDocument();
  }
```

Headers and footers are best implemented inside the loop as shown. Simply draw the header and footer information and then use LockHeader() and LockFooter() to prevent any more drawing in these regions. When the main drawing code calls GetPageInfo(), it will see only the area that remains after removing the header and footer.

OpenDocument() creates a ProgressDisplay window that tells the user which page is being printed and lets the user cancel the process. If the user decides to cancel, NewPage() will automatically call CancelDocument() and then return kFalse. It is important to note that doing anything after printing has been cancelled is considered to be a fatal error. Thus, in the above code, the loop is terminated immediately after cancellation, and CloseDocument() is only called if the process was not cancelled.

19.5.1 Colors

`JColormap.h` defines the colors pre-allocated by JX in every colormap. To allocate other colors, use the widget's `JXColormap` object wich can be accessed through the `JXContainer` member function `GetColormap()`. The colormap is responsible for managing both static and dynamic colors in an efficient way. The widget that allocates a particular color is also responsible for deallocating it, in its destructor at the very latest.

Not all colormaps can allocate dynamic colors. If a widget needs dynamic colors and the default colormap provided by the display cannot provide them, you can create a new colormap and pass this to the window via its constructor. The colormap of a window cannot be changed once the window has been created because X does not support it.

`JXColormap` returns values of type `JColorIndex` instead of raw X pixel values. This is done because `JXDisplay` will automatically switch to a private colormap if the default colormap runs out of space. `JColorIndex` insures that nobody has to recalculate color indices when this happens. (For the efficiency fanatics, using a `JColorIndex` is just as efficient as an X pixel value because it only requires a single lookup in an array to convert a `JColorIndex` to an X pixel value.)

19.5.2 JPainter

This class abstracts the functions required to draw to the screen and to a printer. It supports drawing text and the standard shapes (with or without filling), a clipping rectangle, an adjustable drawing origin, and a pen location, line width, and pen color.

19.5.3 JXFontManager

This class abstracts the interface for accessing fonts. The following functions provide access to information about specific fonts, and about fonts available on the system.

GetFontNames	Return an alphabetical list of all the available font names.
GetFontSizes	Return min avail size, max avail size, and a sorted list of all the available font sizes for the given font name. If all font sizes are supported (e.g. TrueType), return a reasonable min and max, and an empty list of sizes. Return kFalse if there is no such font.
GetFontStyles	Return JFontStyle with available styles set to kTrue. Because of JPainter, underline, strike, and colors are always available.
GetFontID	Return a handle to the specified font. This routine is responsible for finding the best approximation if the specified font is not available.
GetFontName	Return the name of the font with the given id.

GetXFontNames	This provides raw output from XListFonts(), which is a list of all the available fonts.

It is important to remember that the font id will change if the size or style is changed, so you will need to call `GetFontID()` again after changing either of these. `UpdateFontID()` is a utility function to get the new font id for you.

The following functions provide the necessary size information to draw text in a widget.

GetLineHeight	Return the height of a line of text.
GetCharWidth	Return the width of the given character.
GetStringWidth	Return the width of the given string.

19.6 Drawing during a mouse drag

Normally, all drawing for a `JXWidget` should only be done inside `Draw()`. However, it is sometimes necessary to draw things while the user is dragging the mouse. (e.g. a selection rectangle) `JXDragPainter` was created for this purpose. `JXWidget` provides two ways to create a `JXDragPainter`: `CreateDragInsidePainter()` and `Create-DragOutsidePainter()`. The first function creates a `JXDragPainter` for dragging inside the widget's aperture. This is typically what you will use. The second function creates a `JXDragPainter` for dragging inside the widget's enclosure's aperture. This is useful if you need to draw outside the widget's aperture, e.g. if you want to drag the widget itself around.

The only correct way to create a `JXDragPainter` is via `JXWidget`. In addition, you must call `DeleteDragPainter()` when you are done with the object. This is required because `JXWidget` keeps track of the `JXDragPainter` object in order to keep it up to date when the widget scrolls. `JXWidget` only allows one active `JXDragPainter` at a time, so you will trigger an assert if you forget to call `DeleteDragPainter()`. One should never need more than one `JXDragPainter` object because there is only one mouse.

Thus, the simplest approach when using drag painters is to call the appropriate create routine in `HandleMouseDown()`, call `GetDragPainter()` in `HandleMouseDrag()`, and then call `DeleteDragPainter()` in `HandleMouseUp()`.

There are times when a drag painter is not required. If the circumstance is not terribly complicated, the widget's state can be changed and then it can be forced to redraw in the `HandleMouseDrag()` function simply by calling `Redraw()`.

19.7 Partitions

Dividing lines allow the user to decide how an area should be partitioned. JXHorizPartition provides a horizontal partitioning of a rectangle, and JXVertPartition provides a vertical partitioning. These Widgets are not derived from JXScrollableWidget because one only needs adjustable partitions when there is a fixed amount of space.

The default behavior when the entire partition is resized is as follows:

1. If all compartments are elastic, each one changes size by the same amount.

2. If a single compartment is designated as elastic, only this one changes size.

All compartments are elastic if elasticIndex is zero. In this case, the width of every compartment must be specified in the constructor. If elasticIndex is non-zero, then the widths of all but one compartment are specified and fixed, and the width of the elastic compartment is calculated automatically. To change this behavior, derive a new class and override ApertureChanged() to adjust the width of each compartment as follows:

```
void MyHorizPartition::ApertureChanged
  (
  const JRect& newApertureGlobal
  )
{
  // First, let the base class do what-
ever it wants to.
  JXHorizParti-
tion::ApertureChanged(newApertureGlobal);
  // Now call your code to ad-
just the widths of the compartments.
  AdjustCompartmentWidths(newApertureGlobal);
}
```

Since one can place one type of Partition inside the other, one can achieve almost any conceivable geometry. For custom arrangements, the correct way to handle dividing lines is described below. (It is important to remember that the method for adjusting the arrangement must be easy for the user to comprehend.)

1. Derive a class from JXWidgetSet to contain all the other Widgets.

2. In this derived class, implement Draw() to draw the dividing lines.

3. Implement mouse dragging to allow dragging of each dividing line. After a mouse drag, adjust the sizes and positions of the enclosed Widgets.

19.8 Scrolling

Plain widgets do not scroll because the bounds is locked equal to the aperture. To make a scrollable widget, derive the class from `JXScrollableWidget` instead of `JXWidget`. This automatically unlocks the bounds, so the derived class only has to call `SetBounds()`. It also automatically supports the `Page Up`, `Page Down`, `Home`, and `End` keys. If your derived class needs to handle key presses, be sure to call the inherited `HandleKeyPress()` function so that these keys can be processed correctly.

19.8.1 JXScrollbar

Scrollbars take integer values from zero to a client specified maximum. Sliders (see below) provide the general case.

19.8.2 JXScrollbarSet

This manages the geometry of a pair of scrollbars and an enclosure for the widgets that listen to the scrollbars. `GetScrollEnclosure()` returns this enclosure. The scrollable widgets that listen to the scrollbars should all be placed inside this enclosure. To scroll a single widget, pass the `JXScrollbarSet` to the constructor of the `JXScrollableWidget`. The scroll bars will then automatically stay in sync with the bounds of the `JXScrollableWidget`. To scroll several widgets simultaneously, you must write custom code to arrange them inside the `JXContainer` returned by `GetScrollEnclosure()` and then keep the scroll bars in sync with the display.

19.9 Handling events

All events are processed by virtual functions defined in JXContainer and JXWidget. All event handlers are protected unless stated otherwise.

19.9.1 Drawing

```
virtual void Draw(JXWindowPainter& p, const JRect& rect);
virtual void DrawBor-
der(JXWindowPainter& p, const JRect& frame);
virtual void DrawBackground(JXWindowPainter& p,
                            const JRect& frame);
```

`Draw()` and `DrawBorder()` must be overridden by every new derived class, unless they are derived from `JXScrollableWidget`, in which case they only need to override `Draw()`. `DrawBackground()` is implemented by `JXWidget`. Use `SetBackColor()` and `SetFocusColor()` to change the color drawn by `JXWidget`.

19.9.2 Mouse movement

```
virtual void HandleMouseEnter();
virtual void HandleMouse-
Here(const JPoint& pt, const
                             JXKeyModifiers& modi-

fiers);
virtual void HandleMouseLeave();
```

These functions are called when the cursor is inside the Widget's Frame and the mouse is not pressed.

19.9.3 Cursor adjustment

```
virtual void AdjustCursor(const JPoint& pt, const
                          JXKeyModifiers& modi-

fiers);
```

This function is called while the cursor is inside the Widget's Frame.

To change the default cursor for a Widget use SetCursor(). Custom cursors can be created with the functions in JXDisplay. For more control over the cursor (e.g. different cursors in different regions or at different times), override AdjustCursor(). To animate the cursor, use CreateCursorAnimator() to attach a JXCursorAnimator object to the Widget. Use RemoveCursorAnimator() to stop the animation.

19.9.4 Mouse clicks

```
virtual void HandleMouse-
Down( const JPoint& pt, const
                              JXMouseButton but-
ton,
                              const JSize click-
Count,
                              const JXButton-
States& buttonStates,
                              const JXKeyModi-
fiers& modifiers);
virtual void HandleMouseDrag( const JPoint& pt,
                              const JXButton-
States& buttonStates,
                              const JXKeyModi-
fiers& modifiers);
virtual void HandleMouseUp( const JPoint& pt,
                            const JXMouseBut-
ton button,
                            const JXButton-
States& buttonStates,
                            const JXKeyModi-
fiers& modifiers);
```

These functions are called to report mouse clicks. clickCount in HandleMouseDown() specifies the number of consecutive, rapid-fire clicks performed with the same button in the same part of the Widget.

It is safe to delete a Widget in HandleMouseDown() and in HandleMouseUp(), but not in HandleMouseDrag(). The code is written this way because the application should wait until the user makes a decision by releasing the mouse button.

19.9.5 The HitSamePart() function

```
virtual JBoolean HitSamePart(const JPoint& pt1,
                             const JPoint& pt2) const;
```

To control when a set of rapid-fire clicks should increment clickCount in HandleMouse-Down(), override HitSamePart() and return kTrue if the two given clicks should be considered as a double click.

19.9.6 Key press events

```
    virtual void HandleKeyPress(const int key, const
                                JXKeyModifiers& modi-

    fiers);
```

This function is called whenever a key is pressed. If the key is not needed, is must be passed to the base class.

```
    void WantInput(const JBoolean wantInput,
                   const JBoolean wantTab = kFalse,
                   const JBoolean wantModi-
    fiedTab = kFalse);
```

WantInput() must be called in order to receive keypress events. Since JX uses Tab to switch focus between Widgets, there is a special flag for requesting this key. It is suggested that derived classes not request it unless they really need it. This helps to make the user interface more uniform.

19.9.7 Shortcuts

```
    virtual void HandleShortcut(const int key, const
                                JXKeyModifiers& modi-

    fiers);
```

Each Window maintains a list of shortcut keys. These keys are used to activate button, checkboxes, menu items, etc. All shortcut keys are sent to HandleShortcut() instead of HandleKeyPress(). If the key is not recognized, is must be passed to the base class.

Shortcuts are normally specified by a single string with the following format "<modifier><character><modifier><character>...". The modifier is optional and is either an up caret (ˆ) to indicate Control or a hash (#) to indicate Meta. Modifiers that cannot be represented this way can be installed by calling

```
JXWindow::InstallShortcut(JXWidget* wid-
get, const int key,
                          const JXKeyModi-
fiers& modifiers)
```

where key can be any X keysym.

Shortcuts are sent directly to the Widgets that requested them and not to the Widget that currently has focus. It is therefore a good idea to require a modifier for all shortcuts. For consistency with JX, it is suggested that Meta always be used.

Each Widget is responsible for somehow showing the user that it has a shortcut. As an example, each button searches its label for the first character in its shortcut list and underlines it.

19.9.8 Client messages

```
virtual JBoolean HandleClientMes-
sage(const XClientMessageEvent&
                                     clientMes-
sage);
```

This function is called every time an X client message is received. If the message is not recognized, this function should always pass it to the base class. If the message is recognized, it should handle it and then return kJTrue.

Special routines are provided to deal with the X Selection mechanism. Refer to the section on the X Selection for details.

19.10 The X Selection (clipboard)

Support for the X Selection mechanism is provided by `JXSelectionManager` and derived classes of `JXSelectionManager::Data`. To support "copy", an object simply creates an appropriate derived class of `JXSelectionManager::Data` and passes it to the selection manager via `SetData`.

19.11 Drag-And-Drop

```
virtual void  HandleDNDEnter();
virtual void  HandleDNDHere(const JPoint& pt,
                            const JXWid-
get* source);
virtual void  HandleDNDLeave();
virtual void  HandleDNDDrop(const JPoint& pt,
                            const JAr-
ray<Atom>& typeList,
                            const Atom action,
                            const Time time,
                            const JXWid-
get* source);
```

These functions are called to report a drag and drop event. Drag-And-Drop (DND) can be supported by overriding the four HandleDND*() functions. They operate just like the corresponding HandleMouse*() functions. HandleDNDLeave() is called if the mouse leaves. HandleDNDDrop() is called if the mouse is released. The data that was dropped is accessed via the selection manager. (Get the name of the selection from JXDNDManager::GetDNDSelectionName().)

Since each class will only accept certain data types, each derived class should override WillAcceptDrop() and return kTrue if they will accept at least one of the given data types. If the function returns kFalse, the object will not receive any DND messages.

19.11.1 Drop targets

```
virtual JBoolean WillAcceptDrop(const JAr-
ray<Atom>& typeList,
                                Atom* action,
                                const Time time,
                                const JXWid-
get* source);
```

Derived classes that accept drops should override this function and return `kTrue` if they will accept the current drop. If they return `kFalse`, they will not receive any DND messages. The argument `source` is not equal to `NULL` if the drag is between widgets in the same program. This provides a way for compound documents to identify drags between their various parts. This function may modify the `action` parameter.

When the user is dragging over a widget, the three DND functions discussed above will be called, `HandleDNDEnter()`, `HandleDNDHere()`, and `HandleDNDLeave`(). If the user drops on the widget, `HandleDNDDrop()` is called. Typically, `HandleDNDHere()` is used to give the user feedback about what will happen if they drop at the current location. For example, as the user drags over an icon, that icon may become selected, indicating that it is a valid drop target.

19.11.2 Drop sources

```
JBoolean  BeginDND(const JPoint& pt,
                   const JXButtonStates& button-
States,
                   const JXKeyModifiers& modi-
fiers);
```

Call this to begin the DND process. If this returns kTrue, you will not get any more HandleMouseDrag() messages, nor will you get a HandleMouseUp() message.

```
virtual void DNDInit(const JPoint& pt,
                     const JXButtonStates& button-
States,
                     const JXKeyModifiers& modi-
fiers);
```

This is called when DND is initiated. Note that DND cannot be performed until at least one selection target has been added. The targets can be changed in this function, but at least one must have been added at some earlier time (usually in the constructor).

```
virtual void DNDFinish(const JXContainer* target);
```

This is called when DND is terminated by a drop. If it hasn't already, the widget must perform whatever preprocessing is necessary and then copy the dropped data into a separate buffer so that the target can request it whenever it feels like it.

If the copying is always very fast (e.g. if only one item in a list can ever be selected), it can be done in DNDInit(). Otherwise, it should be done here because one should never make the user wait when the drag begins.

Note that the target may request the data while the mouse is being dragged, in which case the source must do the above work and then remember not to do anything extra when DNDFinish() is called. If the drop target is not within the same application, target is NULL. If the target is the same as the source, one can ignore this message because one shouldn't even bother going via JXSelectionManager in this special case.

```
virtual Atom GetDNDAction(const JXContainer* tar-
get,
                          const JXButton-
States& buttonStates,
                          const JXKeyModi-
fiers& modifiers);
```

This is called repeatedly during the drag so the drop action can be changed based on the current target, buttons, and modifier keys. If the drop target is not within the same application, target is NULL. The default implementation is to return the default action: copy.

```
virtual void GetDNDAskActions(const JXButton-
States& buttonStates,
                              const JXKeyModi-
fiers& modifiers,
                              JArray<Atom>* askAc-
tionList,
                              JPtrArray<JString>*
                                askDescription-
List);
```

This is called when the value returned by `GetDNDAction()` changes to `XdndActionAsk`. If `GetDNDAction()` repeatedly returns `XdndActionAsk`, this function is not called again because it is assumed that the actions are the same within a single DND session. This function must place at least 2 elements in `askActionList` and

`askDescriptionList`. The first element should be the default action. The default implementation is to do nothing, so classes that never return `XdndActionAsk` don't have to implement it, and classes that forget to implement it will force an `assert()` in `JXDNDManager` by not returning at least 2 elements.

```
virtual void HandleDNDResponse(const JXCon-
tainer* target,
                               const JBoolean dropAc-
cepted,
                               const Atom action);
```

This is called when the target indicates whether or not it will accept the drop. If `dropAccepted` is `kFalse`, the action is undefined. If the drop target is not within the same application, target is `NULL`. The default implementation is to display the appropriate default cursor provided by `JXDNDManager`. Note that part of the cursor should always be the standard arrow. This way, the user will feel like they really are dragging something.

```
virtual JBoolean ConvertSelection(const Atom name,
                                  const Atom re-
questType,
                                  Atom* return-
Type,
                                  un-
signed char** data,
                                  JSize* dataL-
ength,
                                  JSize* bitsPerBlock);
```

When the user has finished the drop, this is called so that the target can get the data that it needs. Refer to Section 19.10 for more information on using the X selection mechanism.

19.12 Periodic background tasks

Since UNIX is a multi-tasking operating system, one never has to worry about giving time to other programs. However, sometimes one would like to give time to several different tasks within the same program. `JXIdleTask` provides a solution. Simply create a derived class and override `Perform()`. Then use the `JXApplication` member function `InstallIdleTask()` to start the task. Note that this is mainly useful for lightweight tasks such as updating a widget's appearance. Heavy processing should either force the user to wait by displaying a progress display or use `JExecute()` to create a separate process.

19.13 Urgent updating tasks

In some rare cases, one needs to change a Widget in a way that is illegal within the given execution context. As an example, it is illegal to recursively change the size of a widget. In such cases, one should create a derived class of `JXUrgentTask`, override `Perform()`, and use `the JXApplication` member function `InstallUrgentTask()`. `Per-`

`form()` will be called after your code has returned control to the JX event loop. Urgent tasks are automatically deleted after `Perform()` is called.

19.14 Images

`JXImage` provides a clean interface to both `Pixmaps` and `XImages`. There are several different constructors, including ones for `XBM` and `XPM` data. Drawing to the Image is done via a `JXImagePainter` object obtained by calling `CreatePainter()`. There can be an unlimited number of active ImagePainters for each Image.

Each Image can also have a mask specified by a `JXImageMask` object. The constructor

```
JXImageMask(const JXImage& image, const JCol-
orIndex color)
```

creates a mask to remove all pixels of the specified color from the given image. Since `JXImageMask` derives from `JXImage`, one can draw to an ImageMask using an ImagePainter. `JXImageMask::kPixelOff (kJXTransparentColor)` is provided so one can erase pixels from the mask. All other colors add pixels to the mask. This way, the code that draws an image will also draw the mask for that image when passed an ImagePainter associated with the mask.

Conversion between `Pixmaps` and `XImages` is handled internally, so clients never need to worry about it except for efficiency concerns. Drawing to the Image requires a `Pixmap`, while `GetColor()` and `SetColor()` require an `XImage`. To avoid excessive swapping, one should try to group all calls to `GetColor()` and `SetColor()` together.

`SetDefaultState()` allows one to set whether the image data should normally be stored on the server side or the client side. Images that are being drawn in visible windows should be stored on the server side to avoid having to retransmit the data every time the window is redraw.

X also puts another restriction on how Images can be used. Each Image is associated with a particular Display and a particular Colormap. An Image can only be drawn on its own Display and will be garbled if drawn to a Window using a different Colormap.

`JXImageWidget` provides a simple way to display an Image.

19.15 Menus

The JX menu system is very simple and also very powerful. `JXTextMenu` provides all the functionality that one expects from a menu, and it can be placed anywhere in a window or can be used as a sub-menu. `JXMenuBar` manages the geometry of a horizontal list of menus. These two classes are sufficient for most purposes. The power comes from the fact that one can derive classes from `JXMenu` to create menus to do anything. Custom menus are simply derived classes of `JXMenuTable` and work just like any other table.

Menus, like all other widgets, can be placed anywhere in a window. In addition, menus can be attached as sub-menus for items on other menus. This means that once the menu is created, client code cannot tell the difference between a menu in a menu bar, a menu that is acting like a popup somewhere else in the window, and a menu that is a sub-menu of another menu. To make a popup menu of choices display the currently selected choice, call `SetToPopupChoice()`. Menus can be attached as sub-menus either via the second `JXMenu` constructor or by calling `AttachSubmenu()`.

Menus broadcast two messages. Just before the menu opens, it broadcasts `JXMenu::NeedsUpdate`. Clients should respond to this message by activating or deactivating menu items, turning checkboxes on or off, selecting the appropriate items in radio groups, adjusting the text of items, etc. The function `SetUpdateAction()` can be used to set the default state of the menu items. All checkboxes and radio buttons are intially off because it is the responsibility of the client to keep track of the settings. `CheckItem()` turns on both checkboxes and radio buttons.

When an item is selected, the menu broadcasts `JXMenu::ItemSelected` which contains the index of the item. This message will only be broadcast if the user actually chooses a particular item.

`JXTextMenu` supports styled text, an optional Image for each item, key shortcuts for use when the menu is open, and key shortcuts for use when the menu is not open (non-menu shortcuts, NM for short). `SetItemShortcuts()` sets the key shortcuts for an item when the menu is open. If the first character in this list is found in the text of the menu item it is underlined when the menu is open. `SetItemNMShortcut()` sets the single key shortcut for an item when the menu is not open. The string passed to this function is parsed as follows:

Prepended "Ctrl-", "Meta-", "Ctrl-Shift-", and "Meta-Shift-" are converted into modifier flags. If the rest of the string is a single character, then this is the key. The strings "dash", "minus", "plus", "period", and "comma" are translated to characters because the actual characters can confuse the user. (e.g. Use "Ctrl-dash" instead of "Ctrl–") The strings "F1" through "F35" are translated to the corresponding function keys that X provides. The resulting character and modifiers is registered as a shortcut and generates a JXMenu::ItemSelected message when pressed. It is legal to pass in a string that cannot be parsed. (e.g. "Ctrl-middle-click") In this case, the string will simply be displayed without registering a shortcut.

SetMenuItems() is useful for building a menu from a single string. The format is "<item text> <options> | <item text> <options> | ...". The options are as follows:

- %d - item is disabled
- %l - item is followed by a separator line
- %b - item is a checkbox
- %r -item is a radio button
- %h <chars> - specify shortcuts for item (a list of single characters)
- %k <chars> - specify non-menu shortcut
- %i <chars> - specify the item’s id

In addition to `JXTextMenu`, JX also includes `JXImageMenu` which provides a way to display a grid of Images in a menu. You specify the width of the menu and the height is determined by the total number of items.

Note that, since Images are constructed for a particular Colormap, and Menus use the Colormaps of the Windows that they pop up from, Images in a Menu must use the same Colormap as the Menu's Window.

19.15.1 Custom menus

This section describes how to create custom menus.

1. Create a derived class of `JXMenuData` to store the information that the menu will display. `InsertItem()`, `DeleteItem()`, and `DeleteAll()` must call the base class version at some point so that the data stored in `JXMenuData` will stay in sync with the data in your derived class.

2. Create a derived class of `JXMenuTable` to display the information. There is no fixed mapping between menu items and table cells. You can implement `Cell-ToItemIndex()` to implement any mapping you want. (Be sure that the user can understand the it.) `MenuHilightItem()` and `MenuUnhilightItem()` must be implemented so `JXMenuTable` can tell you which item (if any) is currently under the mouse cursor. You must use this information to hilight the item in some way. `GetSubmenuPoints()` must be implemented to return two points (in local coordinates) where a sub-menu could be placed. rightPt should be somewhere to the right of the item's cell, while leftPt should be somewhere to the left of the item's cell. rightPt gets preference. leftPt is used if rightPt would cause part of the sub-menu to be off the screen.

3. Create a derived class of `JXMenuDirector` that overrides `CreateMenuTable()` to create the correct MenuTable object.

4. Create a derived class of `JXMenu` to provide an interface to your derived class of `JXMenuData` and to override `CreateWindow()` to create the correct MenuDirector object

19.16 Text Editor

When of the most powerful feature of JX is its text editor classes, which are all derived from `JTextEditor`. All classes derived from `JTextEditor` support fully styled text, drag and drop, copy/paste of styled text, unlimited depth undo, auto-indent, and automatic right-hand margin adjustment. For many cases, one of the existing derived classes could be used in an application as is. If a custom editor is needed, simply sub-class `JXTEBase`. Typically, such a derived class would merely turn on the desired features on, all of which are available in `JTextEditor`.

19.16.1 JXTEBase

This is the base class for most JX test editor classes. Among other things it implements an edit menu that handles all of the clipboard operations on the text.

19.16.2 JXInputField

This is the base class for all dialog window input fields. Derived classes include `JXString-Input`, `JXFloatInput`, and `JXIntegerInput`.

19.16.3 JXTextEditor

This class has menus for changing the font, font size, and font style.

19.17 Tables

The purpose of a Table class is to display a two dimensional arrangement of items. Each item is displayed in a separate cell of the table. Items are referenced by giving the row index and column index. Each row can have a different height, and each column can have a different width.

JTable implements all the system independent routines for dealing with tables. `JXTable` implements all the system specific routines required by `JTable`, except for `TableDraw-Cell()`. All table classes should therefore be derived from `JXTable` instead of `JTable`.

19.17.1 JTableData

The Table suite is designed according to the Model-View-Controller paradigm. A Table-Data object provides the Model by storing the data to be displayed. A Table object provides the View and updates itself automatically based on messages received from the TableData object. As discussed in the section on cross-platform operation, system dependent derived classes are written to handle mouse clicks, key presses, etc. These derived classes implement the Controllers.

Since drawing to the screen is data dependent, you will always have to create your own class derived from `JTable` to override `TableDrawCell()` and draw the contents of each cell.

`JTable` and `JTableData` provide the foundation for JX's Table suite. When a pair of `JTable` and `JTableData` objects are used in a program, the correct procedure for modifying the data is always to call methods in `JTableData`. `JTable` will then adjust itself automatically because it is designed to operate transparently with any object that follows the `JTableData` message protocol. These messages are described below. (In order to keep `JTableData`'s row and column counts up to date, when one broadcasts the message, one must also call the indicated function.)

Message	Action
RowChanged	The elements in a row have changed
RowInserted	A new row was inserted (also call RowAdded())
RowDuplicated	A row was duplicated (also call RowAdded())
RowRemoved	A row was removed (also call RowDeleted())
AllRowsRemoved	All the rows were removed (also call SetRowCount(0))
RowMoved	A row was moved to a new location
ColChanged	The elements in a column have changed

ColInserted	A new column was inserted (also call ColAdded())
ColDuplicated	A column was duplicated (also call ColAdded())
ColRemoved	A column was removed (also call ColDeleted())
AllColsRemoved	All the columns were removed (also call SetColCount(0))
ColMoved	A column was moved to a new location
RectChanged	The information in the specified rectangle of cells was changed

`JTable` has its own set of messages that it broadcasts when it changes. These can be used to keep several tables synchronized. The most common examples are row and column headers displayed to the right and at the top of the main table.

The most common uses of `JTableData` are provided as part of JX.

19.17.2 JValueTableData and JObjTableData

The `JValueTableData` template is designed to store a dense matrix of values or structs. Just like `JArray`, this class cannot handle anything that requires a copy constructor. For true classes, use the `JObjTableData` template instead. This is designed to store a dense matrix of pointers to objects. (Unlike `JPtrArray`, this class owns all the objects that it stores. i.e. `GetRow()` and `GetCol()` return copies of the objects which must be deleted by the caller, and `SetRow()` and `SetCol()` store copies of the objects that are passed in, so the originals must be deleted by the caller.) In order to store sparse matrices or ragged-edge rows or columns, you must create your own class derived from `JTableData` and follow the above protocol.

19.17.3 JAuxTableData

The `JAuxTableData` template is designed to efficiently store a redundant matrix of values by column. This is often useful for storing style information for a table that displays text in each cell. Most cells will have the default style, with a few cells displaying bold or italicized text. `JAuxTableData<JFontStyle>` stores such information in a memory efficient way. Every `JAuxTableData` object automatically links itself to the Table object that is passed to the constructor, thereby insuring that it automatically stays in sync with the Table and that the Table is redrawn whenever the data stored by `JAuxTableData` changes.

19.17.4 JXEditTable

`JTable` also provides routines for editing the data displayed in each cell. Derived classes decide when to call `BeginEditing()`, `EndEditing()`, and `CancelEditing()`, and then override the editing field manipulation functions described below. Normally, an edit field is a text input box. However, since `JTable` does not understand anything about the edit field object, it could just as easily be a modeless dialog window instead. One should call `EndEditing()` before saving, printing, etc.

`JXEditTable` implements all the system specific editing routines required by `JTable`, so derived classes only need to override the following routines:

CreateXInputField	Create or activate an input field to edit the specified cell and return the object as a `JXInputField`.

ExtractInputData	Check the data in the active input field, and save it if it is valid. Return `kTrue` if it is valid.
DeleteXInputField	Delete or deactivate the active input field. Called when editing is cancelled or when `ExtractInputData()` returns kTrue.

19.17.5 JXStringList and JXStringListTable

`JXStringList` displays a column of strings stored in a `JPtrArray<JString>`. `JXStringTable` works with `JStringTableData` to display a table of strings.

19.17.6 JXFloatTable

`JXFloatTable` works with `JFloatTableData` to display a table of numbers. String and Float tables also implement the routines required by `JXEditTable`, so derived classes only have to call `BeginEditing()` to allow editing of the data displayed in the table.

19.17.7 Input Restrictions

To place simple restrictions on the input (e.g. length of string or range of values), override `CreateXInputField()` to call the inherited function and then install the restrictions. For restrictions beyond those provided by classes derived from `JXInputField`, override the `JTable` member function `ExtractInputData()` and display an error message and return `kFalse` if the data is not acceptable, or use a custom derived class of `JXInputField`.

19.17.8 JXRowHeaderWidget and JXColHeaderWidget

`JXRowHeaderWidget` and `JXColHeaderWidget` are provided to simplify the process of displaying row and column headers for a table. The sample code in the test suite for JX shows how to arrange the Widgets properly inside the scrolling area. To draw the row and column header labels differently, create derived classes and override `TableDrawCell()`. `JXRowHeaderWidget` and `JXColHeaderWidget` also allow the user to change the row heights and column widths, respectively, by dragging the dividing lines between cells. This behavior is off by default, and is usually only useful for column widths since row heights are typically determined by the current font size. To turn on resizing, call `TurnOnRowResizing()` or `TurnOnColResizing()`. `TurnOffRowResizing()` and `TurnOffColResizing()` turn the behavior off again.

19.17.9 Printing

To draw page headers and footers while printing, override the following routines:

GetPrintHeaderHeight	Return the height required for the page header.
DrawPrintHeader	Draw the page header. `JTable` will lock the header afterwards.
GetPrintFooterHeight	Return the height required for the page footer.
DrawPrintFooter	Draw the page footer. `JTable` will lock the footer afterwards.

To include row headers and column headers while printing, call `SetRowHeader()` and `SetColHeader()`. JTable will then position and draw them automatically. These functions are all called by the `JTable` member function `Print()`.

Chapter 20

JX Class Documentation

20.1 Data types

All intrinsic data types are shielded by typedefs to make them system independent.

JBoolean	boolean (kTrue, kFalse)
JCharacter	characters
JFloat	floating point
JIndex	1-based indexing system
JInteger	signed integers
JSignedOffset	offsets that can be positive or negative

JSize	size of something (nonnegative)
JUnsignedOffset	offsets that can only be positive or zero

`JBoolean` is an enum with values kTrue and kFalse. Since comparisons always return int's which cannot be converted to enum's, `JI2B()` is provided to perform the conversion safely. ANSI's bool type is incompatible because it writes different data to a stream and is also unsafe when reading from a stream.

`JIndex` represents a 1-based indexing system, and all functions that take arguments of this type conform to this convention.

20.2 Other conventions

Passing by value, reference, and pointer is used to distinguish what will be done with the data. Intrinsic data types are passed either as const values or as pointers. Objects are passed either as const references or as pointers. When something must be passed as a pointer, it means that it will be modified. Passing by non-const reference is avoided.

The issue of who owns a particular object after a "set" method has been called is also settled by how the object is passed to the "set" method. `x.SetElement1(const T& y)` means that `SetElement1()` makes a copy of y. If y was allocated on the heap, it must be deleted by the caller. `x.SetElement2(T* y)` means that x now owns y and will delete it when it no longer needs it. In this case, the caller must not delete y. It is also an error in this case to allocate y on the stack, because then y will be deleted twice: when the calling function returns and when x is finished with it. Methods like `SetElement1()` are preferred. Methods like `SetElement2()` are used only when copying is too expensive to be reasonable.

When a value is returned by a "get" method, it is sometimes a const reference. This saves time if all one needs to do is look at the value. One should not assume that such references will remain valid, however, so one should only use such references locally inside a function. If one needs to store the value for a longer period of time, one should store a copy of the object rather than the reference.

20.3 JBroadcaster

One major advantage provided by object oriented programming is that data can be decentralized and encapsulated inside objects. This presents a serious problem for large programs, however, because data stored in different objects has to be kept in sync. Writing custom code to handle each such dependency results in tight coupling which has been shown to reduce both reusability and maintainability.

`JBroadcaster` provides a way to implement loose coupling by defining a protocol that objects can use to send messages to each other. With this system, senders only have to provide a useful set of messages without knowing anything about the receivers. To use this system, the class that sends the message and the class that receives it must both be derived from JBroadcaster. To send a message, call `Broadcast()`. To receive a message, an object must first call `ListenTo()` to begin listening for messages and then override `Receive()` to receive and process each message. `StopListening()` can be used to stop listening for messages.

All messages are classes and must be derived from `JBroadcaster::Message`. It is up to each class to define a useful set of messages. Each message should contain whatever information is required to process it. Messages can also provide methods for performing the default response. Refer to the messages defined by `JOrderedSet` for examples (e.g. `JOrderedSetT::ElementInserted::AdjustIndex()`).

It must be mentioned that just because `JBroadcaster` is easy to use does not mean that messages are easy to use. Objects that communicate via messages must be analyzed using the techniques developed for distributed systems, and any computer scientist will gladly explain how careful one must be in order to get such a system to work flawlessly. (In the case of JBroadcaster, messages are received immediately after they are sent, but the order in which listeners are notified is undefined.)

One particularly dangerous case is a message that is broadcast from within a destructor. The best advice is to avoid this at any cost, because objects can depend on themselves, or they can depend on objects that they own and that they therefore delete in their destructor. In both cases, calling `Receive()` while the object is being destructed is usually fatal.

Another deadly possibility is that the message might cause one of the listeners to delete the sender. (This will not cause problems if `Broadcast()` is the last action before returning because then this will not be referenced after the message has been sent.) This often occurs with a "Close window" menu item. The object that receives the message deletes the window and everything in it, including the menu that broadcast the message.

20.4 JString & JRegex

`JString` was written before C++ got a standard string class, and provides essentially the same functionality, except for operator[] since `JString` uses 1-based indexing. It is easy to construct one from the other because each can be constructed from the other's char*. It is important to remember that char* implies a null-terminated array of characters.

`JRegex` provides regular expressions, and the ability to search and replace regular expressions in a JString.

20.5 JCollection derived objects

`JCollection` is the base class for all classes that are primarily collections of other objects. It provides `IsEmpty()` and `GetElementCount()`.

20.5.1 JOrderedSet<T>

This abstract template class defines the interface for all ordered sets (arrays, linked lists, run arrays, etc) and a set of messages for notifying other Broadcasters of changes. It also defines a set of efficient sorting functions.

20.5.2 JOrderedSetIterator<T>

This template class provides a robust way to iterate over the elements in an OrderedSet. `Next()` returns kTrue if there is a next element, and similarly for `Prev()`. Using an Iterator in a while loop is more robust than calling `GetElement()` in a for loop because

Iterators are automatically notified of changes to the OrderedSet. Specifically, they adjust themselves as appropriate when elements are inserted or removed so they will never accidentally attempt to access non-existent elements. In addition, an iterator can also be faster that using `GetElement()` in a for loop. The time required by `GetElement()` can be proportional to the element's index. An iterator can bypass this if it is tightly coupled with the class that stores the data. To get the best iterator for a given OrderedSet, use `NewIterator()`. Note that for Arrays, `GetElement()` requires constant time, so there is no need to use an iterator in this case unless the code inside the loop will have unpredictable effects on the elements in the array.

20.5.3 JArray<T>

This template class implements a bounds checked, 1-based array of elements. It does not understand copy constructors, however, so it should only be used for intrinsic data types and structs containing only intrinsic data types. Objects should be stored in `JPtrArray`'s. In general, storing an array of objects (as opposed to an array of pointers) is dangerous because operator= is not virtual. This is discussed at length in *More Effective C++*.

20.5.4 JPtrArray<T>

This template class implements a bounds checked, 1-based array of pointers. It is designed to store lists of objects, and provides `DeleteElement()` and `DeleteAll()` for automatically disposing of the objects when they are removed from the list. (For objects allocated with operator `new[]`, use `DeleteElementAsArray()` and `DeleteAl-`

`lAsArray()` instead so that operator `delete[]` is called.) Use `RemoveElement()` to remove an object from the list without deleting it. (Somebody else must then delete the object elsewhere.)

20.5.5 JLinkedList<T>

This template class implements a bounds checked, 1-based, doubly linked list of elements. As with `JArray`, `JLinkedList` does not understand copy constructors, however, so it should only be used for intrinsic data types and structs of intrinsic data types. Objects should be stored in `JPtrArray`'s. It is doubly linked so that iterators can efficiently traverse it in either direction and the worst time for `GetElement()` is (# of elements)/2.

20.5.6 JRunArray<T>

This template class implements a bounds checked, 1-based, compressed list of elements. Compression is acheived by storing "runs" of equal elements as a single pair {# of elements, value}. As with `JArray`, `JRunArray` does not understand copy constructors, however, so it should only be used for intrinsic data types and structs of intrinsic data types. Objects should be stored in `JPtrArray`'s.

20.5.7 JQueue<T>

This template class implements a first-in-first-out queue of elements. As with `JArray`, `JQueue` does not understand copy constructors, however, so it should only be used for intrinsic data types and structs of intrinsic data types. Objects should be stored in `JPtrQueue`'s.

20.5.8 JPtrQueue<T>

This template class implements a first-in-first-out queue of pointers to objects. It provides `FlushDelete()` and `DiscardDelete()`.

20.5.9 JStack<T>

This template class implements a first-in-last-out stack of elements. As with `JArray`, `JStack` does not understand copy constructors, however, so it should only be used for intrinsic data types and structs of intrinsic data types. Objects should be stored in `JPtrStack`'s.

20.5.10 JPtrStack<T>

This template class implements a first-in-last-out stack of pointers to objects. It provides `ClearDelete()` and `UnwindDelete()`.

20.6 Instantiating templates

Files ending in .tmpls are provided to simplify the process of instantiating templates. Each such file contains a header explaining how to use it. As an example, to instantiate `JArray<JBoolean>`, the file `Templates-JBoolean.cc` contains the following code:

```
    #include <jTypes.h>
    #define JTemplateType JBoolean
    #include <JArray.tmpls>
    #undef JTemplateType
```

This code can be copied and used to instantiate any other template.

20.7 JContainer

Most classes that qualify as Collections will store their data privately in Arrays or PtrArrays and then implement wrappers and additional methods to access and manipulate the data. In order to keep `GetElementCount()` for the class in sync with `GetElement-Count()` for the internal data, one should derive the class from `JContainer` and then call `InstallOrderedSet()` in the constructor. `JContainer` will then insure that the element counts stay in sync automatically.

20.7.1 JFileArray

This class implements an array stored in a file. All data is stored as ascii text and is transferred via strstream and `JStaticBuffer` objects. Each element of the array can have an arbitrary size. A FileArray can be embedded within another FileArray by storing all the embedded file's data inside one element of the enclosing file. The single base file object is initialized with the file specifications. An embedded file object is initialized with the file object that contains it and the ID of the element that contains it. Embedded files must be opened after the base file and must be closed before the base file.

Each element in a FileArray can be referenced either by its index or its ID. The index gives the logical ordering of the elements in the file. (The physical ordering may be very different since `JFileArray` is optimized to work with a slow disk.) The ID of each element is guaranteed to be unique and to remain the same regardless of how the item's index changes. Thus, an ID provides a way to keep track of an element, regardless of how the elements are rearranged. `JFAIndex` and `JFAID` were created in order to allow function overloading so that an element can be accessed via either its index or its ID.

Storing and retrieving data is quite simple. The following code creates a new element at index 1:

```
ostrstream dataStream;
dataStream << 5 << ' ' << 3;
theFileArray.PrependElement(dataStream);
```

The following code retrieves the data from element 1:

```
JFAIndex index = 1;
JStaticBuffer data;
theFileArray.GetElement(index, &data);
jistrstream(dataStream, data.GetData(), strlen(data));
dataStream >> x >> y;
```

Notice the utility function `jistrstream()`. This function exists because many of the implementations of `istrstream` and `strstream` are broken. This function will use either `istrstream` or `strstream` to return the `dataStream` variable depending on which function is broken on a particular system.

The advantage of sending the data to the FileArray via an ostrstream is that it provides a simple way to package many pieces of information into one element of the file. The only reasonable way to retrieve the data is to use an istrstream. By allocating the ostrstream, the `JStaticBuffer`, and the istrstream on the stack, one never has to remember to delete anything. This provides protection against memory leaks. (`JStaticBuffer` is essentially a "smart pointer," as discussed in *More Effective C++*.)

One important restriction when working with FileArrays is that no two FileArray objects can open the same file. This is because FileArray objects are optimized for use with a slow disk and therefore do not keep the data on the disk up to date. The file is only complete after it is closed.

20.7.2 JPrefsManager

The `JPrefsManager` class buffers the data in a `JPrefsFile` and provides a base class for application-specific preferences management. The functions to access the data are protected because they should be hidden behind a clean interface in the derived class.

`UpgradeData()` is called after the file has been read. Since `UpgradeData()` must work on an empty file, this insures that the program has a valid set of preferences even if the file could not be read.

Some programs enforce that only a single copy is running for each user, so an open prefs file means that the program crashed while editing the preferences. If this is the case, pass `kTrue` for `eraseFileIfOpen` to the constructor. If this is not the case, `DeleteFile()` can be implemented to `assert()` because it should never be called.

20.8 JXUserNotification

This implement `JUserNotification` for JX. It displays a dialog that blocks the entire application until it is dismissed by the user. There are 3 dialogs available.

20.8.1 DisplayMessage

Calling `JUserNotification`'s `DisplayMessage(const JCharacter* message)` function, displays a `JXMessageDialog` dialog that contains the text contained in the `message` argument and a dismiss button.

20.8.2 ReportError

Calling `JUserNotification`'s `ReportError(const JCharacter* message)` function displays a `JXErrorDialog` dialog window that contains the text contained in the `message` argument and a `Dismiss` button.

20.8.3 AskUserYes/AskUserNo

Calling JUserNotification's `AskUserYes(const JCharacter* message)` and `AskUserNo(const JCharacter* message)` functions displays a `JXWarningDialog` dialog window that contains the text contained in the `message` argument and `Yes` and `No` buttons. This function returns a `JBoolean` indicating whether the user pressed the `Yes` or `No` buttons.

20.9 Progress Displays

20.9.1 JXProgressDisplay

This implement the functionality of `JProgressDisplay` for JX, but not the display. The client must create the Widgets to display the progress and then call `SetItems()`. This is useful when one wants to display the progress of an operation without creating a new window.

20.9.2 JXStandAlonePG

This implement `JProgressDisplay` for JX. If the process is to run in the foreground (e.g. a tight loop that ignores the event loop), it displays a dialog window that blocks the entire application until the process is finished. If the process is to run in the background (e.g. via an IdleTask), it displays the same dialog window, but does not block.

20.9.3 JLatentPG

This class encapsulates a progress display that only pops up after N seconds (default 3) and only redraws itself every M increments (default 1).

This allows one to write a loop that doesn't waste time creating a progress display unless it takes a while and doesn't waste time redrawing every time through the loop.

The scale factor is especially useful when one has a loop whose contents execute very quickly on each pass, so that redrawing the screen after every pass adds a huge overhead.

20.10 JXChooseSaveFile

This suite of classes implements `JChooseSaveFile` for JX. It displays a dialog window that blocks the entire application until it is dismissed by the user.

Each dialog window is encapsulated in a separate class: `JXChooseFileDialog`, `JXSaveFileDialog`, and `JXChoosePathDialog`. The unique feature is that these classes are designed to allow further derived classes. This allows one to add functionality to each dialog window. Derived classes must create a `BuildWindow()` function to create the Window. (The constructor must not call this.) Inside `BuildWindow()`, they must call `<inherited>::SetObjects()` and then set up the extra items that the derived class adds. (To allow further derived classes, this code should be put in a `SetObjects()` function instead of being inlined after the Window creation code.)

To use these derived classes, one simply creates a derived class of `JXChooseSaveFile` and overrides the appropriate `Create*Dialog()` function to create the appropriate derived class and then call `BuildWindow()`.

20.11 JXDecorRect

The class itself doesn't actually do anything, but it does define the useful concept of a decorative enclosure for a set of related widgets. Derived classes implement `DrawBorder()`. Derived classes include `JXBorderRect`, `JXDownRect`, `JXEmbossedRect`, `JXEngravedRect`, `JXFlatRect`, and `JXUpRect`.

20.12 JXFileDocument

This class provides functionality for Documents that are stored in a file on disk. It implements all functions required by `JXDocument`. The only function that it requires from derived classes is `WriteFile()`.

`JXFileDocument` handles all the dialog windows: `Save as`, `OK To Close`, and `OK To Revert`. These can be triggered via `SaveInNewFile()`, `OKToClose()`, and `OKToRevert()`. The messages can also be changed. All occurrences of `%f` in these messages are replaced by the name of the file. One can also replace the file chooser object via `SetChooseSaveFile()`. This is required in order to display custom dialog windows.

`SaveInCurrentFile()` saves the data in the current file, if it exists on disk. Otherwise, it calls `SaveInNewFile()`. The user is automatically warned if the file exists on disk but has been modified since it was read in. Call `DataModified()` to specify that the data needs to be saved before closing. `DataReverted()` does the opposite. `DefaultCanReadASCIIFile()` provides a convenient way to check the type of a file if one adheres to the format "`<file signature> <version number> <data>`". `ShouldMakeBackupFile()` can be used to specify whether or not a backup file (`<name>`˜) should be created when saving changes to the data.

The `JXFileDocument` class also implements a `SavetySave()` function that protects against crashes. It also allows for saving when an assert has been triggered.

20.13 JXHelpManager

This provides a simple way to display help for an application. The help system is all hypertext based and is set up via the `JXInitHelp()` global function.

20.14 JXInputField

This is the base class for all dialog window input fields. Derived classes include `JXString-Input`, `JXFloatInput`, and `JXIntegerInput`.

20.15 JXSliderBase

This is the base class for all Slider classes. Sliders provide a way for users to graphically set a value between specified limits with a specified increment between values. The limits and the step size are all floating point numbers.

20.16 JXStaticText

This class displays a non-editable text string. The font is adjustable. The text can be selectable to allow copying and dragging.

20.17 JXWidgetSet

The class itself doesn't actually do anything, but it does define the useful concept of a passive enclosure for a set of related Widgets. Derived classes should construct and arrange the Widgets so that clients can create the entire package with one call. As an example, refer to `JXScrollbarSet` above.

20.18 UNIX pipes

`JInPipeStream` and `JOutPipeStream` are provided to simplify the use of UNIX pipes. Each is constructed with the shell command to be executed. Since both are derived from the appropriate stream classes, `JInPipeStream` acts just like any other istream, and `JOutPipeStream` acts just like any other ostream.

For non-blocking IO, use `JMessageProtocol`, or derive your own class from the ACE library .

Chapter 21

Miscellaneous Details

21.1 Error Handling

JX was built on the principle that code can and should be written to never crash. C++ exceptions are therefore not supported at all. Instead, non-fatal problems are dealt with by returning a boolean value to indicate success or failure. Fatal problems are caught via the assert() macro.

Fatal problems are actions like attempting to access an element outside the boundaries of an array. Trying to exit gracefully (e.g. letting the user save their data before quitting) is impossible because something has been seriously corrupted and the user would be a fool to trust anything written out by the program after the error occurred. If code is written to be able to gracefully deal with a thrown exception in such cases, then the programmers should be shot because they should have redesigned the code to avoid the condition instead.

Non-fatal problems are actions like trying to convert the string "hello" to a number. The function will simply return kFalse. If this string was entered by the user, the problem can very easily be dealt with by printing an error message and asking the user to re-enter the information. If the string was read from a file, it should be treated as if it were entered by the user since the file could get corrupted.

Some will argue that using `assert()` makes software crash prone instead of crash proof since `assert()` always crashes the program. Uncaught exceptions do exactly the same, however, and continuing after an unrecognized exception is just as dangerous as continuing past a failed assert(). Exceptions also cause many other problems, especially inside constructors. (cfr. More Effective C++ by Scott Meyers, ISBN 0-201-63371-X, and Tom Cargill's article on the web "*Exception Handling: A False Sense of Security. - http://cseng.awl.com/bookdetail.qry?ISBN=0-201-63371-X&ptype=636*") If one knows that the code will crash on any error, then this simply provides incentive to design the system correctly the first time.

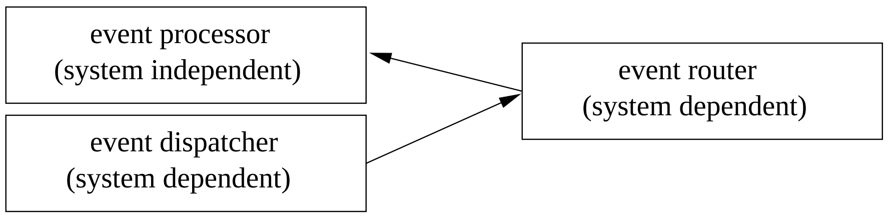
When using JX, use `jAssert.h` instead of `assert.h`. This version of `assert()` attempts to save the user's data in a safety save file before exiting.

One must also remember to `#include <jAssert.h>` as the last `#include` in the source file so that the correct definition of `assert()` is used. (All `assert.h` files throw out any previous definition of `assert()` before defining their own.)

As a final reminder, never do actual work inside an `assert()` because the production version of the code will probably `#define NDEBUG` which will kill all `assert()`'s and everything inside them. You will effectively lose code. This will be nearly impossible to track down because the debug version will work correctly.

21.2 Splitting code into system dependent and system independent parts

Since the basic equipment, such as a keyboard and a mouse, is available on all systems, most of the work of a visual interface can actually be done by system independent code. The trick is to join the system dependent event dispatching code and the system independent event processing code by using multiple inheritance:



The event router inherits from both the event dispatcher and the event processor. The arrows in the diagram show the flow of events. The system generates events which are caught by the event dispatcher. The event dispatcher's functions are declared virtual, so the event router's implementation of these functions is called. These functions interpret the events and call the appropriate functions in the event processor.

The Text Editor and the Table suite are good examples of this style of programming. `JTextEditor` and `JTable` provide all the code required to store the data and keep the display synchronized with the data and leave the issues of scrolling and catching mouse clicks to be implemented by the derived, system dependent classes `JXTEBase` and `JXTable`.

Bibliography

[1] Appel, Andrew W. *Modern Compiler Implementation in C. Cambridge*, UK: Cambridge University Press, 1997.

[2] Bentley, Jon L. *Programming Pearls*. Reading, MA: Addison-Wesley, 1986.

[3] Brooks Jr., Frederick P. *The Mythical Man Month*, 20th Ann. Ed. Reading, MA: Addison-Wesley.

[4] Brown, Chris. *UNIX Distributed Programming*. New York, NY: Prentice Hall, 1994.

[5] Cargill, Tom. *Exception Handling: A False Sense of Security.*

[6] Cooper, Alan. About Face, *The Essentials of User Interface Design*. Foster City, CA: Programmers Press, 1995.

[7] Coplien, James O. *Advanced C++, Programming Styles and Idioms*. Reading, MA: Addison-Wesley, 1992.

[8] Friedl, Jeffery. *Mastering Regular Expressions*. Sebastopol, CA: O'Reilly, 1997.

[9] Glass, Robert L. *Software Runaways.* Upper Saddle River, NJ: Prentice Hall, 1998.

[10] Goldstein, Neal, and Jeff Alger. *Developing Object-Oriented Software for the Macintosh*. Reading, MA: Addison-Wesley, 1992.

[11] Harbison, Samuel P., and Guy L. Steele. *C, A Reference Manual*. Englewood Cliffs, NJ: Prentice Hall, 1995.

[12] Kernighan, Brian W., and P.J. Plauger. *The Elements of Programming Style*. New York, NY: McGraw-Hill, 1978.

[13] Lakos, John. *Large-Scale C++ Software Design*. Reading, MA: Addison-Wesley, 1996.

[14] MacGuire, Steve. *Writing Solid Code*. Redmond, WA: Microsoft Press, 1993.

[15] McConnell, Steve. *Code Complete*. Redmond, WA: Microsoft Press, 1993.

[16] Meyers, Scott. *Effective C++*. Reading, MA: Addison-Wesley, 1992.

[17] Meyers, Scott. *More Effective C++*. Reading, MA: Addison-Wesley, 1996.

[18] Norman, Donald A. *The Design of Everyday Things*. New York, NY: Currency Doubleday, 1988.

[19] Sedgewick, Robert. *Algorithms in C++*. Reading, MA: Addison-Wesley, 1992.

[20] Tognazzini, Bruce. *Tog on Interface*. Reading, MA: Addison-Wesley, 1992.

Index