Building Applications With DUIM

Release 1.0

Dylan Hackers

December 15, 2018

Contents:

CHAPTER

ONE

COPYRIGHT

CHAPTER

TWO

PREFACE

About this manual

This manual, *Building Applications using DUIM*, provides an introduction to developing your own windowed applications using Open Dylan and, in particular, the interface-building functionality provided by the DUIM library suite. It is designed to complement *Getting Started with Open Dylan*, which provides information on using the Open Dylan development environment, and the *DUIM Reference Manual*, which provides a complete reference to the DUIM library suite. You are advised to look at *Getting Started with Open Dylan* before reading this manual in any depth.

This manual is divided into several parts:

Introduction provides an introduction to the concepts behind the DUIM libraries, and their intended use.

The chapters Designing A Simple DUIM Application through to Adding Callbacks to the Application provide an extended example of how to use DUIM to design the user interface to an application. A simple working application is developed from first principles, and this is used as an illustration of some of the most useful features provided by the DUIM libraries. Designing A Simple DUIM Application provides an initial design for the application, Improving The Design improves on this initial design. Adding Menus To The Application shows you how you can add a menu system to an application and Adding Callbacks to the Application demonstrates how to give the application some useful functionality. Using Command Tables introduces the concept of command tables, by re-implementing some of the functionality already described in Adding Menus To The Application. For reference, the full source code of the application described in these chapters is provided in Source Code For The Task List Manager.

A Tour of the DUIM Libraries provides an overall tour of what is available in the suite of DUIM libraries. It provides much less detail than the chapters covering application development, but covers a broader spectrum of functionality. This chapter can be seen as a general introduction to the material covered in the *DUIM Reference Manual*.

The material provided in A Tour of the DUIM Libraries is reasonably independent from the material provided in Chapters Designing A Simple DUIM Application to Adding Callbacks to the Application, and if you wish, you can read through the tour before looking at the example application. Whichever order you approach them in, you should expect some repetition of subject matter, however.

Running examples in this manual

Naturally, when developing your own DUIM applications, you create, edit, and compile files of source code, and organize them as projects based on Dylan libraries and modules, just as you would when developing Dylan code that uses any other library. When developing your application, you can also take advantage of the development environment to make this process smoother, and to execute sections of code using the interactor. Many of the examples in this manual can be run directly from the interactor. Furthermore, this manual assumes that you are reasonably familiar with the development environment provided by Open Dylan. If you are not, please refer to the *Getting Started with Open Dylan* manual.

When developing your own projects using the New Project wizard, new modules that use the DUIM library, and any other relevant libraries are created for you. You may also like to use the Dylan Playground to experiment safely with your development code while keeping your project-specific modules clean. You can open the Dylan Playground by choosing **Tools > Open Playground** from the Dylan the main window.

The full source code for both versions of the application is provided as part of the Open Dylan installation. To load them into the environment, choose **Tools > Open Example Project** and look in the Documentation category, at the examples labeled Task List.

Further reading

For more information about DUIM, you should refer to the **DUIM Reference Manual**. This provides complete reference material on all the libraries and modules provided by DUIM. A wide variety of examples are also provided as part of the standard installation. These can be loaded into the environment by choosing **Tools > Open Example Project** from the main window.

CHAPTER

THREE

INTRODUCTION

Overview of the DUIM libraries

The Dylan User Interface Manager (DUIM—pronounced "dwim") is a Dylan-based programming interface that provides a layered set of portable facilities for constructing user interfaces.

While DUIM provides an API to user interface facilities for the Dylan application programmer, it is not itself a window system toolkit. DUIM uses the service of the underlying window system and UI toolkits as much as possible. DUIM's API is intended to insulate the programmer from most of the complexities of portability, since the DUIM application need only deal with DUIM objects and functions regardless of their operating platform (that is, the combination of Dylan, the host computer, and the host window environment).

DUIM is a high level library that allows you to concentrate on how the interface looks and behaves rather than how to implement it on a particular platform. It abstracts out many of the concepts common to all window environments. The programmer is encouraged to think in terms of these abstractions, rather than in the specific capabilities of a particular host system. For example, using DUIM, you can specify the appearance of output in high-level terms and those high-level descriptions are turned into the appropriate appearance for the given host. Thus, the application has the same fundamental interface across multiple environments, although the details will differ from system to system.

The DUIM programming model

The Dylan User Interface Manager (DUIM) provides a complete functional interface so that you can use Open Dylan to develop and build graphical user interfaces (GUIs) for your applications. It comprises a suite of libraries, each of which provides a specific set of components necessary for developing a GUI.

DUIM has a simple overall design, ensuring that developers who are relatively new to Dylan can produce results quickly and effectively. At the same time, the design is robust enough to allow more experienced developers to extend and use DUIM in non-standard ways when required, in order to produce specific behavior.

Because it is completely written in Dylan, DUIM is able to harness all the power of the Dylan language. This means not only the clean object-oriented design of Dylan, but also the power of functionality such as macros and collections, together with the concise nature of the language syntax. This makes it easy to implement quite complicated GUI designs from the ground up, using small, clear pieces of code. This is in contrast to other GUI design libraries that have to rely on a much more verbose underlying language, such as C, which in turn leads to more complex GUI code that is harder to improve upon and maintain.

In the functionality that it provides, DUIM has a number of goals:

It should be as easy to use as possible.

As well as providing the minimum feature set necessary to build a GUI, DUIM provides functionality that lets you use common GUI features easily.

It should be as compact as possible.

DUIM does not provide *so much* functionality that either you, or the environment, is swamped in complexity.

It should be as portable as possible.

It should be relatively easy to compile code in, and for, as many different hardware and software configurations as possible.

DUIM provides support for all the controls available in every modern GUI environment, and also allows you to develop your own controls as required. As far as possible, DUIM code is not specific to any particular platform, and whenever possible, controls native to the target environment are used in the resulting executable. This has two important consequences for your code:

By using controls native to the target environment, it is easy to develop an application that has the correct look and feel for your platform.

It enables DUIM code to be compiled and run on any platform for which a DUIM backend has been implemented.

A DUIM interface is built from *frames* ; each window in your application is represented by a frame. Each frame contains a hierarchy of *sheets*, in which each sheet represents a unique piece of your window (the menu bar, buttons, and so on). DUIM also handles the event loop for you, allowing you to write methods to handle just the events you wish to treat specially.

The components of the sheet structure itself consist of three types of DUIM object:

- *Gadgets*, which are discrete GUI controls such as buttons, panes, and menus. These are the basic behavioral element of a GUI, and provide methods to handle events such as mouse clicks.
- *Layouts*, which are controls that, rather than having a physical appearance on screen, describe the arrangement of the sheets that are their children.
- *User-defined sheets*, which are implemented by you rather than by DUIM itself.

To implement a user-defined sheet, you create a new class and write methods to handle the different events that it receives, such as repainting itself, supporting mouse events, or handling the clipboard.

To develop an application using DUIM, you typically have to define a number of classes of frame (one for every kind of window or dialog in your application). The definition of each frame class contains a description of the sheet hierarchy that describes the contents of the frame, together with any slots and initial values that are required by the frame class. Once the frame classes are defined, you need to define callback functions that are invoked when certain events occur within the scope of the sheet hierarchy, such as mouse button clicks or textual input. These callback functions encapsulate the behavior of the application.

The chapters Designing A Simple DUIM Application to Using Command Tables provide an extended tutorial that illustrates the basic and most common principles involved in building a GUI for a simple application.

As well as a rich set of GUI controls, DUIM provides support for the following features that are required in GUI design:

- *Dialogs* You can build your own dialogs, wizards, and property frames using pre-supplied DUIM classes. In addition, a number of convenience functions are provided which let you add common dialogs (such as file requesters) to your GUI without having to design the dialog from scratch.

- *Graphics* DUIM provides portable models for colors, fonts, images, and generic drawing operations.
- *Events* DUIM provides portable models for keyboard handling and mouse handling, to simplify the process of writing your own event handling routines.
- *Layouts* DUIM makes it easy to lay out groups of controls in a variety of standard ways, letting you arrange controls in columns, rows, or tables. DUIM takes care of any necessary calculations, ensuring that the size of each control, and the spacing between controls, is correct, without the need for any explicit layout calculation on your part.

CHAPTER

FOUR

DESIGNING A SIMPLE DUIM APPLICATION

Introduction

The next few chapters of the manual introduce you to some of the most important DUIM concepts, and show you how to go about designing and implementing a simple DUIM application. On a first read through, you should work through each chapter in turn, since each chapter relies heavily on the information in the previous chapters.

Design of the application

For the purposes of this example, the application developed is a simple task list manager. The design of the application attempts to achieve the following goals:

- The design is simple enough that the principles of the programming model should not be obscured by the code itself.
- The design attempts to use the most common elements of the various DUIM libraries.
- The design is extensible, so that you can customize it to your own needs.

A task list manager was chosen because it is representative of the sort of GUI application that you will probably want to develop. Although the overall design is quite simple, it demonstrates several commonly used elements and techniques, and is easily extensible beyond the scope of this manual, should you wish to experiment with the code. The concept of a task list manager is familiar to the majority of readers, so you can study the code and the programming model, without having to spend time figuring out what the application itself is supposed to do.

The final task list manager is shown in Designing A Simple DUIM Application. To load the code for the final design into the environment, choose **Tools > Open Example Project** from any window in the environment, and load the Task List 1 project from the Documentation category of the Open Example Project dialog.

The task list manager is very simple to use. You create a list of things that you need to do, assigning a priority to each task as you create it. The application can display the tasks in your list sorted in a variety of ways. You can save your task list to a file on disk, and open files of the same type.

The task list manager demonstrates the use of menus and a variety of button, list, and text controls.

Creating the basic sheet hierarchy

This section shows you how to create gadgets and sheets that make up the overall visual design of the interface. It shows you how to improve upon an initial design, but does not go into any details on the callbacks necessary for the application; at the end of this section you have an initial visual design.

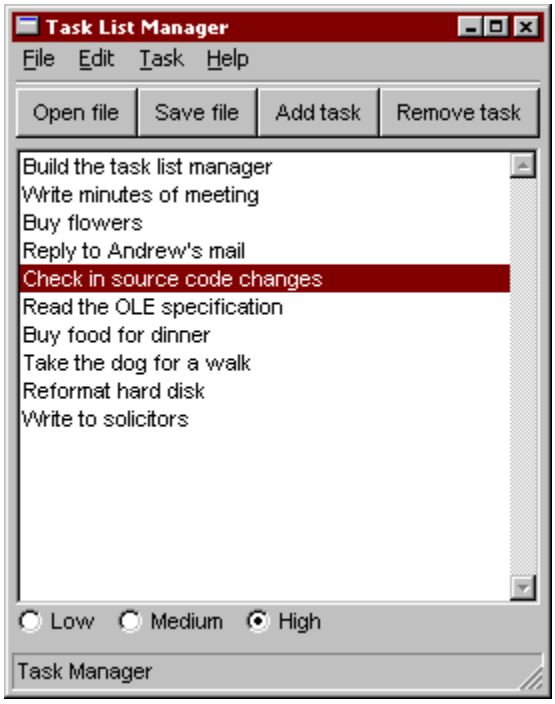

Fig. 4.1: The Task List Manager Application

Placing all the elements in a single layout

The main part of the task list manager is a list box that is used for displaying the tasks that you add in the course of using the program. For the initial design, there are buttons that let you add and remove tasks from the list, and a text field into which you type the text for new tasks.

To begin with, the following code creates all these elements, and places them in a single window, one above the other.

```
make(<column-layout>,
     children: vector (make (<list-box>, items: #(), lines:15),
                       make (<text-field>, label: "Task text:"),
                       make (<push-button>, label: "Add task"),
                       make (<push-button>,
                             label: "Remove task")));
```

You might notice a number of problems with this initial design:

Firstly, the items have all been created correctly, but the resulting window is not particularly attractive. In order to improve the appearance, you need to rearrange the elements in the window by making better use of the layout facilities provided by DUIM.

Secondly, the application does not yet look very much like a typical Windows application. Rather than individual buttons, the application should have a tool bar, and it is not common to have a text field in the main window of an application. There is no menu bar. Currently, the application has more of the feel of a dialog box, than a main application window. These issues are addressed later on in the example.

Redesigning the layout

To address the issue of layout first, you should group the text field and the *Add task* button in a row; since the two elements are inherently connected (the task you add is the one whose text is displayed in the text field), it makes sense

to group them visually as well.

The following code creates the necessary row layout:

```
horizontally ()
  make (<text-field>, label: "Task text:");
  make (<push-button>, label: "Add task");
end
```

Note that the macro `horizontally` has been used here. This macro takes any expressions that are passed to it and creates a row layout from the results of evaluating those expressions. The macro `vertically` works in a similar way, creating a column layout from its arguments. Use `vertically` to combine the row layout you just created with the *Remove task* button that still needs to be incorporated:

```
vertically ()
  horizontally ()
    make (<text-field>, label: "Task text:");
    make (<push-button>, label: "Add task");
  end;
  make (<push-button>, label: "Remove task");
end
```

Finally, you need to add this sheet hierarchy to another row layout, so that the main list box for the application is on the left, and the sheet hierarchy containing the buttons and text field is on the right:

```
horizontally ()
  make (<list-box>, items: #(), lines: 15);
  vertically ()
    horizontally ()
      make (<text-field>, label: "Task text:");
      make (<push-button>, label: "Add task");
    end;
    make (<push-button>, label: "Remove task");
  end;
end
```

In the last few steps, you have exclusively used `horizontally` and `vertically`. In fact, it does not matter if you use these macros, or if you create instances of `<row-layout>` and `<column-layout>` explicitly using `make`.

Note: You may have to resize the window to see everything.

Adding a radio box

There is one aspect of the initial design that you have not yet incorporated into the structure: the radio box. This serves two purposes in the application:

1. It lets you choose the priority for a new task.
2. It displays the priority of any task selected in the list.

The code to create the radio box is as follows:

```
make (<radio-box>, label: "Priority:",
      items: #("High", "Medium", "Low"),
      orientation: #"vertical");
```

Notice that the `orientation:` init-keyword can be used to ensure that each item is displayed one above the other.

It is probably best to place the radio box immediately below the *Remove task* button. To do this, you need to add the definition for the radio box at the appropriate position in the call to `vertically`.

```
(horizontally ()
  make (<list-box>, items: #(), lines: 15);
  vertically ()
    horizontally ()
      make (<text-field>, label: "Task text:");
      make (<push-button>, label: "Add task");
    end;
    make (<push-button>, label: "Remove task");
    make (<radio-box>, label: "Priority:",
          items: #("High", "Medium", "Low"),
          orientation: #"vertical");
  end);
```

Using contain to run examples interactively

You can use the function `contain` to run any of the examples above using the interactor available in the Dylan environment. This function lets you see the results of your work immediately, without the need to compile any source code or build a project, and is extremely useful for experimenting interactively when you are developing your initial ideas for a GUI design.

The `contain` function takes any expression that describes a hierarchy of sheets as an argument. It creates a frame which contains this sheet hierarchy, and displays the resulting frame on the screen.

Thus, to run any of the code segments shown in this chapter, simply pass them to `contain` as an argument. Here are two examples, adapted from earlier examples in this chapter, as illustrations of how to use `contain`.

```
contain (horizontally ()
           make (<text-field>, label: "Task text:");
           make (<push-button>, label: "Add task");
         end);
contain (make (<text-field>, label: "Task text:"));
```

At this point, take a few minutes to go back over this chapter and practice using `contain` to run the code fragments that have already been discussed.

CHAPTER

FIVE

IMPROVING THE DESIGN

The simple layout hierarchy described in Creating the basic sheet hierarchy has a number of problems associated with it, all of which revolve around the fact that the task list manager does not yet look very much like a standard Windows application. Although it is a simple design that does not warrant a complicated user interface, the design you have already seen looks more like a dialog box than an application window.

This section shows you how to improve on the basic design, adding a menu bar and replacing the buttons with a proper tool bar. It also shows you how to move the text field into a separate dialog that pops up when you click the *Add task* button in the tool bar.

From this point on, the interface is defined more formally, using frames. Up to now, the layout hierarchy has been presented informally, and you have used `contain` to display the layout interactively. This is fine for code that you want to evaluate once only, perhaps using the interactor, but for permanent code, a more rigorous framework is preferable.

Defining a project

From this point on, you should put the code for the task list manager into a project, rather than evaluating pieces of code using the interactor. Please refer to the *Getting Started with Open Dylan* for complete details on how to manage projects using the environment; this section provides brief details that tell you how to create a project specifically for the task list manager application. Use the New Project wizard to create a GUI application, and call the project `task-list` when prompted. The New Project wizard offers the option of generating template source code to help you get started. For this example, you must ensure that this option is switched *off* (this is the default setting).

Two versions of the task list manager are included with Open Dylan, so that you can load the code straight into the environment if you wish. These are available in the Open Example Project dialog, in the Documentation category. You can display this dialog by choosing **Tools > Open Example Project** from the environment. The two versions included represent the state of the task list manager at the end of Adding Callbacks to the Application, and at the end of Using Command Tables.

Note: Please note that both projects have the same name within the source code— `task-list` —and you should not load them both into the environment at the same time.

The number of source code files in a given project, and the names you give them, is entirely up to you. For the purposes of this example, you will use the files suggested by the New Project wizard. When you use the New Project wizard, Open Dylan will create a number of files for a project named `task-list`.

`module.dylan` and `library.dylan` These files define the library and modules for the project. For the purposes of this application, you can ignore these files.

`task-list.dylan` Add non-GUI-specific code to this file.

Finally, you need to create the following new file using *File* > *New*, and add it to the project using the *Project* > *Insert File* command. Make sure that this file is the last one listed in the project window.

`frame.dylan` Add the GUI-specific code to this file.

Starting the application

As you add source code to the files in your project, there will be times when you want to build the project to test it. This section defines some methods that let you run the application in a clean way. Add these methods to `frame.dylan`.

The frame class that is used to implement the task list manager is called `<task-frame>`. This class will be introduced in *Defining a new frame class*. You can define a method to create an instance of `<task-frame>` as follows:

```
define method start-task () => ()
  let frame = make(<task-frame>);
  start-frame(frame);
end method start-task;
```

This method is provided as a convenient way to create the frame and then start its event loop. It returns when the event loop shuts down.

Note: Obviously, you should not call this method until you have defined a frame class called `<task-frame>`.

Finally, you can start the application with the following method, and its subsequent call:

```
define method main (arguments :: <sequence>) => ()
  // handle the arguments
  start-task();
end method main;

begin
  main(application-arguments()) // Start the application!
end;
```

Make sure that this is the very last definition in the file `frame.dylan`, and remember that `frame.dylan` should itself be the last file listed in the project window.

Once you have added these methods to your code, you can compile and link the code, and run the application to test it, using the appropriate commands in the Dylan environment.

Note that, unlike languages such as C, Dylan does not insist on a single entrance point to an application such as the one given here. All the same, it is still good practice to define one if you can. The main difference between the use of the method `main` here, and the use of the `main` function in C, is in the arguments that need to be passed. In C, you need to pass two generic arguments: `argc`, which specifies the number of arguments you are passing, and `argv`, an array of strings that define the arguments themselves. In Dylan, however, you only need to pass the second of these arguments; since any Dylan collection already knows its own size, you do not need to pass the number of arguments as an additional parameter.

Adding a default callback

Nothing is more frustrating than designing a user interface that does not respond to user input. Although, in the early stages at least, the user interface does nothing particularly useful, you can at least define a "not yet implemented" message that can be used until you define real behavior for the application.

The definition of the function that gives you this default behavior is as follows:

```
define function not-yet-implemented (gadget :: <gadget>) => ()
  notify-user("Not yet implemented!", owner: sheet-frame(gadget))
end function not-yet-implemented;
```

Add this function to `frame.dylan`.

You can call this function from any gadget in the task list manager by defining it as the activate callback for each gadget. There are several types of callback, and this is the type that is used most in the task list manager. You can define the activate callback for any gadget using the `activate-callback:` init-keyword. More information about callbacks is given in Adding Callbacks to the Application, in which some real callbacks are defined, to make the task list manager do something more substantial.

Defining a new frame class

To begin with, define a frame class using the layout hierarchy you have already created. Although it might seem redundant to implement an inelegant layout again, it is easier to illustrate the basic techniques using a design you are already familiar with. In addition, there are several elements in the design that can be reused successfully.

Add the code described in this section to `frame.dylan`.

Defining a new class of frame is just like defining any Dylan class, except that there are several extra options available beyond the slot options normally available to `define class`. Each of these extra options lets you describe a particular aspect of the user interface. To define the new frame class, use the following structure:

```
define frame <task-frame> (<simple-frame>)
  // definitions of frame slots and options go here
end frame <task-frame>;
```

In this case, `<task-frame>` is the name of the new class of frame, and‘‘<simple-frame>‘‘ is its superclass. Like ordinary Dylan classes, frame classes can have any number of superclasses, with multiple superclasses separated by commas. The superclass of any “standard” frame is usually `<simple-frame>`. If you were designing a dialog box, its superclass would be `<dialog-frame>`. If you were designing a wizard, its superclass would be `<wizard-frame>`.

Adding slots to a frame class is exactly the same as adding slots to a standard Dylan class. You can define slot names, init-keywords, init-functions, default values, and so on. For this example, you are not defining any slots.

Each user interface element in the new class of frame is specified as a pane with a name and a definition. A pane is a sheet within a layout, and you can think of panes as sheets that represent concrete classes in an interface (as opposed to abstract classes). In effect, specifying a pane allows you to group together existing gadgets into some meaningful relationship that effectively creates a new gadget, without actually defining a gadget class.

The name is used to refer to the pane, both from within the frame definition itself, and from other code. The pane definition includes code to create the interface element. A pane specification also includes a place to declare a local variable that can be used within the pane’s definition to refer to the surrounding frame.

The following code fragment defines the two buttons, the text field, the radio box, and the list box from the initial design:

```
pane add-button (frame)
  make(<push-button>, label: "Add task",
       activate-callback: not-yet-implemented);
pane remove-button (frame)
  make(<push-button>, label: "Remove task",
       activate-callback: not-yet-implemented);
pane task-text (frame)
```

```
  make(<text-field>, label: "Task text:",
       activate-callback: not-yet-implemented);
pane priority-box (frame)
  make (<radio-box>, label: "Priority:",
        items: #("High", "Medium", "Low"),
        orientation: #"vertical",
        activate-callback: not-yet-implemented);
pane task-list (frame)
  make(<list-box>, items: #(), lines: 15,
       activate-callback: not-yet-implemented);
```

Note that the definition of each element is identical to the definitions included in the original layout described in Creating the basic sheet hierarchy (except that activate callbacks have been added to the code). Adding `(frame)` immediately after the name of each pane lets you refer to the frame itself within the frame definition using a local variable. This means that you can refer to any pane within the frame using normal slot syntax; that is, a pane called `my-pane` can be referred to as `frame.my-pane` throughout all of the definition of the frame class. This ability is essential when you come to layout each pane in the frame itself.

In addition, you need to define the layout in which to place these panes. This is itself just another pane, and its definition is again identical to the original layout described in Creating the basic sheet hierarchy, with one exception; rather than defining each element explicitly, you just include a reference to the relevant pane that you have already defined using normal slot syntax, thus:

```
pane task-layout (frame)
  horizontally ()
    frame.task-list;
    vertically ()
      horizontally ()
        frame.task-text;
        frame.add-button;
      end;
    frame.remove-button;
    frame.priority-box;
  end;
end;
```

To describe the top-level layout for the frame, you need to refer to this pane using the `layout` option, as follows:

```
layout (frame) frame.task-layout;
```

You actually have a certain amount of freedom when choosing what to define as a pane in the definition of a frame class. For example, the layout in the `task-layout` pane actually contains a number of sub-layouts. If you wanted, each one of these sub-layouts could be defined as a separate pane within the frame definition. Note, however, that you only have to "activate" the top-most layout; there should only be one use of the `layout` option.

Similarly, you are free to use whatever programming constructs you like when defining elements in your code. Just as in the earlier examples, you could define the layouts with explicit calls to `make`, rather than by using the `horizontally` and `vertically` macros. Thus, the following definition of `task-layout` is just as valid as the one above:

```
pane task-layout (frame)
  make(<row-layout>,
       children: vector(frame.task-list,
                        make(<column-layout>,
                             children: vector(make(<row-layout>,
                                                   children: vector(frame.task-text, frame.add-button)))))
```

Notice that this construct is rather more complicated than the one using macros!

Throughout this section, you may have noticed that you can identify a sequence of steps that need to occur inside the definition of a frame. It is good practice to keep this sequence in mind when writing your own frame-based code:

1. Define the content panes
2. Define the layout panes
3. Use the `layout` option

If you glue all the code defined in this section together, you end up with the following complete definition of a frame class.

```
define frame <task-frame> (<simple-frame>)
  pane add-button (frame)
    make(<push-button>, label: "Add task",
         activate-callback: not-yet-implemented);
  pane remove-button (frame)
    make(<push-button>, label: "Remove task",
         activate-callback: not-yet-implemented);
  pane task-text (frame)
    make(<text-field>, label: "Task text:",
         activate-callback: not-yet-implemented);
  pane priority-box (frame)
    make(<radio-box>, label: "Priority:",
         items: #("High", "Medium", "Low"),
         orientation: #"vertical",
         activate-callback: not-yet-implemented);
  pane task-list (frame)
    make (<list-box>, items: #(), lines: 15,
          activate-callback: not-yet-implemented);
  pane task-layout (frame)
    horizontally ()
      frame.task-list;
      vertically ()
        horizontally ()
          frame.task-text;
          frame.add-button;
        end;
        frame.remove-button;
        frame.priority-box;
      end;
    end;
  layout (frame) frame.task-layout;
  keyword title: = "Task List Manager";
end frame <task-frame>;
```

Note the addition of a `title:` keyword at the end of the definition. This can be used to give any instance of the frame class a title that is displayed in the title bar of the frame's window when it is mapped to the screen.

At this stage, the application still has no real functionality, and there is no improvement in the interface compared to the initial design, but by defining a frame class, the implementation is inherently more robust, making it easier to modify and, eventually, maintain.

If you want to try running your code, remember that you need to define some additional methods to create a frame instance and exit it cleanly. Methods for doing this were provided in *Starting the application*. If you define these methods now, you can create running versions of each successive generation of the application as it is developed.

Adding a tool bar

So far, you have seen how to experiment interactively to create an initial interface design. You have also seen how you can take that initial design and turn it into a more rigorous definition, for use within project source code, using a frame class. However, the design of the interface still leaves a lot to be desired, and the application still does not do anything. In this section, you start to look at improving the overall design of the interface.

To begin with, add a tool bar to the interface of the application. Most modern applications have a tool bar that runs along the top edge of the main application window, beneath the application menu bar. Tool bars contain a number of buttons that give you quick access to some of the most common commands in the application. Each button has a label that designates its use, or, more often, a small icon. Although you have already added buttons to the interface that perform important tasks, they have the appearance of buttons in a dialog box, rather than buttons in the main window of an application. The solution is to use a tool bar.

Adding a tool bar to the definition of a frame class is very similar to defining the overall layout of the panes in a frame class. You need to create the tool bar as a pane in the frame definition, and then incorporate it using the `tool-bar` clause, as shown below:

```
pane task-tool-bar (frame)
  make(<tool-bar>, child: ...);
// ...more definitions here...
tool-bar (frame) frame.task-tool-bar;
```

A tool bar has a layout as its child, and each button in the tool bar is defined as a child of that layout. You can either define each button within the definition of the tool bar itself, or, more appropriately, define each button as a pane in the frame, and then refer to the names of these panes in the tool bar definition.

In fact, the buttons you defined in the earlier interface design can be used just as easily in a tool bar as they can within the main layout of the application itself. However, first you must remove the buttons from the task-layout pane of the definition of `<task-frame>`. (If you fail to do this, DUIM attempts to use the same buttons in two different parts of the interface, with undefined results.) A complete definition of a simple tool bar containing two buttons is as follows:

```
pane task-tool-bar (frame)
  make(<tool-bar>,
       child: horizontally ()
                frame.add-button;
                frame.remove-button
              end);
// ...more definitions here...
tool-bar (frame) frame.task-tool-bar;
```

A tool bar that only contains two buttons is on the lean side, however, so let's add two more buttons to open a file and save a file to disk.

```
pane open-button (frame)
  make(<push-button>,
       label: "Open file",
       activate-callback: not-yet-implemented);
pane save-button (frame)
  make(<push-button>,
       label: "Save file",
       activate-callback: not-yet-implemented);
// ...more definitions here...
pane task-tool-bar (frame)
  make(<tool-bar>,
       child: horizontally ()
                frame.open-button;
                frame.save-button;
```

```
                frame.add-button;
                frame.remove-button
              end);
// ...more definitions here...
tool-bar (frame) frame.task-tool-bar;
```

More commonly, an icon is used to label buttons in a tool bar, rather than a text label. You can do this by supplying an instance of `<image>` to the `label:` init-keyword when you define the button, rather than an instance of `<string>`.

So now the application has a tool bar. Somewhat oddly, it does not yet have a menu bar or a system of menus — most tool bars represent a subset of the commands already available from the application's menu system. A menu system is added to the task list manager in Adding Menus To The Application.

Adding a status bar

As well as a tool bar, most applications have a status bar. This is a bar that runs along the bottom edge of the main application window, and displays information about the current status of the application. At its most basic, a status bar provides a label that displays text of some sort. In many applications, status bars contain a number of different fields, providing a wide range of functionality. At their most complex, a status bar may have several different labels that display information about the current state of the application, and labels that display help for the currently selected menu command.

It is worth adding a very simple status bar to the task list application. This contains a label that could eventually be used to display the name of the file currently loaded into the application. Adding a status bar to the definition of a frame class is very similar to adding a tool bar: you need to define a pane that describes the status bar, and then you need to incorporate it using the `status-bar` clause.

```
pane task-status-bar (frame)
  make(<status-bar>, label: "Task Manager");
// ...more definitions here...
status-bar (frame) frame.task-status-bar;
```

Now you have added a status bar to the application. The next step is to glue all the pieces together once again to create your modified frame design.

Gluing the new design together

In improving the initial design of the application, you have added a tool bar and a status bar. Adding a tool bar, in particular, has obviated the need for some of the elements that you added to the earlier version of the frame design. In this section, you throw away those elements that are no longer needed, and add in the new elements, to create a new, improved design for the frame class.

One part of the initial design you have not yet improved on is the radio box that shows the priority of any task in the list. Ideally, rather than using a radio box, you would display the priority of each task alongside the task itself, within the list box. For now, however, keep the radio box.

```
pane priority-box (frame)
  make(<radio-box>,
       items: $priority-items,
       orientation: #"horizontal",
       label-key: first,
       value-key: second,
       value: #"medium",
       activate-callback: not-yet-implemented);
```

Notice that the orientation is no longer constrained to be vertical. In the new design, a horizontal radio box looks better. By default, the orientation of a radio box is horizontal, so you could just completely remove the initialization of the `orientation:` init-keyword from the definition of the pane. In general, though, if you care about the orientation of a gadget, you should specify it explicitly, so leave the init-keyword in the pane definition, and change its value, as shown above.

Next, notice that the items are now specified using a named constant, rather than by embedding literals in the pane definition. The definition of this constant is as follows:

```
define constant $priority-items
  = #(#("Low", #"low"),
      #("Medium", #"medium"),
      #("High", #"high"));
```

Add the definition for this constant to `frame.dylan`.

Using lists of string and symbol values in this constant lets you assign values to the individual components of the radio box elegantly, in conjunction with the other improvements to the definition of `priority-box`.

- The *label key* is a function which is passed an entry from the sequence and returns a string to use as the label.

  Assigning `first` to the label key of `priority-box` ensures that the first element from each sub-list of `$priority-items` (the string) is used as the label for the appropriate item. Thus, the first button in priority box is labeled "Low".

- The *value key* is a function which is passed an entry and returns the logical value of the entry.

  Assigning `second` to the value key of `priority-box` ensures that the second element from each sub-list of `$priority-items` (the symbol) is used as the value for the appropriate item. Thus, the first button in priority box has the value `#"low"`.

Lastly, `priority-box` is given a default value: `#"medium"`. This ensures that the button labeled "Medium" is selected by default whenever `priority-box` is first created.

The definitions for `add-button`, `remove-button`, and `task-list` remain unchanged from the initial design. In addition, you need to add the definitions for `open-button` and `save-button` described in *Adding a tool bar*.

You also need to add in the definitions for the tool bar and status bar themselves, as described in *Adding a tool bar* and *Adding a status bar*.

The definition for `task-layout` has become much simpler. Because you have added buttons to the tool bar, the main layout for the application has reduced to a single column layout whose children are `task-list` and `priority-box`.

The definition for the new design of the frame class now looks as follows (button definitions vary slightly for the Task List 2 project, see A task list manager using command tables):

```
define frame <task-frame> (<simple-frame>)
  // definition of buttons
  pane add-button (frame)
    make(<push-button>, label: "Add task",
         activate-callback: not-yet-implemented);
  pane remove-button (frame)
    make(<push-button>, label: "Remove task",
         activate-callback: not-yet-implemented);
  pane open-button (frame)
    make(<push-button>, label: "Open file",
         activate-callback: not-yet-implemented);
  pane save-button (frame)
    make(<push-button>, label: "Save file",
         activate-callback: not-yet-implemented);
  // definition of radio box
```

```
  pane priority-box (frame)
    make (<radio-box>,
          items: $priority-items,
          orientation: #"horizontal",
          label-key: first,
          value-key: second,
          value: #"medium",
          activate-callback: not-yet-implemented);
  // definition of tool bar
  pane task-tool-bar (frame)
    make(<tool-bar>,
         child: horizontally ()
           frame.open-button;
           frame.save-button;
           frame.add-button;
           frame.remove-button
         end);
  // definition of status bar
  pane task-status-bar (frame)
    make(<status-bar>, label: "Task Manager");
  // definition of list
  pane task-list (frame)
    make (<list-box>, items: #(), lines: 15,
          activate-callback: not-yet-implemented);
  // main layout
  pane task-layout (frame)
    vertically ()
      frame.task-list;
      frame.priority-box;
    end;
  // activation of frame elements
  layout (frame) frame.task-layout;
  tool-bar (frame) frame.task-tool-bar;
  status-bar (frame) frame.task-status-bar;
  // frame title
  keyword title: = "Task List Manager";
end frame <task-frame>;
```

Note that this definition does not incorporate the original `task-text` pane defined in *Defining a new frame class*. In fact, this part of the original interface is handled differently in the final design, and is re-implemented in *Creating a dialog for adding new items*.

Creating a dialog for adding new items

You may be wondering what has happened to `task-text`, the text field in which you typed the text of each new task. In the new design, this is moved to a new dialog box that is popped up whenever you choose a command to add a new task to the list. This section shows you how to define this dialog.

The method `prompt-for-task` below creates and displays a dialog that asks the user to type the text for a new task. The definition of `task-text` is very similar to the definition you provided in the initial design, with the exception that the activate callback exits the dialog, rather than calling the `not-yet-implemented` function.

The method takes two keyword arguments: a title, which is assigned a value by default, and an owner, which is used as the owner for the dialog that is displayed by `prompt-for-task`. Note that the title is never explicitly set by any calls to `prompt-for-task` in the task list manager; it is provided here as an illustration of how you can provide a default value for a keyword argument, rather than requiring that it either always be passed in the call to the method, or

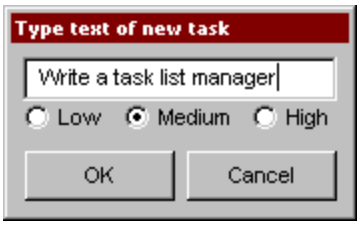

Fig. 5.1: The dialog box created by the prompt-for-task method

that it be hard-wired into the code.

The method returns two values: the name of the new task, that is, the text that the user types into the text field, and the priority of the new task.

Add this method to `frame.dylan`.

Note: The definition of the `prompt-for-task` method uses the `<priority>` type. Note that this type is defined in Defining the underlying data structures for tasks. Until the relevant code in Defining the underlying data structures for tasks is added to your project, any attempt to build it will generate a serious warning.

```
define method prompt-for-task
    (#key title = "Type text of new task", owner)
 => (name :: false-or(<string>),
  priority :: false-or(<priority>))
  let task-text = make(<text-field>,
                       label: "Task text:",
                       activate-callback: exit-dialog);
  let priority-field = make(<radio-box>,
                            items: $priority-items,
                            label-key: first,
                            value-key: second,
                            value: #"medium");
  let frame-add-task-dialog = make(<dialog-frame>,
                                   title: title,
                                   owner: owner,
                                   layout: vertically ()
                                     task-text;
                                     priority-field
                                   end,
                                   input-focus: task-text);
  if (start-dialog(frame-add-task-dialog))
    values(gadget-value(task-text), gadget-value(priority-field))
  end
end method prompt-for-task;
```

Notice that the dialog used in the `prompt-for-task` method is created inline within the method definition. In this particular case, the dialog is only ever needed within the context of `prompt-for-task` and so it is not necessary to use `define frame` to create a distinct class of frame specifically for this dialog.

Note also that *OK* and *Cancel* buttons are generated automatically for a dialog box; you do not need to define them explicitly.

Later on, the activate callback you define for the `add-button` pane calls this method, then inserts the return value into the list `task-list`.

CHAPTER

SIX

ADDING MENUS TO THE APPLICATION

Now it is time to consider adding some menus to your application. There are two basic ways that you can create a system of menus for your application:

- Design a hierarchical series of panes using the `<menu-bar>`, `<menu>`, and various menu buttons classes, and glue the elements of this design together in the correct order.
- Use a command table.

In this chapter, the first of these methods is demonstrated. For information about command tables, refer to Using Command Tables. Before discussing the first method listed above, the overall design of the menu system for the task list manager is discussed.

A description of the menu system

Before implementing the menus for the task list manager, it is worth describing what you are going to implement. The menu system of the task list manager comprises four menus: a *File* menu, *Edit* menu, *Task* menu, and *Help*. Each of these menus contains a number of commands, as follows:

- *File* menu The *File* menu contains four commands that operate on the files loaded into the task list manager. The *Open* command opens a new file. The *Save* command saves the currently loaded file to disk. The *Save As* command saves the currently loaded file to disk under a new name. The *Exit* command quits the task application completely.
- *Edit* menu The *Edit* menu contains the standard clipboard commands: *Cut*, *Copy*, and *Paste*.
- *Task* menu The *Task* menu contains two commands that operate on individual tasks. The *Add* command adds a new task to the list. The *Remove* command removes the selected task from the list.
- *Help* menu In a full-blown application, you would use commands in the *Help* menu as one hook into your online help system (other hooks being provided by buttons in dialog boxes and the F1 key). In this application, the *Help* menu contains a single command that simply displays a simple About dialog for the application.

Creating a menu hierarchy

As you might expect, creating a menu hierarchy in a frame definition is a matter of defining a series of panes for the frame. At the top-most level in the menu hierarchy is the menu bar itself. The menu bar contains each menu defined for the application and each menu contains the menu commands that themselves perform operations. Once the panes have been defined, the menu bar needs to be included in the frame using the `menu-bar` clause.

First of all, you can create a pane that defines the menu bar itself as follows:

```
pane task-menu-bar (frame)
  make(<menu-bar>,
       children: vector(frame.file-menu,
                        frame.edit-menu,
                        frame.task-menu,
                        frame.help-menu));
```

Next, define the File and Tasks menus themselves:

```
pane file-menu (frame)
  make(<menu>, label: "File",
       children: vector(frame.open-menu-button,
                        frame.save-menu-button,
                        frame.save-as-menu-button,
                        frame.exit-menu-button));
pane edit-menu (frame)
  make(<menu>, label: "Edit",
       children: vector(frame.cut-menu-button,
                        frame.copy-menu-button,
                        frame.paste-menu-button));
pane task-menu (frame)
  make(<menu>, label: "Task",
       children: vector(frame.add-menu-button,
                        frame.remove-menu-button));
pane help-menu (frame)
  make(<menu>, label: "Help",
       children: vector(frame.about-menu-button));
```

Finally, you need to define the menu commands themselves. A command that appears on a menu is defined as an instance of `<menu-button>`, and so there is a strong similarity between these buttons and some of the buttons already defined. DUIM also generates mnemonics for each menu item; thus, the items appear as *File* and *Edit*, and so forth. (Note that the `make-keyboard-gesture` function that appears below is defined in *Keyboard accelerators*.)

```
// Commands in the File menu
pane open-menu-button (frame)
  make(<menu-button>, label: "Open...",
       activate-callback: not-yet-implemented,
       accelerator: make-keyboard-gesture(#"o", #"control"),
       documentation: "Opens an existing file.");
pane save-menu-button (frame)
  make(<menu-button>, label: "Save",
       activate-callback: not-yet-implemented,
       accelerator: make-keyboard-gesture(#"s", #"control"),
       documentation: "Saves the current file to disk.");
pane save-as-menu-button (frame)
  make(<menu-button>, label: "Save As...",
       activate-callback: save-as-file,
       documentation: "Saves the current file with a new name.");
pane exit-menu-button (frame)
  make(<menu-button>, label: "Exit",
       activate-callback: not-yet-implemented,
       accelerator: make-keyboard-gesture(#"f4", #"alt"),
       documentation: "Exits the application.");

//Commands in the Edit menu
pane cut-menu-button (frame)
  make(<menu-button>, label: "Cut",
       activate-callback: not-yet-implemented,
       accelerator: make-keyboard-gesture(#"x", #"control"),
```

```
        documentation: "Cut the selection to the clipboard.");
pane copy-menu-button (frame)
  make(<menu-button>, label: "Copy",
        activate-callback: not-yet-implemented,
        accelerator: make-keyboard-gesture(#"c", #"control"),
        documentation: "Copy the selection to the clipboard.");
pane paste-menu-button (frame)
  make(<menu-button>, label: "Paste",
        activate-callback: not-yet-implemented,
        accelerator: make-keyboard-gesture(#"v", #"control"),
        documentation: "Paste the selection in the clipboard at the current position.");

//Commands in the Task menu
pane add-menu-button (frame)
  make(<menu-button>, label: "Add...",
        activate-callback: not-yet-implemented,
        accelerator: make-keyboard-gesture
                       (#"a", #"control", #"shift"),
        documentation: "Add a new task.");
pane remove-menu-button (frame)
  make(<menu-button>, label: "Remove",
        activate-callback: not-yet-implemented,
        accelerator: make-keyboard-gesture
                       (#"d", #"control", #"shift"),
        documentation: "Remove the selected task from the list.");

//Commands in the Help menu
pane about-menu-button (frame)
  make(<menu-button>, label: "About",
        activate-callback: not-yet-implemented,
        accelerator: make-keyboard-gesture(#"f1"),
        documentation:
          "Display information about the application.");
```

Once you have defined the menu bar and all the children that it is to contain, you need to activate the menu bar in the frame by including the following towards the end of the frame definition.

```
menu-bar (frame) frame.task-menu-bar;
```

The definitions of these menu buttons demonstrate two interesting new features: the use of keyboard accelerators, and the use of documentation strings.

Documentation strings

Documentation strings let you provide brief online help for gadgets such as menu buttons. You can specify a documentation string for any gadget using the `documentation:` init-keyword. Although you can make whatever use you want of these strings, using the `gadget-documentation` and `gadget-documentation-setter` methods, documentation strings for menu buttons are used in status bars without any need for special action on your part. If you display a menu and move the mouse pointer over the items in the menu, then the documentation string defined for each item is displayed in the status bar of the frame for as long as the mouse pointer is over the menu item. It is generally good practice to supply documentation strings for all the menu items in a frame. Documentation strings for other gadgets become tooltips in Windows.

Keyboard accelerators

Keyboard accelerators let you define a combination of keys that can be pressed in order to invoke the activate callback of a gadget. This means that you can access the functionality of an application without having to choose commands from menus using the mouse, and can make it much quicker to use an application you are familiar with.

To specify a keyboard accelerator, you need to specify an alphanumeric character, or a function key, together with any modifier keys (such as the CONTROL or ALT keys) that should be held down while the alphanumeric character is pressed. You actually create a keyboard accelerator by calling the `make` method on `<keyboard-gesture>`, though to make it a little easier, define the function below, which is used in the definition of each menu button.

```
define function make-keyboard-gesture
  (keysym :: <symbol>, #rest modifiers)
=> (gesture :: <keyboard-gesture>)
  make(<keyboard-gesture>, keysym: keysym, modifiers: modifiers)
end function make-keyboard-gesture;
```

Add this definition to the file *frame.dylan*.

The keyboard accelerators defined demonstrate the several useful points about keyboard accelerators:

- Whenever possible, use standard keyboard accelerators for standard application commands on your platform. Here, you use CONTROL+O to open a file, CONTROL+S to save a file, and CONTROL+X, CONTROL+C, and CONTROL+V respectively for *Cut*, *Copy*, and *Paste*.
- As well as standard alphanumeric characters, you can use function keys as keyboard accelerators.
- As well as the more common CONTROL key, you can use the ALT and SHIFT keys as modifiers, though you should not use the SHIFT key as the sole modifier.
- You can use more than one modifier key at once.
- If you wish, you need not use any modifier keys at all, as is the case with the (slightly non-standard) keyboard accelerator for the *About* command.

Gluing the final design together

You can now add the definitions of the menu bar, menus, and menu buttons, to the definition of the `<task-frame>` class, to give the code shown below. At this stage, the only thing missing from the final application are real callback functions. Callbacks are dealt with in Adding Callbacks to the Application.

Note that the final definition of `<task-frame>` includes the definition of a slot: `frame-task-list`. This takes an instance of the class `<task-list>` as a value, the default value being an empty `<task-list>`. Although it has not been referred to so far, this class will be used as the basic data structure in which task lists are stored, and a more complete description of these data structures is given in Defining the underlying data structures for tasks. It transpires that defining the `frame-task-list` slot is essential for some of the file handling routines that are described in Handling files in the task list manager.

```
define frame <task-frame> (<simple-frame>)
  slot frame-task-list :: <task-list> = make(<task-list>);

  // definition of menu bar
  pane task-menu-bar (frame)
    make(<menu-bar>,
         children: vector(frame.file-menu,
  frame.edit-menu,
  frame.task-menu,
  frame.help-menu));
```

```
// definition of menus
pane file-menu (frame)
  make(<menu>, label: "File",
       children: vector(frame.open-menu-button,
frame.save-menu-button,
frame.save-as-menu-button,
frame.exit-menu-button));
pane edit-menu (frame)
  make(<menu>, label: "Edit",
       children: vector(frame.cut-menu-button,
frame.copy-menu-button,
frame.paste-menu-button));
pane task-menu (frame)
  make(<menu>, label: "Task",
       children: vector(frame.add-menu-button,
frame.remove-menu-button));

pane help-menu (frame)
  make(<menu>, label: "Help",
       children: vector(frame.about-menu-button));

// definition of menu buttons

// Commands in the File menu
pane open-menu-button (frame)
  make(<menu-button>, label: "Open...",
       activate-callback: not-yet-implemented,
       accelerator: make-keyboard-gesture(#"o", #"control"),
       documentation: "Opens an existing file.");
pane save-menu-button (frame)
  make(<menu-button>, label: "Save",
       activate-callback: not-yet-implemented,
       accelerator: make-keyboard-gesture(#"s", #"control"),
       documentation: "Saves the current file to disk.");
pane save-as-menu-button (frame)
  make(<menu-button>, label: "Save As...",
       activate-callback: save-as-file,
       documentation:
         "Saves the current file with a new name.");
pane exit-menu-button (frame)
  make(<menu-button>, label: "Exit",
       activate-callback: not-yet-implemented,
       accelerator: make-keyboard-gesture(#"f4", #"alt"),
       documentation: "Exits the application.");

//Commands in the Edit menu
pane cut-menu-button (frame)
  make(<menu-button>, label: "Cut",
       activate-callback: not-yet-implemented,
       accelerator: make-keyboard-gesture(#"x", #"control"),
       documentation: "Cut the selection to the clipboard.");
pane copy-menu-button (frame)
  make(<menu-button>, label: "Copy",
       activate-callback: not-yet-implemented,
       accelerator: make-keyboard-gesture(#"c", #"control"),
       documentation: "Copy the selection to the clipboard.");
pane paste-menu-button (frame)
  make(<menu-button>, label: "Paste",
```

```
        activate-callback: not-yet-implemented,
        accelerator: make-keyboard-gesture(#"v", #"control"),
        documentation:
          "Paste the selection in the clipboard at the current position.");

//Commands in the Task menu
pane add-menu-button (frame)
  make(<menu-button>, label: "Add...",
        activate-callback: not-yet-implemented,
        accelerator: make-keyboard-gesture
                       (#"a", #"control", #"shift"),
        documentation: "Add a new task.");
pane remove-menu-button (frame)
  make(<menu-button>, label: "Remove",
        activate-callback: not-yet-implemented,
        accelerator: make-keyboard-gesture
                       (#"d", #"control", #"shift"),
        documentation:
          "Remove the selected task from the list.");

//Commands in the Help menu
pane about-menu-button (frame)
  make(<menu-button>, label: "About",
        activate-callback: not-yet-implemented,
        accelerator: make-keyboard-gesture(#"f1"),
        documentation:
          "Display information about the application.");

// definition of buttons
pane add-button (frame)
  make(<push-button>, label: "Add task",
        activate-callback: not-yet-implemented);
pane remove-button (frame)
  make(<push-button>, label: "Remove task",
        activate-callback: not-yet-implemented);
pane open-button (frame)
  make(<push-button>, label: "Open file",
        activate-callback: not-yet-implemented);
pane save-button (frame)
  make(<push-button>, label: "Save file",
        activate-callback: not-yet-implemented);

// definition of radio box
pane priority-box (frame)
  make (<radio-box>,
        items: $priority-items,
        orientation: #"horizontal",
        label-key: first,
        value-key: second,
        value: #"medium",
        activate-callback: not-yet-implemented);

// definition of tool bar
pane task-tool-bar (frame)
  make(<tool-bar>,
        child: horizontally ()
          frame.open-button;
          frame.save-button;
```

```
            frame.add-button;
            frame.remove-button
          end);

  // definition of status bar
  pane task-status-bar (frame)
    make(<status-bar>, label: "Task Manager");

  // definition of list
  pane task-list (frame)
    make (<list-box>, items: #(), lines: 15,
          activate-callback: not-yet-implemented);

  // main layout
  pane task-layout (frame)
    vertically ()
      frame.task-list;
      frame.priority-box;
    end;

  // activation of frame elements
  layout (frame) frame.task-layout;
  tool-bar (frame) frame.task-tool-bar;
  status-bar (frame) frame.task-status-bar;
  menu-bar (frame) frame.task-menu-bar;

  // frame title
  keyword title: = "Task List Manager";
end frame <task-frame>;
```

CHAPTER

SEVEN

ADDING CALLBACKS TO THE APPLICATION

At this point, the task list manager still does very little. If you try running the code (as described in Starting the application), and interacting with any of the elements in the GUI (clicking on a button, choosing a menu command, and so on), then only the "not yet implemented" message is displayed. This section shows you how to remedy this situation, by adding callback functions to the task list manager.

Getting the application to respond to mouse events such as clicking on a button or choosing a menu command consists of two things:

For each gadget in the GUI, you need to specify which callbacks to use. There are several different types of callback, depending on the type of event for which you want to define behavior.

You need to define the callback functions themselves. These are the functions that are invoked when a particular callback type is detected, and are the functions you use to define the correct behavior for your application.

In addition, you need to set up the basic data structures that allow you to work with tasks in your application.

At this point, you may be wondering exactly what a callback is, and why they are used to respond to application events, rather than event handlers. If you have developed GUI applications using other development environments, you may be more used to writing event handlers that work for a whole class of objects, and discriminating on which instance of a class to work on at any one time by means of case statements.

Writing event handlers in this way can be cumbersome. It turns out to be much simpler to define a function that works only for a particular instance of a class, and then refer to this function when defining the class instance. This function is what is referred to as a callback. This makes the source code for your application much clearer and easier to write, and the only price you pay is that you have to specify a callback for each gadget when you define the gadget itself.

In fact, DUIM provides a complete protocol for defining and handling events of all descriptions. However, you only need to use this protocol if you are creating new classes of gadgets, for which you need to define the event behavior, or new classes of events (for example, support for different input devices or notification of low resources). If you are just using gadgets, then you only ever need to use callbacks.

Defining the underlying data structures for tasks

Before defining any real callbacks, it is time to consider how you can represent task lists, and the information contained in them. This is essential, not just for handling tasks within the application, but for saving task lists to disk, and loading them back into the application.

Add the code described in this section to `task-list.dylan`.

There are two basic kinds of object that you need to model: task lists and tasks. A task list is a collection of one or more tasks. The best way to represent these is by defining a `<task-list>` class and a `<task>` class.

The definition of `<task-list>`, below, contains three slots:

`task-list-tasks`

This slot specifies a sequence of tasks that are contained in the task list. Each object in the sequence will be an instance of `<task>`. The default for new task lists is an empty stretchy vector. An init-keyword has been specified so that this slot can be set when an instance of the class is initialized.

`task-list-filename`

This slot specifies the file on disk to which the task list has been saved, if it has been saved at all. The default for new task lists is `#f`, since the task list has not yet been saved to disk. An init-keyword has been specified so that this slot can be set when an instance of the class is initialized.

`task-list-modified?`

The purpose for this slot is less obvious. It is useful to flag whether or not a task list has been modified so that, for instance, the *Save* command in the application can be disabled if the task list is unmodified. There is no init-keyword defined for this class, because you only ever want to use the supplied default value for new instances of `<task-list>`.

```
define class <task-list> (<object>)
  constant slot task-list-tasks = make(<stretchy-vector>),
    init-keyword: tasks:;
  slot task-list-filename :: false-or(<string>) = #f,
    init-keyword: filename:;
  slot task-list-modified? :: <boolean> = #f;
end class <task-list>;
```

Next, consider the information that needs to be encoded in each individual task. There are two pieces of information that need to be recorded:

- The text of the task, which should be a string.
- The priority, which should be one of high, medium, or low.

Priorities can be recorded using a constant, as shown below:

```
define constant <priority> = one-of(#"low", #"medium", #"high");
```

Notice that it is most straightforward to encode each priority as a symbol. Later on, you will see how you can use `as` to convert each symbol to a format that can be saved to disk and read back into the application as a symbol.

The `<task>` class can then be defined as having two slots: one for the task text itself, and another for the priority. Both have init-keywords so that they can be specified when a new instance is created, and both init-keywords are required; they must be specified whenever a task is created.

```
define class <task> (<object>)
  slot task-name :: <string>,
    required-init-keyword: name:;
  slot task-priority :: <priority>,
    required-init-keyword: priority:;
end class <task>;
```

These three definitions are all that is needed to be able to represent tasks and task lists within the task list application.

In order to handle tasks effectively in the GUI of the task list manager, some changes are necessary to the definition of the `task-list` pane in the definition of `<task-frame>`. These changes are needed to ensure that information about tasks is passed to the `task-list` pane correctly. Make these changes to the existing definition in the file `frame.dylan`.

In Gluing the final design together, the definition of `task-list` was given as:

```
// definition of list
pane task-list (frame)
```

```
  make (<list-box>, items: #(), lines: 15,
        activate-callback: not-yet-implemented);
```

First, you need to ensure that the items passed to `task-list` are the tasks in the `<task-list>` associated with the frame. Recall that a `frame-task-list` slot was specified in the definition of `<task-frame>` ; this slot is used to hold the instance of `<task-list>` that is associated with the `<task-frame>`. The sequence of tasks contained in the associated `frame-task-list` can then be found using the `frame-task-list.task-list-tasks` accessor. To display these tasks in the `task-list` pane, the `items:` init-keyword needs to be set to the value of this accessor:

```
items: frame.frame-task-list.task-list-tasks,
```

Next, you need to ensure that the label for each task in the `task-list` pane is the text of the task itself. As described above, the text of any task is stored in its `task-name` slot. In order to display this text as the label for every item in the list box, you need to specify the `task-name` slot as the `gadget-label-key` of the list box. A label key is a function that is used to calculate the label of each item in a gadget, and it can be specified using the `label-key:` init-keyword:

```
label-key: task-name,
```

This gives the following new definition for the `task-list` pane:

```
// definition of list
pane task-list (frame)
  make (<list-box>,
        items: frame.frame-task-list.task-list-tasks,
        label-key: task-name,
        lines: 15,
        activate-callback: not-yet-implemented);
```

There is one final change that still needs to be made to this pane definition. This is described in *Updating the user interface*.

Specifying a callback in the definition of each gadget

As you have already seen when using the `not-yet-implemented` callback, providing a callback for a gadget is just a matter of specifying another keyword-value pair in the definition of the gadget. There are two ways that you can specify the callback function to use.

If you wish, you can define the callback function inline, making the definition itself the value part of the keyword-value pair.

This can be useful for a simple callback function that you only need to invoke from a single callback type in a single pane. However, if several panes, or several types of callback, need to invoke the same callback function, you need to define the function explicitly in each gadget that uses it.

Alternatively, you can define a callback function explicitly in your application code, and then refer to it by name in the keyword-value pair.

This method is best for portability and reusability of your code, since the same callback function can be referred to by name in as many gadgets as you need to use it in, without having to redefine the callback function in each gadget. It can also lead to more readable source code. This technique is the one used throughout this example application.

As already mentioned, there are a number of different kinds of callback available, depending on the behavior that you want to specify, and the gadget for which you are defining a callback. When defining different callbacks for a gadget, you need to use a different init-keyword for each callback.

As you have already seen, by far the most common callback is the activate callback. This type of callback is invoked when you activate any instance of <action-gadget>. For buttons, the activate callback is invoked when you click on the button. For menu commands, the activate callback is invoked when you choose the command from the menu. The activate callback is the callback that is used most in the task list manager. You can specify an activate callback for any gadget using the activate-callback: init-keyword. In addition, you have seen the value-changed callback, which is invoked when the gadget-value has been changed. You can specify this callback using the value-changed-callback: init-keyword.

You have already defined a callback for all the gadgets in the GUI. All you need to do now is replace the reference to not-yet-implemented with the real function name that should get called when each gadget is activated. Thus, to specify an activate callback for the *Add task* button in the tool bar, redefine the button as follows in the definition of the <task-frame> class:

```
pane add-button (frame)
  make(<push-button>, label: "Add task",
       activate-callback: frame-add-task);
```

You can use exactly the same callback in the new definition of add-menu-button :

```
pane add-menu-button (frame)
  make(<menu-button>, label: "Add...",
       activate-callback: frame-add-task,
       accelerator: make-keyboard-gesture
                      (#"a", #"control", #"shift"),
       documentation: "Add a new task.");
```

Notice how both of these gadgets specify the same activate callback. This is because the *Add* command in the menu should perform exactly the same action as the *Add task* button in the tool bar.

At this point, redefine the callback for each gadget listed in the table below, making sure that you supply the same callback to those gadgets that perform the same functions.

The callback functions used in the Task List Manager

Gadget

Callback

open-menu-button

open-file

save-menu-button

save-file

save-as-menu-button

save-as-file

exit-menu-button

exit-task

add-menu-button

frame-add-task

remove-menu-button

frame-remove-task

about-menu-button

about-task

add-button

frame-add-task

remove-button

frame-remove-task

open-button

open-file

save-button

save-file

The following sections show you how to define the callbacks themselves. You will need to define other functions and methods, as well as the callback functions listed above. These other functions and methods are called by some of the callbacks themselves.

Defining the callbacks

This section shows you how to define the callbacks that are necessary in the task list manager, as well as any other associated functions and methods.

- First, you will look at methods and functions that enable file handling in the task list manager; that is, functions and methods that let you save and load files into the application.
- Next, you will look at methods and functions for adding and removing tasks from the task list.
- Last, you will define a few additional methods that are necessary to update the GUI elegantly, when other operations are performed.

All the code discussed in this chapter is structured so that callbacks which affect the GUI do not also perform other tasks that are not related to the GUI. This helps to keep the design of the application clean, so that you can follow the code more easily, and is recommended for all GUI design. Separating GUI code and non-GUI code also lets you produce code that is more easily reusable, either in other parts of a developing application, or in completely different applications.

Handling files in the task list manager

To begin with, you will define the functions and methods that let you save files to disk and load them back into the task list manager. Once you have added these to your code, you will be able to save and reload your task lists into the application; this type of functionality is essential in even the most trivial application.

There are three methods and two functions necessary for handling files. The methods handle GUI-specific operations involved in loading and saving files. The functions deal with the basic task of saving data structures to disk, and loading them from disk. Add the definitions of the methods to `frame.dylan`, and the definitions of the functions to `task-list.dylan`.

Each method is invoked as a callback in the definition of the `<task-frame>` class:

- `open-file` This method prompts the user to choose a filename, and then loads that file into the task list manager by calling the function `load-task-list`. It is used as the activate callback for both `open-button` (on the application tool bar) and `open-menu-button` (in the *File* menu of the application).
- `save-file` This method saves the task list currently loaded into the application to disk. It is used as the activate callback for both `save-button` (on the application tool bar) and `save-menu-button` (in the *File* menu of the application).

- `save-as-file` This method saves the task list currently loaded into the application to disk, and prompts the user to supply a name. It is used as the activate callback for `save-as-menu-button` (in the *File* menu of the application).

The following functions are called by the methods described above:

- `save-task-list` This function saves an instance of `<task-list>` to a named file. It is called by `save-as-file`.
- `load-task-list` This function takes the contents of a file on disk and converts it into an instance of `<task-list>`. It is called by `open-as-file`.

The following sections present and explain the code for each of these methods and functions in turn.

The open-file method

The code for open-file is shown below. Add this code to `frame.dylan`.

```
define method open-file
    (gadget :: <gadget>) => ()
  let frame = sheet-frame(gadget);
  let task-list = frame-task-list(frame);
  let filename
    = choose-file(frame: frame,
                  default: task-list.task-list-filename,
                  direction: #"input");
  if (filename)
    let task-list = load-task-list(filename);
    if (task-list)
      frame.frame-task-list := task-list;
      refresh-task-frame(frame)
    else
      notify-user(format-to-string("Failed to open file %s", filename),
                  owner: frame)
    end
  end
end method open-file;
```

The method takes a gadget as an argument and returns no values. The argument is the gadget that is used to invoke it, which in the case of the task list manager means either `open-menu-button` (in the *File* menu of the application) or `open-button` (on the tool bar). The `open-file` method then sets three local variables:

- `frame` This contains the frame of which the gadget argument is a part. This is a simple way of identifying the main application frame.
- `task-list` This contains the value of the `frame-task-list` slot for frame. This identifies the instance of `<task-list>` that is being used to hold the task list information currently loaded into the task list manager.
- `filename` This is the name of the file that is to be loaded into the task list manager, and the user is always prompted to supply it.

The method `choose-file` (a method provided by DUIM) is used to prompt for a file to load. The portion of code that performs this task is repeated here:

```
choose-file(frame: frame,
            default: task-list.task-list-filename,
            direction: #"input");
```

This method displays a standard file dialog box so that the user can select a file on any disk connected to the host computer. For `open-file`, you need to supply three arguments to `choose-file` : the frame that owns the dialog, a default value to supply to the user, and the direction of the interaction.

You need to supply a frame so that the system knows how to treat the frame correctly, with respect to the dialog box. Thus, while the dialog is displayed, the frame that owns it cannot be minimized, resized, or interacted with in any way; this is standard behavior for modal dialog boxes.

In this case, supplying a default value is useful in that it lets us supply the filename for the currently loaded task list as a default value. It determines this by examining the `task-list-filename` slot of `task-list` (which, remember, is defined as a local variable and represents the instance of `<task-list>` in use). If this slot has a value, then it is offered as a default. (Note that if the currently loaded task list has never been saved to disk, then this slot is `#f`, and so no default is offered.)

The direction of interaction should also be specified when calling `choose-file`, since the same generic function can be used to prompt for a filename using a standard Open File dialog or a standard Save File dialog. In this case, the direction is `#"input"`, which indicates that data is being read in (that is, Open File is used).

The rest of the `open-file` method deals with loading in the task list information safely. It consists of two nested `if` statements as shown below.

```
if (filename)
  let task-list = load-task-list(filename);
  if (task-list)
    frame.frame-task-list := task-list;
    refresh-task-frame(frame)
  else
    notify-user(format-to-string("Failed to open file %s", filename),
                owner: frame)
  end
end
```

The clause

```
if (filename)
  ...
end
```

is necessary to handle the case where the user cancels the Open file dialog: on cancelling the dialog, the `open-file` method should return silently with no side effects.

If a filename is supplied, then it is read from disk and converted into a format that is readable by the application, in the line that reads

```
let task-list = load-task-list(filename);
```

The function `load-task-list` is described in *The load-task-list function*.

The clause

```
if (task-list)
  ...
else
  ...
end
```

is necessary to handle the case where the filename specified does not contain data that can be interpreted by `load-task-list`. If `task-list` cannot be assigned, then the `else` code is run. This calls the function `notify-user`, which is a simple way to display a short message to the user in a message box.

If `task-list` can be assigned (that is, the contents of the specified file have been successfully read by `load-task-list`), then two lines of code are run. The line

```
frame.frame-task-list := task-list;
```

assigns the `frame-task-list` slot of frame to the value of `task-list`.

The line

```
refresh-task-frame(frame)
```

calls a method that refreshes the list of tasks displayed in the task list manager, so that the contents of the newly loaded file are correctly displayed on the screen. The method `refresh-task-frame` is described in *Updating the user interface*.

The save-file method

The code for `save-file` is as follows:

```
define method save-file
    (gadget :: <gadget>) => ()
  let frame = sheet-frame(gadget);
  let task-list = frame-task-list(frame);
  save-as-file(gadget, filename: task-list.task-list-filename)
end method save-file;
```

Add this code to `frame.dylan`.

This method is very simple, in that it just calls the method `save-as-file`, passing it a filename as an argument. The `save-as-file` method then does the real work of updating the GUI and calling the relevant code to save information to disk.

Just like the `open-file` method, `save-file` takes the gadget used to invoke it as an argument and returns no values. In the case of the task list manager the gadget is either `open-menu-button` (in the *File* menu of the application) or `open-button` (on the tool bar). The `save-file` method sets the following two local variables:

- `frame` The frame of which the gadget argument is a part, so that the main application frame can be identified.
- `task-list` This contains the value of the `frame-task-list` slot for `frame`. This identifies the instance of `<task-list>` that needs to be saved to disk.

Note that similar local variables are used in the definition of `open-file`.

The `save-file` method then calls `save-as-file`, passing it the following two arguments:

- The gadget that invoked `save-file`.
- The filename associated with the instance of `<task-list>` that needs to be saved to disk.

Notice that the second of these arguments may be `#f`, if the task list has not previously been saved to disk.

The save-as-file method

The code for `save-as-file` is as follows:

```
define method save-as-file
    (gadget :: <gadget>, #key filename) => ()
  let frame = sheet-frame(gadget);
  let task-list = frame-task-list(frame);
  let filename
    = filename
      | choose-file(frame: frame,
```

```
                    default: task-list.task-list-filename,
                    direction: #"output");
  if (filename)
    if (save-task-list(task-list, filename: filename))
      frame.frame-task-list := task-list;
      refresh-task-frame(frame)
    else
      notify-user(format-to-string
                    ("Failed to save file %s", filename),
                  owner: frame)
    end
  end
end method save-as-file;
```

Add this code to `frame.dylan`.

Like `open-file` and `save-file`, this method takes a gadget as an argument and returns no values. This argument is the gadget which is used to invoke it. In addition, an optional keyword argument, a filename, can be passed.

This method is a little unusual; as well as being the activate callback for the `save-as-menu-button` (the command *File* > *Save As*), it is also called by the `save-file` method.

- When directly invoked as an activate callback, the `filename` argument is not passed to `save-as-file`. Instead, the user is prompted to supply it. In addition, the `gadget` is `save-as-menu-button`.
- When invoked by `save-file`, a `filename` may be passed, if the associated task list has been saved before. In addition, the gadget is either `save-button` or `save-menu-button`.

As with `open-file`, `save-as-file` sets three local variables:

- `frame` This is the frame containing the gadget passed as an argument.
- `task-list` This contains the value of the `frame-task-list` slot for `frame`, and identifies the instance of `<task-list>` to be saved.
- `filename` The filename to which the instance of `<task-list>` is saved.

Unless `filename` is passed as an optional argument, the user is prompted to supply a filename in which the task list information is to be saved. As with `open-file`, the `choose-file` method is used to do this. In fact, the call to `choose-file` here is identical to the call to `choose-file` in `open-file`, with the exception of the direction argument, which is set to `#"output"`.

The rest of the `save-as-file` method deals with saving the task list information safely. It is similar to the equivalent code in `open-file`, and consists of two nested `if` statements as shown below.

```
if (filename)
  if (save-task-list(task-list, filename: filename))
    frame.frame-task-list := task-list;
    refresh-task-frame(frame)
  else
    notify-user(format-to-string("Failed to save file %s", filename),
                owner: frame)
  end
end
```

As with `open-file`, the clause

```
if (filename)
  ...
end
```

is necessary in case the user cancels the Save file dialog: on cancelling the dialog, `save-as-file` should fail silently with no side effects.

The second `if` statement is more interesting. The body of the `if` statement is like the body of the equivalent `if` statement in `open-file` :

```
frame.frame-task-list := task-list;
refresh-task-frame(frame)
```

This sets the `frame-task-list` slot of `frame` and then calls `refresh-task-frame` to ensure that the correct information is shown on the screen.

Similarly, the body of the `else` clause warns that the task list could not be saved, when the `if` condition does not return true:

```
notify-user(format-to-string("Failed to save file %s", filename),
            owner: frame)
```

The interesting part of this `if` statement is the `if` condition itself:

```
save-task-list(task-list, filename: filename)
```

As well as providing a test for whether the task list frame should be updated, it actually performs the save operation, by calling the function `save-task-list` with the required arguments.

The function `save-task-list` is described in *The save-task-list function* and the method `refresh-task-frame` is described in *Updating the user interface*.

The load-task-list function

The code for `load-task-list` is shown below. Because this function does not use any DUIM code, it is described only briefly.

```
define function load-task-list
    (filename :: <string>) => (task-list :: false-or(<task-list>))
  let tasks = make(<stretchy-vector>);
  block (return)
    with-open-file (stream = filename, direction: #"input")
      while (#t)
        let name = read-line(stream, on-end-of-stream: #f);
        unless (name) return() end;
        let priority = read-line(stream, on-end-of-stream: #f);
        unless (priority)
          error("Unexpectedly missing priority!")
        end;
        let task = make(<task>, name: name,
                        priority: as(<symbol>, priority));
        add!(tasks, task)
      end
    end
  end;
  make(<task-list>, tasks: tasks, filename: filename)
end function load-task-list;
```

Add this code to `task-list.dylan`.

The function `load-task-list` reads a file from disk and attempts to convert its contents into an instance of `<task-list>`, which itself contains any number of instances of `<task>`. It takes one argument, the filename, and returns one value, the instance of `<task-list>`.

This function uses a generic function and a macro from the Streams library to read information from the file. For full information about this library, please refer to the *I/O and Networks Library Reference*.

The file format used by the task list manager is very simple, with each element of a task occupying a single line in the file. Suppose `load-task-list` is called on a file containing the following information:

Wash the dog

medium

Video Men Behaving Badly

high

This would create an instance of `<task-list>` whose `task-list-tasks` slot was a sequence of two instances of `<task>`.

- The first `<task>` would have a `task-name` of *"Wash the dog"* and a `task-priority` of `#"medium"`.
- The second `<task>` would have a `task-name` of *"Video Men Behaving Badly"* and a `task-priority` of `#"high"`.

The `task-list-filename` slot of the `<task-list>` is the filename itself. Note that the `task-list-modified?` slot of the `<task-list>` is set to `#f`, reflecting the fact that the task list is loaded, but unchanged. This does not have to be done explicitly by `load-task-list`, since `#f` is the default value of this slot, as you can see from its definition in *Defining the underlying data structures for tasks*.

The file is opened for reading using the `with-open-file` macro. It is then read a line at a time, setting the local variables `name` and `priority` with each alternate line. After successfully setting both `name` and `priority`, an instance of `<task>` is created, and added to the stretchy vector tasks using `add!`. When the end of the file is reached, `#f` is returned and an instance of `<task-list>` is created from `tasks` and returned by the function.

Note how the `as` method is used to convert a string value such as `"medium"` into a symbol such as `#"medium"`. This is a useful technique to use when you wish to save and load symbol information in an application.

The save-task-list function

The code for `save-task-list` is shown below. Because this function does not use any DUIM code, it is described only briefly.

```
define function save-task-list
    (task-list :: <task-list>, #key filename)
=> (saved? :: <boolean>)
  let filename = filename | task-list-filename(task-list);
  with-open-file (stream = filename, direction: #"output")
    for (task in task-list.task-list-tasks)
      format(stream, "%s\\n%s\\n",
             task.task-name, as(<string>, task.task-priority))
    end
  end;
  task-list.task-list-modified? := #f;
  task-list.task-list-filename := filename;
  #t
end function save-task-list;
```

Add this code to `task-list.dylan`.

The function `save-task-list` takes an instance of `<task-list>` as an argument, and optionally a `filename`. It then attempts to save the instance of `<task-list>` to the file specified by `filename`. It returns a boolean value that indicates whether the file was successfully saved or not. If filename is not passed as an argument to `save-task-list` (in the case where the user has chosen *File* > *Save* or clicked the *Save* button when working

with a task list file that has previously been saved), then the `task-list-filename` slot of the `<task-list>` is used instead.

Like `load-task-list`, this function uses the Streams library to save information to a file. For full information about this library, please refer to the *I/O and Networks Library Reference*. It also uses the `format` function from the Format library, which is described in the same reference.

The file is opened for saving using the `with-open-file` macro (just like `load-task-list`, but in the opposite direction), A `for` loop is used to save each element in each task to the file. The `format` function then writes each element to the file, separated by a newline character. Note how the `as` method is used to convert the `task-priority` symbol to a string when saving each priority value: this is the reverse situation to `load-task-list`, where a method for `as` was used to convert the string to a symbol.

Once every element in the file has been saved, the `task-list-modified` slot of the `<task-list>` is reset to `#f`, and the `task-list-filename` slot of the `<task-list>` is set to the filename used by `save-task-list`. This last step is necessary to allow for the case where the user has chosen the *File* > *Save As* command to save the file under a different name.

Finally, `save-task-list` returns `#t` to indicate that the file has been successfully saved.

Adding and removing tasks from the task list

This section describes the functions and methods necessary for adding to the task list and removing tasks from the task list. A total of two methods and two functions are necessary.

`frame-add-task`

This prompts the user for the details of a new task and adds it to the list.

`frame-remove-task`

This removes the currently selected task from the list, prompting the user before removing it completely.

`add-task`

This adds an instance of `<task>` to an instance of `<task-list>`.

`remove-task`

This removes an instance of `<task>` from an instance of `<task-list>`.

As with the file handling code, DUIM code and non-DUIM code has been separated. The methods beginning with `frame-` deal with the GUI-related issues of adding and removing tasks, and the functions deal with the underlying data structures.

Add the definitions of the methods to `frame.dylan`, and the definitions of the functions to `task-list.dylan`.

DUIM support for adding and removing tasks

This section describes the methods necessary to provide support in the task list manager GUI for adding and removing tasks.

Add the code described in this section to `frame.dylan`.

The code for `frame-add-task` is as follows:

```
define method frame-add-task (gadget :: <gadget>) => ()
  let frame = sheet-frame(gadget);
  let task-list = frame-task-list(frame);
  let (name, priority) = prompt-for-task(owner: frame);
  if (name & priority)
```

```
    let new-task = make(<task>, name: name, priority: priority);
    add-task(task-list, new-task);
    refresh-task-frame(frame);
    frame-selected-task(frame) := new-task
  end
end method frame-add-task;
```

The method takes a gadget as an argument and returns no values. The argument is the gadget which is used to invoke it, which in the case of the task list manager means either `add-menu-button` (in the *Task* menu of the application) or `add-button` (on the tool bar). The `frame-add-task` method then sets a number of local variables:

- `frame` The frame containing the gadget passed as an argument.
- `task-list` The value of the `frame-task-list` slot for `frame`. This identifies the instance of `<task-list>` to which a task is to be added.
- `name` The text of the task to be added.
- `priority` The priority of the task to be added.

As with other DUIM methods you have seen, `frame` and `task-list` are specified using known slot values about the gadget supplied to `frame-add-task`, and the frame that contains the gadget. The `name` and `priority` values are specified by calling the `prompt-for-task` method defined in Creating a dialog for adding new items. This method displays a dialog into which the user types the text for the new task and chooses the priority, both of which values are returned from `prompt-for-task`.

Once all the local variables have been specified, the main body of code for the method, repeated below, is executed.

```
if (name & priority)
  let new-task = make(<task>, name: name, priority: priority);
  add-task(task-list, new-task);
  refresh-task-frame(frame);
  frame-selected-task(frame) := new-task
end
```

This consists of four expressions around which is wrapped an `if` statement.

1. The first expression creates a new task from the values of the `name` and `priority` local variables.
2. The second expression adds the new task to task list, by calling the `add-task` function.
3. The third expression refreshes the display of the task list in the task list manager, so that the new task is displayed on the screen once it has been added.
4. The fourth expression ensures that the new task is selected in the task list manager. The frame-selected–task method is described in *Updating the user interface*.

The `if` statement ensures that all the information needed to construct the new task is specified before the new task is created.

The `add-task` function is described in *Non-DUIM support for adding and removing tasks*.

The code for `frame-remove-task` is as follows:

```
define method frame-remove-task (gadget :: <gadget>) => ()
  let frame = sheet-frame(gadget);
  let task = frame-selected-task(frame);
  let task-list = frame-task-list(frame);
  if (notify-user(format-to-string
                    ("Really remove task %s", task.task-name),
      owner: frame, style: #"question"))
    frame-selected-task(frame) := #f;
    remove-task(task-list, task);
```

```
    refresh-task-frame(frame)
  end
end method frame-remove-task;
```

As with `frame-add-task`, this method takes the gadget that is used to invoke it as an argument and returns no values. In the case of the task list manager, the gadget is either `remove-menu-button` (in the *Task* menu of the application) or `remove-button` (on the tool bar). The `frame-remove-task` method then sets a number of local variables:

- `frame` The frame containing the gadget passed as an argument.
- `task` The task that is to be removed. The task to be removed is the one selected in the list of tasks on screen. The method `frame-selected-task` is called to determine which task this is.
- `task-list` The value of the `frame-task-list` slot for `frame`. This identifies the instance of `<task-list>` from which a task is to be removed.

The method `frame-selected-task` is described in *Updating the user interface*.

Once these local variables have been set, the rest of the code goes about removing the task. The code consists of three expressions around which is wrapped an `if` statement, as shown below.

```
if (notify-user(format-to-string
                  ("Really remove task %s", task.task-name),
    owner: frame, style: #"question"))
  frame-selected-task(frame) := #f;
  remove-task(task-list, task);
  refresh-task-frame(frame)
end
```

Notice here that the method `notify-user` is used as the condition in the `if` statement: if the call to `notify-user` returns `#t`, then the subsequent expressions are executed. This use of `notify-user` illustrates how you can use the method to generate a yes-no question for the user to answer, by using the `style:` init-keyword. You might like to compare the user of `notify-user` in this method with its use in `open-file` or `save-as-file` ; essentially, the only difference is in the use of the `style:` init-keyword.

If the call to `notify-user` returns `#t`, then three expressions are executed:

1. The first calls the setter for `frame-selected-task`, to ensure that no items in the task list are selected.
2. The second calls the function `remove-task`, which removes task from `task-list`.
3. Then, `refresh-task-frame` is called to ensure that the task that has been removed is no longer displayed in the list of tasks on the screen.

The methods defined for `frame-selected-task` are described in *Updating the user interface*. The function `remove-task` is described in *Non-DUIM support for adding and removing tasks*. The `refresh-task-frame` method is described in *Updating the user interface*.

Non-DUIM support for adding and removing tasks

This section describes the functions necessary for adding an instance of `<task>` to a `<task-list>`, and removing a `<task>` from a `<task-list>`. These functions are called by the callback functions `frame-add-task` and `frame-remove-task`, respectively. Because these functions do not use any DUIM code, they are described only briefly.

Add the code described in this section to `task-list.dylan`.

The code for `add-task` is as follows:

```
define function add-task
    (task-list :: <task-list>, task :: <task>) => ()
  add!(task-list.task-list-tasks, task);
  task-list.task-list-modified? := #t
end function add-task;
```

This function takes two arguments, a <task-list> and the <task> that is to be added to it, and returns no values. The add-task function first adds the <task> to the end of the sequence bound to the task-list-tasks slot of the <task-list>, and then sets the task-list-modified? slot of the <task-list> to #t, to indicate that a change in the <task-list> has occurred.

The code for remove-task is as follows:

```
define function remove-task
    (task-list :: <task-list>, task :: <task>) => ()
  remove!(task-list.task-list-tasks, task);
  task-list.task-list-modified? := #t
end function remove-task;
```

This function is analogous to add-task. It takes the same arguments, and returns no values. The function first removes the <task> from the task-list-tasks slot of the <task-list>, and then sets the task-list-modified? slot of the <task-list> to #t, to indicate that a change in the <task-list> has occurred.

Updating the user interface

This section describes a number of miscellaneous methods that are required for smooth operation of the task list manager. Each of the methods defined here ensures that the task list manager displays the correct information and gives the user access to appropriate commands in any given situation. Here is a list of the methods defined in this section, together with a brief description of each one:

- initialize An initialize method is provided for <task-frame> that ensures information is displayed correctly when the task list manager is first displayed. This method is described in *Initializing a new instance of <task-frame>*.

frame-selected-task

This method returns the task that is currently selected in the task list manager. This method is described in *Determining and setting the selected task*.

frame-selected-task-setter

This is a setter method for frame-selected-task, and is used to select or deselect item in the task list manager. This method is described in *Determining and setting the selected task*.

note-task-selection-change

Two methods are defined that deal with updating the GUI whenever a change is made to the task selection state. This method is described in *Enabling and disabling buttons in the interface*.

refresh-task-frame

This method can be called to refresh the task frame at any time. This method is described in Refreshing the list of tasks.

Each of these methods should be added to the file frame.dylan.

Initializing a new instance of <task-frame>

The code below provides an `initialize` method for the class `<task-frame>`. This simply ensures that the display in a `<task-frame>` is refreshed as soon as it is created, and calls any subsequent methods that may be defined for it (although, in the case of the task list manager, there are none). While not strictly necessary, this `initialize` method illustrates general good practice when defining your own classes of frame. If the application was associated with files of a particular type on disk, then the `initialize` method would be necessary to ensure that tasks were displayed correctly after starting the task list manager by double-clicking on a file of tasks.

```
define method initialize
    (frame :: <task-frame>, #key) => ()
  next-method();
  refresh-task-frame(frame);
end method initialize;
```

Add the code for this method to `frame.dylan`.

Determining and setting the selected task

Two methods are used to determine which task is selected in the task list manager, and to set a specific task in the task list manager: `frame-selected-task` and `frame-selected-task-setter`.

The `frame-selected-task` method returns the task that is currently selected in the task list manager, or `#f` if no task is selected. This method is used by `frame-remove-task` to determine which task should be deleted from the task list. It is also used by `note-task-selection-change` to determine whether or not a task is selected.

```
define method frame-selected-task
    (frame :: <task-frame>) => (task :: false-or(<task>))
  let list-box = task-list(frame);
  gadget-value(list-box)
end method frame-selected-task;
```

The `frame-selected-task` method works by determining the `gadget-value` of the list box that displays the tasks in the task list manager. The `gadget-value` of a collection such as a list box is the selected item. Notice how you can access the value of a pane in a frame instance in exactly the same way that you can access the value of a slot in a class instance; the definition of the pane creates an accessor that is just like a slot accessor. Recall that the name of the list box in the definition of the `<task-frame>` class is `task-list`.

A setter method is also defined for `frame-selected-task`, as shown below:

```
define method frame-selected-task-setter
    (task :: false-or(<task>), frame :: <task-frame>)
=> (task :: false-or(<task>))
  let list-box = task-list(frame);
  gadget-value(list-box) := task;
  note-task-selection-change(frame);
  task
end method frame-selected-task-setter;
```

This method takes two arguments: the `task` to select in the task list manager, and the `frame` to which the task belongs. It returns the task. The method determines the list box used to display tasks in `frame`, and then sets the `gadget-value` of that list box to `task`. Finally, it calls `note-task-selection-change`, described below, to update other parts of the user interface appropriately, such as buttons on the tool bar.

As with most setter methods, `frame-selected-task-setter` is not called directly. Instead, it is called implicitly by setting a value using `frame-selected-task`. For example,

```
frame-selected-task(frame) := #f;
```

ensures that no tasks are selected in `frame`.

The `frame-selected-task-setter` method is called by two other methods: `frame-add-task` (to ensure that the task added is subsequently selected) and `frame-remove-task` (to ensure that no tasks are selected once a task has been removed from the list). These methods are described in *DUIM support for adding and removing tasks*.

Add the code for these methods to `frame.dylan`.

Enabling and disabling buttons in the interface

The two methods for `note-task-selection-change` make a number of changes to the GUI of the task list manager, to ensure that the correct information is displayed to the user. In particular, they perform any changes necessary after an item in the task list has been selected or deselected. They ensure that the correct priority is displayed in the radio box, depending on whether there is a task currently selected, and they also enable or disable the *Remove task* button and its equivalent command in the *Task* menu, depending on whether there is a task selected or not (if there is no task selected, then the button and menu command should both be disabled).

There are two methods defined, one on an instance of `<task-frame>`, and one on an instance of `<gadget>`. The Task List 1 project requires both of these methods. For the Task List 2 project, however, the first method requires a slightly different definition, and the second method is not required at all.

The `note-task-selection-change` method defined on `<task-frame>` is called by `refresh-task-frame`, described on Refreshing the list of tasks. The refresh-task-frame method is called whenever the list of tasks needs to be refreshed for whatever reason. This happens most commonly after adding or removing a task from the list, or loading in a new task list from a file on disk. The method `refresh-task-frame` takes an instance of `<task-frame>` as an argument and returns no values. For the Task List 1 project, the `note-task-selection-change` method is defined:

```
define method note-task-selection-change
    (frame :: <task-frame>) => ()
  let task = frame-selected-task(frame);
  if (task)
    frame.priority-box.gadget-value := task.task-priority;
  end;
  let selection? = (task ~= #f);
  frame.remove-button.gadget-enabled? := selection?;
  frame.remove-menu-button.gadget-enabled? := selection?;
end method note-task-selection-change;
```

For the Task List 2 project the `note-task-selection-change` method is defined:

```
define method note-task-selection-change
    (frame :: <task-frame>) => ()
  let task = frame-selected-task(frame);
  if (task)
    frame.priority-box.gadget-value := task.task-priority;
  end;
  command-enabled?(frame-remove-task, frame) := task ~= #f;
end method note-task-selection-change;
```

The method takes an instance of `<task-frame>` as an argument, and returns no values. It works by calling `frame-selected-task` to determine which, if any, task is currently selected, and sets that to a local variable, `task`.

The expression

```
if (task)
  frame.priority-box.gadget-value := task.task-priority;
end;
```

sets the gadget value of the priority-box pane in the task list manager to the value of the task-priority slot of the selected task, if a task is selected. This ensures that if a task is selected, its priority is displayed correctly beneath the list of tasks. Note that priority-box may take the same set of values as the task-priority slot, namely #"low", #"medium", and #"high", so it is straightforward to make this kind of assignment.

The rest of the method deals with enabling or disabling gadgets that let the user remove a task from the task list. If there is no task selected, then remove-button and remove-menu-button need to be disabled. If there is a task selected, then they need to be enabled. This behavior is achieved by converting the value of the variable task, which can take a value of false-or(<task>), into a boolean value, called selection?. This is done in the expression

```
let selection? = (task ~= #f);
```

This sets selection? to the result of performing an inequality comparison on task and #f. Thus, if task is #f (there is no task selected), then selection? is #f, but if task is an instance of <task> (there is a task selected), then selection? is #t.

The two calls to gadget-enabled? then set the gadget-enabled slot of the appropriate gadgets to the value of selection?, enabling or disabling each gadget as appropriate.

The second method for note-task-selection-change is defined for an instance of <gadget>, as follows:

```
define method note-task-selection-change
    (gadget :: <gadget>) => ()
  let frame = gadget.sheet-frame;
  note-task-selection-change(frame)
end method note-task-selection-change;
```

This takes a gadget as an argument. It simply finds the frame that the gadget belongs to, and calls the other method for note-task-selection-change on that frame.

The second method for note-task-selection-change needs to be used as the value-changed callback of the task-list pane in the definition of <task-frame> ; a value-changed callback is invoked whenever the gadget-value of a gadget changes. Because the gadget-value of a list box is the currently selected item, whenever a different item is selected in the list box, note-task-selection-change is called.

In order to achieve this, a small change is needed to the definition of the task-list pane in frame.dylan. In this definition for the Task List 1 project, change the line that reads:

```
activate-callback: not-yet-implemented);
```

to

```
value-changed-callback: note-task-selection-change);
```

and for the Task List 2 project change the line to

```
value-changed-callback: method (gadget)
  note-task-selection-change(frame) end);
```

to give a final definition for this pane as follows:

```
// definition of list
pane task-list (frame)
  make (<list-box>,
        items: frame.frame-task-list.task-list-tasks,
        label-key: task-name,
```

```
        lines: 15,
        value-changed-callback: note-task-selection-change);
```

Add the code for these methods to `frame.dylan`.

Refreshing the list of tasks

The `refresh-task-frame` method is called whenever the list of tasks needs to be refreshed for whatever reason. This happens most commonly after adding or removing a task from the list, or loading in a new task list from a file on disk. The method `refresh-task-frame` takes an instance of `<task-frame>` as an argument and returns no values. For the Task List 1 project the definition is:

```
define method refresh-task-frame
    (frame :: <task-frame>) => ()
  let list-box = frame.task-list;
  let task-list = frame.frame-task-list;
  let modified? = task-list.task-list-modified?;
  let tasks = task-list.task-list-tasks;
  if (gadget-items(list-box) == tasks)
    update-gadget(list-box)
  else
    gadget-items(list-box) := tasks
  end;
  gadget-enabled?(frame.save-button) := modified?;
  gadget-enabled?(frame.save-menu-button) := modified?;
  note-task-selection-change(frame);
end method refresh-task-frame;
```

However, the Task List 2 project requires a call to `command-enabled?`, so the definition is:

```
define method refresh-task-frame
    (frame :: <task-frame>) => ()
  let list-box = frame.task-list;
  let task-list = frame.frame-task-list;
  let modified? = task-list.task-list-modified?;
  let tasks = task-list.task-list-tasks;
  if (gadget-items(list-box) == tasks)
    update-gadget(list-box)
  else
    gadget-items(list-box) := tasks
  end;
  command-enabled?(save-file, frame) := modified?;
  note-task-selection-change(frame);
end method refresh-task-frame;
```

To begin, `refresh-task-frame` sets a number of local variables:

- `list-box` The list box used to display the list of tasks in task list manager.
- `task-list` The task list currently loaded in the task list manager.
- `modified?` The value of the `task-list-modified?` slot of `task-list`.
- `tasks` The sequence of tasks stored in `task-list`.

Next, the following code is executed:

```
if (gadget-items(list-box) == tasks)
  update-gadget(list-box)
```

```
else
  gadget-items(list-box) := tasks
end;
```

This code ensures that if the items in the list box are the same as the sequence of tasks in the task list, then the display in the list box is updated to ensure all the items are displayed correctly. If the items in the list box are not the same as the sequence of tasks, then the items in the list box are updated to reflect the current task list. The items in the list box could be different if a task had been added or removed from the list, or if a completely new set of tasks had been loaded into the task list manager.

Lastly, the following three lines

```
gadget-enabled?(frame.save-button) := modified?;
gadget-enabled?(frame.save-menu-button) := modified?;
note-task-selection-change(frame);
```

ensure that the *Save* button and *File* > *Save* menu command are enabled if the task list has been modified, and then any changes that need to be made to the GUI as a result of changing the selected item are performed, by calling `note-task-selection-change`.

Add the code for this method to `frame.dylan`.

Creating an information dialog

The following function displays a simple dialog box that provides information about the application. This dialog is displayed when you choose the *Help* > *About* menu command.

```
define function about-task (gadget :: <gadget>) => ()
  notify-user("Task List Manager", owner: sheet-frame(gadget))
end function about-task;
```

Exiting the task list manager

The `exit-task` method allows you to exit the task list manager. It is invoked by choosing *File* > *Exit*. The definition of this method is quite simple.

```
define method exit-task (gadget :: <gadget>) => ()
  let frame = sheet-frame(gadget);
  let task-list = frame-task-list(frame);
  save-file (gadget);
  exit-frame(frame)
end method exit-task;
```

Add this method to the file `frame.dylan`.

The method takes the gadget used to invoke it and returns no values. In this case, `exit-task` is only ever invoked by the `exit-menu-button` gadget.

As with many other callbacks in this example, `exit-task` sets a number of local variables:

- `frame` The frame that the gadget argument belongs to.
- `task-list` The task list associated with `frame`.

The method begins by calling the `save-file` method (defined in *The save-file method*) to save the current task list to disk. This ensures that the user does not lose any work. Next, the `exit-frame` generic function is invoked to exit the task list manager window.

Enhancing the task list manager

This concludes the tutorial on building application with DUIM. At this point, you can build and run a functional task list manager, but it is a very basic application. Using Command Tables introduces command tables as a way of defining hierarchies of menu commands. To do this, it re-implements the menu hierarchy you defined in Adding Menus To The Application, but does not add any new functionality to the application.

There are many ways that the task list manager could be extended, and you might like to try experimenting with the code. To begin with, very little error checking has been written into the application, and you might like to add some in order to make the task list manager more robust. For example, it is currently possible to exit the task list manager and lose any changes in an unsaved list of tasks.

In addition to error checking, there is a wide range of new functionality you might like to add. A few ideas are listed below:

- Re-implement the list box and radio box in the main window of the task list manager as a table control, so that the priority of each task is displayed next to the text for the task.
- Implement the facility to define categories, so that tasks could be assigned categories such as "Home" and "Business". Categories could be listed in the table control alongside priorities.
- Allow sorting the list of tasks according to a key. Tasks could then be sorted by priority or category.
- Implement the ability to mark tasks as complete.
- Allow users to add text memos to any task.

This is only a very limited list of ideas. After learning about command tables in Using Command Tables, read through A Tour of the DUIM Libraries to learn more about the features that DUIM provides. Then, using the *DUIM Reference Manual* as your reference source, get coding!

CHAPTER

EIGHT

USING COMMAND TABLES

Introduction

Another way that you can define a set of menus is by defining a *command table*. A command table lets you create the complete set of commands for an application in a more compact and reusable way than the standard menus you have seen so far. As well as making the definition of each command in a menu shorter and easier to code, it lets you handle effects such as the disabling of menu commands more elegantly, by removing the need to use `gadget-enabled`. You can include a command table in the definition of a frame in the same way that you can include a tool bar, or a status bar, and because of this, and the fact that you can include command tables within other command tables, it is easy to reuse the same command table across different frames in your application.

Command tables are best used in the following situations:

- If your menu commands do not use check or radio buttons.
- If the menu bar in your application is not context sensitive (that is, the available commands on the menu remain consistent as the application state changes).

In other cases, you should define your menu hierarchy by defining panes that combine specific gadgets, as demonstrated in Adding Menus To The Application. Using a combination of command tables and standard menu definitions in a GUI design is not recommended.

The task list manager application does not use check or radio buttons in any of its menu commands, and the menu bar is not context sensitive. This means that, if you wish, you can define the commands in the task list manager using command tables, rather than standard menus.

This chapter provides an introduction to command tables by showing you how to re-implement the menu system of the task list manager as a set of command tables. It does not provide a complete copy of all the source code necessary to implement the task list manager. For a complete copy of the code, please refer to Source Code For The Task List Manager. To load the code into the environment, choose **Tools > Open Example Project** from any window in the environment, and load the Task List 2 project from the Documentation category of the Open Example Project dialog.

Note: Please note that this project, like the Task List 1 project, is called `task-list` within the source code, and you should not load them both into the environment at the same time.

Implementing a command table

You use `define command-table` to define a new command table. Consider the following command table defined for the *File* menu in the task list manager:

```
define command-table *file-command-table* (*global-command-table*)
  menu-item "Open" = open-file,
    accelerator: make-keyboard-gesture(#"o", #"control"),
    documentation: "Opens an existing file.";
  menu-item "Save" = save-file,
    accelerator: make-keyboard-gesture(#"s", #"control"),
    documentation: "Saves the current file to disk.";
  menu-item "Save As..." = save-as-file,
    documentation: "Saves the current file with a new name.";
  menu-item "Exit" = exit-task,
    accelerator: make-keyboard-gesture(#"f4", #"alt"),
    documentation: "Exits the application.";
end command-table *file-command-table*;
```

This defines a command table, called `*file-command-table*`, that contains all the menu commands required in the *File* menu of the task list manager. It replaces the definition of each menu button, as well as the definition of the *File* menu itself, in the original implementation of the task list manager application that was given in Adding Menus To The Application. As you can see, this definition is considerably shorter than the individual definitions of the menu and menu buttons previously required,

When defining a command table, you should provide a list of other command tables from which the command table you are defining inherits. This is done in the clause

```
define command-table *file-command-table* (*global-command-table*)
```

above. This is analogous to the way that the superclasses of any frame class are listed in the frame's definition.

Any items defined by the command tables which are to be inherited are automatically added to the command table being defined.

In the example above, `*file-command-table*` inherits from only one command table: `*global-command-table*`. This is defined globally for the whole Dylan environment, and every command table that does not explicitly inherit from other command tables must inherit from this command table.

Each menu item is introduced using the `menu-item` option, and a command is specified for each menu item immediately after the = sign. Each command is just the activate callback that was defined for the equivalent menu button gadget in Adding Callbacks to the Application.

Notice that you can use the `accelerator:` and `documentation:` init-keywords to specify a keyboard accelerator and a documentation string for each menu item in the command table, just like you can when you define each menu button in a menu using a specific gadget. In the same way, you can specify the value of any init-keyword that can be specified for an instance of `<menu-button>`.

Re-implementing the menus of the task list manager

The code below provides definitions for the entire menu hierarchy of the task list manager, using the same activate callbacks that are described and implemented in Adding Callbacks to the Application. Note that the labels, documentation strings, and keyboard accelerators for each menu item are identical to the ones used in the original implementation of the task list manager. For completeness, the definition of `*file-command-table*`, described in Using Command Tables, is repeated below.

```
define command-table *file-command-table* (*global-command-table*)
  menu-item "Open" = open-file,
    accelerator: make-keyboard-gesture(#"o", #"control"),
    documentation: "Opens an existing file.";
  menu-item "Save" = save-file,
    accelerator: make-keyboard-gesture(#"s", #"control"),
```

```
    documentation: "Saves the current file to disk.";
  menu-item "Save As..." = save-as-file,
    documentation: "Saves the current file with a new name.";
  menu-item "Exit" = exit-task,
    accelerator: make-keyboard-gesture(#"f4", #"alt"),
    documentation: "Exits the application.";
end command-table *file-command-table*;

define command-table *edit-command-table* (*global-command-table*)
  menu-item "Cut" = cut-command,
    accelerator: make-keyboard-gesture(#"x", #"control"),
    documentation: "Cut the selection to the clipboard.";
  menu-item "Copy" = copy-command,
    accelerator: make-keyboard-gesture(#"c", #"control"),
    documentation: "Copy the selection to the clipboard.";
  menu-item "Paste" = paste-command,
    accelerator: make-keyboard-gesture(#"v", #"control"),
    documentation: "Paste the selection in the clipboard at the current position.";
end command-table *edit-command-table*;

define command-table *task-command-table* (*global-command-table*)
  menu-item "Add..." = frame-add-task,
    accelerator: make-keyboard-gesture(#"a", #"control", #"shift"),
    documentation: "Add a new task.";
  menu-item "Remove" = frame-remove-task,
    accelerator: make-keyboard-gesture(#"d", #"control", #"shift"),
    documentation: "Remove the selected task from the list.";
end command-table *task-command-table*;

define command-table *help-command-table* (*global-command-table*)
  menu-item "About" = about-task,
    accelerator: make-keyboard-gesture(#"f1"),
    documentation: "Display information about the application.";
end command-table *help-command-table*;
```

The definitions above can be used in place of the definition of each menu and menu button in the original implementation of the task list manager. You must place the command table definitions provided above after the callback definitions themselves, to avoid forward references.

Including command tables in frame definitions

In the previous section, you defined four command tables: one for each menu in the task list manager. Next, you need to combine these command tables and include them in the definition of the `<task-frame>`. The way to do this is to define an additional command table which has each of the other command tables as its components, and then supply this command table as an option in the definition of `<task-frame>`.

```
define command-table *task-list-command-table* (*global-command-table*)
  menu-item "File" = *file-command-table*;
  menu-item "Edit" = *edit-command-table*;
  menu-item "Task" = *task-command-table*;
  menu-item "Help" = *help-command-table*;
end command-table *task-list-command-table*
```

Just like the menu commands in each menu, every menu in the menu bar is defined as a menu item in the definition of the command table.

You can add a command table to the definition of a frame class in much the same way as you add a layout, tool bar, status bar, or menu bar, using the `command-table` option. In the definition of `<task-frame>`, replace the line that reads:

```
menu-bar (frame) frame.task-menu-bar;
```

with

```
command-table (frame) *task-list-command-table*;
```

A complete listing of the implementation of `<task-frame>` using command tables is given in Source Code For The Task List Manager.

Changes required to run Task List 2

In order for the Task List 2 project to run properly, you must modify some of the definitions you constructed in Adding Callbacks to the Application. This section outlines the required changes. For your convenience, the complete source code for both of the Task List projects is provided in Source Code For The Task List Manager.

Changes to button definitions

The definition of each button in the definition of `<task-frame>` needs to be modified compared to their definition in the Task List 1 project, as described in Gluing the new design together.

Broadly speaking, you need to update the `command:` keyword/argument pair for each button gadget, and you need to redefine the activate callback to allow for the fact that the callbacks now take frames as arguments.

Thus, for a button that is defined as:

```
pane add-button (frame)
  make (<push-button>, label: "Add task",
        activate-callback: frame-add-task);
```

the new definition is:

```
pane add-button (frame)
  make(<push-button>, label: "Add task",
       command: frame-add-task,
       activate-callback: method (gadget)
         frame-add-task(frame)
       end);
```

This change must also be made for the definition of radio box, which then becomes:

```
// Definition of radio box
pane priority-box (frame)
  make(<radio-box>,
       items: $priority-items,
       orientation: #"horizontal",
       label-key: first,
       value-key: second,
       value: #"medium",
       command: not-yet-implemented
       activate-callback: method (gadget)
         not-yet-implemented(frame)
       end);
```

For complete definitions, you should refer to the source code available in Appendix A or from the Open Example Project dialog in the environment.

Changes to callback definitions

The following callbacks should be redefined so as to take an instance of <task-frame> as an argument, rather than an instance of <gadget>.

- frame-add-task
- frame-remove-task
- open-file
- save-file
- save-as-file
- about-task
- exit-task

For complete definitions of these callbacks, you should refer to the source code available in Appendix A or from the Open Example Project dialog in the environment.

Changes to method definitions

The definitions for the methods given in Chapter 5 must be redefined so as to take an instance of <frame> as an argument, rather than an instance of <gadget>. This change results in these new definitions:

```
define method open-file
    (frame :: <task-frame>) => ()
  let task-list = frame-task-list(frame);
  let filename
    = choose-file(frame: frame,
                  default: task-list.task-list-filename,
                  direction: #"input");
  if (filename)
    let task-list = load-task-list(filename);
    if (task-list)
      frame.frame-task-list := task-list;
      refresh-task-frame(frame)
    else
      notify-user(format-to-string("Failed to open file %s", filename),
                  owner: frame)
    end
  end
end method open-file;

define method save-file
    (frame :: <task-frame>) => ()
  let task-list = frame-task-list(frame);
  if (task-list.task-list-modified?)
    save-as-file(frame, filename: task-list.task-list-filename)
  end
end method save-file;

define method save-as-file
    (frame :: <task-frame>, #key filename) => ()
```

```
  let task-list = frame-task-list(frame);
  let filename
    = filename | choose-file(frame: frame,
                             default: task-list.task-list-filename,
                             direction: #"output");
  if (filename)
    if (save-task-list(task-list, filename: filename))
      frame.frame-task-list := task-list;
      refresh-task-frame(frame)
    else
      notify-user(format-to-string
                    ("Failed to save file %s", filename),
                  owner: frame)
    end
  end
end method save-as-file;

define method frame-add-task (frame :: <task-frame>) => ()
  let task-list = frame-task-list(frame);
  let (name, priority) = prompt-for-task(owner: frame);
  if (name & priority)
    let new-task = make(<task>, name: name, priority: priority);
    add-task(task-list, new-task);
    refresh-task-frame(frame);
    frame-selected-task(frame) := new-task
  end
end method frame-add-task;

define method frame-remove-task (frame :: <task-frame>) => ()
  let task = frame-selected-task(frame);
  let task-list = frame-task-list(frame);
  if (notify-user(format-to-string
                    ("Really remove task %s", task.task-name),
                  owner: frame, style: #"question"))
    frame-selected-task(frame) := #f;
    remove-task(task-list, task);
    refresh-task-frame(frame)
  end
end method frame-remove-task;

define method note-task-selection-change
    (frame :: <task-frame>) => ()
  let task = frame-selected-task(frame);
  if (task)
    frame.priority-box.gadget-value := task.task-priority;
  end;
  command-enabled?(frame-remove-task, frame) := task ~= #f;
end method note-task-selection-change;
```

For details about `note-task-selection-change`, see Enabling and disabling buttons in the interface. See A task list manager using command tables for the complete source code for the Task List 2 project.

CHAPTER

NINE

A TOUR OF THE DUIM LIBRARIES

Introduction

This chapter provides an overview of the gadgets and functionality that are provided by DUIM. Of necessity, it covers a lot of ground in as short a space as possible, and does not attempt to place any information in the more general context of application development.

To gain an understanding of how different pieces of DUIM functionality can be glued together to create a working application, you should follow the extended example given in this manual in Designing A Simple DUIM Application through Adding Callbacks to the Application. If you need more complete information on any particular aspect of DUIM, you should refer to the DUIM Reference Manual.

The most important DUIM classes are as follows:

- `<frame>` A window in your application.
- `<sheet>` A unique piece of any window.
- `<gadget>` Sheets that are window controls.
- `<layout>` Sheets that control the arrangement of other sheets in the sheet hierarchy.

All of these are subclasses of `<object>`, except `<layout>` which is a subclass of `<sheet>`.

As with any other Dylan class, use `make` to create an instance of a DUIM class.

This chapter introduces you to the most important and useful of all these elements.

- *A tour of gadgets* describes many of the gadgets available in DUIM. A wide variety of different gadgets are available in DUIM, to enable you to create applications that utilize all of the controls for the target operating system.
- *A tour of layouts* describes layouts. These are classes that allow you to group together other sheets hierarchically (typically gadgets and other layouts) in order to put together the elements in any window.
- *A tour of sheets* introduces you to the more general concept of sheets. If you intend defining your own sheet classes (for instance, to design your own controls), then you will need to understand how to handle sheets on a more general level than is needed to use gadgets or layouts.
- *A tour of frames* introduces the different kinds of frame available. There are two basic types of frame: normal windows and dialog boxes. This section also describes how to create your own classes of frame.

You can use the Dylan Playground to run the examples in this chapter. *Reminder:* to interactively run the segments of example code presented in this chapter, you must pass them to `contain` (see Using contain to run examples interactively for details).

A tour of gadgets

The DUIM-Gadgets library provides you with all the controls you can use to create an interface. Objects like buttons, menus, boxes, and other common interface elements are defined as subclasses of the base class `<gadget>`.

General properties of gadgets

Each class of gadget has a set of associated slots that help define the properties for that class. Different classes of gadget have different sets of slots. This section describes some of the more important slots available. The following slots are common across most (though not necessarily all) gadget classes.

`gadget-label` This slot holds the label that is associated with a gadget.

For an item on a menu or a button, for example, this label appears on the gadget itself. For a gadget such as a text field or a border, the label may be displayed next to the gadget.

A label is usually a text string, but can often be an icon, such as is often found on the buttons of an application's toolbar.

If a gadget does not have a label, `gadget-label` returns `#f`.

`gadget-enabled?` This slot specifies whether or not the gadget is active—that is, whether the user of your application can interact with the gadget. All gadgets have a `gadget-enabled?` slot. The `gadget-enabled?` slot returns either `#t` or `#f`. When a gadget is disabled, it is usually grayed out on the screen, and cannot be interacted with in any way.

`gadget-value` This slot holds the value of the gadget. Most gadgets can have a value of some kind; these are general instances of the `<value-gadget>` class. However, gadgets such as borders that are placed around elements have no associated value.

Generally speaking, you can think of the gadget value as a value that the user of your application has assigned to the gadget. The `gadget-value-type` depends on the class of gadget involved. For a text field, the gadget value is the string typed into the text field. For a gadget with several items (see `gadget-items` below), such as a list, the gadget value is the selected item. For a radio button, the gadget value is a boolean that denotes whether the button is selected or not.

If a gadget does not have any values, `gadget-value` returns `#f`.

All of the slots described above can also be specified as init-keyword values when creating an instance of a gadget. In all cases, the init-keyword name is the same as the slot name, but without the preceding "gadget-". Thus, a gadget can be enabled or disabled when it is first created by specifying the `enabled?:` init-keyword appropriately.

Gadgets can also have a variety of associated callbacks. A callback is a function that is invoked when a particular event occurs that is associated with a given gadget, such as pressing a button. It is the primary technique you use to make your applications "do something". Like gadget properties, different classes of gadget can have different callback types available. For an introduction to callbacks, see *Assigning callbacks to gadgets*.

Button gadgets

Broadly speaking, these are gadgets whose value can be changed, or for which some user-defined functionality can be invoked, by clicking on the gadget. Button gadgets encompass obvious controls such as push buttons, radio buttons, and check boxes, and, less obviously, menu items.

Standard buttons

DUIM provides three standard button gadget classes:

- <push-button> Sometimes referred to as *command button* in Microsoft documentation.
- <radio-button> Sometimes referred to as *option button* in Microsoft documentation.
- <check-button> Sometimes referred to as *check box* in Microsoft documentation.

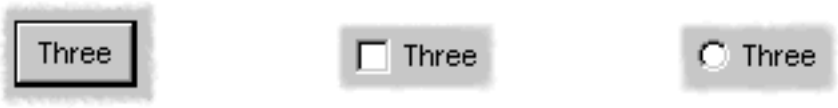

Fig. 9.1: A push button, a radio button, and a check button

The chapters covering the task list manager application (chapters Designing A Simple DUIM Application to Using Command Tables) introduced you to the <push-button> class. This is the default type of button (that is, creating an instance of <button> actually creates an instance of <push-button>).

```
make(<push-button>, label: "Hello");
```

Radio buttons let you choose one option out of a group of several. They are usually implemented in groups of several buttons (using the <radio-box> class), although they can also be created singly, as shown in *A push button, a radio button, and a check button*. For more information about creating groups of radio buttons, see *Button boxes*.

```
make(<radio-button>, label: "Hello");
```

Check buttons are buttons whose setting can be toggled on and off. Like radio buttons, they are often implemented in groups, although unlike radio buttons, they are frequently used individually. For more information about creating groups of check buttons, see *Button boxes*.

```
define variable *my-check-button*
  := make(<check-button>, label: "Hello"
          value: #f);
```

Remember that you can use gadget-label to set or return the label for any button. As demonstrated in the examples above, it is also good practice to set the label when defining any button, using the label: init-keyword.

Radio and check buttons have a gadget-value of #t or #f, depending on whether or not the button is selected. For example:

```
gadget-value(*my-check-button*)
```

returns #f if the check button is not selected.

You can set the gadget-value with the := operator.

```
gadget-value(*my-check-button*) := #t;
```

Supplying a value for a push button is a useful way of sending information to your application. The value of a push button can be used by any callback defined on the push button.

You can make any push button the default option for the frame it is a part of using the default?: init-keyword when defining the button. By default, this is #f, but if specified as #t, the button is displayed on the screen with a heavier border, and any callback defined for the button is invoked by pressing the RETURN key on the keyboard, as well as by clicking the button itself.

```
define variable *my-default-button*
  := make(<push-button>,
          label: "Click me or press Return",
          default?: #t));
```

It is good practice to define a default button in most dialog boxes, so that the user can easily perform a default action. Generally, the *OK* or *Yes* button in a dialog box is the most acceptable default button, though for particularly destructive operations you should consider another choice.

Buttons are intrinsically "non-stretchy" objects. That is, the width of a button is computed from the length of its label, and the button will not automatically size itself according to the size of the sheet that it is a part of. You should use the `max-width:` init-keyword to make a button fill all the available space, by setting it to the constant `$fill`.

Thus, the button created by

```
make(<button>, label: "Red");
```

will only be as wide as the label it is given—"Red", in this case—but the button created by

```
make(<button>, label: "Red", max-width: $fill);
```

will have a width that is determined by the sheet that it is a child of and will still have the same minimum width, so it cannot be resized too small.

Menu buttons

Standard buttons described buttons that are all displayed in windows on the screen. For each of those buttons, there is an analogous type of button that is displayed as an item in a menu.

The `<push-menu-button>` class is used to create a standard menu item. This class is the menu-specific equivalent to `<push-button>`.

Like push buttons, you can make a given push menu button the default command in a menu by specifying the `default?:` init-keyword. The label for a default menu button is highlighted in the menu that it is displayed in, usually by displaying the label using a bold font.

The `<radio-menu-button>` class is used to create a menu item that has the properties of a radio button. The value of a radio menu button may be toggled on and off, just like a radio button, and from any group of radio menu buttons, only one may be on at any one time.

In appearance, a selected radio menu button is usually shown with a small dot to the left of the command name on the menu.

As with radio buttons, radio menu buttons are most useful when used in group form. The class `<radio-menu-box>` is provided for this purpose. See *Menu boxes* for more details.

The `<check-menu-button>` class is used to create a menu item that has the properties of a check button. The value of a check menu button may be toggled on and off, just like a check button, by repeatedly choosing the menu item. In a group of check menu buttons, any number may be on at any one time.

In appearance, a selected check menu button is usually shown with a check mark to the left of the command name on the menu.

For more information about creating menus, see Adding Menus To The Application.

Collection gadgets

Collection gadgets are gadgets whose items can consist of any Dylan collection. They are typically used to group together a number of related objects, such as items in a list or a group of buttons. All collection gadgets are general instances of the protocol class `<collection-gadget>`.

Note that collection gadgets are not actually defined as collections of gadgets, as you might assume. Instead, they contain a sequence of items, such as strings, numbers, or symbols, that describe the contents of the collection gadget. It is worth emphasizing this distinction since, visually, collection gadgets often look like groups of individual gadgets.

Useful properties of collection gadgets

All collection gadgets share certain essential properties. These can either be specified when an instance of a gadget is created, using an init-keyword, or set interactively via a slot value.

gadget-items This slot contains a Dylan collection representing the contents of a collection gadget.

gadget-label-key The label key is a function that is used to compute the label of each item in a collection gadget, and therefore defines the "printed representation" of each item. If `gadget-label-key` is not explicitly defined for a collection gadget, its items are labeled numerically.

gadget-value-key Similar to the label key, the value key is used to compute a value for each item in a collection gadget. The gadget value of a collection gadget is the value of any selected items in the collection gadget.

gadget-selection-mode The selection mode of a collection gadget determines how many items in the gadget can be selected at any time. This takes one of three symbolic values: `#"single"` (only one item can be selected at any time), `#"multiple"` (any number of items can be selected at once), `#"none"` (no items can be selected at all).

Note that you can use `gadget-selection-mode` to read the selection mode of a gadget, but you cannot reset the selection mode of a gadget once it has been created. Instead, use the `selection-mode:` init-keyword to specify the selection mode when the gadget is created.

Generally, different subclasses of collection gadget specify this property automatically. For example, a radio box is single selection, and a check box is multiple selection.

To specify any of these slot values as an init-keyword, remove the "gadget-" prefix. Thus, the `gadget-value-key` slot becomes the `value-key:` init-keyword.

Button boxes

Groups of functionally related buttons are placed in button boxes. The superclass for button boxes is the `<button-box>` class. The two most common types of button box are `<check-box>` (groups of check buttons) and `<radio-box>` (groups of radio buttons). In addition, `<push-box>` (groups of push buttons) can be used.

Fig. 9.2: A push box

Note: You should be aware of the distinction between the use of the term "box" in DUIM, and the use of the term "box" in some other development documentation (such as Microsoft's interface guidelines). *In the context of DUIM, a box always refers to a group containing several gadgets* (usually buttons). In other documentation, a box may just be a GUI element that looks like a box. For example, a *check button* may sometimes be called a *check box*.

A `<radio-box>` is a button box that contains one or more radio buttons, only one of which may be selected at any time.


```
define variable *my-radio-box*
  := make(<radio-box>, items: #[1, 2, 3],
          value: 2);
```

Note the use of `value:` to choose the item initially selected when the box is created.

For all boxes, the `gadget-value` is the selected button. In the illustration above the `gadget-value` is 2.

```
? gadget-value(*my-radio-box*);
=> 2
```

You can set the `gadget-value` to 3 and the selected button changes to 3:

```
gadget-value(*my-radio-box*) := 3;
```

As with all collection gadgets, use `gadget-items` to set or return the collection that defines the contents of a radio box.

```
? gadget-items(*my-radio-box*);
=> #[1, 2, 3]
```

If you reset the `gadget-items` in a collection gadget, the gadget resizes accordingly:

```
gadget-items(*my-radio-box*) := range(from: 5, to: 20, by: 5);
```

A check box, on the other hand, can have any number of buttons selected. The following code creates a check box. After creating it, select the buttons labelled 4 and 6, as shown below.

```
define variable *my-check-box*
  := make(<check-box>, items: #(4, 5, 6));
```

You can return the current selection, or set the selection, using `gadget-value`.

```
gadget-value(*my-check-box*);
=> #[4, 6]
gadget-value(*my-check-box*) := #[5, 6];
```

Remember that for a multiple-selection collection gadget, the gadget value is a sequence consisting of the values of all the selected items. The value of any given item is calculated using the value key.

Menu boxes

In addition to groups of buttons, groups of menu items can be created. All of these are subclasses of the class `<menu-box>`.

 A `<push-menu-box>` is a group of several standard menu items. A `<push-menu-box>` is the menu-specific version of `<push-box>`. This is the default type of `<menu-box>`.

 A `<radio-menu-box>` is a group of several radio menu items. A `<radio-menu-box>` is the menu-specific version of `<radio-box>`.

 A `<check-menu-box>` is a group of several check menu items. A `<check-menu-box>` is the menu-specific version of `<check-box>`.

All the items in a menu box are grouped together on the menu in which they are placed. A divider separates these items visually from any other menu buttons or menu boxes placed above or below in the menu. It is useful to use push menu boxes to group together related menu commands such as *Cut*, *Copy*, and *Paste*, where the operations performed by the commands are related, even though the commands themselves do not act as a group. Note that you can also use command tables to create and group related menu commands. See Using Command Tables for more details.

Lists

A <list-box>, although it has a different appearance than a <radio-box>, shares many of the same characteristics:

```
make(<list-box>, items: #(1, 2, 3));
```

Fig. 9.3: A list box

As with other boxes, gadget-value is used to return and set the selection in the box, and gadget-items is used to return and set the items in the box.

Like button boxes, list boxes can be specified as either single, multiple, or no selection when they are created, using the selection-mode: init-keyword. Unlike button boxes, different values for selection-mode: do not produce gadgets that are different in appearance; a single selection list box is visually identical to a multiple selection list box.

Two init-keywords let you specify different characteristics of a list box.

The borders: init-keyword controls the appearance of the border placed between the list itself, and the rest of the gadget. It takes a number of symbolic arguments, the most useful of which are as follows:

- #"sunken" The list looks as if it is recessed compared to the surrounding edge of the gadget.
- #"raised" The list looks as if it is raised compared to the surrounding edge of the gadget.
- #"groove" Rather than raising or lowering the list with respect to its border, a groove is drawn around it.
- #"flat" No border is placed between the list and the edges of the gadget.

The scroll-bars: init-keyword controls how scroll bars are placed around a list box. It takes the following values:

- #"vertical" The list box is given a vertical scroll bar.
- #"horizontal" The list box is given a horizontal scroll bar.
- #"both" The list box is given both vertical and horizontal scroll bars.
- #"none" The list box is given no scroll bars.
- #"dynamic" The list box is given vertical and horizontal scroll bars only when they are necessary because of the amount of information visible in the list.

 The <option-box> class is another list control that you will frequently use in your applications. This gadget is usually referred to in Microsoft documentation as a *drop-down list box*. It differs from a standard list box in that it looks rather like a text field, with only the current selection visible at any one time. In order to see the entire list, the user must click on an arrow displayed to the right of the field.

```
make(<option-box>, items: #("&Red", "&Green", "&Blue"));
```

Notice the use of the & character to denote a keyboard shortcut. Pressing the R key when the option box has focus selects Red, pressing G selects Green, and pressing B selects Blue.

Like list boxes, option boxes also support the borders: and scroll-bars: init-keywords.

The <combo-box> class is visually identical to the <option-box> class, except that the user can type into the text field portion of the gadget. This is a useful way of allowing the user to specify an option that is not provided in the list, and a common technique is to add any new options typed by the user into the drop-down list part of the gadget for future use.

Like list boxes and option boxes, combo boxes support the borders: and scroll-bars: init-keywords.

Display controls

Display controls describe a set of collection gadgets that provide a richer set of features for displaying more complex objects, such as files on disk, that may have properties such as icons associated with them.

A number of display controls are available that, like lists, are used to display information in a variety of ways.

Tree controls

The `<tree-control>` class (also known as a tree view control in Microsoft documentation) is a special list control that displays a set of objects in an indented outline based on the logical hierarchical relationship between the objects. A number of slots are available to control the information that is displayed in the control, and the appearance of that information.

Fig. 9.4: A tree control

The `tree-control-children-generator` slot contains a function that is used to generate any children below the root of the tree control. It is called with one argument, which can be any instance of `<object>`.

The `icon-function:` init-keyword specifies a function that returns an icon to display with each item in the tree control. The function is called with the item that needs an icon as its argument, and it should return an instance of `<image>` as its result. Typically, you might want to define an icon function that returns a different icon for each type of item in the control. For example, if the control is used to display the files and directories on a hard disk, you would want to return the appropriate icon for each registered file type.

Typically, icons should be no larger than 32 pixels high and 32 pixels wide: if the icon function returns an image larger than this, then there may be unexpected results.

Note that there is no setter for the icon function, so the function cannot be manipulated after the control has been created. In the example below, `$odd-icon` and `$even-icon` are assumed to be icons that have been defined.

```
make(<tree-control>,
     roots: #[1],
     children-generator:
       method (x) vector(x * 2, 1 + (x * 2)) end,
     icon-function: method (item :: <integer>)
       case
         odd?(item) => $odd-icon;
         even?(item) => $even-icon;
       end);
```

Like list boxes and list controls, tree controls support the `scroll-bars:` init-keyword.

List controls

 The `<list-control>` class is used to display a collection of items, each item consisting of an icon and a label. In Microsoft documentation, this control corresponds to the List View control in its "icon", "small icon", and "list" views. Like other collection gadgets, the contents of a list control is determined using the `gadget-items` slot.

Like tree controls, list controls support the `icon-function:` init-keyword. Note, however, that unlike tree controls, you can also use the `list-control-icon-function` generic function to retrieve and set the value of this slot after the control has been created.

A number of different views are available, allowing you to view the items in different ways. These views let you choose whether each item should be accompanied by a large or a small icon. You can specify the view for a list control when it is first created, using the `view:` init-keyword. After creation, the `list-control-view` slot can be used to read or set the view for the list control.

The list control in the example below contains a number of items, each of which consists of a two element vector.

- The first element (a string) represents the label for each item in the list control.
- The second element (beginning with "reply-") represents the value of each item in the list control—in this case the callback function that is invoked when that item is double-clicked.

The example assumes that you have already defined these callback functions elsewhere.

```
make(<list-control>,
     items: vector(vector("Yes or No?", reply-yes-or-no),
                   vector("Black or White?",
                          reply-black-or-white),
                   vector("Left or Right?", reply-left-or-right),
                   vector("Top or Bottom?", reply-top-or-bottom),
                   vector("North or South?",
                          reply-north-or-south)),
     label-key: first,
     value-key: second,
     scroll-bars: #"none",
     activate-callback: method (sheet :: <sheet>)
         gadget-value(sheet)(sheet-frame(sheet))
       end);
```

In the example above, `first` is used to calculate the label that is used for each item in the list, and `second` specifies what the value for each item is. The activate callback examines this gadget value, so that the callback specified in the `items:` init-keyword can be used. Note that the `scroll-bars:` init-keyword can be used to specify which, if any, scroll bars are added to the control.

Like list boxes, and tree controls, list controls support the `borders:` and `scroll-bars:` init-keywords.

Table controls

The `<table-control>` class (which corresponds to the List View control in its "report" view in Microsoft documentation) allows you to display items in a table, with information divided into a number of column headings. This type of control is used when you need to display several pieces of information about each object, such as the name, size, modification date and owner of a file on disk. Typically, items can be sorted by any of the columns shown, in ascending or descending order, by clicking on the column header in question.

Because a table control displays more complex information than a list control, two init-keywords, `headings:` and `generators:` are used to create the contents of a table control, based on the control's items.

- `headings:` This takes a sequence of strings that are used as the labels for each column in the control.
- `generators:` This takes a sequence of functions. Each function is invoked on each item in the control to calculate the information displayed in the respective column.

Thus, the first element of the `headings:` sequence contains the heading for the first column in the control, and the first function in the `generators:` sequence is used to generate the contents of that column, and so on for each element in each sequence, as shown here:

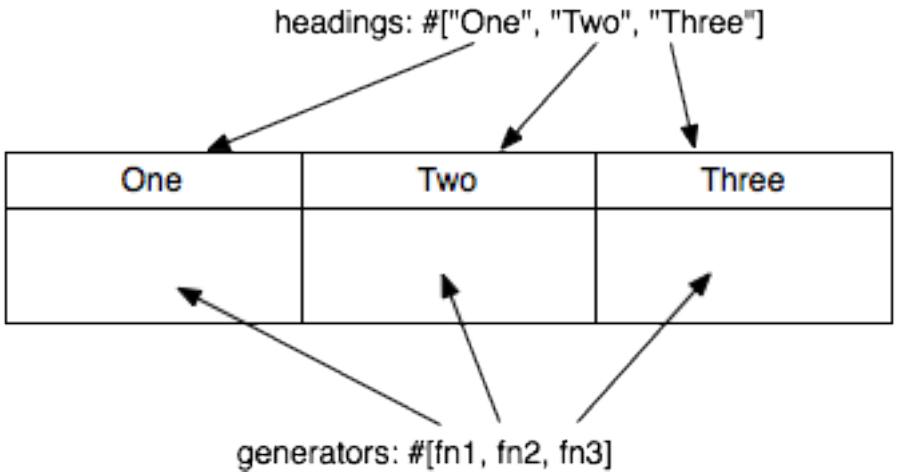

Fig. 9.5: Defining column headings and contents in table controls

Note that the sequences passed to both of these init-keywords should contain the same number of elements, since there must be as many column headings as there are functions to generate their contents.

Like list boxes and list controls, table controls support the `borders:` and `scroll-bars:` init-keywords. Like list controls, the `view:` init-keyword and `table-control-view` slot can be used to manipulate the view used to display the information: choose between `#"table"`, `#"small-icon"`, `#"large-icon"`, and `#"list"`. The `widths:` init-keyword can be used to determine the width of each column in a table control when it is created. This column takes a sequence of integers, each of which represents the width in pixels of its respective column in the control.

Spin boxes

A `<spin-box>` is a collection gadget that only accepts a limited set of ordered values as input. To the right of the text field are a pair of buttons depicting an upward pointinglimage11| arrow and a downward pointing arrow. Clicking on the buttons changes the value in the text field, incrementing or decrementing the value as appropriate.

A typical spin box might accept the integers 0-50. You could specify a value in this spin box either by typing it directly into the text field, or by clicking the up or down arrows until the number 50 was displayed in the text field.

The `gadget-items` slot is used to specify the possible values that the spin box can accept.

Consider the following example:

```
make(<spin-box>, items: range(from: 6, to: 24, by: 2));
```

This creates a spin box that accepts any even integer value between 6 and 24.

Text gadgets

Several text gadgets are provided by the DUIM-Gadgets library. These represent gadgets into which the user of your application can type information. The superclass of all text gadgets is the `<text-gadget>` class.

There are three kinds of text gadget available: text fields, text editors, and password fields.

Useful properties of text gadgets

You an initialize the text string in a text gadget using the `text:` init-keyword. The `gadget-text` slot can then be used to manipulate this text after the gadget has been created.

The `value-type:` init-keyword (and the `gadget-value-type` slot) is used to denote that a given text gadget is of a particular type. Currently, three types are supported: `<string>`, `<integer>`, and `<symbol>`. The type of a text gadget defines the way that the text typed into a text gadget is treated by `gadget-value`. The default is `<string>`.

The `gadget-text` slot *always* returns the exact text contents of a text gadget. However, `gadget-value` interprets the text and returns a value of the proper type, depending on the `gadget-value-type`, or `#f` if the text cannot be parsed. Setting the `gadget-value` "prints" the value and inserts the appropriate text into the text field.

For example, if you specify `value-type: <integer>`, then `gadget-text` always returns the exact text typed into the text gadget, as an instance of `<string>`, even if the text contains non-integer characters. However, `gadget-value` can only return an instance of `<integer>`, having interpreted the `gadget-text`. If the `gadget-text` contains any non-integer characters, then interpretation fails, and `gadget-value` returns `#f`.

Note that the combo boxes and spin boxes also contains a textual element, though they are not themselves text gadgets.

Text fields

The `<text-field>` class is a single line edit control, and is the most basic type of text gadget, consisting of a single line into which you can type text.


```
make(<text-field>, value-type: <integer>, text: "1234");
```

Use the''x-alignment:'' init-keyword to specify how text typed into the field should be aligned. This can be either `#"left"`, `#"center"`, or `#"right"`, the default being `#"left"`.

Text editors

The `<text-editor>` class is a multiple line edit control, used when more complex editing controls and several lines of text are needed by the user.

The `columns:` and `lines:` init-keywords control the size of a text editor when it is created. Each init-keyword takes an integer argument, and the resulting text editor has the specified number of character columns (width) and the specified number of lines (height).

In addition, text editors support the `scroll-bars:` init-keyword described in *Lists*.

```
make(<text-editor>, lines: 10, fixed-height?: #t);
```

Password fields

The `<password-field>` class provides a specialized type of single line edit control for use in situations where the user is required to type some text that should not be seen by anyone else, such as when typing in a password or identification code. Visually, a password field looks identical to a text field. However, when text is typed into a password field, it is not displayed on the screen; a series of asterisks may be used instead.

Range gadgets

Range gadgets are gadgets whose `gadget-value` can be any value on a sliding scale. The most obvious examples of range gadgets are scroll bars and sliders. The protocol class of all range gadgets is the class `<value-range-gadget>`.

Useful properties of range gadgets

When creating a range gadget, you must specify the range of values over which the `gadget-value` of the gadget can vary, using the `gadget-value-range` slot. An instance of type `<range>` must be passed to this slot. You can initialize this value when creating a value range gadget using the `value-range:` init-keyword. The default range for any value range gadget is the set of integers from 0 to 100.

When first created, the value of a range gadget is the minimum value of the `gadget-value-range` of the gadget, unless `value:` is specified. As with all other gadgets, use `gadget-value` to return or set this value, as shown in *Returning or setting the gadget-value of a scroll-bar*, which illustrates this behavior for a scroll bar.

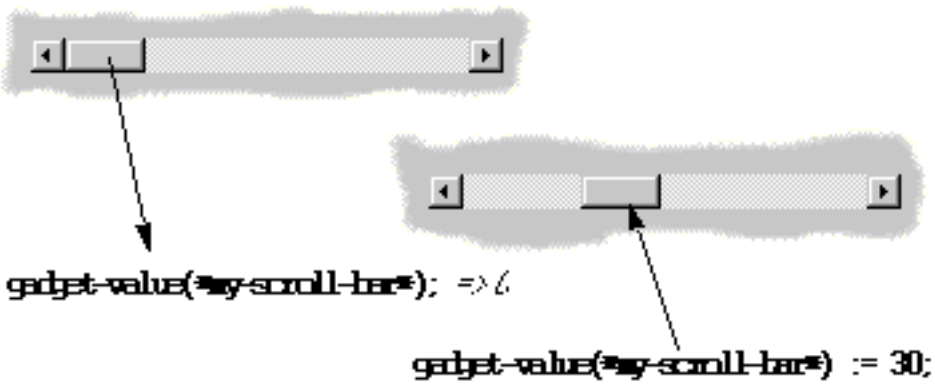

Fig. 9.6: Returning or setting the gadget-value of a scroll-bar

Scroll bars

The `<scroll-bar>` class is the most common type of value range gadget. Interestingly, it is probably also the class that is explicitly used the least. Because most gadgets that make use of scroll bars support the `scroll-bars:` init-keyword; you rarely need to explicitly create an instance of `<scroll-bar>` and attach it to another gadget.

```
define variable *my-scroll-bar* :=
  contain(make(<scroll-bar>,
               value-range: range(from: 0, to: 50)));
```

On the occasions when you do need to place scroll bars around a gadget explicitly, use the `scrolling` macro.

```
scrolling (scroll-bars: #"vertical")
  make(<radio-box>,
       orientation: #"vertical",
       items: range(from: 1, to: 50))
end
```

Sliders

Sliders can be created in much the same way as scroll bars. By default, the gadget value is displayed alongside the slider itself.

 You can display tick marks along the slider using the `tick-marks:` init-keyword, which is either `#f` (no tick marks are displayed) or an integer, which specifies the number of tick marks to display. The default is not to show tick marks.

If tick marks are used, they are distributed evenly along the length of the slider. You can use as many or as few tick marks as you wish, and you are advised to use a number that is natural to the user, such as 3, 5, or 10. While it is possible to use oddball numbers such as 29, this could confuse the user of your application, unless there is a compelling reason to do so.

```
define variable *my-slider*
  := make(<slider>,
          value-range: range(from: 0, to: 50)
          tick-marks: 10);
```

Progress bars

 The `<progress-bar>` class is used to display a dialog that provides a gauge illustrating the progress of a particular task. Possible uses for progress bars include the progress of an installation procedure, downloading e-mail messages from a mail server, performing a file backup, and compiling one or more files of source code. Any situation in which the user may have to wait for a task to complete is a good candidate for a progress bar.

Assigning callbacks to gadgets

To make gadgets actually do something, you have to assign them callback functions. A callback is a function that is invoked when a particular event occurs on a gadget, such as pressing a button. When the user presses a button, the appropriate callback method is invoked and some behavior, defined by you, occurs. It is the main way of providing your applications with some kind of interactive functionality. Most classes of gadget have a number of different callbacks available. Like gadget properties, different classes of gadget can have different callback types available.

The most common type of callback is the activate callback. This is the callback that is invoked whenever a general instance `<action-gadget>` is activated: for instance, if a push button is clicked. All the gadget classes you have seen so far are general instances of `<action-gadget>`.

The following code creates a push button that has an activate callback defined:

```
make(<push-button>,
     label: "Hello",
     activate-callback: method (button)
         notify-user("Pressed button!",
         owner: button)
       end)));
```

The `notify-user` function is a useful function that lets you display a message in a dialog.

Now when you click on the button, a notification pops up saying “Pressed button!”

Two callbacks are unique to general instances of `<value-gadget>` : the value-changing and the value-changed callbacks. The value-changing callback is invoked as the gadget value of the gadget changes, and the value-changed callback is invoked when the value has changed, and is passed back to the gadget.

Fig. 9.7: Simple behavior of notify-user

In practice, a value-changing callback is of most use in a gadget whose value you need to monitor constantly, such as a <value-range-gadget>. A value-changed callback is of most use when the user enters a value explicitly and returns it to the application, for instance by clicking on a button or pressing RETURN.

In a text field, for example, a value-changing callback would be invoked whenever a character is typed in the text field, whereas a value-changed callback would be invoked once the user had finished typing and had returned the value to the gadget. For a text field, the value-changed callback is usually more useful than the value-changing callback.

```
contain(make(<text-field>,
 value-changed-callback:
   method (gadget)
     notify-user
       ("Changed to %=", gadget-value(gadget))
   end));
```

A tour of layouts

Layouts determine how the elements that make a GUI are presented on the screen. Together with gadgets, layouts are an important type of sheet that you need to be familiar with in order to develop basic DUIM applications. Support for layouts is provided by the DUIM-Layouts library.

You can think of layouts as containers for gadgets and other layouts. They have little or no physical substance on the screen, and simply define the way in which other elements are organized. The sheet at the top of the sheet hierarchy will always be a layout.

Any layout takes a number of children, expressed as a sequence (usually a vector), and lays them out according to certain constraints. Each child must be an instance of a DUIM class. Typically, the children of any layout will be gadgets or other layouts.

There are six main classes of layouts, as follows:

<column-layout> This lays out its children in a single column, with all its children left-aligned by default.

<row-layout> This lays out its children in a single row.

<pinboard-layout> This does not constrain the position of its children in any way. It is up to you to position each child individually, like pins on a pinboard.

<fixed-layout> This class is similar to pinboard layouts, in that you must specify the position of each child. Unlike pinboard layouts, however, you must also specify the size of each child.

<stack-layout> This lays out its children one on top of another, with all the children aligned at the top left corner by default. It is used to design property sheets, tab controls, or wizards, which contain several layouts, only one of which is visible at any one time.

<table-layout> This lays out its children in a table, according to a specified number of rows and columns.

Row layouts and column layouts

Create a column layout containing three buttons as follows:

```
contain(make(<column-layout>,
             children: vector(make(<push-button>, label: "One"),
                              make(<push-button>, label: "Two"),
                              make(<push-button>, label: "Three"))));
```

Fig. 9.8: Three button arranged in a column layout

Similarly, `<row-layout>` can be used to lay out any number of children in a single row.

A number of different init-keywords can be used to specify the initial appearance of any layouts you create. Using these init-keywords, you can ensure that all children are the same size in one or both dimensions, and that a certain amount of space is placed between each child. You can also place a border of any width around the children of a layout.

To equalize the heights or widths of any child in a layout, use `equalize-heights?: #t` or `equalize-widths?: #t` respectively. To ensure that each child is shown in its entirety, the children are sized according to the largest child in the layout, for whatever dimension is being equalized.

The `equalize-heights?:` and `equalize-widths?:` init-keywords are particularly useful when defining a row of buttons, when you want to ensure that the buttons are sized automatically. In addition, remember that each button can be specified as `max-width: $fill` to ensure that the button is sized to be as large as possible, rather than the size of its label.

To add space between each child in a layout, use `spacing:`, which takes an integer value that represents the number of pixels of space that is placed around each child in the layout. Use `border:` in much the same way; specifying an integer value creates a border around the entire layout which is that number of pixels wide. Notice that while `spacing:` places space around each individual child in the layout, `border:` creates a border around the entire layout. You can use `border-type:` to specify whether you want borders to appear sunken, raised, or flat.

Each of the init-keywords described above apply to both row layouts and column layouts. The following init-keywords each only apply to one of these classes.

Use `x-alignment:` to align the children of a column layout along the `x` axis. This can be either `#"left"`, `#"right"`, or `#"center"`, and the children of the column layout are aligned appropriately. By default, the children of a column layout are aligned along the left hand side.

Use `y-alignment:` to align the children of a row layout along the `y` axis. This can be either `#"top"`, `#"bottom"`, or `#"center"`, and the children of the column layout are aligned appropriately. By default, the children of a row layout are aligned along the top.

Stack layouts

The `<stack-layout>` class is provided to let you create layout classes in which only one child is visible at a time. They are used to implement tab controls and wizards. In a stack layout, all children are placed on top of one another, with each child aligned at the top left corner by default.

```
make(<stack-layout>,
     children: vector(make(<list-box>, label: "List 1"
                           items: #("One", "Two",
                                    "Three", "Four"),
                      make(<list-box>, label: "List 2"
                           items: #("Five", "Six",
```

```
                                    "Seven", "Eight"),
                            make(<push-button>, label: "Finish")));
```

Pinboard layouts and fixed layouts

A pinboard layout is a framework that serves as a place to locate any number of child gadgets. It has no built in layout information, so, unless you specify coordinates explicitly, any object placed in a pinboard layout is placed at the coordinates 0,0 (top left), with the most recently created object on top.

In normal use, you should supply coordinate information for each child to determine its position in the layout. You have complete flexibility in positioning objects in a pinboard layout by giving each object coordinates, as shown in the following example:

```
contain
  (make
    (<pinboard-layout>,
     children:
       vector (make(<push-button>, label: "One", x: 0, y: 0),
               make(<push-button>, label: "Two", x: 50,y: 50),
               make(<push-button>, label: "Three",
                    x: 50, y: 100))));
```

Fig. 9.9: Three buttons arranged in a pinboard layout

Any child in a pinboard layout obeys any size constraints that may apply to it, whether those constraints have been specified by you, or calculated by DUIM. For instance, any button you place on a pinboard layout will always be large enough to display all the text in its label, as shown in *Three buttons arranged in a pinboard layout*. The `<fixed-layout>` class takes generalization of layouts a step further, by requiring that you specify not only the position of every child, but also its size, so that DUIM performs no constraint calculation at all. This class of layout should only be used if you know exactly what size and position every child in the layout should have. It might be useful, for instance, if you were setting up a resource database in which the sizes and positions of a number of sheets were specified, and were to be read directly into your application code from this database. For most situations, however, you will not need to use the `<fixed-layout>` class.

Using horizontally and vertically macros

The macros `horizontally` and `vertically` are provided to position objects sequentially in a column layout or row layout. Using these macros, rather than creating layout objects explicitly, can lead to shorter and more readable code.

```
horizontally ()
  make(<push-button>, label: "One");
  make(<push-button>, label: "Two");
  make(<push-button>, label: "Three")
end;
```


Fig. 9.10: Three buttons arranged in a horizontal layout

```
vertically ()
  make(<push-button>, label: "One");
  make(<push-button>, label: "Two");
  make(<push-button>, label: "Three")
end;
```

You can specify any init-keywords that you would specify for an instance of `<row-layout>` or `<column-layout>` using `vertically` and `horizontally`. To do this, just pass the init-keywords as arguments to the macro. The following code ensures that the row layout created by `horizontally` is the same width as the button with the really long label. In addition, the use of `max-width:` in the definitions of the two other buttons ensures that those buttons are sized so as to occupy the entire width of the row layout.

```
vertically (equalize-widths?: #t)
  horizontally ()
    make(<button>, label: "Red", max-width: $fill);
    make(<button>, label: "Ultraviolet", max-width: $fill);
  end;
  make(<button>,
       label: "A button with a really really long label");
end
```

A tour of sheets

Each unique piece of a window is a sheet. Thus, a sheet creates a visible element of some sort on the screen. In any frame, sheets are nested in a parent-child hierarchy. The DUIM-Sheets library provides DUIM with many different types of sheet, and defines the behavior of sheets in any application.

For basic DUIM applications, you do not need to be aware of sheet protocols, and you do not need to define your own sheet classes, since most of the sheet classes you need to use have been implemented for you in the form of gadgets (*A tour of gadgets*) and layouts (*A tour of layouts*).

Basic properties of sheets

All sheets, including gadgets and layouts, have a number of properties that deal with the fairly low level implementation behavior of sheet classes. When developing basic DUIM applications, you do not need to be concerned with these properties for the most part, since gadgets and layouts have been designed so as to avoid the need for direct low level manipulation. However, if you design your own classes of sheet, you need to support these properties.

`sheet-region` The sheet region is used to define the area of the screen that "belongs to" a sheet. This is essential for deciding in which sheet a particular event occurs. For example, the `sheet-region` for a gadget defines the area of the screen in which its callbacks are invoked, should an event occur.

The sheet region is expressed in the sheet's own coordinate system. It can be an instance of any concrete subclass of `<region>`, but is usually represented by the region class `<bounding-box>`.

The sheet-region is defined relative to the region of its parent, rather than an absolute region of the screen.

`sheet-transform` This maps the sheet's coordinate system to the coordinate system of its parent. This is an instance of a concrete subclass of `<transform>`.

Providing the sheet transform means that you do not have to worry about the absolute screen position of any given element of an interface. Instead, you can specify its location relative to its parent in the sheet hierarchy. For example, you can arrange gadgets in an interface in terms of the layout that contains them, rather than in absolute terms.

sheet-parent This is #f if the sheet has no parent, or another sheet otherwise. This slot is used to describe any hierarchy of sheets.

sheet-mapped? This is a boolean that specifies whether the sheet is visible on a display, ignoring issues of occluding windows.

sheet-frame This returns the frame a sheet belongs to.

Many sheet classes, such as <menu-bar> or <tool-bar>, have single or multiple children, in which case they have additional attributes:

- **sheet-children The value of this slot is a sequence of sheets. Each** sheet in the sequence is a child of the current sheet.
- Methods to add, remove, and replace a child.
- Methods to map over children.

Some classes of sheet — usually gadgets — can receive input. These have:

sheet-event-queue

- This is a list of all the events currently queued and waiting for execution for a given sheet.

Methods for <handle-event>

- Each class of sheet must have methods for <handle-event> defined, so that callbacks may be described for the sheet class.

Sheets that can be repainted have methods for handle-repaint. Sheets that can display output have a sheet-medium slot. As a guide, all gadgets can be repainted and can display output, and no layouts can be repainted or display output.

A tour of frames

As you will have seen if you worked through the task list manager example application, frames are the basic components used to display DUIM objects on-screen. Every window in your application is a general instance of <frame>, and contains a hierarchy of sheets. Frames control the overall appearance of the entire window, and organize such things as menu bars, tool bars, and status bars.

A subclass of <frame>, <simple-frame>, is the way to create basic frames. Usually, you will find it most convenient to define your own classes of frame by subclassing <simple-frame>.

The event loop associated with a frame is represented by a queue of instances, each instance being a subclass of <event>. The most important events are subclasses of <device-event>, for example, <button-press-event> and <key-press-event>. Unless you intend defining your own event or sheet classes, you do not need to understand events.

Different types of frame are provided, allowing you to create normal windows, as well as dialog boxes (both modal and modeless), property pages and wizards.

Support for frames is provided by the DUIM-Frames library.

Creating frames and displaying them on-screen

To create an instance of a frame class, use make, as you would any other class. To display an instance of a frame on the screen, use the function start-frame. This takes as an argument a name bound to an existing frame, or an expression (including function and macro calls) that evaluates to a frame instance.

For example, to create a simple frame that contains a single button, use the following code:

```
start-frame(make(<simple-frame>,
                 title: "Simple frame",
                 layout:
                   make(<push-button>,
                        label: "A button on a simple frame")));
```

Fig. 9.11: A simple frame

Note that normally you should define your own subclasses or <simple-frame> and call start-frame on instances of these, rather than creating direct instances of <simple-frame>.

Useful properties of frames

You can specify a wide variety of properties for any instance or class of frame. This section describes some of the most common properties you might want to use. Naturally, when you create your own classes of frame by subclassing <simple-frame>, you can define new properties as well. For more information on creating your own frame classes, see *Defining new classes of frame*, and review the description of the task list manager in Improving The Design and Adding Menus To The Application.

The frame-pane property is used to define every discrete element in a frame class. Exactly what constitutes a discrete element is, to a large extent, up to the programmer. As a guide, every pane definition creates an accessor just like a slot accessor, and so any element whose value you might want to retrieve should be defined as a pane. Individual gadgets, layouts, and menus are all generally expressed as panes in a frame definition. When defining a frame class, use the pane option to define each pane.

The frame-layout property is used to specify the topmost layout in the sheet hierarchy that forms the contents of a frame class. This take an instance of any subclass of <layout> which may itself contain any number of gadgets or other layouts as children. The children of this layout are themselves typically defined as panes within the same frame definition. When defining a frame class, use the layout option to define the topmost layout.

Other major components of a frame can be specified using frame-menu-bar, frame-tool-bar, and frame-status-bar. Each property takes an instance of the corresponding gadget class as its value. You can also use frame-command-table to specify a command table defining all the menu commands available in the menu bar. All of these slots have corresponding options you can set when creating your own frame classes.

To determine the initial size and position of any frame, use frame-width, frame-height, frame-x, and frame-y. Each of these properties takes an integer argument that represents a number of pixels. Note that frame-x and frame-y represent the position of the frame with respect to the top left hand corner of the screen.

Sometimes, it may be useful to fix the height or width of a frame. This can be done using frame-fixed-width? and frame-fixed-height?, both of which take a boolean value. Setting frame-resizable? to #f fixes both the width and height of a frame.

Defining new classes of frame

As described in Defining a new frame class, the define frame macro is used to create new classes of frame. The bulk of the definition of any new frame is split into several parts:

- The definition of any slots and init-keywords you want available for the new class of frame.
- The definition of any panes that should be used in the new class of frame.

- The definition of other components that you wish to include, such as a menu bar, status bar, and so on.

Slots and init-keywords can be used to let you (or the user of your applications) set the properties of any instances of the new frame class that are created.

Panes control the overall appearance of the new class of frame. You need to define panes for any GUI elements you wish to place in the frame.

Specifying slots for a new class of frame

As with any other Dylan class, you can use standard slot options to define slots for any new class of frame. This includes techniques such as setting default values, specifying init-keyword names, and specifying whether or not an init-keyword is required.

The following example defines a subclass of `<simple-frame>` that defines an additional slot that can be set to a date and time. The default value of the slot is set to the current date and time using an init expression. So that you can provide an initial value for the slot, it is defined with an init-keyword of the same name.

```
define frame <date-frame> (<simple-frame>)
  slot date :: <date> = current-date(),
    init-keyword: date:;
  // Other stuff here
end class <date-frame>;
```

Specifying panes for a new class of frame

In the same way that you can define slots, you can define panes for a frame class using pane options. Panes may be used to define all the visual aspects of a frame class, including such things as:

- The layouts and gadgets displayed in the frame
- The menu bar, menus, and menu commands available in the frame
- Additional components, such as tool bars or status bars

Typically, the definition for any pane has the following syntax:

```
pane *pane-name* (*pane-owner* ) *pane-definition* ;
```

This breaks down into the following elements:

- The reserved word `pane`.
- The name you wish to give the pane, which acts as a slot accessor for the frame, to let you retrieve the pane.
- A space in which you can bind the owner of the pane (usually the frame itself) to a local variable for use inside the pane definition
- The definition of the pane

Once you have defined all the visual components of a frame using an arrangement of panes of your choice, each major component needs to be included in the frame using an appropriate clause. For example, to include a tool bar, having created a pane called `app-tool-bar` that contains the definition of the tool bar itself, you need to include the following code at the end of the definition of the frame:

```
tool-bar (frame) frame.app-tool-bar;
```

The major components that need to be activated in any frame definition are the top level layout, menu bar, tool bar, and status bar.

The following example shows how to define and activate panes within a frame.

Three panes are defined:

- `button` A push button that contains a simple callback.
- `status` A status bar.
- `main-layout` A column layout that consists of the `button` pane, together with a drawing pane.

```
define frame <example-frame> (<simple-frame>)
  ... other code here

  // pane definitions
  pane button (frame)
    make(<push-button>,
         label: "Press",
         activate-callback:
           method (button)
             notify-user (format-to-string ("Pressed button"),
                          owner: frame)
           end);

  pane status (frame)
    make(<status-bar>);

  pane main-layout (frame)
    vertically (spacing: 10)
      horizontally (borders: 2, x-alignment: #"center")
        frame.button;
      end;
      make(<drawing-pane>,
           foreground: $red);
    end;

  ... other code here

  // activate components of frame
  layout (frame) frame.main-layout;
  status-bar (frame) frame.status;

  // frame title
  keyword title: = "Example Frame";
end frame <example-frame>;
```

The following method creates an instance of an `<example-frame>`.

The simplest way to create an example frame is by calling this method thus: `make-example-frame();`.

```
define method make-example-frame => (frame :: <example-frame>)
  let frame
    = make(<example-frame>);
  start-frame(frame);
end method make-example-frame;
```

For a more complete example of how to define your own class of frame for use in an application, see the chapters that cover the development of the Task List Manager in this manual (Chapters Designing A Simple DUIM Application to Using Command Tables).

Overview of dialogs

Dialog boxes are a standard way of requesting more information from the user in order to proceed with an operation. Typically, dialog boxes are modal — that is, the operation cannot be continued until the dialog is dismissed from the screen. Whenever an application requires additional information from the user before carrying out a particular command or task, you should provide a dialog to gather information.

For general purposes, you can create your own custom dialog boxes using frames: the class `<dialog-frame>` is provided as a straightforward way of designing frames specifically for use as dialogs. See *A tour of frames* for an introduction to frames.

For commonly used dialog boxes, DUIM provides you with a number of convenience functions that let you use predefined dialogs in your applications without having to design each one specifically. These convenience functions use pre-built dialog interfaces supplied by the system wherever possible,. This not only makes them more efficient, it also guarantees that the dialogs have the correct look and feel for the system for which you are developing.

Many systems, for example, provide pre-built interfaces for the Open, Save As, Font, and similar dialog boxes. By using the functions described in this section, you can guarantee that your application uses the dialog boxes supplied by the system wherever they are available.

The most commonly used convenience function is `notify-user`, which you have already seen. This function provides you with a straightforward way of displaying an alert message on screen in whatever format is standard for the target operating system.

```
contain(make(<push-button>,
             label: "Press me!",
             activate-callback:
               method (gadget)
                 notify-user
                   (format-to-string ("You pressed me!"))
               end));
```

The example above creates a push button which, when pressed, calls `notify-user` to display message.

The common Open File and Save File As dialogs can both be generated using `choose-file`. The `direction:` keyword lets you specify a direction that distinguishes between the two types of dialog: thus, if the direction is `#"input"`, a file is opened, and if the direction is `#"output"` a file is saved.

```
choose-file(title: "Open File", direction: #"input");
choose-file(title: "Save File As", direction: #"input");
```

Note that DUIM provides default titles based on the specified direction, so you need only specify these titles if you want to supply a non-standard title to the dialog.

Further examples of this function can be found in Handling files in the task list manager.

The convenience functions `choose-color` and `choose-text-style` generate the common dialogs for choosing a color and a font respectively. Use `choose-color` when you need to ask the user to choose a color from the standard color palette available on the target operating system, and use `choose-text-style` when you want the user to choose the font, style, and size for a piece of text.

Several other convenience dialogs are provided by DUIM. The following is a complete list, together with a brief description of each. For more information on these dialogs, please refer to the *DUIM Reference Manual*.

`choose-color` — Choose a system color.

`choose-directory` — Choose a directory on disk.

`choose-file` — Choose an input or output file.

`choose-from-dialog` — Choose from a list presented in a dialog.

`choose-from-menu` — Choose from a list presented in a popup menu

`choose-text-style` — Choose a font.

`notify-user` — Provide various kinds of notification to the user.

There are a number of standard dialogs provided by Windows that are not listed above. If you wish to use any of them, you must either use the Win32 control directly, or you must emulate the dialog yourself by building it using DUIM classes.

Where to go from here

This concludes a fairly basic tour of the major functionality provided by DUIM. Other topics that have not been covered in this tour include colors, fonts, images, generic drawing properties, and the functionality provided to for defining your own sheets and handling events.

From here, you can refer to two other sources of information.

- If you have not already done so, go back and study the chapters that cover the development of the Task List Manager application (Designing A Simple DUIM Application to Adding Callbacks to the Application inclusive). Try building the project in the development environment, experiment with the code, and extend the application in any way you wish.
- A number of DUIM examples are supplied with Open Dylan, in addition to those discussed in this book. In the environment, choose **Tools > Open Example Project** to display the Open Example Project dialog, and try some of the examples listed under the DUIM category.
- For complete information on everything provided by DUIM, look at the *DUIM Reference Manual*. This contains a complete description of every interface exported by DUIM, together with examples where relevant. The reference manual also provides further information about how you should use DUIM, and the organization of the DUIM class hierarchy.

CHAPTER

TEN

SOURCE CODE FOR THE TASK LIST MANAGER

For completeness, here is the full source code for both versions of the task list manager. If you have followed the example given in Designing A Simple DUIM Application through Using Command Tables from the beginning, then your code should be the same as the code given in *A task list manager using menu gadgets*. The source code for the second version of the task list manager, using command tables, is given in *A task list manager using command tables*.

A task list manager using menu gadgets

This section contains the complete source code to the first complete design of the task list manager, described in Chapters Improving The Design to Adding Callbacks to the Application. To load this code into the environment, choose **Tools > Open Example Project** from any window in the environment. The code in this section can be loaded by choosing Task List 1 in the Documentation category of the Open Example Project dialog.

Contents of the file *frame.dylan* :

```
Module:    task-list
Synopsis:  Task List Manager.
Author:    Functional Objects, Inc.
Copyright:    Original Code is Copyright (c) 1995-2004 Functional Objects, Inc.
              All rights reserved.
License:      See License.txt in this distribution for details.
Warranty:     Distributed WITHOUT WARRANTY OF ANY KIND

define constant $priority-items
  = #(#("Low", #"low"),
      #("Medium", #"medium"),
      #("High", #"high"));

define frame <task-frame> (<simple-frame>)
  slot frame-task-list :: <task-list> = make(<task-list>);

  // definition of menu bar
  pane task-menu-bar (frame)
    make(<menu-bar>,
         children: vector(frame.file-menu,
                          frame.edit-menu,
                          frame.task-menu,
                          frame.help-menu));
  // definition of menus
  pane file-menu (frame)
    make(<menu>, label: "File",
```

```
        children: vector(frame.open-menu-button,
                         frame.save-menu-button,
                         frame.save-as-menu-button,
                         frame.exit-menu-button));
pane edit-menu (frame)
  make(<menu>, label: "Edit",
        children: vector(frame.cut-menu-button,
                         frame.copy-menu-button,
                         frame.paste-menu-button));
pane task-menu (frame)
  make(<menu>, label: "Task",
        children: vector(frame.add-menu-button,
                         frame.remove-menu-button));
pane help-menu (frame)
  make(<menu>, label: "Help",
        children: vector(frame.about-menu-button));

// definition of menu buttons

// Commands in the File menu
pane open-menu-button (frame)
  make(<menu-button>, label: "Open...",
        activate-callback: open-file,
        accelerator: make-keyboard-gesture(#"o", #"control"),
        documentation: "Opens an existing file.");
pane save-menu-button (frame)
  make(<menu-button>, label: "Save",
        activate-callback: save-file,
        accelerator: make-keyboard-gesture(#"s", #"control"),
        documentation: "Saves the current file to disk.");
pane save-as-menu-button (frame)
  make(<menu-button>, label: "Save As...",
        activate-callback: save-as-file,
        documentation:
          "Saves the current file with a new name.");
pane exit-menu-button (frame)
  make(<menu-button>, label: "Exit",
        activate-callback: exit-task,
        accelerator: make-keyboard-gesture(#"f4", #"alt"),
        documentation: "Exits the application.");

//Commands in the Edit menu
pane cut-menu-button (frame)
  make(<menu-button>, label: "Cut",
        activate-callback: not-yet-implemented,
        accelerator: make-keyboard-gesture(#"x", #"control"),
        documentation: "Cut the selection to the clipboard.");
pane copy-menu-button (frame)
  make(<menu-button>, label: "Copy",
        activate-callback: not-yet-implemented,
        accelerator: make-keyboard-gesture(#"c", #"control"),
        documentation: "Copy the selection to the clipboard.");
pane paste-menu-button (frame)
  make(<menu-button>, label: "Paste",
        activate-callback: not-yet-implemented,
        accelerator: make-keyboard-gesture(#"v", #"control"),
        documentation: "Paste the selection in the clipboard at the current position.");
```

```
//Commands in the Task menu
pane add-menu-button (frame)
  make(<menu-button>, label: "Add...",
       activate-callback: frame-add-task,
       accelerator: make-keyboard-gesture
                      (#"a", #"control", #"shift"),
       documentation: "Add a new task.");
pane remove-menu-button (frame)
  make(<menu-button>, label: "Remove",
       activate-callback: frame-remove-task,
       accelerator: make-keyboard-gesture
                      (#"d", #"control", #"shift"),
       documentation:
         "Remove the selected task from the list.");

//Commands in the Help menu
pane about-menu-button (frame)
  make(<menu-button>, label: "About",
       activate-callback: about-task,
       accelerator: make-keyboard-gesture(#"f1"),
       documentation:
         "Display information about the application.");

// definition of buttons
pane add-button (frame)
  make(<push-button>, label: "Add task",
       activate-callback: frame-add-task);
pane remove-button (frame)
  make(<push-button>, label: "Remove task",
       activate-callback: frame-remove-task);
pane open-button (frame)
  make(<push-button>, label: "Open file",
       activate-callback: open-file);
pane save-button (frame)
  make(<push-button>, label: "Save file",
       activate-callback: save-file);

// definition of radio box
pane priority-box (frame)
  make (<radio-box>,
        items: $priority-items,
        orientation: #"horizontal",
        label-key: first,
        value-key: second,
        value: #"medium",
        activate-callback: not-yet-implemented);

// definition of tool bar
pane task-tool-bar (frame)
  make(<tool-bar>,
       child: horizontally ()
                frame.open-button;
                frame.save-button;
                frame.add-button;
                frame.remove-button
              end);

// definition of status bar
```

```
  pane task-status-bar (frame)
    make(<status-bar>, label: "Task Manager");

  // definition of list
  pane task-list (frame)
    make (<list-box>,
          items: frame.frame-task-list.task-list-tasks,
          label-key: task-name,
          lines: 15,
          value-changed-callback: note-task-selection-change);

  // main layout
  pane task-layout (frame)
    vertically ()
      frame.task-list;
      frame.priority-box;
    end;

  // activation of frame elements
  layout (frame) frame.task-layout;
  tool-bar (frame) frame.task-tool-bar;
  status-bar (frame) frame.task-status-bar;
  menu-bar (frame) frame.task-menu-bar;

  // frame title
  keyword title: = "Task List Manager";
end frame <task-frame>;

define method initialize
    (frame :: <task-frame>, #key) => ()
  next-method();
  refresh-task-frame(frame);
end method initialize;

define method prompt-for-task
   (#key title = "Type text of new task", owner)
 => (name :: false-or(<string>),
     priority :: false-or(<priority>))
  let task-text
    = make(<text-field>,
           label: "Task text:",
           activate-callback: exit-dialog);
  let priority-field
    = make(<radio-box>,
           items: $priority-items,
           label-key: first,
           value-key: second,
           value: #"medium");
  let frame-add-task-dialog
    = make(<dialog-frame>,
           title: title,
           owner: owner,
           layout: vertically ()
                     task-text;
                     priority-field
                   end,
           input-focus: task-text);
  if (start-dialog(frame-add-task-dialog))
```

```
    values(gadget-value(task-text), gadget-value(priority-field))
  end
end method prompt-for-task;

define function make-keyboard-gesture
    (keysym :: <symbol>, #rest modifiers)
 => (gesture :: <keyboard-gesture>)
  make(<keyboard-gesture>, keysym: keysym, modifiers: modifiers)
end function make-keyboard-gesture;

define function not-yet-implemented (gadget :: <gadget>) => ()
  notify-user("Not yet implemented!", owner: sheet-frame(gadget))
end function not-yet-implemented;

define method start-task () => ()
  let frame
    = make(<task-frame>);
  start-frame(frame);
end method start-task;

define method frame-add-task (gadget :: <gadget>) => ()
  let frame = sheet-frame(gadget);
  let task-list = frame-task-list(frame);
  let (name, priority) = prompt-for-task(owner: frame);
  if (name & priority)
    let new-task = make(<task>, name: name, priority: priority);
    add-task(task-list, new-task);
    refresh-task-frame(frame);
    frame-selected-task(frame) := new-task
  end
end method frame-add-task;

define method frame-remove-task (gadget :: <gadget>) => ()
  let frame = sheet-frame(gadget);
  let task = frame-selected-task(frame);
  let task-list = frame-task-list(frame);
  if (notify-user(format-to-string
                    ("Really remove task %s", task.task-name),
                  owner: frame, style: #"question"))
    frame-selected-task(frame) := #f;
    remove-task(task-list, task);
    refresh-task-frame(frame)
  end
end method frame-remove-task;

define method frame-selected-task
    (frame :: <task-frame>) => (task :: false-or(<task>))
  let list-box = task-list(frame);
  gadget-value(list-box)
end method frame-selected-task;

define method frame-selected-task-setter
    (task :: false-or(<task>), frame :: <task-frame>)
 => (task :: false-or(<task>))
  let list-box = task-list(frame);
  gadget-value(list-box) := task;
  note-task-selection-change(frame);
  task
```

```
end method frame-selected-task-setter;

define method refresh-task-frame
    (frame :: <task-frame>) => ()
  let list-box = frame.task-list;
  let task-list = frame.frame-task-list;
  let modified? = task-list.task-list-modified?;
  let tasks = task-list.task-list-tasks;
  if (gadget-items(list-box) == tasks)
    update-gadget(list-box)
  else
    gadget-items(list-box) := tasks
  end;
  gadget-enabled?(frame.save-button) := modified?;
  gadget-enabled?(frame.save-menu-button) := modified?;
  note-task-selection-change(frame);
end method refresh-task-frame;

define method note-task-selection-change
    (gadget :: <gadget>) => ()
  let frame = gadget.sheet-frame;
  note-task-selection-change(frame)
end method note-task-selection-change;

define method note-task-selection-change
    (frame :: <task-frame>) => ()
  let task = frame-selected-task(frame);
  if (task)
    frame.priority-box.gadget-value := task.task-priority;
  end;
  let selection? = (task ~== #f);
  frame.remove-button.gadget-enabled? := selection?;
  frame.remove-menu-button.gadget-enabled? := selection?;
end method note-task-selection-change;

define method open-file
    (gadget :: <gadget>) => ()
  let frame = sheet-frame(gadget);
  let task-list = frame-task-list(frame);
  let filename
    = choose-file(frame: frame,
                  default: task-list.task-list-filename,
                  direction: #"input");
  if (filename)
    let task-list = load-task-list(filename);
    if (task-list)
      frame.frame-task-list := task-list;
      refresh-task-frame(frame)
    else
      notify-user(format-to-string("Failed to open file %s", filename),
                  owner: frame)
    end
  end
end method open-file;

define method save-file
    (gadget :: <gadget>) => ()
  let frame = sheet-frame(gadget);
```

```
  let task-list = frame-task-list(frame);
  save-as-file(gadget, filename: task-list.task-list-filename)
end method save-file;

define method save-as-file
    (gadget :: <gadget>, #key filename) => ()
  let frame = sheet-frame(gadget);
  let task-list = frame-task-list(frame);
  let filename
    = filename
      | choose-file(frame: frame,
                    default: task-list.task-list-filename,
                    direction: #"output");
  if (filename)
    if (save-task-list(task-list, filename: filename))
      frame.frame-task-list := task-list;
      refresh-task-frame(frame)
    else
      notify-user(format-to-string
                    ("Failed to save file %s", filename),
                  owner: frame)
    end
  end
end method save-as-file;

define function about-task (gadget :: <gadget>) => ()
  notify-user("Task List Manager", owner: sheet-frame(gadget))
end function about-task;

define method exit-task (gadget :: <gadget>) => ()
  let frame = sheet-frame(gadget);
  let task-list = frame-task-list(frame);
  save-file (gadget);
  exit-frame(frame)
end method exit-task;

define method main (arguments :: <sequence>) => ()
  // handle the arguments
  start-task();
end method main;

begin
  main(application-arguments()) // Start the application!
end;
```

Contents of the file *task-list.dylan* :

```
Module:    task-list
Synopsis:  Task List Manager.
Author:    Functional Objects, Inc.
Copyright:    Original Code is Copyright (c) 1995-2004 Functional Objects, Inc.
              All rights reserved.
License:      See License.txt in this distribution for details.
Warranty:     Distributed WITHOUT WARRANTY OF ANY KIND

define class <task-list> (<object>)
  constant slot task-list-tasks = make(<stretchy-vector>),
    init-keyword: tasks:;
  slot task-list-filename :: false-or(<string>) = #f,
```

```
    init-keyword: filename:;
  slot task-list-modified? :: <boolean> = #f;
end class <task-list>;

define constant <priority> = one-of(#"low", #"medium", #"high");

define class <task> (<object>)
  slot task-name :: <string>,
    required-init-keyword: name:;
  slot task-priority :: <priority>,
    required-init-keyword: priority:;
end class <task>;

define function add-task
    (task-list :: <task-list>, task :: <task>) => ()
  add!(task-list.task-list-tasks, task);
  task-list.task-list-modified? := #t
end function add-task;

define function remove-task
    (task-list :: <task-list>, task :: <task>) => ()
  remove!(task-list.task-list-tasks, task);
  task-list.task-list-modified? := #t
end function remove-task;

define function save-task-list
    (task-list :: <task-list>, #key filename)
 => (saved? :: <boolean>)
  let filename = filename | task-list-filename(task-list);
  with-open-file (stream = filename, direction: #"output")
    for (task in task-list.task-list-tasks)
      format(stream, "%s\n%s\n",
             task.task-name, as(<string>, task.task-priority))
    end
  end;
  task-list.task-list-modified? := #f;
  task-list.task-list-filename := filename;
  #t
end function save-task-list;

define function load-task-list
   (filename :: <string>) => (task-list :: false-or(<task-list>))
  let tasks = make(<stretchy-vector>);
  block (return)
    with-open-file (stream = filename, direction: #"input")
      while (#t)
        let name = read-line(stream, on-end-of-stream: #f);
        unless (name) return() end;
        let priority = read-line(stream, on-end-of-stream: #f);
        unless (priority)
          error("Unexpectedly missing priority!")
        end;
        let task = make(<task>, name: name,
                        priority: as(<symbol>, priority));
        add!(tasks, task)
      end
    end
  end;
```

```
  make(<task-list>, tasks: tasks, filename: filename)
end function load-task-list;
```

A task list manager using command tables

This section contains the complete source code of the task list manager when command tables have been used to implement the menu system, rather than explicit menu gadgets. To load this code into the environment, choose **Tools > Open Example Project** from any window in the environment. The code in this section can be loaded by choosing Task List 2 in the Documentation category of the Open Example Project dialog.

The command tables used in this implementation are described in Using Command Tables. You should refer to Improving The Design and Adding Callbacks to the Application, for a full description of the rest of the code shown here. Note that, apart from code specific to command tables and callbacks, the code listed in this section is a repeat of code listed in *A task list manager using menu gadgets*.

Contents of the file *frame.dylan* :

```
Module:    task-list
Synopsis:  Task List Manager.
Author:    Functional Objects, Inc.
Copyright:    Original Code is Copyright (c) 1995-2004 Functional Objects, Inc.
              All rights reserved.
License:      See License.txt in this distribution for details.
Warranty:     Distributed WITHOUT WARRANTY OF ANY KIND

define constant $priority-items
  = #(#("Low", #"low"),
      #("Medium", #"medium"),
      #("High", #"high"));

define frame <task-frame> (<simple-frame>)
  slot frame-task-list :: <task-list> = make(<task-list>);

  // Note: no definition of menu buttons in this implementation,
  // See definition of command tables instead.

  // definition of buttons
  pane add-button (frame)
    make(<push-button>, label: "Add task",
     command: frame-add-task,
     activate-callback: method (gadget) frame-add-task(frame) end);
  pane remove-button (frame)
    make(<push-button>, label: "Remove task",
     command: frame-remove-task,
         activate-callback:  method (gadget) frame-remove-task(frame) end);
  pane open-button (frame)
    make(<push-button>, label: "Open file",
     command: open-file,
         activate-callback: method (gadget) open-file(frame) end);
  pane save-button (frame)
    make(<push-button>, label: "Save file",
     command: save-file,
         activate-callback: method (gadget) save-file(frame) end);
```

```
  // definition of radio box
  pane priority-box (frame)
    make(<radio-box>,
     items: $priority-items,
     orientation: #"horizontal",
     label-key: first,
     value-key: second,
     value: #"medium",
     activate-callback: method (gadget) not-yet-implemented(frame) end);

  // definition of tool bar
  pane task-tool-bar (frame)
    make(<tool-bar>,
         child: horizontally ()
                  frame.open-button;
                  frame.save-button;
                  frame.add-button;
                  frame.remove-button
                end);

  // definition of status bar
  pane task-status-bar (frame)
    make(<status-bar>, label: "Task Manager");

  // definition of list
  pane task-list (frame)
    make (<list-box>,
          items: frame.frame-task-list.task-list-tasks,
          label-key: task-name,
          lines: 15,
          value-changed-callback: method (gadget) note-task-selection-change(frame) end);

  // main layout
  pane task-layout (frame)
    vertically ()
      frame.task-list;
      frame.priority-box;
    end;

  // activation of frame elements
  layout (frame) frame.task-layout;
  tool-bar (frame) frame.task-tool-bar;
  status-bar (frame) frame.task-status-bar;
  command-table (frame) *task-list-command-table*;

  // frame title
  keyword title: = "Task List Manager";
end frame <task-frame>;

define method initialize
    (frame :: <task-frame>, #key) => ()
  next-method();
  refresh-task-frame(frame);
end method initialize;

define method prompt-for-task
   (#key title = "Type text of new task", owner)
 => (name :: false-or(<string>),
```

```
     priority :: false-or(<priority>))
  let task-text
    = make(<text-field>,
           label: "Task text:",
           activate-callback: exit-dialog);
  let priority-field
    = make(<radio-box>,
           items: $priority-items,
           label-key: first,
           value-key: second,
           value: #"medium");
  let frame-add-task-dialog
    = make(<dialog-frame>,
           title: title,
           owner: owner,
           layout: vertically ()
                     task-text;
                     priority-field
                   end,
           input-focus: task-text);
  if (start-dialog(frame-add-task-dialog))
    values(gadget-value(task-text), gadget-value(priority-field))
  end
end method prompt-for-task;

define function not-yet-implemented (frame :: <task-frame>) => ()
  notify-user("Not yet implemented!", owner: frame)
end function not-yet-implemented;

define method start-task () => ()
  let frame
    = make(<task-frame>);
  start-frame(frame);
end method start-task;

define method frame-add-task (frame :: <task-frame>) => ()
  let task-list = frame-task-list(frame);
  let (name, priority) = prompt-for-task(owner: frame);
  if (name & priority)
    let new-task = make(<task>, name: name, priority: priority);
    add-task(task-list, new-task);
    refresh-task-frame(frame);
    frame-selected-task(frame) := new-task
  end
end method frame-add-task;

define method frame-remove-task (frame :: <task-frame>) => ()
  let task = frame-selected-task(frame);
  let task-list = frame-task-list(frame);
  if (notify-user(format-to-string
                    ("Really remove task %s", task.task-name),
                  owner: frame, style: #"question"))
    frame-selected-task(frame) := #f;
    remove-task(task-list, task);
    refresh-task-frame(frame)
  end
end method frame-remove-task;
```

```
define method frame-selected-task
    (frame :: <task-frame>) => (task :: false-or(<task>))
  let list-box = task-list(frame);
  gadget-value(list-box)
end method frame-selected-task;

define method frame-selected-task-setter
    (task :: false-or(<task>), frame :: <task-frame>)
 => (task :: false-or(<task>))
  let list-box = task-list(frame);
  gadget-value(list-box) := task;
  note-task-selection-change(frame);
  task
end method frame-selected-task-setter;

define method refresh-task-frame
    (frame :: <task-frame>) => ()
  let list-box = frame.task-list;
  let task-list = frame.frame-task-list;
  let modified? = task-list.task-list-modified?;
  let tasks = task-list.task-list-tasks;
  if (gadget-items(list-box) == tasks)
    update-gadget(list-box)
  else
    gadget-items(list-box) := tasks
  end;
  command-enabled?(save-file, frame) := modified?;
  note-task-selection-change(frame);
end method refresh-task-frame;

define method note-task-selection-change
    (frame :: <task-frame>) => ()
  let task = frame-selected-task(frame);
  if (task)
    frame.priority-box.gadget-value := task.task-priority;
  end;
  command-enabled?(frame-remove-task, frame) := task ~= #f;
end method note-task-selection-change;

define method open-file
    (frame :: <task-frame>) => ()
  let task-list = frame-task-list(frame);
  let filename
    = choose-file(frame: frame,
                  default: task-list.task-list-filename,
                  direction: #"input");
  if (filename)
    let task-list = load-task-list(filename);
    if (task-list)
      frame.frame-task-list := task-list;
      refresh-task-frame(frame)
    else
      notify-user(format-to-string("Failed to open file %s", filename),
                  owner: frame)
    end
  end
end method open-file;
```

```
define method save-file
    (frame :: <task-frame>) => ()
  let task-list = frame-task-list(frame);
  if (task-list.task-list-modified?)
    save-as-file(frame, filename: task-list.task-list-filename)
  end
end method save-file;

define method save-as-file
    (frame :: <task-frame>, #key filename) => ()
  let task-list = frame-task-list(frame);
  let filename
    = filename
        | choose-file(frame: frame,
                      default: task-list.task-list-filename,
                      direction: #"output");
  if (filename)
    if (save-task-list(task-list, filename: filename))
      frame.frame-task-list := task-list;
      refresh-task-frame(frame)
    else
      notify-user(format-to-string
                    ("Failed to save file %s", filename),
                  owner: frame)
    end
  end
end method save-as-file;

define function about-task (frame :: <task-frame>) => ()
  notify-user("Task List Manager", owner: frame)
end function about-task;

define method exit-task (frame :: <task-frame>) => ()
  let task-list = frame-task-list(frame);
  save-file(frame);
  exit-frame(frame)
end method exit-task;

define function make-keyboard-gesture
    (keysym :: <symbol>, #rest modifiers)
 => (gesture :: <keyboard-gesture>)
  make(<keyboard-gesture>, keysym: keysym, modifiers: modifiers)
end function make-keyboard-gesture;

// Definition of the File menu

define command-table *file-command-table* (*global-command-table*)
  menu-item "Open" = open-file,
    accelerator:   make-keyboard-gesture(#"o", #"control"),
    documentation: "Opens an existing file.";
  menu-item "Save" = save-file,
    accelerator:   make-keyboard-gesture(#"s", #"control"),
    documentation: "Saves the current file to disk.";
  menu-item "Save As..." = save-as-file,
    documentation: "Saves the current file with a new name.";
  separator;
  menu-item "Exit" = exit-task,
    accelerator:   make-keyboard-gesture(#"f4", #"alt"),
```

```
    documentation: "Exits the application.";
end command-table *file-command-table*;

// Definition of the Edit menu

define command-table *edit-command-table* (*global-command-table*)
  menu-item "Cut" = not-yet-implemented,
    accelerator:   make-keyboard-gesture(#"x", #"control"),
    documentation: "Cut the selection to the clipboard.";
  menu-item "Copy" = not-yet-implemented,
    accelerator:   make-keyboard-gesture(#"c", #"control"),
    documentation: "Copy the selection to the clipboard.";
  menu-item "Paste" = not-yet-implemented,
    accelerator:   make-keyboard-gesture(#"v", #"control"),
    documentation: "Paste the selection in the clipboard at the current position.";
end command-table *edit-command-table*;

// Definition of the Task menu

define command-table *task-command-table*
    (*global-command-table*)
  menu-item "Add..." = frame-add-task,
    accelerator: make-keyboard-gesture(#"a", #"control", #"shift"),
    documentation: "Add a new task.";
  menu-item "Remove" = frame-remove-task,
    accelerator: make-keyboard-gesture(#"d", #"control", #"shift"),
    documentation: "Remove the selected task from the list.";
end command-table *task-command-table*;

// Definition of the Help menu

define command-table *help-command-table* (*global-command-table*)
  menu-item "About" = about-task,
    accelerator:   make-keyboard-gesture(#"f1"),
    documentation: "Display information about the application.";
end command-table *help-command-table*;

// Definition of the overall menu bar

define command-table *task-list-command-table*
    (*global-command-table*)
  menu-item "File" = *file-command-table*;
  menu-item "Edit" = *edit-command-table*;
  menu-item "Task" = *task-command-table*;
  menu-item "Help" = *help-command-table*;
end command-table *task-list-command-table*;


define method main (arguments :: <sequence>) => ()
  // handle the arguments
  start-task();
end method main;

begin
  main(application-arguments()) // Start the application!
end;
```

Contents of the file *task-list.dylan* :

The file *task-list.dylan* is identical to the listing shown in *A task list manager using menu gadgets*, and so is not repeated here.

CHAPTER

ELEVEN

INDICES AND TABLES

- genindex
- search