CHAPTER 6

Laying Out Applications

In the previous chapter, we looked at creating and setting up windows to provide a framework for an application. In this chapter, we will begin looking at how to add other widgets to the newly created windows.

PHTP-GTK provides various specialty widgets that give you control over not only their placement within the container, but also how they react when the container is resized. This chapter introduces the `GtkFrame`, `GtkVBox`, `GtkHBox`, `GtkButtonBox`, `GtkTable`, `GtkFixed`, and `GtkNotebook` widgets. We will also begin to implement the Crisscott PIMS application, focusing on setting up the application so that the functional pieces can just be dropped into place.

The Sample Application Layout

The Crisscott PIMS application needs to have the following elements as part of the main window:

- A menu for opening and saving files, changing settings, and so on
- A toolbar for quick access to commonly used commands
- An area for navigating to individual products
- An area to display a summary of the current inventory data
- An area to display a summary of the currently selected product
- An area to display news and important messages
- An area for adding and editing product information
- An area for displaying status messages

Adding these eight areas to the window is more involved than it appears. We need to do more than simply decide which pieces go where. We need to consider the relative size and importance of each section, as well as whether or not an area should be able to shrink or grow with the application. These decisions are often made by considering what the final product might look like and how it will be used.

For instance, what type of information will the news section include? Will it just show headlines, or will it also display article bodies? If the news section will show summaries of the articles, or even the full articles, it will need to be much larger than it would if it were to simply show headlines. This decision of how to size and place the news section will have a significant

impact on the user experience. Whereas a large prominently displayed news section will draw the user's attention, it is more likely that Crisscott, Inc. wants the suppliers to remain focused on the product information instead of the news and updates.

Figure 6-1 shows one possible skeleton layout for the PIMS application. Different interpretations of the importance and role of each section of the application will result in widely varied final products, but the layout in Figure 6-1 is what we will try to achieve by the end of this chapter.

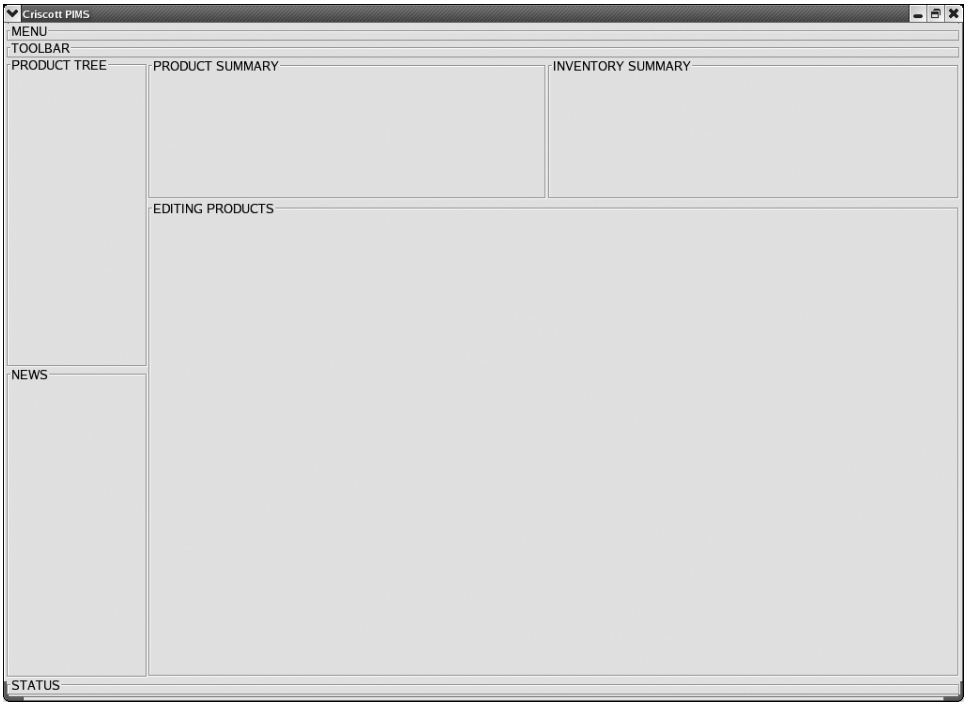

Figure 6-1. *One possible layout for the PIMS application*

Discussion of application layout revolves mainly around containers, which provide the framework and structure for the application by holding and positioning the other elements on the screen. The choice of which container to use depends greatly on the needs of the application and can vary from one section of the application to another. Before deciding which route to go, you must determine levels of priority for the program. Influencing factors are the expected size of the application's window, whether or not certain elements should shrink or grow when the window changes size, and the space between elements on the screen.

Frames

A `GtkFrame` widget is a simple container that exists mostly for decoration purposes, but also allows related content to be grouped together on the screen. `GtkFrame` widgets are bin containers, meaning they can contain only one child widget. As far as providing structure for an application, a `GtkFrame` plays no role. However, it is useful for enhancing usability.

The GtkFrame has a thin border and a label at the top. The border helps to group the frame's contents as a single unit. The label is most often used to describe the contents of the frame.

Most containers do not take up space or have any size without having children added first, but because GtkFrame has a border and a label, it is able to take up screen space on its own. This makes GtkFrame an excellent choice as a placeholder during development. Throughout the next several sections, we'll use frames to hold the place of application elements while we develop the layout. Figure 6-2 shows GtkFrames in action as part of the PHP-GTK 2 demo application.

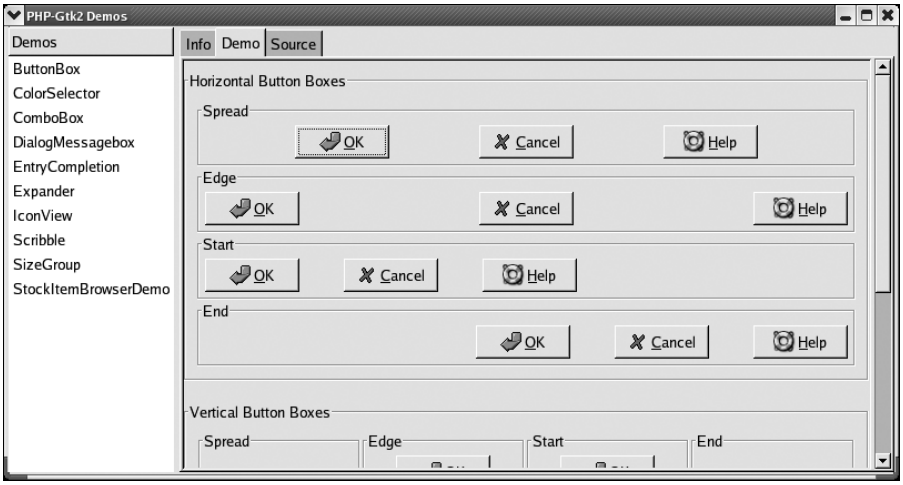

Figure 6-2. *GtkFrames in action*

Creating and using a GtkFrame is simple. The constructor takes the label to be displayed. You can add a child with the add method.

Setting the Label Section

While frames may be simple, they are flexible. Even though technically GtkFrame is a bin container, it is really more of a bin-and-a-half, because the label section of the frame can be set with either a string value or a widget.

You can add any type of widget as the frame's label. This opens up a whole world of possibilities—both good and bad. For example, adding a button to the frame's label area can increase the functionality of an application but can also distort the layout. So, you need to understand exactly what may happen if you add a widget as the frame's label.

To add a widget as the label, call set_label_widget and pass the new label widget. If all you want is a simple string, you can call set_label, passing the new label string. The following line uses set_label to change a frame's label from its current value to the string 'Buttons'.

```
$frame->set_label('Buttons');
```

And to set the frame's label to a GtkButton, you can use the following:

```
$frame->set_label_widget(new GtkButton('Click Me'));
```

You can also move the label around within the top edge of the frame, aligning it both horizontally and vertically. By using set_label_align, you can move the label left or right and up or down. The two values passed to set_label_align determine the label's relative distance from the upper-left corner of the frame. The values must be decimal numbers between zero and one, inclusive. The numbers represent a percentage of the frame's width. If .5 is passed as the first argument, the label will appear centered along the top edge of the frame. If 1 is passed, the label will be aligned to the right. The second argument determines the vertical positioning. A value of 0 means the label's top edge will align with the top border. A value of 1 means that the label's bottom edge will align with the top border. If the alignment is not specifically set, the frame's label will be left aligned and vertically centered. This is equivalent to calling set_label_align(0, .5).

Setting the Border Type

Aside from allowing you to change the label and reposition it, GtkFrame also lets you control the border that is displayed. You can set five border types:

- Gtk::SHADOW_NONE: No border will be visible.
- Gtk::SHADOW_IN: The border is beveled inward.
- Gtk::SHADOW_OUT: The border is beveled outward.
- Gtk::SHADOW_ETCHED_IN: A thin, inward border surrounds the frame.
- Gtk::SHADOW_ETCHED_OUT: A thin, outward border surrounds the frame.

To set the frame's border, use set_shadow_type. By default, a frame's border is set to Gtk::SHADOW_ETCHED_IN. Figure 6-3 shows five simple frames with varying borders and label positions.

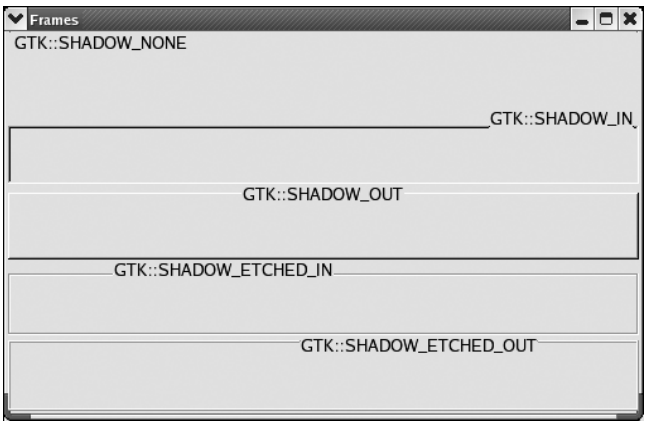

Figure 6-3. *Several GtkFrames with different alignments and border types*

Boxes

The box container is one of the simplest types of containers available to PHP-GTK, and it is also one of the most versatile. PHP-GTK offers three types of box containers, each designed to fulfill a specific need:

- `GtkVBox`: Used to display widgets in a vertical column.
- `GtkHBox`: Used to display widgets in a horizontal row.
- `GtkButtonBox`: Used to display a set of buttons.

Creating Vertical and Horizontal Boxes

`GtkVBox` and `GtkHBox` are probably the two most commonly used containers in PHP-GTK. Their simplicity and ability to be nested makes them extremely powerful tools for laying out an application. Part of the power in these two classes is that they are not picky about what types of widgets are added to them, meaning any widget that does not require a specific type of parent can be added to a `GtkVBox` or `GtkHBox`. This includes buttons, labels, trees, text entries, and even other boxes. In fact, after creating the main application window, most applications immediately add either a `GtkVBox` of `GtkHBox` to the main `GtkWindow`, and then add boxes within the box. Actually, that is the fastest way to achieve the layout shown earlier in Figure 6-1.

Let's start from the outside and work our way in. Remember from the previous chapter that almost all applications start with a `GtkWindow`. Also remember that `GtkWindow` is a bin, meaning that it can have only one child directly. Therefore, to place more than one widget inside a window, the window's only child must be a container. The container can then take one or more children, which themselves can be containers, and so on. Nesting containers allows an application to be built up from a single-child widget to a collection of containers and children that work together as an application. The quickest way to fill a window with multiple widgets is to add a widget that is not a bin—namely, a `GtkVBox` or `GtkHBox`.

Look at Figure 6-1 again. We can break down the application into four main rows. The first two are the menu and toolbar. The last row is the status bar. The third row is everything in between. While rows tend to make you think of horizontal displays, rows are actually created by stacking items one on top of another vertically. Since the rows are elements stacked vertically, they must be packed into a `GtkVBox`.

Packing Widgets into a Box

While to most people, the terms *packing* and *adding* may have the same meaning, in the PHP-GTK world, *packing* has a slightly different connotation. Think about how you get ready for a vacation. Sure you add items to your suitcase, but things are not just haphazardly thrown in. They are placed neatly and with a purpose. The items are packed, not just added. The same thing can be said about adding widgets to a box. They can be simply added, but more commonly they are packed. This means that their order is carefully considered and thought is given to how they will react within the container.

You pack widgets into a box by using the `pack_start` method. Each call to `pack_start` adds its widget to the container, starting at the top and working down toward the bottom of the box. The first call to `pack_start` places its widget at the top of the box, the second call to `pack_start` adds the widget passed as the second element from the top, and so on. There is no limit to how many elements can be added this way.

A similar method is called pack_end. It works in the same way as pack_start, except that the widgets are packed from the bottom up. The first element added with pack_end will be the last element in the container, the second element added with pack_end will be the second to last element in the container, and so on.

Regardless of what order the method calls are made, elements added to a GtkVBox with pack_start will always appear above elements added with pack_end.

Listing 6-1 shows the code that will create four rows within our sample application's window. GtkFrame widgets are used as temporary placeholders. The frames are packed using the pack_start method.

Listing 6-1. *Creating Rows Within a Window*

```
<?php
class Crisscott_MainWindow extends GtkWindow {

    public function __construct()
    {
        // Call the parent constructor.
        parent::__construct();

        // Set the size of the window.
        $this->set_size_request(500, 300);
        // Position the window.
        $this->set_position(Gtk::WIN_POS_CENTER);
        // Give the window a title.
        $this->set_title('Criscott PIMS');

        // Add window's children
        $this->_populate();

        // Make the window expand to the limits of the screen.
        $this->maximize();

        // Set up the application to close cleanly.
        $this->connect_object('destroy', array('Gtk', 'main_quit'));
    }

    private function _populate()
    {
        // Create a vertical box.
        $vb1 = new GtkVBox();

        // Pack some frames in the box.
        $vb1->pack_start(new GtkFrame('MENU'), false, false, 0);
        $vb1->pack_start(new GtkFrame('TOOLBAR'), false, false, 0);
        $vb1->pack_start(new GtkFrame('MAIN'), true, true, 0);
        $vb1->pack_start(new GtkFrame('STATUS'), false, false, 0);
```

```
        // Add the box to the window.
        $this->add($vb1);
    }
    // ...
}
?>
```

A closer look at the calls to `pack_start` shows that there are four arguments passed, instead of just the widget that is added. The following three optional parameters allow `pack_start` and `pack_end` to be more powerful than the generic `add` method:

- `expand`: A Boolean value that determines whether or not the widget will take up all of the space available to it.
- `fill`: A Boolean value that determines whether or not the widget will shrink and grow along with its parent.
- `padding`: The amount of padding, in pixels, that will surround the child widget.

As I mentioned, packing implies consideration of how widgets will react within their parent container. A child widget will be given the opportunity to interact with its parent container at two times: first, when the child is added, and then when the container is resized. When these events occur, the children of the container will be given the opportunity to automatically adjust their size or the amount of space each occupies within the box.

When a widget is packed into a box, by default, it tries to take up as much room as is available. Also by default, the widget will try to resize itself to fill in that space. Taking up space and filling in space are not necessarily the same thing.

Taking up space simply means that the widget reserves a given amount of space for itself. It tells other widgets, "This area is mine. Stay out!"

Filling up the space means that the widget shrinks or grows to fit within the space it has reserved. If the widget is only 50 pixels square, but the space available is 100 pixels wide by 200 pixels high, the widget will try to expand to fit the 100×200 space. When more than one widget wants to take up all the space available, just as with children on a long car trip, the parent steps in. The parent splits up the available space evenly among the children that want to occupy as much screen space as possible.

The second and third arguments to the `pack_start` and `pack_end` methods determine whether a widget may reserve the maximum amount of available space, known as *expanding*, or resize itself to fill the space allotted to it, known as *filling*, respectively. By default, both of these arguments are passed as `true`. This means that when a widget is packed into a box, it will be automatically resized to be as big as the box and other children will allow.

However, in cases where a widget has been specifically sized, allowing the widget to fill the available space may have undesired effects. Passing `false` as the second argument tells the container that the given child widget does not want more space than it needs to display its content. Passing `false` as the third argument tells the application that the widget being packed should not be resized to fill the space available. Both of these requests are honored when the child is packed and when the container is resized. If when the child was packed, it was allowed to fill the available space, it will resize itself again when the container is resized and more (or less) space is available.

In Listing 6-1, most parts of the application should take up as little space as needed to allow for the more important pieces to be the focus of the application. That is why only the frame labeled `MAIN` is allowed to expand and fill the space available to it. All other sections of the application are forced to remain just large enough to show their contents.

■ **Tip** All boxes have a `set_homogeneous` method, which makes the box's children share the space equally. This is the same as setting all of the fill arguments (the third argument) to `true` for every widget packed into the box. If `set_homogeneous` is passed `true`, all children will share the space equally, regardless of what options were passed when the widget is packed.

The final argument that may be passed to `pack_start` and `pack_end` is an integer that defines the amount of padding that will surround the child widget. The padding that surrounds a child is given in pixels. In a box container, padding exists only in the direction of the box. So, for a `GtkVBox`, padding will appear only above and below the child widget. For a `GtkHBox`, padding will appear only to the left and right of the widget. Padding between widgets does not collapse. If widget A is packed with a padding of 10 pixels and widget B is packed with a padding of 20 pixels, the total space between the two widgets will be 30 pixels. Figure 6-4 shows several widgets packed into a `GtkHBox` with different padding values. In Listing 6-1, the maximum amount of screen space is used by setting all of the padding to 0.

Figure 6-4. *Different padding values*

Nesting Boxes

Nesting boxes is the practice of putting one box inside another box. Box classes are highly specialized, as this makes managing their children easier. While specialization may be good for managing children, it makes layout a little more difficult. For instance, creating rows *or* columns is easy, but creating rows *and* columns with boxes requires a little patience and planning.

To create rows within a column, a `GtkVBox` must be placed inside a `GtkHBox`. To create columns within a row, simply do the opposite—put a `GtkHBox` inside `GtkVBox`. While this may not sound very complicated, it can become quite difficult to track all the different boxes when the levels go beyond one or two deep.

Listing 6-2 shows a reworked version of the `_populate` method used in Listing 6-1. The new method uses `GtkVBox` and `GtkHBox` widgets nested inside one another to add more sections to the application.

Listing 6-2. *Nesting Boxes*

```
<?php
//...
private function _populate()
{
```

```
        // Create several boxes for nesting.
        $vb1 = new GtkVBox();
        $vb2 = new GtkVBox();
        $vb3 = new GtkVBox();
        $hb1 = new GtkHBox();
        $hb2 = new GtkHBox();

        // Add some frames to the first vBox.
        $vb1->pack_start(new GtkFrame('MENU'), false, false, 0);
        $vb1->pack_start(new GtkFrame('TOOLBAR'), false, false, 0);

        // Nest an hBox inside the vBox.
        $vb1->pack_start($hb1);

        // Add another frame after the hBox.
        $vb1->pack_start(new GtkFrame('STATUS'), false, false, 0);

        // Nest a vBox inside the hBox
        $hb1->pack_start($vb2, false, false, 0);

        // Nest another vBox inside the hBox.
        $hb1->pack_start($vb3);

        // Pack some frames into one of the nested vBoxes.
        $vb2->pack_start(new GtkFrame('PRODUCT TREE'));
        $vb2->pack_start(new GtkFrame('NEWS'));

        // Set the size of the vBox.
        $vb2->set_size_request(150, -1);

        // Nest an hBox inside one of the nested vBoxes.
        $vb3->pack_start($hb2, false, false, 0);

        // Add a frame after the nested box.
        $vb3->pack_start(new GtkFrame('EDITING PRODUCTS'));

        // Add a few frames to the nested hBox.
        $hb2->pack_start(new GtkFrame('PRODUCT SUMMARY'));
        $hb2->pack_start(new GtkFrame('INVENTORY SUMMARY'));

        // Set the nested hBox's size.
        $hb2->set_size_request(-1, 150);

        // Add the vBox to the window.
        $this->add($vb1);
    }
    // ...
    ?>
```

Figure 6-5 shows the new layout using nested boxes.

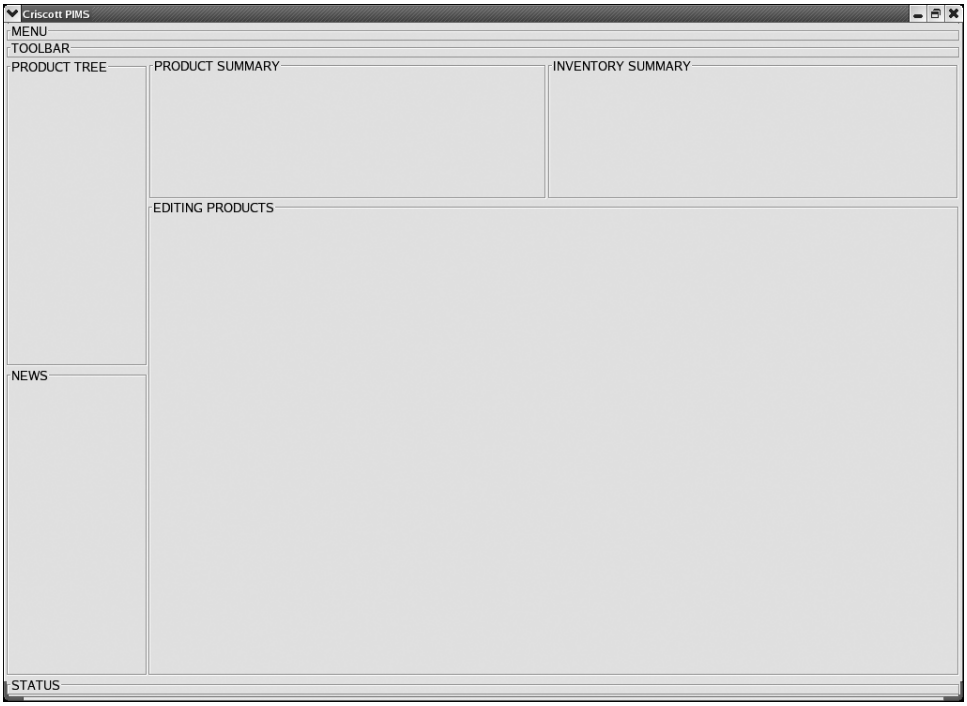

Figure 6-5. *Using nested boxes for layout*

In Listing 6-2, the third row, which used to have a title of `MAIN`, is now split into two columns. The third row was changed from a simple frame to a `GtkHBox`. A `GtkHBox` was used because the application needs to display things next to each other horizontally. By placing two `GtkVBox` widgets next to each other, you can create columns.

The items packed into the `GtkVBox` widgets will be separated distinctly into items on the left and items on the right. Notice that the first `GtkVBox` packed is told not to expand or fill. The box also has its size explicitly set using `set_size_request`. The combination of proper packing and `set_size_request` ensures that the box will appear just as you intended. The box will not shrink or grow within its parent box, and its size is not dependent on other children in the container.

Within each of the two columns are two rows. Since `$vb2`, the column on the left, is very simple, two frames are added directly to the `GtkVBox`. `$vb3`, on the other hand is slightly more

complex. Its first row contains two columns again. Once again, the rows are created by nesting GtkHBox widgets inside the GtkVBox.

Pay close attention to when and how the extra arguments for pack_start are used in this listing. Try switching the values from true to false and vice versa, and add a little (or a lot) of padding here and there. Changing one Boolean value can have a huge impact on the rest of the application.

■ **Note** In Listing 6-2, because the child box was packed into a GtkHBox, the expand and fill arguments apply only in the horizontal direction. The vertical expand and fill arguments trickle down from the parent when it was packed into the GtkVBox. Since the parent was told that it may expand and fill within the GtkVBox, the child will shrink and grow vertically depending on the size of its parent.

Button Boxes

The application isn't quite ready to start adding buttons, but this is a good point to discuss button boxes. GtkButtonBox is a descendant of GtkBox, just like GtkVBox and GtkHBox. The difference is that button boxes can use special layouts that are often helpful when displaying a group of buttons.

There are two varieties of GtkButtonBox: GtkVButtonBox and GtkHButtonBox. Each type functions in the same way as its regular box counterpart. Widgets are packed into a GtkButtonBox using pack_start and pack_end.

The advantage to using a button box is in the set_layout method. set_layout determines how the buttons will be shown within the box. The layout can be one of four values:

- spread: The buttons will be distributed evenly in the box.
- edge: The buttons will be as far apart as possible. The first button will be against the beginning edge of the box, and the last button will be against the ending edge of the box. All of the buttons in between will be as far apart from each other as they can.
- start: The buttons appear toward the starting edge of the box.
- end: The buttons appear toward the ending edge of the box.

For a better understanding of how to use button boxes, take a look at phpgtk2-demo.php, located in the demos directory of the PHP-GTK source. The button box demo shows the many different layouts of a GtkButtonBox, as you can see in Figure 6-6.

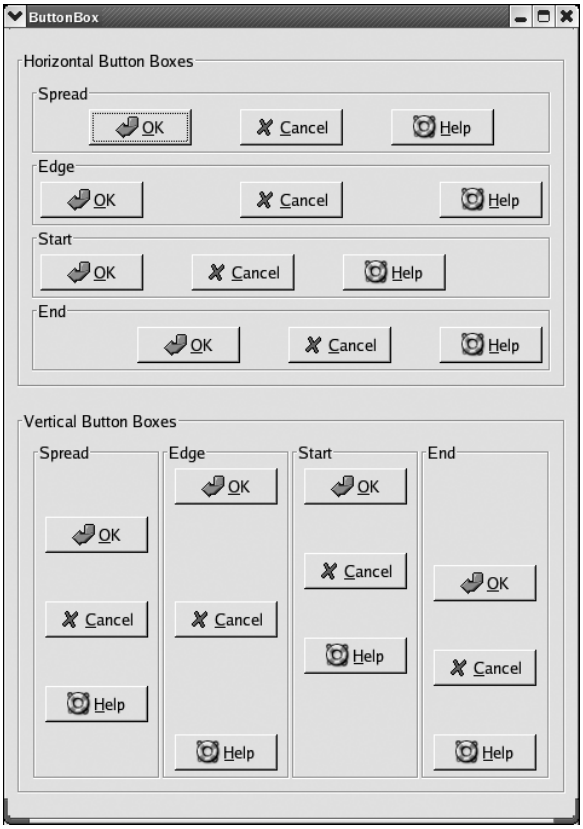

Figure 6-6. *The different layouts of button boxes*

Tables

It is often possible to achieve the desired layout of an application using nested boxes, but setting up the application and keeping things organized gets more and more difficult as the levels of nesting get deeper. As with most things in life, there is more than one way to reach the same result.

`GtkTable` is a container class designed specifically for laying out an application, unlike its HTML counterpart, which is designed for organizing data. A `GtkTable` can be used to more easily display widgets in rows and columns. A `GtkTable` container is similar to a table in HTML. It has rows and columns made up of individual cells, and each cell can span more than one row and/or column. While the contents of each cell are independent, the dimensions of a row or column are determined by the largest cell in that row or column.

Listing 6-3 is an implementation of the `_populate` method from the earlier listings, but it uses a `GtkTable` instead of nested boxes. At first glance, the new version appears to be quite complicated because of all of the integers floating around, but once these numbers are explained, the picture clears up rather quickly.

Listing 6-3. *Laying Out an Application Using GtkTable*

```
<?php
// ...
private function _populate()
{
    // Create a new table with 5 rows and 3 columns.
    $table = new GtkTable(5, 3);

    // Make it easier to set both expand and fill at the same time.
    $expandFill = Gtk::EXPAND|Gtk::FILL;

    // Attach two frames to the table.
    $table->attach(new GtkFrame('MENU'), 0, 2, 0, 1, $expandFill, 0, 0, 0);
    $table->attach(new GtkFrame('TOOLBAR'), 0, 2, 1, 2, $expandFill, 0, 0, 0);

    // Create a new frame and set its size.
    $productTree = new GtkFrame('PRODUCT TREE');
    $productTree->set_size_request(150, -1);

    // Attach the frame to the table.
    $table->attach($productTree, 0, 1, 2, 3, 0, $expandFill, 0, 0);

    // Create a new frame and set its size.
    $news = new GtkFrame('NEWS');
    $news->set_size_request(150, -1);

    // Attach the frame to the table.
    $table->attach($news, 0, 1, 3, 4, 0, $expandFill, 0, 0);

    // Create a subtable.
    $table2 = new GtkTable(2, 2);

    // Create a new frame and set its size.
    $productSummary = new GtkFrame('PRODUCT SUMMARY');
    $productSummary->set_size_request(-1, 150);

    // Attach the frame to the subtable.
    $table2->attach($productSummary, 0, 1, 0, 1, $expandFill, 0, 1, 1);

    // Create a new frame and set its size.
    $inventorySummary = new GtkFrame('INVENTORY SUMMARY');
    $inventorySummary->set_size_request(-1, 150);

    // Attach the frame to the subtable.
    $table2->attach($inventorySummary, 1, 2, 0, 1, $expandFill, 0, 1, 1);
    $table2->attach(new GtkFrame('EDITING PRODUCTS'), 0, 2, 1, 2,
                    $expandFill, $expandFill, 1, 1);
```

```
    // Attach the subtable to the main table.
    $table->attach($table2, 1, 2, 2, 4, $expandFill, $expandFill, 0, 0);

    // Attach another frame to the main table.
    $table->attach(new GtkFrame('STATUS'),  0, 2, 4, 5, $expandFill, 0, 0, 0);

    // Add the table to the window.
    $this->add($table);
}
// ...
?>
```

Figure 6-7 gives an idea of what the end result of this section looks like. Notice that even though the code has changed, the result is the same.

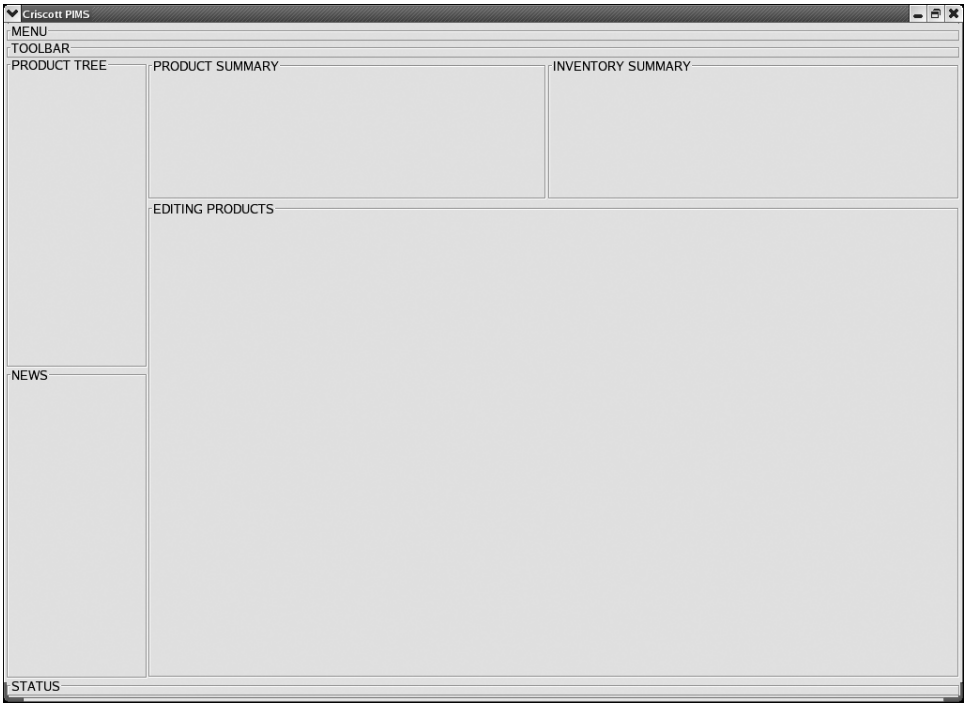

Figure 6-7. *The Crisscott PIMS application using a GtkTable for layout*

Constructing the Table

The first step in using a `GtkTable` is to create a new instance. The constructor for a `GtkTable` widget takes three optional parameters.

- `rows`: The number of rows the table should have initially. The value must be an integer between `1` and `65535`, inclusive. It defaults to `1`.
- `columns`: The number of columns the table should have initially. The value must be an integer between `1` and `65535`, inclusive. It defaults to `1`.
- `homogeneous`: A Boolean value that if set to `true` will force all cells to be the same size. It defaults to `false`.

The first two parameters are the number of rows and columns that the table should have. If at some point the table needs an additional row or column, you can easily change the dimensions of the table using `resize`. `resize` sets the new number of rows and columns to the two integer values passed. In both the constructor and `resize`, the first argument is the number of rows and the second argument is the number of columns.

■ **Note** It isn't strictly necessary to resize the table when a new row or column is added. If a child is added into a cell that doesn't exist, the row and/or column needed for that cell will be added automatically.

The final argument for the `GtkTable` constructor is the Boolean `homogeneous` value. This value defaults to `false` and has the same effect that `set_homogeneous` has for boxes. If the homogeneous argument is set to `true`, all cells in the table will be the same size. In Listing 6-3, the table is created with five rows and three columns, for a total of fifteen cells. No value is passed to tell the table whether or not the cells should be homogeneous, so they will default to being as tall as the tallest cell in their row and as wide as the widest cell in their column. The height and width of the largest cell in a row or column are determined by the cell's content.

Attaching Children

The next step in laying out the application is adding children to the table. Just as boxes have their own terminology for adding children, so does `GtkTable`. In a table, children are not added—they are *attached*, and this is accomplished with the `attach` method.

Attaching a child gives greater control over the location and the way the child reacts within the table. The first priority in attaching a child to a table is putting it in the right place. When putting a widget in a table, all four sides of the widget must be specifically positioned. The `attach` method takes a whopping nine arguments:

- `child`: The widget to be added to the table.
- `col_start`: The starting column to attach the child to.
- `col_end`: The ending column to attach the child to.
- `row_start`: The starting row to attach the child to.
- `row_end`: The ending row to attach the child to.
- `x_options`: Whether or not the child should expand and fill in the x direction.
- `y_options`: Whether or not the child should expand and fill in the y direction.

- `x_padding`: The amount of padding on the left and right of the child widget.
- `y_padding`: The amount of padding on the top and bottom of the child widget.

The first argument to `attach` is the widget that will be added to the table. The other arguments specify the child's placement within the cell, whether it expands and fills, and its padding.

Cell Placement

After the `child` argument, the next four arguments to the `attach` method correspond to the four sides of the child widget. The `col_start` argument tells the table in which column the left side of the child should start. If the `col_start` argument is `0`, the child will be in the leftmost column. If the `col_start` argument is `1`, the child will start in the second column. The `col_end` argument tells the table where the child should stop. The value passed is one greater than the column in which the widget should end.

For instance, if a child should be in only the first column of a table, the `col_start` and `col_end` arguments should be `0` and `1`. If a child should span the second and third columns, the `col_start` and `col_end` arguments should be `1` and `3`. This tells the table that the child should start in column 1 (rows and columns are indexed starting with 0, just like most things in programming) and end before column 3.

The `row_start` and `row_end` arguments to `attach` are similar to `col_start` and `col_end`, except they determine the row or rows that the child will occupy.

The first call to `attach` in Listing 6-3 places a `GtkFrame` in the first row of the table, spanning all three columns. This is done by telling the child to start in column 0 and end just before column 3. The child is also told to start in row 0 and end just before row 1. With these four values, you can place a child in any cell and have it span as many rows and/or columns as needed.

Expanding and Filling

Assigning a widget to a cell in a table is only half the goal. The other half involves explaining how the widget should react within the table. Similar to packing items in boxes, attaching widgets to a table also involves determining whether or not the child should expand and fill the maximum amount of space available.

With boxes, the space for each element is either part of a row (`GtkHBox`) *or* part of a column (`GtkVBox`). A table cell is the intersection of both a row *and* a column. Therefore, the expand and fill attributes need to be set for both the row, or x-axis, and column, or y-axis.

The `x_options` argument passed to `attach` tells the table whether the child should expand and/or fill the cell in the x direction. The value passed should be made up of one or more constant values. If the widget should expand but not fill the cell, the value should be `Gtk::EXPAND`. If the widget should fill the cell but not expand, the value should be set to `Gtk::FILL`. If the widget should both expand and fill the cell, the value should be `Gtk::EXPAND|Gtk::FILL`.

The `y_options` argument sets the same values for the y-axis. Passing 0 to either of these values tells the table not to allow the child to expand or fill the cell in that direction. By default, a child will expand and fill a cell in both directions. In Listing 6-3, the menu and toolbar frames are told to expand and fill their cells only along the x-axis. The product tree frame is told to expand and fill only along the y-axis. Because the product tree frame is set to 150 pixels wide and is told not to expand or fill on the x-axis, it will always remain 150 pixels wide. The height

of the frame is set to -1, which means that its height is not to be strictly controlled. Coupled with the expand and fill properties for the y-axis, this allows the frame to stretch when the window is resized.

Padding

The final task when attaching a widget to a table is setting the amount of padding that each cell should have. When packing a widget in a box, the padding is set equally on two sides to the value of the last argument passed to `pack_start` or `pack_end` (which two sides depends on the type of box). When attaching a widget to a table, padding can be set for both the x and y directions, just as with the expand and fill properties.

The `x_padding` and `y_padding` arguments passed to `attach` determine the x and y padding, respectively. If either of these values is omitted, the padding for that direction will default to 5 pixels.

Tables vs. Boxes

You can use tables and boxes in a similar manner to create similar output. Listing 6-3 even has a table nested inside another table to show how similar the two can be. Despite their similarities, `GtkTable` is often a better choice when setting up an application.

Using tables gives you more control over the placement of children within the application than creating the layout with boxes. With tables, it is possible to have a good idea of where the widgets will appear before the application starts up. Boxes usually require much more trial and error.

Tables also lend themselves better to more readable code. It is easy to tell where in the application a table cell will be by looking at the row and column to which it is attached. With boxes, it is usually more difficult and requires a little bit of tracing through the code.

While tables may have several advantages over boxes, they are not the only other choice. An alternative is to use a fixed container, as described next.

Fixed Containers

`GtkTable` is very effective in lining up widgets into rows and columns. But sometimes the relationships between children in the same row or column can be a problem, because the dimensions of a cell in a table are determined by the largest cell in a cell's row and column. That is where `GtkFixed` comes in.

Like `GtkTable`, `GtkFixed` allows for precise positioning, but it does not strictly align elements with each other. The elements in a `GtkFixed` container have no influence on one another. The height and width of a given child do not depend on another element, because each child is placed independently of the other children.

A `GtkFixed` widget is similar to a bulletin board. Each child is put in a specific location and stays there, somewhat oblivious to its surroundings. Free-form layout is quick and easy with a `GtkFixed` widget. Simply put a widget in its place, and that is it.

Listing 6-4 re-creates Figure 6-1, but this time uses a `GtkFixed` container instead of boxes or tables. The end result is exactly the same in appearance.

Listing 6-4. *Using GtkFixed to Lay Out the Application*

```
<?php
// ...
private function _populate()
{
    // Create a GtkFixed container.
    $fixed = new GtkFixed();

    // Create a frame, set its size, and put it in the fixed container.
    $menu = new GtkFrame('MENU');
    $menu->set_size_request(GDK::screen_width() - 10, -1);
    $fixed->put($menu, 0, 0);

    // Create a frame, set its size, and put it in the fixed container.
    $toolbar = new GtkFrame('TOOLBAR');
    $toolbar->set_size_request(GDK::screen_width() - 10, -1);
    $fixed->put($toolbar, 0, 18);

    // Create a frame, set its size, and put it in the fixed container.
    $pTree = new GtkFrame('PRODUCT TREE');
    $pTree->set_size_request(150, GDK::screen_height() / 2 - 54);
    $fixed->put($pTree, 0, 36);

    // Create a frame, set its size, and put it in the fixed container.
    $news = new GtkFrame('NEWS');
    $news->set_size_request(150, GDK::screen_height() / 2 - 54);
    $fixed->put($news, 0, GDK::screen_height() / 2 - 18);

    // Create a frame, set its size, and put it in the fixed container.
    $status = new GtkFrame('STATUS');
    $status->set_size_request(GDK::screen_width() - 10, -1);
    $fixed->put($status, 0, GDK::screen_height() - 72);

    // Create a frame, set its size, and put it in the fixed container.
    $pSummary = new GtkFrame('PRODUCT SUMMARY');
    $pSummary->set_size_request(GDK::screen_width() / 2 - 90, 150);
    $fixed->put($pSummary, 152, 36);

    // Create a frame, set its size, and put it in the fixed container.
    $iSummary = new GtkFrame('INVENTORY SUMMARY');
    $iSummary->set_size_request(GDK::screen_width() / 2 - 75, 150);
    $fixed->put($iSummary, GDK::screen_width() / 2 - 90 + 154, 36);
```

```
    // Create a frame, set its size, and put it in the fixed container.
    $edit = new GtkFrame('EDIT PRODUCTS');
    $edit->set_size_request(GDK::screen_width() - 150, GDK::screen_height() - 262);
    $fixed->put($edit, 152, 190);

    // Add the fixed container to the window.
    $this->add($fixed);
}
// ...
?>
```

To create this version of the application, each child widget needs to be sized and placed individually. This is different from the other examples. In the previous two listings, only a few widgets were specifically sized, and even then, it was either their height or width, not both.

With a `GtkFixed`, children are not able to react to their surroundings. Children cannot expand or fill an area. They simply get put into the container and remain there. Therefore, if a child should take up a certain amount of the screen space, its size must be explicitly set.

Putting Widgets in a Fixed Container

When you use boxes, you *pack* widgets, which implies that the widgets are added to the container one after another. When you use tables, you *attach* widgets, which implies that they become part of the table. When you use a fixed container, the widgets are *put*, which implies that a location was selected and the widget was placed in that specific spot.

To put a widget into a `GtkFixed` container, call the `put` method and pass the x and y coordinates for the upper-left corner of the child. The child will be put directly into the container at the location given. Its size will remain the same as it was before it was added. If the container is resized, the child will not change. It will still be *x* pixels from the left and *y* pixels from the top of the container.

In Listing 6-4, each element is sized and put individually. Calculating the position for a given element can be difficult and often requires some advance knowledge of the other children in the container.

Using Fixed Containers

Fine-grained control is the strong point of `GtkFixed`; however, as the name implies, flexibility is its weakness. Listing 6-4 is admittedly a poor use of `GtkFixed`. The PIMS application does not need such complete control over the application layout. Instead, it needs less control and more flexibility.

Run the application using the `_populate` method from Listing 6-4. When the application has loaded, try unmaximizing or resizing the window. The elements within the `GtkFixed` do not resize. As you can see in Figure 6-8, the product edit area quickly gets cut off when the window is resized even slightly smaller. This obviously is a bad design.

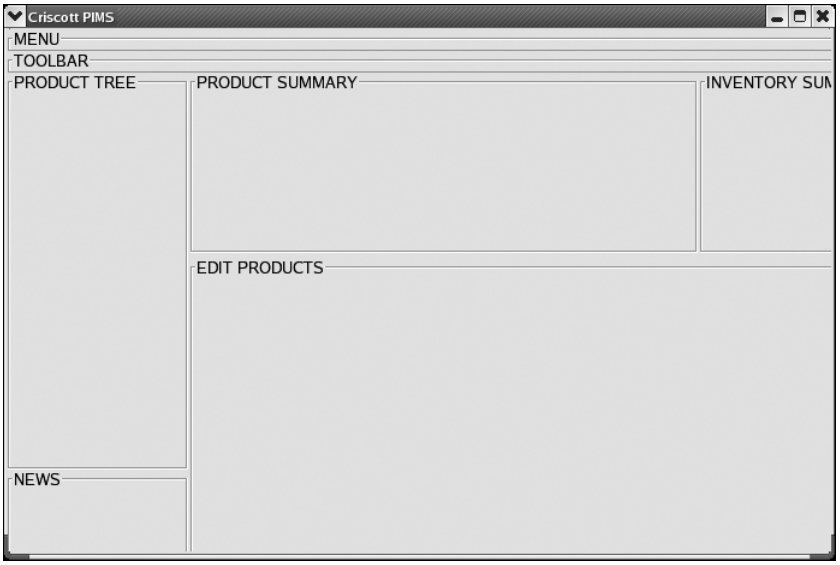

Figure 6-8. *An example of the issues inherent in GtkFixed*

`GtkFixed` has its uses, but laying out a large application probably isn't one of them. `GtkFixed` should instead be used in places where position is much more important than size or where the container cannot be resized. For instance, the splash screen might make good use of a `GtkFixed` container. The user cannot resize the window, and the splash screen will likely show a corporate logo. Corporations often have specific rules about their logos, which define sizes and distances between the logo and other elements. A `GtkFixed` container can be used to ensure that the corporate rules are followed.

For the Crisscott PIMS application, Listing 6-3 is probably the best solution. Because of the complexity of the layout, the box approach is just too much work to keep organized. The desire to keep the application highly usable and flexible rules out the `GtkFixed` approach.

Notebooks

Now that the general layout for the application is set, it is time to think about how to best fit all the varying pieces into the limited real estate. For several parts of the application, this is a simple matter. The menu goes in the area we blocked out for the menu, and the news section goes where the news frame is. But the main reason for building this application is not to show a menu or to distribute news; it is to manage product data. That means we are going to need one or more areas to modify product data and other information. Trying to show all of the tools in the product-editing area at the same time would be difficult at best. Additionally, it would make the application rather confusing to use.

One approach could be to make the product-editing area scroll to give elements more room, but that wouldn't really improve the usability. A more helpful approach would be to show only one set of tools at a time. Tools that are not in use should be hidden to avoid confusion and brought to the forefront when needed. Hiding and displaying groups of widgets may

sound difficult, but there is a highly specialized container widget that makes it easy. That widget is GtkNotebook.

Figure 6-9 shows the PHP-GTK 2 Dev_Inspector (http://cweiske.de/phpgtk2_devinspector.htm). This application uses a GtkNotebook widget to organize reflection data.

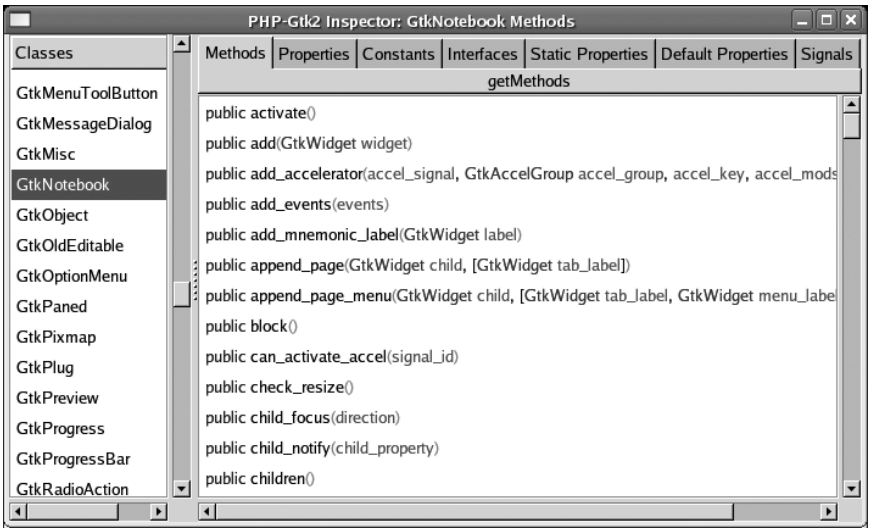

Figure 6-9. *GtkNotebook in the PHP-GTK 2 Dev_Inspector*

GtkNotebook is a container that organizes its children into pages. Each page is itself a bin container and can hold one child. What makes GtkNotebook so powerful is that at any given time, a specific page can be brought to the front of the screen. Only the selected page will be seen by the user. All other pages will remain intact but out of view. The GtkNotebook can have tabs that allow the user to select a given page, or the tabs can be hidden. If the tabs are hidden, the application will control which page is currently displayed.

GtkNotebook is very good for organizing groups of widgets into task-oriented blocks. Being able to control which group of widgets is currently visible also allows you to force the user to step through a process in an ordered manner. The user will not be able to skip ahead, because the next step is not yet available.

GtkNotebook should be thought of more as a three-ring binder than an actual notebook. It consists of pages that can be added, removed, and reordered. The GtkNotebook can have all of its pages marked, or tabbed, or the tabs can be hidden. If the tabs are visible, a user can click one to jump to that page. The tabs can also be moved to any side of the page.

A GtkNotebook widget can have as many pages as needed. Each page holds any arbitrary data and exists independently of the other pages. While the notebook may have more than one child page, the pages themselves are bins and may only have one child each. Just as with GtkWindow and GtkFrame, to have more than one widget show up in the page, another container must be added as the page's child.

In the Crisscott PIMS application, the main area in the application will serve multiple purposes. It will be used to add and edit product information, update supplier data, transmit inventory data, and perform a few other tasks. Figure 6-10 shows what the application will

look like with the GtkNotebook widget added. To keep these tasks organized, the main area will use GtkNotebook, with each task assigned to one or more pages. This approach will maximize the amount of space available and will also help improve usability by forcing the user to focus on one task at a time.

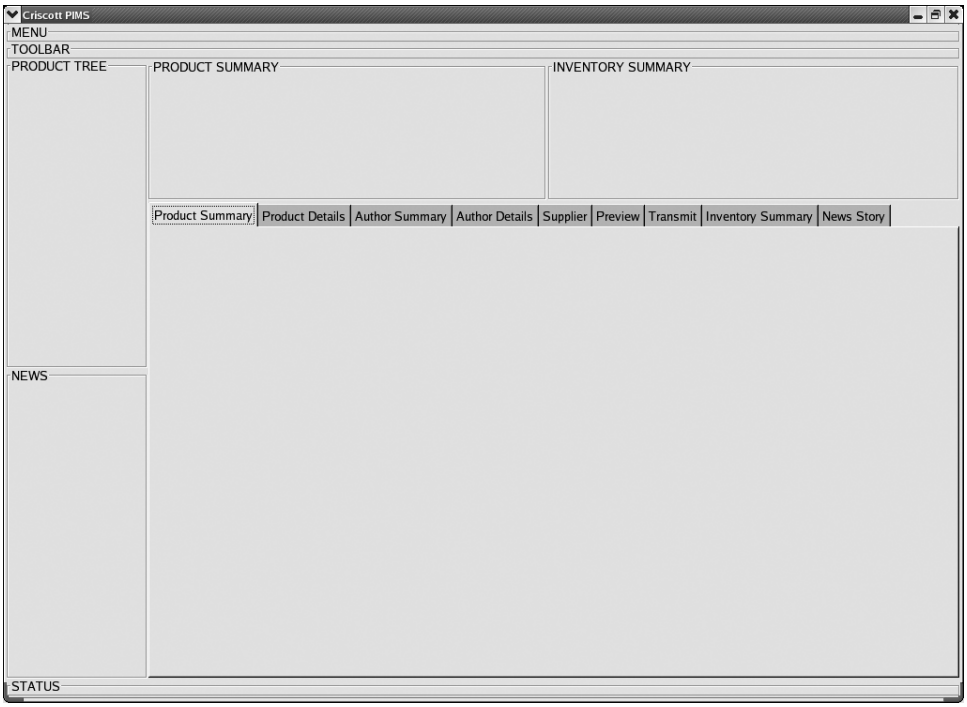

Figure 6-10. *The PIMS application with a simple notebook*

The amount of effort that must be put into organizing widgets with GtkNotebook is considerably less than trying to group, hide, and show widgets by hand. In Listing 6-5, the GtkFrame widget that was labeled EDITING PRODUCTS has been replaced with a custom object that extends GtkNotebook. The custom class helps to make the PIMS-specific organization a little easier. Note also that the notebook is stored as a member variable. This is because items will need to be added, removed, and accessed by other parts of the application.

Listing 6-5. *Adding GtkNotebook to the Application*

```
<?php
// ...
private function _populate()
{
    // Create a new table.
    $table = new GtkTable(5, 3);
```

```
// Make it easier to set both expand and fill at the same time.
$expandFill = Gtk::EXPAND|Gtk::FILL;

// Attach a few frames to the table.
$table->attach(new GtkFrame('MENU'),    0, 2, 0, 1, $expandFill, 0, 0, 0);
$table->attach(new GtkFrame('TOOLBAR'), 0, 2, 1, 2, $expandFill, 0, 0, 0);

// Create a new frame and set its size.
$productTree = new GtkFrame('PRODUCT TREE');
$productTree->set_size_request(150, -1);

// Attach the frame to the table.
$table->attach($productTree, 0, 1, 2, 3, 0, $expandFill, 0, 0);

// Create a new frame and set its size.
$news = new GtkFrame('NEWS');
$news->set_size_request(150, -1);

// Attach the frame to the table.
$table->attach($news, 0, 1, 3, 4, 0, $expandFill, 0, 0);

// Create a new subtable.
$table2 = new GtkTable(2, 2);

// Create a new frame and set its size.
$productSummary = new GtkFrame('PRODUCT SUMMARY');
$productSummary->set_size_request(-1, 150);

// Attach the frame to the subtable.
$table2->attach($productSummary, 0, 1, 0, 1, $expandFill, 0, 1, 1);

// Create a new frame and set its size.
$inventorySummary = new GtkFrame('INVENTORY SUMMARY');
$inventorySummary->set_size_request(-1, 150);

// Attach the frame to the subtable.
$table2->attach($inventorySummary, 1, 2, 0, 1, $expandFill, 0, 1, 1);

// Create a new instance of the main notebook.
require_once 'Crisscott/MainNotebook.php';
$this->mainNotebook = new Crisscott_MainNotebook();

// Attach the notebook to the subtable.
$table2->attach($this->mainNotebook, 0, 2, 1, 2,
                $expandFill, $expandFill, 1, 1);

// Attach the subtable to the main table.
$table->attach($table2, 1, 2, 2, 4, $expandFill, $expandFill, 0, 0);
```

```
    // Attach a new frame to the main table.
    $table->attach(new GtkFrame('STATUS'),  0, 2, 4, 5, $expandFill, 0, 0, 0);

    // Add the table to the window.
    $this->add($table);
}
// ...
?>
```

Defining the Notebook

The next step is defining the `Crisscott_MainNotebook` class. To start, the notebook will be very simple and use the default settings. Then we will look at customizing the notebook to better suit our needs.

Listing 6-6 is a first run at putting the notebook together. The `Crisscott_MainNotebook` class is a simple wrapper around `GtkNotebook` that adds a few pages and tracks them using an array. The constructor for `GtkNotebook` takes no arguments and returns a notebook that has no pages. The idea behind the `Crisscott_MainNotebook` class is just to make development a little more organized. Each page is given a text label, which is also used as the array index. Now instead of having to search for a page by number, you can search a small array by its label.

Listing 6-6. *Organizing Tools with GtkNotebook*

```
<?php
class Crisscott_MainNotebook extends GtkNotebook {

    public $pages = array();

    public function __construct()
    {
        // Call the parent constructor.
        parent::__construct();

        // Create an array of tab labels.
        $titles = array(
                        'Product Summary',
                        'Product Details',
                        'Author Summary',
                        'Author Details',
                        'Supplier',
                        'Preview',
                        'Transmit',
                        'Inventory Summary',
                        'News Story'
                        );

        // Add a page for each element in the array and put
        // it in the pages array for easier access later.
```

```
        foreach ($titles as $title) {
            $pageNum = $this->append_page(new GtkVBox(), new GtkLabel($title));
            $page    = $this->get_nth_page($pageNum);
            $this->pages[$title] = $page;
        }
    }
}
?>
```

Adding, Moving, and Removing Notebook Pages

Each page of a notebook has two elements: the tab and the content. The page tab is usually a string that describes the contents of the page. The tab is the main method by which a user will select a page. The content of the page can be anything. Usually, the content that is added directly to the page is some sort of container.

A page can be added at the beginning of a notebook, at the end of a notebook, or anywhere in between using the methods `prepend_page`, `append_page`, and `insert_page`, respectively. All require a widget for the page body and a widget for the page tab. In Listing 6-6, several pages are appended to the notebook. Each page is added in turn to the back of the notebook with `append_page`. The pages could have just as easily been added to the front of the notebook with `prepend_page`. To insert a page in any arbitrary position, use `insert_page`, passing the body widget, the tab widget, and the page position. Pages are indexed starting from 0, so the first page in the notebook is actually page 0.

When a page is added to a `GtkNotebook` widget, its page index is returned. The value returned from `append_page` is always the total number of pages minus one. The return value from `prepend_page` is always 0. The return value from `insert_page` is not always the same as the position passed to it. If a page is inserted with a position of 12 but there are only eight pages in the notebook, the new page will be added as the last page. The page indexes are always collapsed. For example, the newly inserted page may be inserted in position 12, but it will immediately be moved to position 8. If no position value is passed to `insert_page`, the position will default to –1, which means the page should be appended to the back of the notebook. The value that will be returned will be the same as if `append_page` were used.

Knowing the page index is useful because you can use it to retrieve a page from the notebook. Listing 6-6 uses the `get_nth_page` method to return the page body widget after it has been added to the notebook. This is done to make accessing the page contents easier later on.

Once the page is found, it can then be used to grab the label or the label text. `get_tab_label` takes a notebook child, usually returned by `get_nth_page`, and returns the tab widget. `get_tab_label_text` will return just the text string for the same page if passed the same child widget. Just as you can get a tab label or its text, you can set a tab label or its text. `set_tab_label` and `set_tab_label_text` work in much same way as their get counterparts. Each expects a widget that has already been prepended, appended, or inserted into the notebook as the first argument and either a tab widget or a string of text to be set as the tab label. The index of a page is the key to being able to make changes.

■ **Tip** To get the total number of pages, use `get_n_pages`. Keep in mind that this is the total number of pages, not the index of the last page. The index of the last page is `get_n_pages() - 1`.

While the index of a page is the key to accessing the page, the index may change. If a new page is prepended or inserted in front of a given page, the index will be incremented. If a page in front of a given page is removed or moved to the back of the notebook, the page's index will be decremented. To get the index of a specific child, use the `page_num` method. `page_num` takes a widget as the only argument and returns the page index. Regrabbing the index this way comes in handy when pages in the notebook are moved.

You can reorder pages by using the `reorder_child` method. `reorder_child` takes the child given as the first argument and puts it in the position given by the second argument. The page that previously occupied the position will be moved backward in the notebook. If the position passed to `reorder_child` is greater than the total number of pages, the page will be moved to the back of the notebook.

You can also remove pages from the notebook. To do this, call `remove_page` and pass the page index.

Whenever either the `reorder_child` or `remove_page` method is called, the pages are reindexed. This means, for example, that the page that was previously in position 5 may now be in position 4, 5, or 6, even though that particular page was never moved. Because of this reindexing, the code in Listing 6-6 uses a separate array with associative keys to keep track of the pages. Without this separate array, finding a particular page may require cycling through all the pages in the notebook.

Navigating Notebook Pages

In reality, a `GtkNotebook` widget can do only three things: go back one page, go forward one page, or jump to a particular page. What makes `GtkNotebook` such an excellent tool is the number of ways in which these three simple tasks can be accomplished. The most obvious method to get from one page to another is by clicking the tab for a given page. This is a simple user interaction and requires no special programming.

`GtkNotebook` containers are powerful because of their ability to bring a particular group of widgets to the front while hiding all others. While showing and hiding groups is a nice feature, it is the ease with which the top page can be changed that makes `GtkNotebook` so powerful.

In some cases, an application may need to display a given page of the notebook. For instance, when a user wants to edit a product in the Crisscott PIMS application, the page that is currently being shown should be hidden, and the product-editing page should be brought to the screen. Likewise, if there were a page specifically for error messages, that page would be shown whenever an error is encountered.

Moving to the Next, Previous, or Specific Page

Another reason to change a page automatically is to step through a process. `GtkNotebook` makes it easy to move through a series of steps that together make up one complete process. The way to step through something is to complete the requirements for one page, and then go to the next. Moving to the next page is done by calling `next_page`. The `next_page` method hides the current page and shows the page with the next index. The `previous_page` method is similar to `next_page`, except it goes to the previous page. If there is not a previous page, the first page is shown again. Neither `next_page` nor `previous_page` will cycle around the pages.

Listing 6-7 rewrites the constructor of the `Crisscott_MainNotebook` class and adds two buttons to each page. One is connected to the `previous_page` method, while the other is connected to the `next_page` method.

■ **Tip** If `next_page` is called and the last page of the `GtkNotebook` is already being shown, nothing happens. This can be shown by creating a signal handler for the `switch-page` signal and trying to go past the end of the `GtkNotebook`. When trying to go beyond the last page, the signal handler will not be called.

Listing 6-7. *Moving to the Next or Previous Page*

```
<?php
// ...
public function __construct()
{
    // Call the parent constructor.
    parent::__construct();

    // Create an array of tab labels.
    $titles = array(
                    'Product Summary',
                    'Product Details',
                    'Author Summary',
                    'Author Details',
                    'Supplier',
                    'Preview',
                    'Transmit',
                    'Inventory Summary',
                    'News Story'
                    );

    // Add a page for each element in the array and
    // put it in the pages array for easier access
    // later.
    foreach ($titles as $title) {
        $pageNum = $this->append_page(new GtkVBox(), new GtkLabel($title));
        $page    = $this->get_nth_page($pageNum);
        $this->pages[$title] = $page;

        // Create a previous page button.
        $button = new GtkButton('PREVIOUS');

        // Create a signal handler that will bring the previous page to the
        // front of the notebook when the button is clicked.
        $button->connect_object('clicked', array($this, 'prev_page'));

        // Pack the button into the page.
        $page->pack_start($button, false, false);
```

```
        // Create a next button.
        $button = new GtkButton('NEXT');

        // Create a signal handler that will bring the next page to the front of the
        // notebook when the button is clicked.
        $button->connect_object('clicked', array($this, 'next_page'));

        // Pack the button into the page.
        $page->pack_start($button, false, false);
    }
}
// ...
?>
```

Moving to the next or previous page is good enough for stepping through a process, but there are cases when relative movements are not enough. Sometimes it is necessary to jump to a particular page. To jump to a particular page, use `set_current_page`. When passed an integer, `set_current_page` will bring the page with that index to the screen. If an index of `-1` is passed, the last page will be shown.

`set_current_page` has a corresponding `get_current_page` method. This method returns the page index of the page that is currently visible. Listing 6-8 shows a method that can be connected to any signal, such as the `clicked` signal of a button, and jumps to a random page in the notebook. This method is not all that practical, but it does show how the `$pages` array can be used with `set_current_page` to easily navigate to any page in the notebook.

Listing 6-8. *A Method for Jumping to a Random Page*

```
<?php
public function goToRandomPage()
{
    // Pick an array key at random and jump to that page.
    $rndIndex = array_rand($this->pages);
    $this->set_current_page($this->page_num($this->pages[$rndIndex]));
}
?>
```

So, to summarize, the following methods can be used to access pages in `GtkNotebook`:

- `prev_page`: Brings the previous page to the front of the notebook.
- `next_page`: Brings the next page to the front of the notebook.
- `set_current_page`: Brings the page with the given index to the front of the notebook.

Using a Pop-Up Menu

Navigating in a notebook is easy because of the many ways there are to get from one page to another. Not only are there plenty of class methods to switch pages, but there are also multiple ways for the user to get from one page to the next.

Normally, a user clicks the page's tab to bring that page to the front of the screen, but if the notebook is set up properly, another method may be available to the user. If `popup_enable` is called, a menu will pop up when the user right-clicks in the tab area. This pop-up menu will have an entry for each page in the notebook. When a user selects an entry from this menu, a built-in signal handler is fired and shows the corresponding page. To see this menu, the user doesn't actually need to click a tab. If there is empty space in the tab area, the user can click there as well. If at some point the menu is no longer needed or shouldn't be available, `popup_disable` will make the menu unavailable.

By default, the text that is used for the pop-up menu is copied from the tab label. If you want to use different text or a different type of widget for a particular page in the menu, use the `*_page_menu` methods. You can prepend, append, and insert pages with the methods discussed earlier, but you can also perform the same tasks with the menu sister methods: `prepend_page_menu`, `append_page_menu`, and `insert_page_menu`. The `*_page_menu` methods can take an additional parameter that is not available with their nonmenu counterparts: a widget that will be used as the menu label. The widget for the menu is usually a `GtkLabel` widget, but it could be any type of widget.

At this point, you might be thinking, "What is the point of having a pop-up menu when the user can just click one of the tabs?" That is a valid question and one that will be answered in the next section.

Decorating a Notebook

The power of `GtkNotebook` is not only in the way it shows and hides different pages, but also in the amount of customization it allows. Having tabs at the top of the notebook may not work for an application. You can put the tabs at the bottom. If that doesn't work, you can move the tabs to one of the sides. You can even get rid of them entirely. Maybe that isn't good enough. Maybe all of the tabs need to be the same size. Maybe the tabs need some padding. Or there may even be too many tabs to show at once. `GtkNotebook` has methods to help with each of these situations.

First, let's tackle moving the tabs away from the top of the notebook.

Repositioning the Tabs

Having the tabs at the top of the notebook may be popular on web pages, but many desktop applications move the tabs to another side. For instance, Microsoft Excel uses tabs at the bottom of the notebook for accessing worksheets within a workbook.

Repositioning the tabs is a simple matter of calling `set_tab_pos` and passing a `GtkPositionType`. A `GtkPositionType` is just a constant value that defines a position such as top, right, bottom, or left. The names of the constants are `Gtk::POS_TOP`, `Gtk::POS_RIGHT`, `Gtk::POS_BOTTOM`, and `Gtk::POS_LEFT`. When the tabs are moved to another side of the notebook, they keep their order and orientation; that is, on the left and right the tab for the first page is on the top, and the tab for the last page is on the bottom.

Because the tabs keep their orientation, moving the tabs to the left or right of the notebook may not have the desired effect. The tabs will appear to stick out from the side of the notebook instead of lying along it. This may be nice if the notebook has many pages, but in most cases, it won't be the intended result. It is possible to make the tabs lay flat against the side of the notebook by angling text so that it will run vertically, as discussed in the next chapter.

Hiding the Tabs

Just because you're using `GtkNotebook` doesn't mean that you must show the tabs. When using a notebook to control a user's movement through a step-by-step process, it probably isn't a good idea to allow the user to jump around using the notebook tabs. In that case, you may want to hide the tabs.

When the tabs are hidden, all built-in user navigation is taken away. Because the pop-up menu requires the user to click the tab area, if the tabs are hidden, there is no way for the user to access the menu.

You turn off the tabs by calling the `set_show_tabs` method and passing `false`. Passing `true` to the method turns the tabs back on. Unfortunately, with `GtkNotebook`, it is either all or nothing. There is no way to turn off a particular tab while leaving the rest visible.

Adjusting the Border and Sizing the Tabs

The all-or-nothing rule applies to other aspects of tabs as well. You can adjust the padding, or border, around the contents of a tab on all four sides. The border can be changed for the top and bottom independently of the left and right, but individual sides cannot be manipulated. Also, setting the tab border changes the border for all tabs, not just the tab for a specific page.

To change the border on all four sides of every tab at once, use `set_tab_border`. To change the border for just the left and right, use `set_tab_hborder`. To change the padding on the top and bottom of the tabs, use `set_tab_vborder`. All three methods take one argument that defines the number of pixels of padding that should be set. The default border for all sides of the tabs is 2 pixels.

Setting a larger border value will increase the dimensions of all tabs, regardless of their contents. Normally, the contents of the tab determine its width because the tab shrinks to fit the contents so that it takes up as little space as possible. Using `set_homogeneous_tabs` and passing `true`, will make all of the tabs the same width. The space that is available for the tabs will be divided equally and shared by every tab. The border on the left and right (or top and bottom depending on where the tabs have been positioned) will be automatically adjusted. If a new page is added to the notebook, the tabs will be readjusted to account for the new tab. When `set_homogeneous` tabs is passed `true`, all tabs will be the same size. This is true even if one tab's text is much longer than the text on the rest of the tabs. If one tab takes up more than its fair share of real estate, the width of that tab (plus any border) will become the new width of all tabs.

But how can the width of the tabs be more than the width of the notebook? Normally, the notebook simply stretches to accommodate the extra width, but this can lead to unwanted results. Depending on how the notebook was added to its parent container, it may stretch out the window and throw off the layout. If the notebook isn't able to stretch the window, one or more of the tabs may simply be cut off and not be accessible to the user. This is obviously a problem, but not one without a solution.

Using Scrolling Tabs

The `set_scrollable` method sounds like it would make the pages within a notebook scroll when the contents are too much to show on one screen, but that is not quite what it does. `set_scrollable` applies only to the tab area of the notebook.

Calling `set_scrollable` and passing `true` will allow some of the tabs to be hidden yet accessible through the use of two scrolling icons. Take a look at Figure 6-11. Notice the two

icons to the left and right of the tabs. Clicking one of these arrows will select the next page in the notebook. If the tab for the next page is hidden, the tabs will be shifted so that the new current page's tab is shown. In fact, any time a page is shown, the tabs will be shifted so that the tab for that page is shown. This is where the pop-up menu comes in handy again. Scrolling through the pages requires the users to move one by one through every page until they reach their destination. Using the pop-up, the users can quickly jump directly to the desired page.

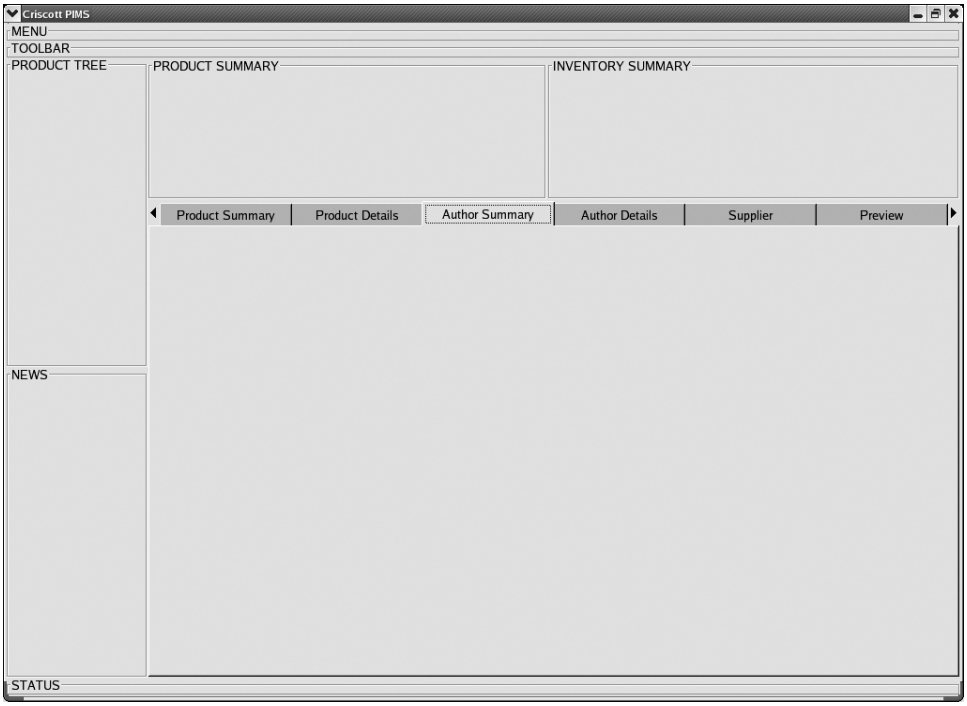

Figure 6-11. *GtkNotebook with scrolling tabs*

The Crisscott PIMS application can certainly make good use of `GtkNotebook`. There are several distinct and independent tools that the application will have that can benefit from the organization `GtkNotebook` offers. The tools in the application need to be somewhat controlled, so the tabs should probably be hidden. Other features, such as buttons or menus, can serve to bring pages to the screen when they are needed.

Summary

Containers are arguably the most essential pieces of an application. They provide structure and organization for an application. Aside from these two very important features, you simply can't build a PHP-GTK application without using at least one container.

In this chapter, you learned how to lay out an application through the use of containers. There is the fast and free method of using boxes, which is good for smaller applications with less rigid design constraints. Then there is the extremely structured `GtkFixed` approach, where

everything has a place and that place doesn't change. And there is a compromise between the two using `GtkTable`. `GtkTable` provides a good balance between structure and flexibility. Widgets are easy to align, but are also free to shrink or grow as the application needs.

Finally, we looked at `GtkNotebook`. This handy widget makes organizing an application a snap. Tools within the application can be organized into pages, which can then be accessed in a variety of ways.

The choice of which container to use for a particular piece of an application plays a very big role in how it will interact with the rest of the application and the user.

In the next chapter, we will begin to look at entering and displaying data. The chapter will focus on how to collect data from the user and return it to be shown in the application. Finally, the application will go from a static window on the screen to an interactive program capable of collecting, analyzing, and presenting data. By the end of the next chapter, we will have an application by all definitions of the word.