Pro PHP-GTK

Scott Mattocks

Apress®

Pro PHP-GTK

ISBN-13: 978-1-59059-613-5

ISBN-10: 1-59059-613-7

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Jason Gilmore
Technical Reviewers: Christian Weiske, Steph Fox
Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Jason Gilmore, Jonathan Hassell, James Huddleston, Chris Mills, Matthew Moodie, Dominic Shakeshaft, Jim Sumser, Matt Wade
Project Manager: Kylie Johnston
Copy Edit Manager: Nicole LeClerc
Copy Editors: Marilyn Smith, Jennifer Whipple
Assistant Production Director: Kari Brooks-Copony
Production Editor: Ellie Fountain
Compositor: Kinetic Publishing Services, LLC
Proofreader: Dan Shaw
Indexer: Valerie Perry
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com` in the Source Code section.

CHAPTER 3

Understanding PHP-GTK Basics

In the previous chapter, you installed and tested a PHP-GTK environment, setting the stage for writing some code. However, before rushing into creating an application, you should understand the basic relationships between PHP-GTK classes.

PHP-GTK is a complex hierarchy of classes. If you want to understand why your Save As window isn't showing up properly, you need to know what its base classes are doing.

Inheritance isn't the only relationship in PHP-GTK. Classes can be wrappers around other classes; some classes will have instances of another class as properties; and other classes may exist only to manipulate other objects. It is important to know how these classes interact, because changing one object can have a profound effect on many others.

PHP-GTK defines many class families, which are based on the libraries that the classes hook into. The two most important families from a developer's standpoint are `Gdk` and `Gtk`. The `Gdk` family of classes consists of low-level classes that interact very closely with the windowing system. These classes are responsible for displaying windows and showing colors on the screen.

The `Gtk` family is a grouping of higher-level objects. These objects represent application components such as text, menus, or buttons.

The `Gtk` classes will often contain one or more `Gdk` classes as members. Although it does happen, it is rare that a developer works directly with a `Gdk` class. In most cases, manipulation of a `Gdk` instance is done through a `Gtk` class. The `Gtk` classes are the ones that create and manage the pieces of an application that you are used to seeing. If `Gtk` is the movie star of PHP-GTK, `Gdk` is the personal assistant. `Gdk` does half of the work, while `Gtk` gets all of the attention.

Widgets and Objects

The `Gtk` family tree starts with one class: `GtkObject`. Every class in the `Gtk` family extends `GtkObject`. Some classes extend directly from `GtkObject`, while others are grandchild classes. Members of the `Gtk` family can basically be broken into two major groups: objects and widgets.

The GtkObject Class

`GtkObject` defines a few basic methods and declares a few signals (We'll talk more about signals in the next chapter; for now, just keep in mind that a signal is used to let PHP-GKT know that some important event has occurred). Having one base class is nice not only for the GTK developers, but also for the users. We know that any class we instantiate that extends from `GtkObject` will have these few methods that we can call when needed.

Let's take a look at what the class definition for this object might look like if it were written in PHP. Then we can talk about how it works and what role it plays in our development. Take a look at Listing 3-1.

Listing 3-1. *Definition of GtkObject*

```
<?php
class GtkObject {

    private $flags;
    private $refCounter;

    public function destroy()
    {
        unset($this);
    }

    public function flags()
    {
        return $this->flags;
    }

    public function set_flags($flags)
    {
        $this->flags = $this->flags | $flags;
    }

    public function sink()
    {
        if (--$this->refCounter < 1) {
            $this->destroy();
        }
    }

    public function unset_flags($flags)
    {
        $this->flags = $this->flags & ~$flags;
    }
}
?>
```

As you can see, the class defines a handful of public methods. These public methods are available to all other classes in the `Gtk` family. Just because they are available, however, doesn't mean that you will ever use them. Most of the methods defined by `GtkObject` are used primarily by PHP-GTK itself. We will still take a closer look at them, though, because it is important to know why PHP-GTK calls them.

The destroy Method

The `destroy` method is probably the only `GtkObject` method you will ever call explicitly in your code. It does exactly what you would expect: destroys the object. This method will be overridden by some classes that extend `GtkObject`.

Some classes, known as *containers*, exist just to group other objects logically and visually. When you destroy a container, it will destroy all of the objects it contains. For instance, destroying the main window of your application will basically delete every class in your application.

The sink Method

The `destroy` method is called by PHP-GTK when an object is no longer needed. Determining when an object is no longer needed (or wanted) is done in two ways.

The first way is by tracking the reference counter. The reference counter is the number of objects (including the object itself) that reference a given object. When the reference counter hits zero, it pretty much means that no one cares about the object anymore. Since no one cares, PHP-GTK destroys the object. This type of action is pretty rare. The reference counter is maintained using the `sink` method. Calling the `sink` method decrements the reference counter. Usually, PHP-GTK does this during the execution of an application. Incrementing the counter is always done by PHP-GTK. There is no method for "unsinking" an object.

The other way PHP-GTK knows that an object isn't needed is when someone or something tells it to kill the object. For example, let's say an instance of class A contains an instance of class B. When you destroy the class A instance, you no longer need the class B instance. While object A is in the process of deconstructing, it is going to tell PHP-GTK to get rid of the class B instance. Another example is when the user clicks the *x* in the upper-right corner of a window. That tells your application that the user is finished using it. Under most circumstances, the application will shut down. It does this by destroying the main window. When the main window is destroyed, it destroys everything contained within it.

The flags, set_flags, and unset_flags Methods

The `flags`, `set_flags`, and `unset_flags` methods do exactly what their names suggest: they return, set, and unset flags associated with an object, respectively. Flags are used to track object attributes, such as whether or not the object is visible, or whether or not it can accept drag-and-drop objects. They offer a simple way to track object properties without using a lot of memory.

If you wanted to know the current status of an object, you could call the `flags` method and compare the result to some known state. In practice, though, you probably won't ever use this method. Similarly, you probably won't use the `set_flags` or `unset_flags` method either. In fact, setting the flags doesn't mean you have changed any object properties. Setting or unsetting flag values will likely just confuse your application.

On the other hand, PHP-GTK will use these methods. Before it does any operation that requires the object to be in a certain state, such as displaying the object on the screen, it will compare the current set of flags. If they aren't right, PHP-GTK will call the method needed to get the object in the right state. For instance, before a button can be shown on the screen, it must be inside a window. PHP-GTK uses the `flags` method to quickly check if the button is ready to go. Any method that changes an object's state will call `set_flags` or `unset_flags` as needed. The parameter that is passed to `set_flags` and `unset_flags` is an integer, and it's used to change the value of the object flags through bitwise operations. PHP-GTK will call `set_flags`

when you show or realize an object. PHP-GTK will call `unset_flags` when you hide that object later. Before PHP-GTK does either of these operations, it will check the flags, using the `flags` method, to verify whether it actually needs to do any work.

The object flags allow PHP-GTK to quickly and easily manage object properties. These few simple methods are integral to being able to control and manipulate all of the classes that inherit from `GtkObject`.

Objects

Objects are the classes that extend directly from `GtkObject` and their children, except for `GtkWidget` and classes that extend from it, as described in the next section.

Objects are usually considered helper classes. They don't have any visual components that can be shown on the screen. A buffer for holding and manipulating text is an example of an object.

Objects cannot receive direct user interactions. Since they have no visual components, there is nothing for the user to click on or select.

Objects typically store data such as number ranges or text. They are used to encapsulate data and provide a consistent interface for manipulating that data. It is much easier to pass around a bundle of numbers than it is to pass around several numbers while trying to keep them organized.

Widgets

Widgets are classes that extend from `GtkWidget` (which extends from `GtkObject`). Widgets technically are objects because they inherit from `GtkObject`, but they deserve special treatment in the world of GTK.

Widgets are objects with visual representations and can react to user input. Widgets are the classes you are familiar with through your use of GUI applications. A button is a widget, as are labels, menus, and windows. Widgets are the objects with which your application's users will be directly interacting. They can listen for interaction events, such as user clicks, resizing, and even dragging and dropping.

Widgets can be shown or hidden. They can be given keyboard focus or have it taken away. You can also control the look and feel of an individual widget or all widgets of a certain type.

Widgets and objects need each other to make an application work. Data that no one can see or interact with isn't very useful. A button that doesn't have a label or change any data doesn't do much good either. Widgets and objects must work together to make a successful application.

Widgets, being visual objects, can be shown on the screen. This isn't always what you want, though, and it isn't how they start. For instance, you may not want a button to show up until the user enters some data in a text field. You don't have to show the button until you are ready. To help you manage what is shown and what isn't, widgets have three basic states: *realized*, *unrealized*, and *shown*. These three states represent the visual status of a widget, and we'll take a closer look at them here.

■ **Note** I apologize for the confusing naming conventions. It is true that all widgets are objects, but they are a special and quite large subset of objects. Keep in mind that when I refer to *objects* in PHP-GTK, I am talking about those classes that do not inherit from `GtkWidget`.

The Realized State

A realized widget is a widget that has valid `window` and `allocation` properties. It isn't yet visible on the screen, but it is ready to be shown. The `window` property of a widget is a `GdkWindow` instance. Remember that the `Gdk` classes are the ones that actually take care of the visual representations on the screen. The `allocation` property, an instance of the `GdkAllocation` class, holds the location and dimensions of a widget. It has value for the x and y coordinates, and the height and width of the widget. The `window` and `allocation` properties are responsible for telling PHP-GTK how the widget is going to be displayed and where.

The realized state may also be called the *hidden* state. This is because it is pretty similar to a widget in the shown state, but it just isn't shown on the screen. Only realized widgets can be shown. Fortunately, the `GtkWidget` base class is smart enough to realize a widget before you try to show it.

Listing 3-2 shows how you can move from one state to another.

Listing 3-2. *Changing Widget States*

```
<?php
$widget = new GtkWindow();

// If you try to grab the window before realizing
// the object, you will get nothing.
var_dump($widget->window);
var_dump($widget->flags());

$widget->realize();

// Now that the widget is realized, you can grab
// the window property.
var_dump($widget->window);
var_dump($widget->flags());

$widget->show();

// Showing and hiding a widget changes the value
// of its flags.
var_dump($widget->flags());

$widget->hide();
```

```
var_dump($widget->flags());

$widget->unrealize();

// Now that the widget is realized, you can grab
// the window property.
var_dump($widget->window);
var_dump($widget->flags());
?>
```

Some appropriately named methods help you change a widget's state. The `realize` method tells PHP-GTK to make room in memory for the widget, because you plan on showing it soon. If you call the `realize` method of a widget, the widget will be moved into the realized state but will be hidden; that is, PHP-GTK will assign it a valid `GdkWindow` and `GdkAllocation`. The widget will not be displayed on the screen, though.

We talked earlier about how widgets may be members of other widgets and how the `destroy` method can affect those members. The `realize` method can also have an effect on widgets other than the calling widget. A widget cannot be realized until its parent widget is realized. If you think about it logically, there is pretty good reason for this. Realizing a widget tells you where it will be on the screen. But how can you know where it will be if you don't know where its parent will be? If you try to realize a widget whose parent hasn't yet been realized, PHP-GTK will help you out by realizing that parent also. It will do this recursively, all the way up to the top-level widget, which is usually a window.

In most applications, the `realize` method doesn't need to be called directly. The exception is when you need to know where a widget will be on the screen or how much space it will take up before it is displayed. For instance, say you wanted to print the dimensions of an image as a caption. Before the image is displayed on the screen, you can get its size and location by calling the `realize` method and then checking the value of its `allocation` property.

The Unrealized State

When you first create a widget, it is unrealized. *Unrealized* means that no memory has been allocated for the visual parts of the widget. The unrealized widget doesn't have a size, and it doesn't have a position. The most important thing to know about an unrealized widget is that it doesn't have a valid value for its `window` property. As noted in the previous example of printing the dimensions of an image as a caption, you cannot get those dimensions from an unrealized object.

Just as there is a `realize` method, there is an `unrealize` method. Calling `unrealize` removes the widgets `window` and `allocation` properties. If a widget is shown when `unrealize` is called, it will first be hidden, and then unrealized.

Just like the `realize` method, the `unrealize` method has an effect on widgets other than the calling widget. If a widget that is unrealized has children, they will be unrealized, too. The positioning and size information are no longer relevant if the parent doesn't contain any position or size information. These rules do not apply in reverse, however. Realizing a parent does not realize the child, just as unrealizing the child does not unrealize the parent.

Look at the simple example in Listing 3-3. It shows how realizing and unrealizing have a recursive effect on multiple widgets.

Listing 3-3. *Recursively Realizing and Unrealizing Widgets*

```
<?php
$window = new GtkWindow();
var_dump($window->window);
var_dump($window->flags());

$button = new GtkButton();
$window->add($button);

$button->realize();
var_dump($window->window);
var_dump($window->flags());

$window->unrealize();
var_dump($button->window);
var_dump($button->flags());
?>
```

First, we show the window in its initial unrealized state. Then, after adding a button and realizing the button, we check the window's state again. The presence of a `window` property is evidence that the window has been realized, even though we didn't call the `realize` method explicitly. Next, we show the button in its realized state, and then unrealize the window. When we try to view the button's `window` property again, it is gone.

The Shown State

To arrive at the shown state, all you need to do is call the `show` method of a widget. If the widget hasn't yet been realized, PHP-GTK will realize the widget first.

When you no longer want the widget to be shown, you can call the `hide` method. That will move the widget back to the realized or hidden state. The widget will still have its `window` and `allocation` properties, but will not be displayed on the screen.

Showing or hiding a widget doesn't have quite the same recursive effect that realizing a widget does. If you show a widget whose parent has not been shown, nothing seems to happen. The parent isn't realized, and neither is the child. PHP-GTK will queue up this request and show the widget when the parent is ready.

Try changing the `realize` call in Listing 3-3 to a call to `show`. You will see that the `window` property of our window object is still `null`. If you then change the `unrealize` call to `show`, you will see that the `window` property for both the window and the button will be objects. Also try showing the window without showing the button. That will demonstrate that showing widgets is not recursive. The button is not shown unless you call `show` explicitly for the button; that is, unless you use the `show_all` method.

The `show_all` method shows the calling widget, and then recursively shows all of the widget's children. The corollary to the `show_all` method is the `hide_all` method. Calling `hide_all` will hide the calling widget and all of its children recursively. Keep in mind that `hide` and `hide_all` are just moving the widget back to the realized state. They are not unrealizing the widgets. The widget will still have its `window` and `allocation` properties.

Parents and Children

We've been talking a lot about parent widgets and child widgets, but we haven't given this relationship much attention. As you have seen from the previous examples, the parent-child relationship is very important to any application. Making changes to a parent can have effects that trickle down to many other pieces of the application. Understanding the implications of making a change will save you countless hours of debugging.

In the parent-child relationship, a widget that has another widget as a member is a *parent*. A widget that is a member of another is a *child*. The parent doesn't just provide a place for the child to hang out while waiting for the code to be executed. The parent provides a visual context in which the child will appear. In some cases, the child widget exists only to assist the parent with some critical function. The nodes of a tree, for instance, exist only to represent data in the tree and don't have a use elsewhere. Without the nodes, the tree would be useless. Without the tree, the nodes would be unorganized and be almost impossible to manage.

Containers

While most widgets may be children, a few widgets may be parents. Only those widgets that extend from the `GtkContainer` class may be parents. These widgets are called *containers*, of which there are many types. Here are a few examples:

- Bin containers, such as `GtkWindow`, `GtkFrame`, and `GtkButton`, can have only one child at a time.
- Box containers, such as `GtkHBox` and `GtkVBox`, display their children one after the other in a given direction.
- Special containers, like `GtkTable` and `GtkTree`, manage their children in unique ways, such as rows and columns for tables and nodes for trees.

How a container manages its children is often a function of how the container is used. `GtkWindow`, for instance, is designed to provide a window for all the other widgets in the application. It isn't designed for laying out or organizing any data. Because of this, it accepts only one child. It relies on the addition of a widget designed specifically to control the layout.

The `GtkWindow` class extends the `GtkBin` class. `GtkBin` is a specialized class for all containers that accept only one child. Any class that inherits from `GtkBin` is known as a *bin*. If you try to add two children to a bin, you get a warning message. Try executing Listing 3-4 by saving the code to a file, and then running `php filename.php`. If you try running the example with a `GtkHBox` instance instead of a `GtkWindow` instance, you won't see the warning.

Listing 3-4. *Adding Two Children to a Bin*

```
<?php
$window = new GtkWindow();
$window->add(new GtkButton());
$window->add(new GtkButton());
/*
 Prints a warning:
 Gtk-WARNING **: Attempting to add a widget with type GtkButton to a
 GtkWindow, but as a GtkBin subclass a GtkWindow can only contain one
```

```
 widget at a time; it already contains a widget of type GtkButton
*/
?>
```

The number of children that a container may have varies depending on the role of the container. But the number of parents a widget may have is much more controlled.

Top-Level and Parent Widgets

A widget either may not have any parents or it may have one. Widgets that cannot have a parent are called *top-level widgets*. `GtkWindow` is a top-level widget, because it doesn't make sense to put a window inside another widget. A window provides a framework for the application. Putting a window inside another widget would be like putting your garage inside your car.

If a widget is not a top-level widget, it may have one and only one parent. Remember that the parent not only keeps the children organized, but also provides a visual context for the child. If a widget had two parents, it wouldn't know where to show up. If you try to assign two parents to a widget, as shown in Listing 3-5, a rather informative message will be printed to the terminal. The message tells you exactly what you did wrong. You can't put a widget directly into a container while it is still inside another container. You can, however, put the first container into the second with the child still in it, because each widget will still have only one parent.

Listing 3-5. *Trying to Give a Widget Two Parents*

```
<?php
// Create some containers.
$window = new GtkWindow();
$frame  = new GtkFrame();
// Create our test widget.
$button = new GtkButton('Button');

// Try giving the widget two parents.
$window->add($button);
$frame->add($button);

/*
Prints a warning message:
 Gtk-WARNING **: Attempting to add a widget with type GtkButton to a
 container of type GtkFrame, but the widget is already inside a
 container of type GtkWindow, the GTK+ FAQ at http://www.gtk.org/faq/
 explains how to reparent a widget.
*/
?>
```

Adding a widget to a container doesn't mean that the widget is stuck there, as you'll learn next.

Adding and Removing Widgets

Containers and widgets have some handy methods to help you move widgets into and out of containers. Some are specialized for the container type, and we will cover those in later chapters. For now, we will look at the methods that come with the base `GtkContainer`, `GtkBin`, and `GtkWidget` classes.

The add, remove, and reparent Methods

You have already seen one of the methods for adding a child in some of the previous examples. The appropriately named `add` method will take a widget and make it a child of the container. `add` is a method of the base class `GtkContainer`. Because it is part of the base class, it needs to be very generic. `add` doesn't worry about placement or positioning; it simply puts the widget into the container. It is up to the classes that extend `GtkContainer` to worry about positioning and ordering. We will go into much more detail about how to lay out children inside a container in Chapter 6.

The equally well-named `remove` method will remove a given widget from the container. For both of these methods, you need to know the container and the widget that you want to add or remove. With adding, obviously, you need to know which widget you want to add, but removing may be different. You may just want to remove a bin's child so that you can put something else in that container.

Containers have a `children` method that returns the children of the container in an array. There is also a `get_children` method which has the same result. Bins have an extra method for getting the child. Since a bin can have only one child, it is kind of silly to return an array. You can use the `get_child` method to return the container's child widget. Once you know what is in the container, you can then remove its contents.

If you just want to move a widget from one container to another, you don't have to go through the process of removing the widget from one container and adding it to the other. There is a neat little helper method that makes changing the widget's parent, or "reparenting" the widget, a simple one-step process. The `reparent` method removes the widget from its current parent container and adds it to the container that you pass as the method's only parameter. `reparent` does all of the dirty work for you behind the scenes.

■ **Tip** If your container is a bin, you can also add a child using the `set_child` method. For bins, `add` and `set_child` do the same thing, so I usually just stick with `add` all of the time to avoid confusion.

Let's take a look at how you can control the parent-child relationship with container methods. Listing 3-6 is a simple script that demonstrates the use of `add`, `remove`, and `reparent`.

Listing 3-6. *Adding and Removing Widgets from a Container*

```
<?php
function testForParent($widget)
{
        $parent = $widget->get_parent();

        echo 'The ' . get_class($widget) . ' has ';
        if (isset($parent)) {
                echo 'a ' . get_class($parent);
        } else {
                echo 'no';
        }
```

```
        echo " parent.\n";
}

// Start with three widgets.
$window = new GtkWindow();
$frame  = new GtkFrame('I am a frame');
$button = new GtkButton("I'm a button");

testForParent($button);

$frame->add($button)
testForParent($button);

// What if we want the button to be added directly to
// the window?
$frame->remove($button);
$window->add($button);
testForParent($button);

// Now switch it back to the frame.
$button->reparent($frame);
testForParent($button);
?>
```

The function at the top, `testForParent`, is used to show which type of container is the parent of the widget that is passed in. You might use a method like this to figure out what role a widget is playing in your application. Say you have a method that changes a label's text. You may want to know if the label is just text from the application or is part of a button. If it is part of a button, you may want to shorten the text that you set for the label. It prints out a simple message that tells the class of the widget you are testing and the class of its parent, if it has one. In the rest of the script, we use this function to report the parent information every time we `add`, `remove`, or `reparent` a widget.

When you run the script, you will see that the button starts off with no parent. This is what you would expect, since we did the first test immediately after creating the button. Next, we call the frame's `add` method and pass in the button. When we test for the parent this time, the function tells us that the button has a frame for a parent. After removing the button from the frame and adding it to the window, we test again. This time, as expected, the button's parent is a window object. Finally, we add the button back to the frame using the `reparent` method. The test again shows that the frame is the button's parent.

Notice that when we moved the button to the window, we had to first remove it from the frame. In this simple example, it isn't that big of a deal, because we know which container is the button's parent. In a real-world situation, you probably won't know which container is the widget's parent. You would probably need to use a function similar to `testForParent`, which returns the parent container. Using `reparent`, all we needed to do was pass in the new parent container. PHP-GTK took care of tracking down the old parent and removing the widget first.

The set_parent and unparent Methods

Adding a widget to a container using the `add` method isn't the only way to accomplish the task. A few methods that belong to widget can be used to create a parent-child relationship.

Calling `set_parent` and passing the container has a similar effect to calling the container's `add` method and passing the widget. It adds the calling widget as a child of the container. Similarly, the `unparent` method will remove a child widget from its container. The `unparent` method doesn't need any parameters, because a widget can have only one parent.

These widget methods for controlling the parent-child relationship should be used with caution. While it is true that they have the same effect on the widget, they also have some side effects that can make debugging difficult. If you use `set_parent` to set a widget's parent, trying to remove the widget from the container using the container's `remove` method will not work. If you use `add` to assign a widget to a container, calling `unparent` will not work.

In short, if you set a widget's parent with a widget method, you must remove the widget from the container with a widget method. If you use a container method to add the widget, you must use a container method to remove the widget. However, there is an exception to the rule. In Listing 3-6, we used `reparent` to move the button back into the frame. `reparent` is really just a shortcut method for calling `remove` on the widget's parent container and then adding it to the new container. Since `reparent` uses the container methods internally, you can't use it when you have used `set_parent`.

Listing 3-7 is a reworked version of Listing 3-6. It uses `set_parent` and `unparent` instead of `add` and `remove`. At the end, there is a call to `reparent`.

Listing 3-7. *Using set_parent and unparent*

```
<?php
function testForParent($widget)
{
        $parent = $widget->get_parent();

        echo 'The ' . get_class($widget) . ' has ';
        if (isset($parent)) {
                echo 'a ' . get_class($parent);
        } else {
                echo 'no';
        }
        echo " parent.\n";
}

// Start with three widgets.
$window = new GtkWindow();
$frame  = new GtkFrame('I am a frame');
$button = new GtkButton("I'm a button");

testForParent($button);

$button->set_parent($frame)
testForParent($button);
```

```
// What if we want the button to be added directly to
// the window?
$button->unparent();
$button->set_parent($window);
testForParent($button);
$button->unparent();
testForParent($button);
$button->set_parent($frame);

// This line will throw an error message.
$button->reparent($window);
?>
```

As you can see, the call to `reparent` will throw an error, saying something to the effect that you are trying to give a widget two parents. Since the button was added using `set_parent`, when `reparent` calls the container's `remove` method, nothing happens. Then when `reparent` calls `add` on the new container, PHP-GTK balks, as it should. Under no circumstances can a widget have two parents. Because of this little problem, I recommend that you use only `add`, `remove`, and `reparent`. It will make your life just a little easier.

Summary

PHP-GTK is a well-structured hierarchy. The classes that make up PHP-GTK all have a relationship to one another and all depend on each other to make an application a success. Some classes are designed to organize data (objects); others are created specifically to interact with the user (widgets).

Aside from their class definitions, widgets also relate to each other through a parent-child relationship. Containers provide a context for their child widgets and give them an area in which to be displayed. The parent also plays a key role in whether or not the child is displayed at all. The parent-child relationship is one of the key elements in the makeup of a PHP-GTK application.

In Chapter 4, we will look at another fundamental principle of PHP-GTK: events. We will discuss what exactly an event-driven model is and how it is used in PHP-GTK. We will start to look at how your application will be able to respond to the user's actions. By the end of the next chapter, you will have all of the pieces you need to make a fully interactive application.