PowerPlant X 1.0
Developer's Guide

Revised 2003/08/13

metrowerks
Software Starts Here

Metrowerks, the Metrowerks logo, and CodeWarrior are registered trademarks of Metrowerks Corp. in the US and/or other countries. All other tradenames and trademarks are the property of their respective owners.

Copyright © Metrowerks Corporation. 2003. ALL RIGHTS RESERVED.

The reproduction and use of this document and related materials are governed by a license agreement media, it may be printed for non-commercial personal use only, in accordance with the license agreement related to the product associated with the documentation. Consult that license agreement before use or reproduction of any portion of this document. If you do not have a copy of the license agreement, contact your Metrowerks representative or call 800-377-5416 (if outside the US call +1-512-996-5300). Subject to the foregoing non-commercial personal use, no portion of this documentation may be reproduced or transmitted in any form or by any means, electronic or mechanical, without prior written permission from Metrowerks.

Metrowerks reserves the right to make changes to any product described or referred to in this document without further notice. Metrowerks makes no warranty, representation or guarantee regarding the merchantability or fitness of its products for any particular purpose, nor does Metrowerks assume any liability arising out of the application or use of any product described herein and specifically disclaims any and all liability. **Metrowerks software is not authorized for and has not been designed, tested, manufactured, or intended for use in developing applications where the failure, malfunction, or any inaccuracy of the application carries a risk of death, serious bodily injury, or damage to tangible property, including, but not limited to, use in factory control systems, medical devices or facilities, nuclear facilities, aircraft navigation or communication, emergency systems, or other applications with a similar degree of potential hazard**.

USE OF ALL SOFTWARE, DOCUMENTATION AND RELATED MATERIALS ARE SUBJECT TO THE METROWERKS END USER LICENSE AGREEMENT FOR SUCH PRODUCT.

How to Contact Metrowerks

Corporate Headquarters	Metrowerks Corporation 7700 West Parmer Lane Austin, TX 78729 U.S.A.
World Wide Web	`http://www.metrowerks.com`
Sales	Voice: 800-377-5416 Fax: 512-996-4910 Email: sales@metrowerks.com
Technical Support	Voice: 800-377-5416 Email: support@metrowerks.com

Table of Contents

1

Introduction

Mac OS X is an impressive operating system. It is responsive, stable, and offers a rich variety of sophisticated and intuitive managers and services to interact with the user.

Designing and developing an application for Mac OS X is not easy, however. To be successful and effective, an application must carefully handle countless details. Consequently, you, the application's developer, must also work hard. Thankfully, the PowerPlant X framework solves many of the problems you normally handle during development.

PowerPlant X is an application framework written in C++. It uses the Carbon interfaces in Mac OS X to provide significant parts of an application for you. The PowerPlant X framework is simple and flexible to use, but powerful and expressive. Let the PowerPlant X framework handle the details while you concentrate on the bigger, more interesting issues.

Before You Begin

Before you begin reading this book, you should be familiar with some topics that this book does not cover:

- Mac OS X software development, including the Carbon layer and HIToolbox features
- the C++ programming language, including the Metrowerks CodeWarrior C/C++ compilers and other tools for Mac OS X software development
- the CodeWarrior Integrated Development Environment (IDE)

How to Use This Book

To learn why you should take advantage of the PowerPlant X framework, read this chapter and "PowerPlant X Overview." These chapters introduce the PowerPlant X framework. They describe the framework's design, its conventions, and other information that you will find helpful while using the PowerPlant X framework.

After reading these general chapters, pick and choose among the remaining chapters to learn about specific parts of the framework. Each of these chapters introduces a small part of the framework, how it works, and how to use it in your application.

These chapters illustrate how to use the PowerPlant X framework with source code listings. Many of these listings are taken from the example projects for the PowerPlant X framework. To examine and try out these projects, go to this folder

`Metrowerks Folder``/(CodeWarrior Examples)/Mac OS X Examples/PowerPlant X/`

where *Metrowerks Folder* is the folder where you installed your CodeWarrior software and documentation for Mac OS X.

Using Other Documentation

Table 1.1 list other CodeWarrior documentation that is related to this guide.

Table 1.1 Related CodeWarrior documentation

To learn about...	refer to this documentation
using CodeWarrior tools and libraries to develop Mac OS software	*Targeting Mac OS*
CodeWarrior C/C++ compilers	*C Compilers Reference*
the CodeWarrior IDE	*IDE User Guide*
the Metrowerks Standard Library for C++ (MSL C++)	*MSL C++ Reference*

Table 1.2 lists the documentation for Mac OS X application programming interfaces (APIs) that the PowerPlant X framework uses. Apple Computer, Inc. publishes this documentation at

`http://developer.apple.com`.

Table 1.2 Related Mac OS documentation

To learn about...	refer to this documentation
Mac OS X in general	*Mac OS X: An Overview for Developers*
user interface conventions for Mac OS X applications	*Aqua Human Interface Guidelines*

Table 1.2 Related Mac OS documentation (*continued*)

To learn about...	refer to this documentation
HIToolbox, the object-oriented system on which Mac OS X bases its user interface	*Introducing HIView* *HIObject Reference* *HIView Reference*
the Carbon Event manager	*Handling Carbon Events* *Carbon Event Manager Reference*
the File manager	*File Manager Reference*
Navigation Services	*Navigation Services for Carbon: An Overview* *Programming With Navigation Services*
the Apple Event manager	*Apple Event Manager Reference*
Bundles and property lists	*Bundles* *Property Lists*

Conventions Used In This Book

Table 1.3 lists the typographical formats that this book uses to denote types of information.

Table 1.3 Typographical conventions

Information	Examples

items that appear in source code, including class names, variable names, and literal values	`PPx::Window`
	`apples != oranges`
	`"untitled"`

file names	`CustomViews.mcp` `cassert`
Internet addresses	`http://www.metrowerks.com`
items that appear on the screen, including menu commands, labels for controls, and window titles	the **File** menu's **Close** command **Preferences** window
new expressions or terminology	We call an object *persistent* if it is able to save its state in a permanent place, typically a file, so that it may be perfectly restored later.
placeholders for other values	*newtitle* is the new name for the object
titles of books	*CodeWarrior IDE User Guide*

2

PowerPlant X Overview

The PowerPlant X framework gets its elegance, flexibility, and power from its strong design and implementation. Read this chapter to introduce yourself to the PowerPlant X design and implementation.

- Design Principles
- Naming Conventions
- Mac OS Interfaces

Design Principles

The PowerPlant X framework follows a few principles in its design and takes advantage of some Mac OS and Standard C++ technologies in its implementation:

- Small Classes Give Big Flexibility and Power
- Mac OS X, Mach-O, Carbon, and HIToolbox
- Standard C++ Library

Small Classes Give Big Flexibility and Power

PowerPlant X classes are small, independent, and focused on a single purpose or task. PowerPlant X framework gets its power, simplicity, and flexibility by encouraging you to choose and combine only the classes you need to solve your programming problems rather than relying on overburdened classes with unused members.

For example, the PowerPlant X `Window` class encapsulates a Mac OS window. The `Window` class creates and removes a window on the screen. It also offers member functions to hide and show the window, and to retrieve and change its title.

By itself, such a class would only be a convenient set of function wrappers for a few Mac OS system calls. But the PowerPlant X `Window` class inherits from the `EventTarget`, `Attachable`, and `WindowCloserDoer` classes, too (Listing 2.1).

Listing 2.1 From class Window’s declaration

```
class Window:
  public Attachable, // Add and remove Attachment objects.
  public EventTarget, // Receive and handle events.
  public WindowCloseDoer // Handle window-closing behavior.
{
  // ...
};
```

Inheriting from class `Attachable` allows you to add objects derived from class `Attachment` to a `Window` object. By adding and removing `Attachment` objects, you gain the ability to customize an `Attachable` object’s behavior at runtime instead of customizing it through inheritance.

By inheriting from class `EventTarget`, Window objects become capable of receiving events from the Carbon Event manager. Also, class `EventTarget`, in turn, inherits from class `Persist`, which declares functions for saving and reconstructing an object from an external source, such as a file.

Class `WindowCloseDoer` handles the `Window` object’s behavior when the window closes. `WindowCloseDoer`, in turn, inherits from `SpecificEventDoer`, a class for handling a precise Carbon Event. The PowerPlant X framework declares a subclass of `SpecificEventDoer` for each kind of event that the Carbon Event manager generates. A class gains tremendous power by inheriting from `EventTarget` and any number of subclasses of `SpecificEventDoer`, without becoming difficult to implement or modify.

By inheriting from just 3 classes, a subclass becomes a sophisticated interface that is simple to implement and use.

Mac OS X, Mach-O, Carbon, and HIToolbox

The PowerPlant X framework takes advantage of the great, new technologies that Mac OS X introduces. PowerPlant X applications use the Mach-O executable format, the native format for Mac OS X applications.

PowerPlant X classes use the Carbon interfaces to interact with Mac OS managers and services. The PowerPlant X classes use the Mac OS HIToolbox object system to implement user interface views and controls.

Standard C++ Library

To manage its internal data structures, the PowerPlant X framework relies on the stability and flexibility of the ISO Standard C++ Library's container classes and other utilities.

Items in the ISO Standard C++ Library are in the `std` namespace.

Naming Conventions

- Namespaces
- Files, Classes, and Structures
- Function Names
- Variable and Argument Names
- Data Type and Template Names
- Macro and Constant Names

Namespaces

The PowerPlant X header and source code files use C++ namespaces to organize its classes.

The top PowerPlant X namespace is the `PPx` namespace. All PowerPlant X classes, templates, functions, and global variables are declared and defined in this namespace. (Preprocessor macros are defined outside of C++ namespaces.)

Listing 2.2 shows examples of using this namespace.

Listing 2.2 Using the PowerPlant X namespace

```
void BeginProgram()
{
  PPx::RegisterCommonXMLDecoders(); // 1
  PPx::RegisterCommonXMLEncoders(); // 2

  PPx_RegisterPersistent_(PPx::Window); // 3
  PPx_RegisterPersistent_(PPx::WindowContentView); // 4
  PPx_RegisterPersistent_(PPx::BindingsFrameAdapter); // 5
}
```

Items 1 and 2 call functions in the `PPx` namespace.

Items 3, 4, and 5 combine preprocessor macro calls (`PPx_RegisterPersistent_`) with names of classes in the `PPx` namespace (`Window`, `WindowContentView`, and `BindingsFrameAdapter`).

The PowerPlant X source files often use unnamed namespaces to restrict the scope of the items in the unnamed namespace to the file that they appear in.

Files, Classes, and Structures

The names of source code files end with `.cp`. The names of header files end with `.h`. PowerPlant X files have names that begin with `PPx`. Some files contain classes, functions, and structures to simplify the Mac OS interfaces. The names of these files begin with `Sys`.

Examples:

```
PPxView.h
PPxView.cpp
SysCFData.h
SysCFData.cp
```

Class and structure names begin with an uppercase letter. Classes and structures that encapsulate Mac OS functions and data structures begin with `Sys`.

Examples:

```
View
DataFork
SysAEHandler
```

Function Names

PowerPlant X functions and member functions begin with an uppercase letter. Function names are usually verbs.

Examples:

```
SetMenuCommandStatus()
RegisterCommonXMLDecoders()
```

Member functions that retrieve and change values in an object begin with `Get` and `Set`, respectively. Member functions that retrieve a logical state of an object begin with `Is` or `Has`. Examples:

`GetClassName()`

`SetBindings()`

Variable and Argument Names

Variable names begin with a lowercase letter. Variable members in a class begin with a lowercase `m` followed by an uppercase letter. Variable members in a structure do not begin with `m`. Class variables, variables declared with `static` in the class declaration, begin with an `s` followed by an uppercase letter. Examples:

`mSubViews`

`sRootObject`

Function argument names begin with `in`, `out`, or `io` to indicate input, output, and input/output values, respectively. Examples:

`inPreviousState`

`outResult`

`ioCurrentName`

Data Type and Template Names

Type names begin with an uppercase letter. Examples:

`ObjectMapT`

`BaseT`

Enumeration type names begin with an uppercase `E`. Examples:

`EMetaTarget`

`ESockState`

Template parameter types begin with an uppercase `T`. Template parameters specified with `class` should be a class type. Template parameters specified with `typename` may be any type. Examples:

`TEventClass`

`TEventKind`

Macro and Constant Names

Macro names begin with `PPx_`.

Macros that are used like functions use the same naming convention as functions, but begin with `PPx_` and end with `_`. Macro arguments follow the same naming conventions as the items they represent. Examples:

```
PPX_Version
```

```
#define PPx_Throw_(ExceptionClass, inWhat, inWhy)
```

Macros that are used to control conditional compilation use underscores to separate words. Examples:

```
PPx_Debug_Exceptions
```

```
PPx_Debug_Signals
```

Constant names begin with a description of the constant's data type, which begins with a lowercase letter, followed by an underscore, followed by a description of the constant, which begins with an uppercase letter. Examples:

```
err_BadParam
```

```
dataValue_True
```

If the constant represents the ID of a resource, the name begins with the resource type. Examples:

```
MBAR_Main
```

```
ALRT_Exception
```

Mac OS Interfaces

- System Wrappers
- Referring to Mac OS Interfaces

System Wrappers

The PowerPlant X framework offers several wrapper classes that simplify your application's interaction with Mac OS. For example, the PowerPlant X `SysAppleEvent` class encapsulates the Mac OS `AppleEvent` data structure, making it easier to create and manipulate. For more information, see "System Wrappers."

Referring to Mac OS Interfaces

Mac OS interfaces are in the global namespace. To refer to a Mac OS function, data type, structure or other item, make sure the name of the item has a "`::`" prefix. Listing 2.3 shows an example.

Listing 2.3 Using Mac OS interfaces with “::”

```
void MyLoadMenuBar()
{
  MenuBarHandle mbar = ::GetNewMBar(128);
}
```

3

Converting Interface Builder Files

The PowerPlant X View Converter application converts `.nib` files into PowerPlant X source code and data files. With the view converter, you can import user interface layouts that you create with Apple Computer's Interface Builder application into your PowerPlant X application. This chapter shows you how to use this tool.

- Using the Converter
- Adding Converted Files to Your Project
- Constructing Windows Generated by the View Converter

Using the Converter

The PowerPlant X View Converter application converts Carbon window layouts in `.nib` files into PowerPlant X data and source code files.

To convert a `.nib` file into PowerPlant X data and source code files, follow these steps:

1. Drag and drop the `.nib` file onto this application:

   `Metrowerks Folder``/Other Metrowerks Tools/PPxViewConverter`

   where *Metrowerks Folder* is the name of the folder where you have installed the CodeWarrior tools.

2. In the window that appears, select the items in the `.nib` file that you want to convert to PowerPlant X files by clicking their checkboxes in the **Use** column

3. To change the name of the C++ class that the view converter creates for a window layout, click its name in the **Class Name** column to type the new class name.

   The new class name must be a valid C++ class name. In other words, the name must begin with an alphabetic character or underscore, followed by alphanumeric and underscore characters.

4. Click **Convert to XML**.

For each window in the `.nib` file, the view converter creates a new PowerPlant X class that is a subclass of `Window`. Each new class contains data members for the subviews within the window.

The view converter stores the information for these new classes in source code and data files:

- a header file (`.h`) declares a new subclass of the PowerPlant X `Window` class
- a source code file (`.cp`) defines the functions of the new `Window` subclass
- a data file (`.pobj`) contains the layout information of the new `Window` subclass

The view converter stores these files in the same folder as the `.nib` file they were converted from. The new files are in a new subfolder.

Adding Converted Files to Your Project

After converting a .nib file to PowerPlant X files, you are ready to add the PowerPlant X files to your project:

- Adding Data Files
- Adding Header and Source Code Files

Adding Data Files

To add the converted `.pobj` files to your project, follow these steps.

1. In the CodeWarrior IDE, open the project for your PowerPlant X application.
2. In the project window, click the **Package** tab to see the hierarchy of folders for the application's bundle.
3. Place `.pobj` files that apply to all regions in this folder in the project window:

```
ApplicationName.app
  Contents
    Resources
```

   where *ApplicationName* is the name of your application.

4. Place region-specific `.pobj` files to the appropriate package folder in the project window:

```
ApplicationName.app
  Contents
```

```
Resources
  RegionName.lproj
```

where *ApplicationName* is the name of your application and *RegionName* is the name of region.

Adding Header and Source Code Files

To add the converted `.h` and `.cp` files to your project, follow these steps.

1. In the CodeWarrior IDE, open the project for your PowerPlant X application.
2. In the project window, click the **Files** tab to see the hierarchy of folders for the application's bundle.
3. Add the new `.h` and `.cp` files in the project window.

Constructing Windows Generated by the View Converter

After adding the converted files to your project, you are ready to refer to them in your PowerPlant X application's source code. To use the new classes, you must first register them with the PowerPlant X persistence mechanism before instantiating them.

Listing 3.1 shows an example that registers some converted classes with the PowerPlant X persistence mechanism.

Listing 3.1 Registering Converted Window Classes

```
// Include the files generated by the PowerPlant X view converter.
#include "MyDocumentWindow.h"
#include "MyProgressWindow.h"

void MyRegisterConvertedWindows()
{
  PPx_RegisterPersistent_(MyDocumentWindow);
  PPx_RegisterPersistent_(MyProgressWindow);
}
```

`MyRegisterConvertedWindows()` registers window classes named `MyDocumentWindow` and `MyProgressWindow`. Call this function before instantiating one of these classes, typically at application startup.

Listing 3.2 shows an example that instantiates a converted window class.

Listing 3.2 Instantiating a Converted Window Class

```
MyDocumentWindow* MyCreateDocWindow()
{
  // Create a window safely.
  std::auto_ptr < MyDocumentWindow > safewind =
    PPx::XMLSerializer::ResourceToObjects < MyDocumentWindow > (
      CFSTR("MyDocumentWindow"));

  // Window was successfully created, so we no longer need the auto_ptr.
  MyDocumentWindow* wind = safewind.release();

  // Make the new window visible.
  wind->Show();

  return wind;
}
```

`MyCreateDocWindow()` uses the automatic pointer class in the ISO Standard C++ Library to ensure exception safety. It calls the `XMLSerializer` class's `ResourceToObjects()` function to create a `MyDocumentWindow` object. `ResourceToObjects()` creates the object by reading the contents of the `MyDocumentWindow.pobj` file in the application's bundle, which was generated by the view converter application.

4

Views and Controls

The `View` class manages user interface items that appear in a window. For example, the most frequently-used subclasses of `View` manage Mac OS X controls, such as radio buttons, image wells, scroll bars, and so on.

This chapter describes the `View` class's properties and capabilities:

- View Characteristics
- Constructing Views and Controls
- Deleting Views
- Manipulating Controls
- Managing Hierarchical Views
- Creating Custom Views

View Characteristics

Class `View` and its subclasses implement the user interface elements that appear in a window and interact with the user:

- Controls are Subclasses of View
- Class View Uses HIToolbox
- The Superview and Subviews
- Views Receive and Act on Events
- Views are Persistent
- Views May Be Manipulated
- View Construction Requirements

Controls are Subclasses of View

Class `View` is an abstract class. This class implements the default behaviors that are common to all views. The PowerPlant X framework also implements subclasses of

`View` for each kind of Mac OS control. To create your own custom views and controls, the framework offers class `BaseView`. `BaseView` is a concrete subclass of class `View`.

Class View Uses HIToolbox

The PowerPlant X framework uses the Mac OS HIToolbox object system to manage the parts of a `View` object that interact with the operating system. Because it is based on HIViewRef, PowerPlant X classes use the `HIPoint`, `HIRect`, and `HISize` data types to specify coordinates and dimensions when manipulating objects derived from class `View`.

The Superview and Subviews

A view has a superview and may contain subviews. The PowerPlant X framework offers powerful features for managing this hierarchy:

- A view may add itself to a superview and a superview may remove some or all of its subviews.
- A view's visibility and whether or not is enabled are dominated by its superview; when a view becomes invisible or disabled, its superviews also become invisible or disabled, respectively.
- When a view's superview changes dimensions, it automatically resizes its subviews.

"Managing Hierarchical Views" describes the member functions that perform these tasks.

Views Receive and Act on Events

Because they inherit from `EventTarget`, view objects can handle commands and Carbon Events. `View` objects are also attachable, allowing them to be customized at runtime.

Views are Persistent

Inheriting from `EventTarget` also allows objects derived from `View` to be persistent.

Views May Be Manipulated

Views may be shown, hidden, activated, deactivated, enabled, and disabled. Some subclasses of views, for example, views that implement Mac OS controls, may also have their values changed and retrieved.

"Manipulating Controls" describes the member functions that perform these tasks.

View Construction Requirements

Like other PowerPlant X classes that are capable of persistence, classes you derive from `View` must meet these requirements to be properly constructed:

- a default constructor
- override the `InitState()`, `Initialize()`, and `FinishInit()` functions

Override the `InitState()` function to initialize data members from persistent external data that superclasses of your `View` subclass do not read. Your `InitState()` function should call the `InitState()` function of your class's immediate superclass.

Override `Initialize()` to initialize data from function arguments. Your `Initialize()` function should call the `Initialize()` function of your class's immediate superclass.

Override `FinishInit()` to complete any initialization your object needs that is common to both persistent and regular construction. In other words, `FinishInit()` allows you to avoid redundant initialization tasks shared by `InitState()` and `Initialize()`.

"Creating Custom Views" shows you how to use this style of object construction.

Constructing Views and Controls

To instantiate a control and place it in a superview, use the `CreateView()` function. Listing 4.1 shows an example.

Listing 4.1 Creating a control

```
void MyAddLabel(
  PPx::View* inSuperView,
  CFStringRef inText,
  const HIRect& inFrame)
{
```

```
  PPx::StaticText* label = PPx::CreateView < PPx::StaticText > (
    inSuperView,
    inFrame,
    PPx::visible_Yes,
    PPx::enabled_Yes,
    inText,
    nil);
}
```

The `CreateView()` function instantiates a new view object on the heap, adds it to a superview, configures its appearance and state, then calls its `Initialize()` and `FinishInit()` functions.

To place a `View` object in a window's content are, call the window's `GetContentView()` function. Use the result of this function as the superview argument when calling the `CreateView()` function.

Deleting Views

To remove the view from its superview without destroying it, call the view's `RemoveFromSuperView()`.

To destroy a view, including removing it from its superview, use the `delete` operator. The `View` class's destructor destroys its subviews recursively, then removes itself from its superview.

Manipulating Controls

The `View` class provides functions that directly manipulate the HIView object that a `View` object contains:

- Hiding and Showing Controls
- Enabling and Disabling Controls
- Examining and Changing a View's Value
- Responding to User Interaction

Hiding and Showing Controls

To show or hide a view, call the `SetVisible()` function. To check to see if a view is visible, call its `IsVisible()` function.

A view retains the status of its visibility even when its superview's visibility changes. When a superview becomes invisible, all of its visible subviews also become invisible. When a superview is made visible, any of its previously-visible subviews become visible again. Subviews that were previously invisible remain invisible.

Enabling and Disabling Controls

To enable and disable a view, call the `SetEnabled()` function. To check to see if a view is enabled, call its `IsEnabled()` function.

Like its visibility, a subview retains its enabled status when its superview's enabled state changes. For example, when a superview becomes disabled, its subviews retain their status even though they are also disabled. When the superview later becomes enabled, its subviews that were previously enabled become enabled again and its subviews that were previously disabled remain disabled.

Examining and Changing a View's Value

Table 4.1 lists the member functions in the `View` class that manipulate a view's value. Many Mac OS user interface controls use the value to specify the control's appearance and behavior. For example class `Slider` provides a Mac OS slider control. The control's minimum and maximum specify the slider control's range. The control's value specifies the position of the slider's thumb.

Table 4.1 Examining and changing a control's value

To do this...	call this member function
change the control's value	`SetValue()`
retrieve the control's value	`GetValue()`
change the control's minimum	`SetMinValue()`
retrieve the control's minimum	`GetMinValue()`
change the control's maximum	`SetMaxValue()`
retrieve the control's maximum	`GetMaxValue()`
change the control's text label	`SetTitle()`
retrieve the control's text label	`GetTitle()`

Responding to User Interaction

To respond to the events generated when a user interacts with a view, you may either create a custom view that inherits from the appropriate subclass of `EventDoer`, or you may add an attachment that intercepts the appropriate event. Listing 4.2 and Listing 4.3 show an example of inheriting from a subclass of `EventDoer` to allow a control to respond to a user-generated event.

Listing 4.2 Declaring an event handler to respond to user interaction

```
class MyWizardLauncher: public PPx::RoundButton,
    public PPx::ControlHitDoer
{
protected:
  virtual OSStatus DoControlHit(
    PPx::SysCarbonEvent& ioEvent,
    ControlRef inControl,
    ControlPartCode inPartCode,
    UIn32 inKeyModifiers);
};
```

Listing 4.3 Defining an event handler to respond to user interaction

```
OSStatus MyWizardLauncher::DoControlHit(
    PPx::SysCarbonEvent& /* ioEvent */,
    ControlRef /* inControl */,
    ControlPartCode /* inPartCode */,
    UIn32 /* inKeyModifiers */)
{
    MyShowWizWindow();
    return noErr; // handled event, do not propagate
}
```

Managing Hierarchical Views

Some views are containers for other views. For example, the PowerPlant X `RadioGroup` and `RadioButton` classes are both subclasses of `View`. A `RadioGroup` object displays a group of `RadioButton` objects. The `RadioGroup` object acts as a *superview* for a group of `RadioButton` objects, which are *subviews*.

A `View` object may have only one superview and 0 or more subviews. The PowerPlant X framework conveniently handles many of the details of working with a view's superview and subviews:

- Changing a View Hierarchy
- Resizing Views

Changing a View Hierarchy

`View::AddSubView()` adds a subview (and the sub-subviews within it) to a view. If the subview already belongs to another superview, it is detached from its old superview before being added to its new superview.

`View::RemoveFromSuperView()` removes a subview from its superview, making the subview invisible. The subview is not destroyed, it only detaches itself from its superview.

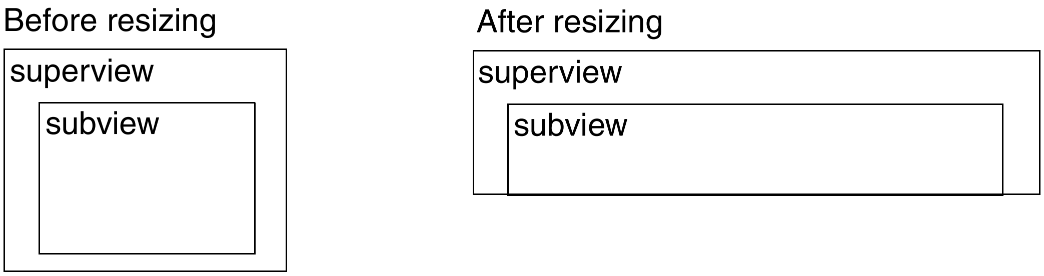
Resizing Views

To specify how a `View` object behaves when the screen dimensions of its superview changes, use a `FrameAdapter` object to calculate a `View` object's new dimensions. The `FrameAdapter` class's `AdaptFrame()` member accepts the original and updated superview dimensions and the current dimensions of the view being resized. Using these arguments, `AdaptFrame()` calculates the view's new dimensions.

Class `FrameAdapter` is an abstract class. Subclasses of `FrameAdapter` define specific resizing policies by overriding `AdaptFrame()`. Class `BindingsFrameAdapter` is a subclass of `FrameAdapter` that updates a view's frame based on whether or not each side of a subview is at a fixed distance to its superview's corresponding side.

Figure 4.1 shows an example. With a `BindingsFrameAdapter` object, the subview locks its left, top, and right sides to its superview's corresponding sides. After resizing the superview to increase its width and reduce its height, the subview's left, top, and right sides move to stay the same distance from the superview's corresponding sides. However, the subview's bottom side remains unchanged. Consequently, the superview clips the bottom part of the subview.

Figure 4.1 Using a BindingsFrameAdapter



Listing 4.4 shows the source code that implements this behavior.

Listing 4.4 Example of using BindingsFrameAdapter

```
void MyLockLeftTopRight(View* inSubView)
{
  PPx::BindingsFrameAdapter* adapter = new PPx::BindingsFrameAdapter;

  // left, top, right, bottom
  adapter->SetBindings(true, true, true, false);
  inSubView->SetFrameAdapter(adapter);
}
```

Behind the scenes, the `View` class relies on the `ControlBoundsChangedDoer` and `FrameAdapter` classes to manage the way it reacts to changes in its screen dimensions.

The `View` class inherits from `ControlBoundsChangedDoer` to react to `kEventControlBoundsChanged` events from the Mac OS Carbon Event manager. Class `View` overrides this event handler class's member function, `DoControlBoundsChanged()`, to notify each of its subviews by calling their `AdoptToSuperFrameSize()` functions. This function checks to see if the subview has a `FrameAdapter` object. If so, it calls the `FrameAdapter` object's `AdaptFrame()` function.

Creating Custom Views

To create your own view, use the `BaseView` class. `BaseView` is a concrete subclass of `View`. By itself, `BaseView` object does nothing. To specify how a `BaseView` object behaves you may either

- declare a subclass of `BaseView` that also inherits from event-handling classes at compile time
- add (and remove) event-handling attachments to a `BaseView` object at runtime

This section describes which events to handle and the kinds of custom view arrangements that the PowerPlant X framework offers:

- Choosing Which Events to Handle
- Customizing Views With Inheritance
- Customizing Views With Attachments

Choosing Which Events to Handle

Use event handlers to implement how your custom view appears and behaves. Whether you use inheritance or attachments, your custom view must handle events to draw the view and, optionally, interact with the user. The header file `PPxViewEvents.h` declares event-handling classes for views. This file offer a broad range of event handlers, but this section covers the most commonly used handlers, listed in Table 4.2.

Table 4.2 Deciding which events to handle in a custom view

To implement this behavior...	inherit from this class
the view's appearance on the screen	`ControlDrawDoer`
respond to a user's click in the control	`ControlHitDoer`
determine if a point is in the control	`ControlHitTestDoer`
change the control's state in response to a mouse click	`ControlHiliteChangedDoer`

Customizing Views With Inheritance

When declaring your custom view's class, derive it from `BaseView` and the appropriate event-handling classes that you want your custom view to act on. Listing 4.5 shows an example of a custom view's class declaration.

Listing 4.5 Declaration of a custom view using class inheritance

```
class  MyButton : public PPx::BaseView,
        public PPx::ControlDrawDoer,
        public PPx::ControlHitTestDoer,
        public PPx::ControlHiliteChangedDoer {
```

```
protected:
  virtual OSStatus DoControlDraw(
    PPx::SysCarbonEvent&  ioEvent,
    ControlRef  inControl,
    ControlPartCode  inPartCode,
    RgnHandle inClipRgn,
    CGContextRef inContext);

  virtual OSStatus DoControlHitTest(
    PPx::SysCarbonEvent& ioEvent,
    ControlRef inControl,
    const HIPoint& inHitPoint,
    ControlPartCode& outPartCode);

  virtual OSStatus DoControlHiliteChanged(
    PPx::SysCarbonEvent& ioEvent,
    ControlRef inControl);

  // Other member functions...
};
```

Class `MyButton` inherits from `BaseView`. From `BaseView`, `MyButton` receives the abilities of all objects derived from `View`.

`MyButton` also inherits from `ControlDrawDoer`, `ControlHitTestDoer`, and `ControlHiliteChangeDoer`, which allow `MyButton` to specify how it will be drawn and how it reacts to mouse movements and clicks. Listing 4.6 shows the definitions of the related member functions that handle these events.

Listing 4.6 Handling inherited events in a custom control

```
OSStatus
MyButton::DoControlDraw(
  PPx::SysCarbonEvent& /* ioEvent */,
  ControlRef inControl,
  ControlPartCode /* inPartCode */,
  RgnHandle /* inClipRgn */,
  CGContextRef inContext)
{
  HIRect frame;
  GetLocalFrame(frame);

  if (::IsControlHilited(inControl)) {
    ::CGContextSetRGBFillColor(inContext, 0.1, 0.1, 1.0, 0.3);
  } else {
```

```
    ::CGContextSetGrayFillColor(inContext, 0.5, 0.3);
  }

  ::CGContextFillRect(inContext, frame);

  return noErr;
}

OSStatus
MyButton::DoControlHitTest(
  PPx::SysCarbonEvent& /* ioEvent */,
  ControlRef /* inControl */,
  const HIPoint& inHitPoint,
  ControlPartCode& outPartCode)
{
  OSStatus result = eventNotHandledErr;

  HIRect  frame;
  GetLocalFrame(frame);
  if (::CGRectContainsPoint(frame, inHitPoint)) {
    outPartCode = kControlButtonPart;
    result = noErr;
  }
  return result;
}

OSStatus
MyButton::DoControlHiliteChanged(
  PPx::SysCarbonEvent& /* ioEvent */,
  ControlRef inControl)
{
  ::HIViewSetNeedsDisplay(inControl, true);
  return noErr;
}
```

Customizing Views With Attachments

An alternative to implementing event handling behaviors with class inheritance is to use attachments. `Attachment` objects offer an advantage over class inheritance: they can be added to and removed from an `Attachable` object at runtime. Listing 4.7 shows an example.

Listing 4.7 Using an Attachment object with a View

```
void MyWorkWithFrame(PPx::View* inView)
{
  // Create and attach the attachment.
  MyDrawFrameAttachment*  frameAtt = new MyDrawFrameAttachment;
  frameAtt->Initialize(inView);
  frameAtt->FinishInit();
  inView->AddAttachment(frameAtt);

  // Use the inView object with its new attachment.

  // Note: RemoveAttachment also destroys the attachment!
  inView->RemoveAttachment(frameAtt);
}
```

`MyDrawFrameAttachment` draws a rectangle around the object it is attached to. Although `MyWorkWithFrame()` accepts an argument of type `View`, `MyDrawAttachment` can be attached to any object that inherits from `EventTarget` and `Attachable`. (Of course, `MyDrawAttachment` will have no effect on the object it is attached to unless that object receives the events that `MyDrawAttachment` handles.)

Listing 4.8 shows the `MyDrawFrameAttachment` class used in Listing 4.7.

Listing 4.8 Declaration for an Attachment object

```
class MyDrawFrameAttachment :
  public PPx::TargetAttachment,
  public PPx::ControlDrawDoer {

public:
  void Initialize(PPx::EventTarget* inTarget);

protected:
  virtual void InitState(PPx::ObjectState& inState);
  virtual OSStatus DoControlDraw(
    PPx::SysCarbonEvent& ioEvent,
    ControlRef inControl,
    ControlPartCode inPartCode,
    RgnHandle inClipRgn,
    CGContextRef inContext);

private:
  virtual CFStringRef ClassName() const;
```

```
  void FinishInit();

  PPx::SysEventTargetRef mEventTarget;
};
```

`MyDrawFrameAttachment` inherits from `TargetAttachment` and `ControlDrawDoer`. `TargetAttachment` inherits from `Attachment`. With `TargetAttachment`’s members, `MyDrawFrameAttachment` manages an event target. With `ControlDrawDoer`, `MyDrawFrameAttachment` handles draw events from the Carbon Event manager.

`Initialize()` accepts the object that will receive the events that the attachment handles. `FinishInit()` allows the attachment object to initialize other data members (Listing 4.9).

Listing 4.9 Initializing an attachment

```
void
MyDrawFrameAttachment::Initialize(
  PPx::EventTarget* inTarget)
{
  SetEventTarget(inTarget);
}

void
MyDrawFrameAttachment::FinishInit()
{
  PPx::EventTarget* target = GetEventTarget();

  if (target != nil) {
    ::EventHandlerRef eventHandler =
      PPx::ControlDrawDoer::Install(target->GetSysEventTarget());
    mEventHandler.Adopt(eventHandler);
  }
}
```

`FinishInit()` installs the `MyDrawFrameAttachment` object as a `ControlDrawDoer` event handler and remembers the result in `mEventHandler`, a `SysEventHandler` object. A `SysEventHandler` data member ensures that the event handler will be properly uninstalled when a `MyDrawFrameAttachment` object is destroyed.

`ClassName()` and `InitState()` do the tasks needed for the PowerPlant X persistence mechanism (Listing 4.10). Because a `MyDrawFrameAttachment` object does not have any of its own data that needs to be saved and restored, its `InitState()` function only calls the `InitState()` function of its parent class.

Listing 4.10 Handling persistence in an attachment

```
CFStringRef
MyDrawFrameAttachment::ClassName() const
{
  return CFSTR("MyDrawFrameAttachment");
}

void
MyDrawFrameAttachment::InitState(
  PPx::ObjectState& inState)
{
  PPx::TargetAttachment::InitState(inState);
}
```

`DoControlDraw()`, inherited from `ControlDrawDoer`, implements the behavior for reacting to a control draw event (Listing 4.11).

Listing 4.11 Handling an event in an attachment

```
OSStatus
MyDrawFrameAttachment::DoControlDraw(
  PPx::SysCarbonEvent& /* ioEvent */,
  ControlRef inControl,
  ControlPartCode /* inPartCode */,
  RgnHandle /* inClipRgn */,
  CGContextRef inContext)
{
  HIRect frame;
  ::HIViewGetFrame(inControl, &frame);
  frame.origin.x = 0;
  frame.origin.y = 0;
  ::CGContextStrokeRect(inContext, frame);
  return eventNotHandledErr; // Propagate the event.
}
```

`DoControlDraw()` returns `eventNotHandledErr`, which indicates that the event should also be sent to the next event handler in the Carbon Event manager's event stack. By allowing this event to propagate, the `MyDrawFrameAttachment` object allows other handlers the opportunity to receive this event, including other attachments and the object to which the `MyDrawFrameAttachment` is attached.

Windows

In this chapter we learn how to manipulate windows using the PowerPlant X `Window` class. The `Window` class encapsulates the usual Mac OS X Window Manager features. It also adds persistence along with view and event handling capabilities. Thanks to the PowerPlant X framework, a sophisticated user interface element becomes surprisingly simple to manipulate and customize.

The sections in this chapter show you how to use the PowerPlant X features that manage Mac OS windows:

- Window Characteristics
- Constructing Windows
- Closing Windows
- Adding Subviews to Windows
- Customizing Window Behavior

Window Characteristics

This section introduces class Window:

- Windows and Views
- Window Construction Requirements
- Common Window Tasks

Windows and Views

In the PowerPlant X framework, the `Window` class is not derived from the `View` class. In other words, the Mac OS Window managed by a PowerPlant X `Window` object, by itself, is blank. To display its contents, the `Window` class contains a single root view that acts as a superview for everything that appears in the window's content area.

In the PowerPlant X framework, this superview is called the *content view*. To retrieve a `Window` object's content view, call the object's `GetContentView()` member.

Window Construction Requirements

Classes derived from `Window` must meet these requirements to be properly constructed:

- a default constructor
- override the `InitState()`, `Initialize()`, `FinishInit()` functions

The `InitState()` function initializes a newly-constructed object's data members from external data. Make sure that your `InitState()` function also calls the `InitState()` function of your `Window` subclass's immediate superclass.

The `Initialize()` function initializes a new object from its arguments. Make sure that your `Initialize()` function also calls the `Initialize()` function of your `Window` subclass's immediate superclass.

Override `FinishInit()` to do initialization tasks that are common to both the `Initialize()` and `InitState()` functions.

Common Window Tasks

The `Window` class has member functions that perform some commonly-used tasks for window manipulation. Table 5.1 lists those features.

Table 5.1 Commonly-used window manipulations

To do this...	call this function in class `Window`
make the window visible	`Show()`
make the window invisible	`Hide()`
check to see if the window is visible	`IsVisible()`
change the window's title	`SetTitle()`
get the window's title	`GetTitle()`
get the reference to the Mac OS window being managed by a PowerPlant X Window object	`GetSysWindow()`
get the PowerPlant X Window object that manages a Mac OS window	`GetWindowObject()`

Constructing Windows

Listing 5.1 shows an example of instantiating a new window.

Listing 5.1 Creating a new window

```
PPx::Window* MyMakeWindow()
{
  // Construct and set up the window.
  Rect myBounds = { 100, 50, 400, 450 }; // top, left, bottom, right

  PPx::Window* myWindow = new PPx::Window;
  myWindow->Initialize(
    kDocumentWindowClass, // class
    Window::GetDefaultAttributes(),
    myBounds, // dimensions, in global coordinates
    CFSTR("My Window"));  // title
  myWindow->FinishInit();

  // Add subviews to the window here.

  // Make the window visible.
  myWindow->Show();
}
```

To create a new window, just construct an object of class `Window` using the `new` operator, then call the object's `Initialize()` and `FinishInit()` functions. The window is ready to accept subviews at this point. See "Adding Subviews to Windows" for more information. After adding the subviews to the window's content area and the window is ready to be displayed, call the `Show()` function.

Closing Windows

To close a window, call its `Close()` function. To customize how a window reacts when it is about to be closed, override the `Window` class's `DoWindowClose()` function.

Among other classes, the `Window` class inherits from `WindowCloseDoer`. `WindowCloseDoer` is a subclass of `EventDoer` that handles the Carbon Event manager's window close event (kind `kEventWindowClose`, class `kEventClassWindow`). Consequently, the member function in class `Window` that implements the event handler for `WindowCloseDoer` is `DoWindowClose()`.

The rest of this section describes when to call the `Window::Close()` function and how to implement a window's behavior when it closes:

- When to Close a Window

- Customizing a Window's Close Behavior

When to Close a Window

The `Window` class's `Close()` function does not close a window directly. Instead, it posts a window close event, allowing the `Window` class's `WindowCloseDoer` handler to act on the event.

Typically, the user closes a window by

- clicking the window's close button
- choosing **Close** from the **File** menu

The PowerPlant X framework handles clicks on a window's close button for you. When the user clicks a window's close button, the Carbon Event manager issues a `kEventWindowClose` event. The Carbon Event manager dispatches this event to the `Window::DoWindowClose()` function.

The PowerPlant X framework defers handling menu commands to your application, however, including the **File** menu's **Close** command. To handle this command, mix your application's class with the `CommandHandler` class to implement behavior for the `kHICommandClose` Carbon Event. Listing 5.2 shows the parts of a window class's declaration that handle the **Close** command.

Listing 5.2 Declarations for handling the File menu's Close command.

```
class MyApp:
  public PPx::Application,
  public PPx::CommandHandler < kHICommandClose >
  // Other superclasses...
{
protected:
  virtual OSStatus DoSpecificCommand(
    PPx::CommandIDType < kHICommandClose >,
    PPx::SysCarbonEvent& ioEvent);

  virtual OSStatus DoSpecificCommandStatus(
    PPx::CommandIDType < kHICommandClose >,
    PPx::SysCarbonEvent& ioEvent);

  // Other member functions...
}
```

With the `CommandHandler` member functions, `DoSpecificCommand()` and `DoSpecificCommandStatus()`, you determine the **Close** command's status and what it does when the user chooses it. Specifically, the **Close** command should be enabled if there is a window active and it should call the front-most window's `Close()` member. 5.3 shows the functions in the `MyApp` class that implement this behavior.

Listing 5.3 Definitions for handling the File menu's Close command

```
OSStatus
MyApp::DoSpecificCommand(
    PPx::CommandIDType < kHICommandClose >,
    PPx::SysCarbonEvent& /* ioEvent */)
{
  PPx::Window* wind = PPx::Window::GetWindowObject(::FrontWindow());
  if (wind != nil) {
    wind->Close();
  }
  return noErr;
}

OSStatus
MyApp::DoSpecificCommandStatus(
    PPx::CommandIDType < kHICommandClose >,
    PPx::SysCarbonEvent& /* ioEvent */)
{
  bool isPPxWind = PPx::Window::GetWindowObject(::FrontWindow());

  PPx::EventUtils::SetMenuCommandStatus(kHICommandClose, isPPxWind);
  return noErr;
}
```

Customizing a Window's Close Behavior

The default implementation for `DoWindowClose()` destroys the window object by using the `delete` operator. The `Window` class's destructor ensures that the window's subviews are destroyed.

A typical reason to change a window's close behavior is to prompt the user to save changes to the window's contents. If the user's choice eventually leads to the window being closed, delete the window. To prevent the close event from propagating to previously-installed handlers in the Carbon Event stack, return `noErr`. Listing 5.4 shows an example.

Listing 5.4 Customizing how a window closes

```
// In a header file...
class MyCustomWindow: public PPx::Window
{
protected:
  virtual OSStatus DoWindowClose(
    SysCarbonEvent& ioEvent,
    WindowRef inWindow);
};

// In a source code file...
OSStatus
MyCustomWindow::DoWindowClose(
  SysCarbonEvent& ioEvent,
  WindowRef inWindow)
{
  bool cancel = MyDontSaveCancelSave();
  if (cancel)
  { // User chooses to close the window.
    delete this;
  }
  return noErr;
}
```

Adding Subviews to Windows

To add a subview to a window's content view, call the `Window` object's `AddSubView()` function.

Alternatively, PowerPlant X classes that derive from the `View` class have an `Initialize()` function that requires an argument specifying a superview. When creating a subview derived from class `View`, use the `CreateView()` function with the value returned from your window's `GetContentView()` function. Listing 5.5 shows an example.

Listing 5.5 Adding subviews to windows

```
  PPx::Window* myWindow1 = MyMakeWindow();
  PPx::Window* myWindow2 = MyMakeWindow();

  // Create a view, in this case a class derived from View.
  HIRect myViewBounds = { 1, 1, 25, 25}; // left, top, width, height
```

```
  // Set up the new view, adding it as a subview of myWindow1’s
  // content view.
  PPx::ChasingArrows* myArrows =
    PPx::CreateView < PPx::ChasingArrows > (
      myWindow1->GetContentView(),
      myViewBounds,
      PPx::visible_Yes,
      PPx::enabled_Yes);

  // Add a view to a window after the view has been created.
  // In this case, myArrows will be removed from myWindow1 and
  // added to myWindow2.
  myWindow2->AddSubView(myArrows);
```

Customizing Window Behavior

Thanks to the `EventDoer` class, changing the way a window behaves is simple: just derive a new class from `Window` and the appropriate subclasses of `EventDoer`. The PowerPlant X framework offers many subclasses of `EventDoer` customize practically any facet of a window’s appearance and interaction with the user:

- the classes in `PPxWindowEvents.h` customize the way a window behaves
- the classes in `PPxWindowDefEvents.h` customize the window’s definition

Like many other PowerPlant X objects, a `Window` object can be customized through inheritance or through attachments. Listing 5.6 and Listing 5.7 show an example of customizing a window’s behavior by inheriting from a subclass of `EventDoer`.

Listing 5.6 Declarations for customizing a window’s behavior

```
class MyWizWindow: public PPx::Window,
    public PPx::WindowGetIdealSizeDoer
{
protected:
  // Compute the preferred size of my wizard window.
  virtual OSStatus DoWindowGetIdealSize(
    SysCarbonEvent& ioEvent,
    WindowRef inWindow,
    Point& outIdealSize);

private:
  // Custom initialization goes here.
```

```
  virtual void FinishInit();
};
```

Listing 5.7 Definitions for customizing a window's behavior

```
OSStatus
MyWizWindow::DoWindowGetIdealSize(
  SysCarbonEvent& /* ioEvent */,
  WindowRef /* inWindow */,
  Point& outIdealSize)
{
  outIdealSize.v = 300;
  outIdealSize.h = 500;
};

void
MyWizWindow::FinishInit()
{
  // Register our event handler.
  EventTargetRef targRef = GetSysEventTarget();
  PPx::WindowGetIdealSizeDoer::Install(targRef);
}
```

Applications

The PowerPlant X `Application` class handles application-level capabilities. This chapter describes this class and how to use it.

- Application Characteristics
- Handling Custom Commands
- Launching
- Quitting

Application Characteristics

The `Application` class implements the behaviors for launching, quitting, and other application-level tasks. Class `Application` inherits from `ApplicationEventTarget` and `Attachable`:

- `ApplicationEventTarget` allows applications to receive events from the Mac OS Carbon Event manager and handles persistence by eventually inheriting from class `Persistent`.
- `Attachable` allows `Attachment` objects to be associated with applications.

Handling Custom Commands

By itself, class `Application` only defines member functions for handling persistence and for starting the Carbon Event manager's event loop. To customize your application, create a subclass of `Application`. Listing 6.1 shows an example of an application that handles an **Import** command.

Listing 6.1 Declaring a custom application class to handle a command

```
class MyApp:
  public PPx::Application,
  public PPx::CommandConverter,
  public PPx::CommandHandler < cmd_Import >
  // other superclasses
{
public:
  const UInt32 cmd_Import = 'impt';
```

```
  virtual OSStatus DoSpecificCommand(
    PPx::ComandIDType < cmd_Import >,
    PPx::SysCarbonEvent& ioEvent);

  virtual OSStatus DoSpecificCommandStatus(
    PPx::ComandIDType < cmd_Import >,
    PPx::SysCarbonEvent& ioEvent);

  void DoImport();

  bool CanImport() const;

  // Other member functions...
};
```

Class `MyApp` inherits from `CommandConverter` to translate raw Carbon Events into events that contain custom commands.

`MyApp` also inherits from `CommandHandler` to implement a handler for a specific command, in this case, the **Import** command. (Instead of class inheritance, you could also create a separate attachment class that handles the **Import** command.)

The `DoSpecificCommand()` function, inherited from class `CommandHandler`, implements `MyApp`'s **Import** command. `DoSpecificCommandStatus()`, also inherited from `CommandHandler`, computes whether or not the `Import` command is enabled (Listing 6.2).

Listing 6.2 Handling a command in an application

```
OSStatus
MyApp::DoSpecificCommand(
  PPx::CommandIDType < MyApp::cmd_Import >,
  PPx::SysCarbonEvent& /* ioEvent */)
{
  DoImport();
}

OSStatus
MyApp::DoSpecificCommandStatus(
  PPx::CommandIDType < MyApp::cmd_Import >,
  PPx::SysCarbonEvent& /* ioEvent */)
{
  PPx::EventUtils::SetMenuCommandStatus(
      cmd_Import, CanImport());
```

```
  return noErr; // Do not propagate the event.
}
```

Launching

Starting a PowerPlant X application requires a few steps:

- initialize Mac OS services
- set PowerPlant X debugging options
- register PowerPlant X persistence services and classes
- instantiate an `Application` object and call its `Run()` function

Listing 6.3 shows a simple definition for a PowerPlant X application's `main()` function.

Listing 6.3 Starting an application

```
const short MBAR_menuBar = 128;

void InitMacOS();
void InitDebug();
void InitPersistence();
void RunApp();

int main(void)
{
  InitMacOS();
  InitDebug();
  InitPersistence();
  RunApp();
}
```

The `InitMacOS()` function initializes the Mac OS services that the application needs and installs the application's menu bar (Listing 6.4).

Listing 6.4 Setting up Mac OS services and loading a menu bar

```
void InitMacOS()
{
  ::InitCursor();
```

```
  MenuBarHandle menuBar = ::GetMenuBar(MBAR_menuBar);
  ::SetMenuBar(menuBar);
}
```

The `InitDebug()` function configures the PowerPlant X signal and exception macros (Listing 6.5). A preprocessor macro, `My_Debuggable_Build_`, controls the source code generated by the signal and exception macros. See "Testing and Debugging" and "Exception and Error Handling" for more information on these macros.

Listing 6.5 Specifying the behavior of PowerPlant X signals and exceptions

```
void InitDebug()
{
#if My_Debuggable_Build_
  PPx_SetDebugThrow_Alert_();
  PPx_SetDebugSignal_Alert_();
#else
  PPx_SetDebugThrow_Nothing_();
  PPx_SetDebugSignal_Nothing_();
#endif
}
```

The `InitPersistence()` function registers the PowerPlant X decoders, encoders and the persistent-capable classes that the application uses (Listing 6.6).

Listing 6.6 Registering decoders, encoders, and classes for persistence

```
// Call once, at application startup.
void InitPersistence()
{
  // Register data decoders and encoders.
  PPx::RegisterCommonXMLDecoders();
  PPx::RegisterCommonXMLEncoders();

  // Register PowerPlant X classes that the application uses.
  PPx_RegisterPersistent_(PPx::Window);
  PPx_RegisterPersistent_(PPx::WindowContentView);

  // Register custom classes.
  PPx_RegisterPersistent_(MyDrawFrameAttachment);
  PPx_RegisterPersistent_(MyCustomApp);
}
```

Calling `InitPersistence()` when initializing an application allows the application to save and restore objects. The external representation of objects will be in XML format, and will be capable of saving and restoring objects of class `Window`, `WindowContentView`, `MyDrawFrameAttachment`, and `MyCustomApp`.

The `RunApp()` function instantiates and runs the application object (Listing 6.7).

Listing 6.7 Running an Application object

```
void RunApp()
{
  MyApp app;

  app.Run();
}
```

Quitting

There are two occasions when an application quits:

- the user chooses the **Quit** command from the application menu, causing the Carbon Event manager to issue a `kHICommandQuit` event
- an application or Mac OS sends an Apple Event to quit, `kAEQuitApplication`

Your application does not need handlers for both the `kHICommandQuit` and `kAEQuitApplication` events, however. Instead, your application can take advantage of the Mac OS default command handler for the `kHICommandQuit` event. This default handler for `kHICommandQuit` issues a `kAEQuitApplication` Apple Event. So, to handle both of these events, make sure your subclass of `Application` also inherits from `AEQuitApplicationDoer`.

To always enable the **Quit** command in the application menu, your class should also inherit from `SpecificCommandEnableDoer`.

Listing 6.8 shows the parts of the `MyApp` class declaration that handle quitting.

Listing 6.8 Declaring a custom application class to handle the Quit command

```
class MyApp:
  public PPx::Application,
  // other superclasses
  public PPx::AEQuitApplicationDoer,
  public PPx::SpecificCommandEnableDoer < kHICommandQuit >
```

```
{
public:
  // Other member functions...

  virtual OSStatus DoAEQuitApplication(
    const AutoAEDesc&  inAppleEvent,
    AutoAEDesc&  outAEReply);

  void Disconnect();
  void ReleaseSubsystems();

};
```

The `DoAEQuitApplication()` cleans up the application’s state before it quits and stops the Carbon Event manager loop, which returns control to the application class’s `Run()` function. (Listing 6.9).

Listing 6.9 Handling the command and Apple Event for quitting an application

```
OSStatus MyApp::DoAEQuitApplication(
  const AutoAEDesc&  /* inAppleEvent */,
  AutoAEDesc&  /* outAEReply */)
{
  Disconnect();
  ReleaseSubsystems();

  // Leave the Carbon Event manager’s event loop.
  ::QuitApplicationEventLoop();

  return noErr;
}
```

Utility and Operating System Classes

Besides classes that implement an application's appearance and behavior, PowerPlant X offers utility classes to simplify tedious tasks and handle tasks not directly related to implementing application features:

- Testing and Debugging
- Exception and Error Handling
- Character Strings
- System Wrappers

Testing and Debugging

Use the `PPx_Signal` macros to verify the integrity of your program as it runs, like `assert()` in Standard C++ Library.

- Verifying With Signals
- Controlling Signals

Verifying With Signals

Use the `PPx_Signal` macros to test the assumptions you rely on while developing your application. A signal does nothing if its test succeeds and reports failed tests to you, allowing you to detect bugs more easily. Some common uses for PowerPlant X signals include

- verifying the validity of arguments passed to a function
- verifying the value returned by a function
- verifying that the computations performed in a function are correct before the function returns

---

NOTE Like most assertion facilities, make sure that the signal's test condition has no side effects. In other words, the evaluation of the condition that is passed to a `PPx_SignalIf_` or `PPx_SignalIfNot_` macro must not change the program's state.

Table 7.1 lists describes these macros.

Table 7.1 PPx_Signal macros

To do this...	use this macro
verify that a condition is true	`PPx_SignalIf_(`*`cond`*`)` where *cond* is a boolean expression
verify that a condition is false	`PPx_SignalIfNot_(`*`cond`*`)` where *cond* is a boolean expression
always report a signal	`PPx_SignalString_(`*`str`*`)` where *str* is a character string

Controlling Signals

To activate these macros, define the `PPx_Debug_Signals` preprocessor macro with a true value. To turn off these macros, define `PPx_Debug_Signals` with `0` or undefine it. Make sure the definition of this macro occurs before including the `PPxDebugging.h` header file.

The signal macros report the cause of their signal in an alert box, in the debugger, to the standard error stream, or not at all. By default, the `PPx_Signal` macros do nothing, even if they are activated. Table 7.2 lists the macros that control the behavior of the `PPx_Signal` macros.

Table 7.2 Controlling PPx_Signal behavior

To specify that signals should...	use this macro
not be reported	`PPx_SetDebugSignal_Nothing_()`
appear in an alert box	`PPx_SetDebugSignal_Alert_()`
be reported in the debugger	`PPx_SetDebugSignal_Debugger_()`
be output to the error stream, `stderr`	`PPx_SetDebugSignal_Console_()`

When an alert box appears to report a signal, you are prompted to continue executing the application, stop the application, go to the debugger, or continue without reporting future signals.

When reporting signals to the standard error stream, signal output appears in one of a few ways:

- the SIOUX window, if you are using the CodeWarrior SIOUX library in your application

- the CodeWarrior IDE's **Log Window**, if you are using the CodeWarrior IDE's debugger without the SIOUX library
- in a terminal window, if you have launched your PowerPlant X application from the command line
- redirected to a file, if you have launched your PowerPlant X application from the command line and redirected the standard error stream's output

Exception and Error Handling

`PPx_Throw` macros raise a C++ exception, allowing you to add error-handling capabilities in your application.

- Throwing Exceptions
- Controlling Exception Behavior
- Exception Classes
- Getting Location Information

Throwing Exceptions

Use `PPx_Throw` macros to check for and respond to error conditions while your program runs. For example, if your application attempts to read a file but fails, throw an exception so that your application's error-handling features can correct the problem or report it to the user.

Table 7.3 lists describes these macros.

Table 7.3 PPx_Throw macros

To do this...	use this macro
check that a condition is true	`PPx_ThrowIf_(`*`cond, except, what, why`*`)` where *cond* is a boolean expression, *except* is the type of exception to throw, *what* is the exception ID, and *why* is a character string describing the exception
check that a pointer is not nil	`PPx_ThrowIfNil_(`*`ptr, except, what, why`*`)` where *ptr* is a pointer expression, *except* is the type of exception to throw, *what* is the exception ID, and *why* is a character string describing the exception

Table 7.3 PPx_Throw macros (*continued*)

To do this...	use this macro
check an Mac OS error code	`PPx_ThrowIfOSError_(`*`error, why`*`)` where *error* is a value of type `OSStatus` and *why* is a character string describing the exception. This macro throws an exception of type `OSError`.
always throw an exception	`PPx_Throw_(`*`except, what, why`*`)` where *except* is the type of exception to throw, *what* is the exception ID, and *why* is a character string describing the exception

Controlling Exception Behavior

Like `PPx_Signal` macros, `PPx_Throw` macros optionally report the cause of an exception. (Although `PPx_Signal` macros can be turned off, `PPx_Throw` macros always throw exceptions even when they do not report them.)

`Exception` objects optionally store a description of the cause of the exception and the source code location where the exception was thrown. Define `PPx_Debug_Exceptions` macro to a true value to record this information. To declare `Exception` classes that do not record this information, define `PPx_Debug_Exceptions` to `0` or undefine it. Make sure the definition of this macro occurs before including the `PPxDebugging.h` header file.

These macros report the cause of the signal in an alert box, in the debugger, or not at all. By default, the `PPx_Throw` macros do not report exceptions. Table 7.4 lists the macros that control the behavior of the `PPx_Throw` macros.

Table 7.4 Controlling how PPx_Throw macros report an exception to the user

To specify that exceptions should...	use this macro
not be reported	`PPx_SetDebugThrow_Nothing_()`
appear in an alert box	`PPx_SetDebugThrow_Alert_()`
be reported in the debugger	`PPx_SetDebugThrow_Debugger_()`
be output to the error file, `stderr`	`PPx_SetDebugThrow_Console_()`

When an alert box appears to report an exception, you are prompted to continue executing the application, stop the application, go to the debugger, or continue without reporting future exceptions (although any future exceptions will still be thrown).

When reporting exceptions to the standard error stream, exception output appears in one of a few ways:

- the SIOUX window, if you are using the CodeWarrior SIOUX library in your application
- the CodeWarrior IDE's **Log Window**, if you are using the CodeWarrior IDE's debugger without the SIOUX library
- in a terminal window, if you have launched your PowerPlant X application from the command line
- redirected to a file, if you have launched your PowerPlant X application from the command line and redirected the standard error stream's output

Exception Classes

The `PPx_Throw` macros throw C++ exceptions with objects derived from the PowerPlant X `Exception` class.Listing 7.1 lists the hierarchy of exception classes.

Listing 7.1 PowerPlant X Exception Class Hierarchy

```
Exception
  OSError
    OSErrorCode
  LogicError
  RuntimeError
  DataError
```

When thrown in an exception with the PowerPlant X `PPX_Throw` macros, `Exception` objects, and objects of its subclasses, contain information about the exception. Table 7.5 and Table 7.6 list the member functions that retrieve this information.

If the `PPx_Debug_Exceptions` macro is undefined or defined with a false value, the `Why()` function returns an empty string and the `Where()` function returns `sourceLocation_Nothing`, which has nil pointers for the file and function names, and 0 for the line number.

Table 7.5 Getting information about an exception

To get this information...	use this member function in class Exception
the exception ID	`What()`
Mac OS error code that caused the exception (available in `OSError` and `OSErrorCode` classes only)	`GetOSErrorCode()`

Table 7.6 Additional information about exceptions

To get this information when `PPx_Debug_Exceptions` is defined with a true value...	use this member function in class Exception
a textual description	`Why()`
location in source code where the exception was thrown	`Where()`

Getting Location Information

If the `PPx_Debug_Exceptions` preprocessor symbol is defined with a true value, class `Exception` and its subclasses store information about an exception's source code location. The `SourceLocation` structure has members that specify a function name, line number, and source code file.

Character Strings

The PowerPlant X `CFString` class makes the Core Foundations string type in the Mac OS Core Foundations interfaces convenient to use.

`PPX::CFString` is derived from `PPx::CFMutableObject`, which declares a conversion operator for returning a pointer to a `CFString`. In other words, to the compiler, a `PPx::CFString` object can be converted to an object of type `CFStringRef`. Thanks to this conversion operator, you may pass `PPx::CFString` objects to Mac OS routines that require `CFStringRef` arguments.

`CFString` declares several functions to construct, append, assign, search, compare, and perform many other manipulations on character strings. The PowerPlant X framework overloads many of these functions to allow Core Foundation strings to be used with C strings, Pascal strings, and other character string formats.

System Wrappers

In your PowerPlant X application you will often need to call Mac OS routines directly. Consequently, your application will also need to manipulate Mac OS data types and structures. To simplify the system calls and data structure manipulations, the PowerPlant X framework offers a range of wrapper classes. A wrapper class encapsulates a service or manager in Mac OS, making it simpler and more convenient to use.

The PowerPlant X wrapper classes are organized into groups.These groups are arranged in subfolders with the same name in the PowerPlant X folder on your computer.Table 7.7 describes these groups.

Table 7.7 Wrapper class groups

This group...	offers classes to simplify using this part of Mac OS
CoreFoundation	Mac OS Core Foundation services and data structures
Events	Carbon Events and Apple Events
HIToolbox	Mac OS HIToolbox object system for user interface items
OSServices	other, smaller services and manager in Mac OS

Index