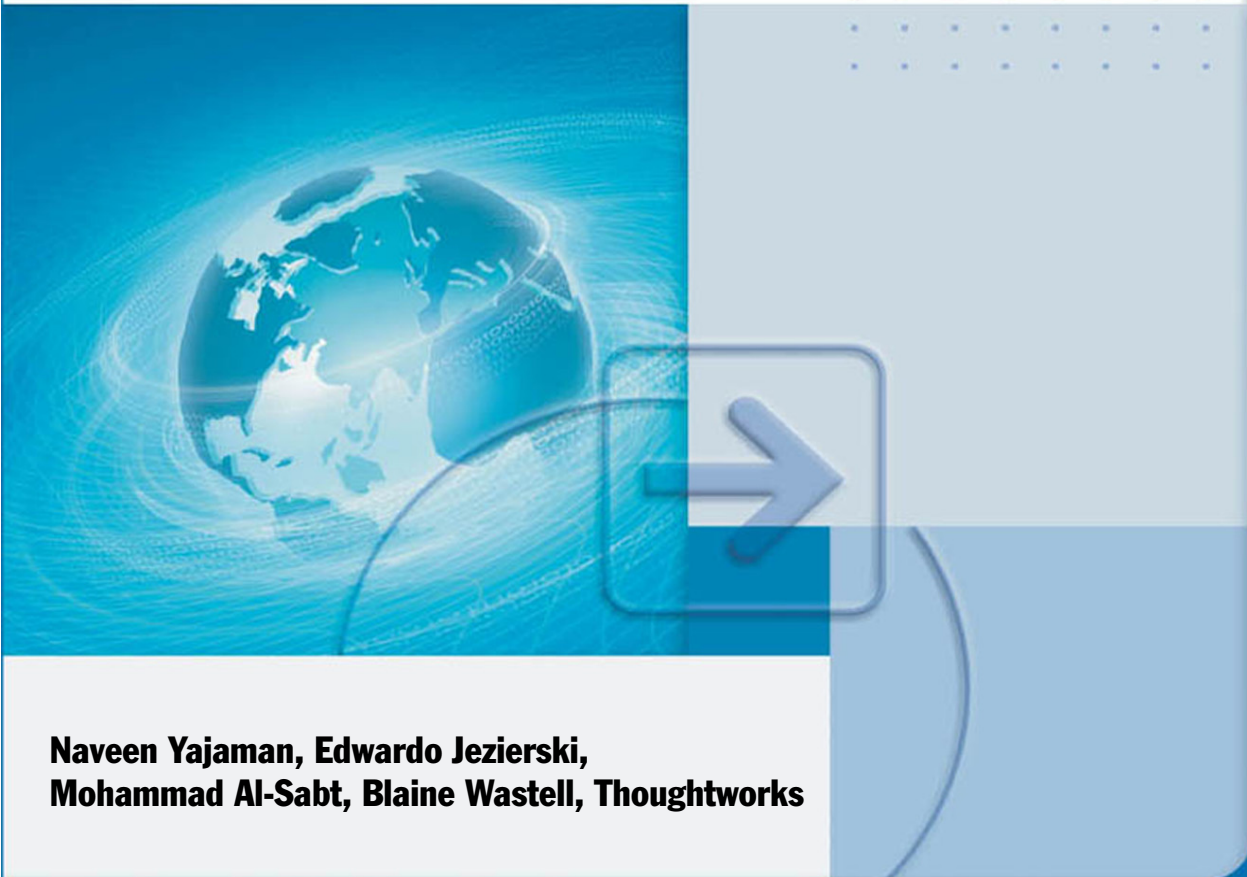


Microsoft®

User Interface Process Application Block

Version 2.0



patterns & practices

**Naveen Yajaman, Edwardo Jezierski,
Mohammad Al-Sabt, Blaine Wastell, Thoughtworks**

Information in this document, including URL and other Internet Web site references, is subject to change without notice. Unless otherwise noted, the example companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted herein are fictitious, and no association with any real company, organization, product, domain name, e-mail address, logo, person, place, or event is intended or should be inferred. Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

© 2004 Microsoft Corporation. All rights reserved.

Microsoft, MS-DOS, Windows, Windows NT, Windows Server, MSDN, Visual Basic, Visual C#, and Visual Studio are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

The names of actual companies and products mentioned herein may be the trademarks of their respective owners.

Contents

Chapter 1

Introduction to the UIP Application Block **1**

Who Should Read This Guide	2
Prerequisites	2
User Interface Process Concepts	3
Model-View-Controller	5
Goals of the UIP Application Block	6
Abstracting Navigation and Workflow Code	6
Using a Single Programming Model	6
Removing State Management From the User Interface	7
Persisting a Snapshot of State Across Processes	7
Features of the User Interface Application Block	8
Chapter Overview	9
Summary	10

Chapter 2

Design of the UIP Application Block **11**

UIP Application Block Architecture	11
Elements of the UIP Application Block	15
UIP Manager	15
UIP Configuration Classes	16
Navigator	16
View Manager	24
Views	28
Controller Base	32
State	34
State Persistence Providers	36
Additional Interfaces	41
Summary	43

Chapter 3**Developing Applications with the UIP Application Block 44**

Preparing to Build the UIP Application Block	45
Migrating Applications That Use UIP Application Block, Version 1	45
Developing an Application with the UIP Application Block	46
Separating the Application into Distinct User Interface Processes	46
Selecting a Navigator for Each User Interface Process	47
Determining the State Persistence Provider	48
Building and Referencing the UIP Application Block	49
Creating the Controller	50
Starting and Resuming Tasks	52
Linking Tasks	54
Creating Views	58
Creating the Configuration File	60
Adding Additional Functionality	72
QuickStarts for the UIP Application Block	77
AdvancedHostDemo QuickStart	78
InsuranceClientManagement QuickStart	79
InsurancePurchaseWizard QuickStart	79
NoNavGraph QuickStart	79
MultiNavGraph QuickStart	79
Store QuickStart	80
Summary	80

Chapter 4**UIP Application Block Deployment and Operations 81**

Deployment Requirements	81
Deploying the UIP Application Block	81
Deploying the UIP Application Block as a Private Assembly	82
Deploying the UIP Application Block as a Shared Assembly	82
Updating the UIP Application Block	84
Updating Private Assemblies	84
Updating Shared Assemblies	84
See Also	85
Security Considerations	85
SQL Server Security	85
Configuration File Security	85
Security Permissions	86
Security Threats and Countermeasures	86
Summary	88

Appendix A

Extending the UIP Application Block	89
Customizing State	89
Creating a Custom State Class	90
Creating a Strongly Typed Class	92
Customizing State Persistence	97
Creating a Custom State Persistence Provider	97
Customizing When State Is Persisted	99
Customizing Views and View Management	103
Creating a Custom IView Implementation	104
Create the Application Views	107
Creating a Custom IViewManager Implementation	108
Custom View Management Code	113
Persisting Task Information Between User Sessions	114

Appendix B

UIP Application Block Terminology	117
Additional Resources	124

1

Introduction to the UIP Application Block

Applications typically contain code to manage user interaction in the presentation layer; for example, one form can determine the next form that a user navigates to. Developers often write all of this code within the user interface (UI) code itself. However, if a developer includes navigation code in the user interface, the resulting code can be complex and very difficult to reuse, maintain, or extend. In addition, the application may be difficult to port to a different platform because the application control logic and state cannot be reused.

In many cases, applications contain state that must be maintained. If state is stored in a form, the code must access the form to retrieve state. This can be difficult to implement and tends to result in inelegant code, which again affects the extensibility and reusability of a user interface. For example, if, you develop several forms to be viewed sequentially, and then you need to insert a new form in the sequence, you must recode or modify both the previous and subsequent forms to incorporate the new form.

As a user interacts with an application, he or she may begin a task, put it on hold for a while, and then return to it. If the user closes the application between sessions, he or she may lose state and need to restart the entire task from the beginning. This is tedious for the user, and in some cases, it is unacceptable for your business logic.

Therefore, as you design your application, consider workflow, navigation, and interaction with business services and components as separate concerns from how data is acquired and presented to the user.

The User Interface Process (UIP) Application Block, version 2, provides an extensible framework to simplify the process of separating business logic code from the user interface. You can use the block to write complex user interface navigation and workflow processes that can be reused in multiple scenarios and extended as your application evolves. This guide describes the design and features of the UIP Application Block and demonstrates how you can use the block in your applications.

The UIP Application Block addresses a specific set of challenges that you will encounter in user interface development. These include:

- Navigation and workflow control — This should not be embedded in the user interface, but often is because the decision about which view to display next is based on business logic. This results in inelegant and unmanageable code.
- Navigation and workflow changes — Reformatting the layout of an application (changing the sequence of pages or adding new pages) is very difficult with traditional user interface techniques.
- State management — Passing state and maintaining consistency of state between views is difficult and is different for Windows-based applications and Web applications.
- Saving a snapshot of current interaction — You may want to capture a snapshot of an interaction and recreate it elsewhere, across time, machine, or logon boundaries.

Who Should Read This Guide

This guide is targeted at software developers and architects who are developing applications with complex user interface interactions. Specifically, the target customer for the block is a developer who designs and writes code to create Web applications or applications designed to run on the Microsoft® Windows® operating system (including smart client applications).

Prerequisites

To fully benefit from this guide, you should have an understanding of the Microsoft .NET Framework (including .NET Framework security concepts). The specific technologies mentioned in the guide are the Microsoft Visual C#® development tool, the Microsoft Visual Basic® .NET development system, the .NET Framework, and XML.

User Interface Process Concepts

To help manage the presentation layer in your application, you should split it up into manageable units. A number of terms related to this concept are heavily used in this guide. These include:

- *A user interface process* — A set of user-related activities that achieve a goal. By grouping together activities in this way, you can define the views that are used in each user interface process and determine how navigation should occur between those views.
- *A UIP task* — A running instance of a user interface process.
- *UIP state* — A snapshot of a task that can be persisted.

These terms, along with other important UIP-related terms, are discussed in more detail in Appendix B, “UIP Application Block Terminology.”

The UIP Application Block allows you to abstract the presentation layer code of your application into a separate layer. Applications that use the UIP Application Block have a presentation layer that is further divided into the following layers:

- *User interface components* — These components make up the user interface of the application. Users see and interact with these components. User interface components are responsible for acquiring data from the user and rendering data to the user.
- *User interface process components* — These components orchestrate the user interface elements and coordinate background activities, such as navigation and workflow control and state and view management. Users do not see user interface process components; however, these components perform a vital supportive role to user interface components.

The following diagram shows these two layers in the architecture of a .NET-based distributed application.

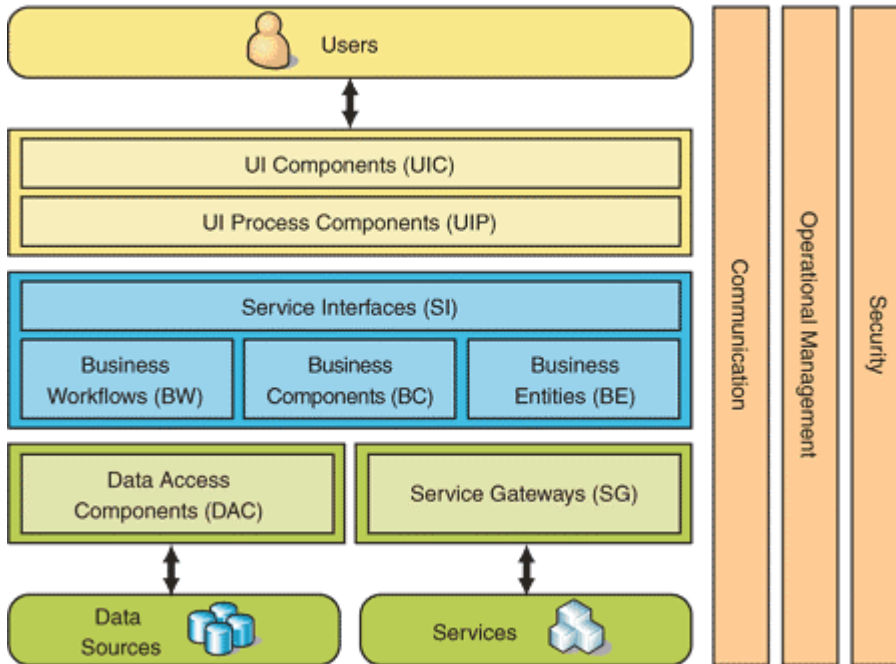


Figure 1.1

Component layers found in distributed applications and services built with .NET

The UIP Application Block is a framework for building user interface process components. These components are responsible for:

- Managing the flow of information through the user interface components
- Managing the transitions between stages of a user interface process
- Modifying user process flow in response to exceptions
- Separating the conceptual user interaction flow from the implementation or device where it occurs
- Maintaining internal business-related state, usually by holding on to one or more business entities that are affected by the user interaction

This means they also manage:

- Accumulating data taken from many UI components to perform a batch update at the server
- Keeping track of the progress of a task in a user interface process

By keeping your user interface process components separate from your user interface components, you can abstract the code that controls the underlying behavior of the application from the user interface itself. This allows you to write reusable code for the control flow and state management of different types of applications, independent of the user interface used. It also allows you to change user interfaces and manage multiple user interfaces without changing large amounts of code and gives you more flexibility in testing your applications because you can split test procedures into logical units independent of the user interface.

Model-View-Controller

The UIP Application Block is based on the model-view-controller (MVC) pattern. This is a fundamental design pattern for the separation of user interface logic from business logic. The pattern separates the modeling of the application domain, the presentation of the application, and the actions based on user input into three distinct objects:

- **Model** — This object knows all about the data to be displayed and is responsible for managing the data and the actions of the application. It can be thought of as the processing part of an input-process-output system.
- **View** — This object manages the information displayed to users. Multiple views can be used to display the same information in different ways. It can be thought of as the output part of an input-process-output system.
- **Controller** — This object allows the user to interact with the application. It takes input from the user and passes instructions to the model. In the input-process-output system, this is the input part.

Figure 1.2 depicts the structural relationship between the three objects.

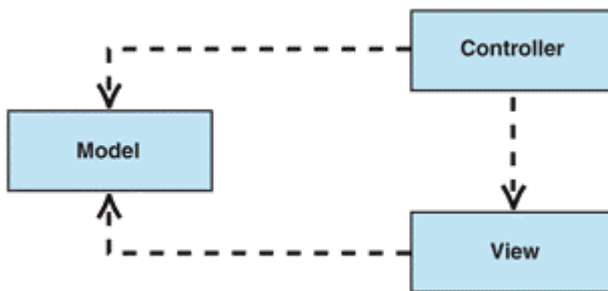


Figure 1.2

MVC class structure

Both the view and the controller depend on the model. However, the model depends on neither the view nor the controller. This is one of the key benefits of the separation. This separation allows the model to be built and tested independently from the visual presentation.

Exactly how you implement the model-view-controller pattern depends on your application type. For example, the separation between view and controller is secondary in many rich client and smart client applications, and many user interface frameworks implement the roles as one object. However, in Web applications, the separation between view (the browser) and controller (the server-side components handling the HTTP request) is very well defined.

Note: For more information about MVC, see “Model-View-Controller” at <http://msdn.microsoft.com/practices/type/patterns/enterprise/desmvc/>.

Goals of the UIP Application Block

The UIP Application Block is designed to help you:

- Abstract all navigation and workflow code from the user interface
- Enable the use of the same programming model for different types of application
- Remove all state management from the user interface
- Persist snapshot state across processes

It is useful to discuss each of these in more detail.

Abstracting Navigation and Workflow Code

The workflow of an application is a business process, not a user interface process, and the control of it should lie outside of the user interface layer. Code abstraction also helps you to maintain and extend your existing applications more easily.

The UIP Application Block abstracts workflow code from the user interface layer into the user interface process layer. As the navigation within the workflow is not hard coded into the user interface elements themselves, if the flow of your application changes, you need to recode the process only, not the elements within the process.

Using a Single Programming Model

You may create a Windows-based application for an internal audience, and later need to develop a Web-based front end for external users. You may develop applications for a Web browser that later need to be ported to a device application. If you use the same programming model for all application types, you can more easily port applications from one platform to another.

By using the UIP Application Block, you can ensure that all the navigation and workflow code is extracted from the user interface. This allows you to plug in new views (pages, forms, or whatever the application requires), and reuse the rest of the existing application.

Removing State Management From the User Interface

In many applications, state is stored in the user interface layer. This means that complex code is required to store, access, and manipulate that state. In addition, state is dependent on the user interface type, which can make it difficult to transfer code between application types.

The UIP Application Block abstracts UIP state from the user interface and stores it in a generic state object that can be accessed by using the block classes. This means that any view can work with the UIP state without knowing which other view is storing that information, or how to access it. If the view type is changed, the state management code can be left untouched.

Persisting a Snapshot of State Across Processes

In many applications, it is very difficult to store the state of a task at any given time. Most applications use transactions to encapsulate each task or sub-task, but only allow a complete transaction to either succeed or fail. This means that if a system shuts down or a user navigates away from a Web site, it is often difficult for the system or user to return to the point at which they left that process.

The UIP Application Block includes state persistence mechanisms that allow code to persist UIP state. This allows you to develop applications that persists UIP state at pre-determined intervals, meaning that you can restore the application at a later time. In addition to storing state in the application's memory, your code can save it in isolated storage or a database for later reuse.

Note: The UIP Application Block was designed based on reviews of successful .NET applications. It is provided as source code that you can use as-is or you can customize the code to more closely fit your specific requirements. It can be used to solve a number of challenges presented to developers, and enhances the reusability and maintainability of your code. It is not an indication of future direction for user interface processes within the Microsoft .NET Framework. Future releases of the .NET Framework may address user interface processes using a different model.

Features of the User Interface Process Application Block

The User Interface Process Application Block provides you with a number of features that will help you develop User Interface Process Components, and therefore distributed applications, more efficiently. The following features were in the first version of UIP and continue to be part of UIP version 2.

- Web session resume — Allows users to save information in a browser, so that next time they start a browser session, they can continue from where they left off.
- Web session transfer — Helps one user to suspend a session, and another user to pick it up on another computer. Also allows you to suspend a session on one device, and resume it on a different type of device.
- Reuse of code between application types — Helps you port the user interface process control logic from one type of application to another (for example, from a Windows application to a Web application).
- Development of discrete tasks — Helps you develop encapsulated tasks (for example, a Register User task and a Checkout task) that can be linked together to complete a job. You can pass information between tasks as required (for example, you can pass user details to the Checkout task).
- State persistence providers — Allows you to store and retrieve state from a SQL Server database, Web Session object, or from memory.

The following features are new to UIP version 2:

- Expanded navigation management — Gives you a variety of options for specifying the transitions between views in Windows-based applications and Web applications. Each navigator provides a different level of locking the user into the pre-defined flow of transitioning between views. In the previous version of the UIP Application Block, navigation management was provided solely by the navigation graph. In UIP version 2, the navigation graph is one of several navigation options.
- Additional state persistence providers — Gives you the ability to store and retrieve state from isolated storage and gives you the option to encrypt the data.
- Layout Managers — Gives you the ability to add custom layout logic to views in your applications
- User Navigation management — Uses conditional logic to test whether the user is allowed to navigate to a view by specifying a URL or by clicking on the back or forward buttons.
- Usability enhancements — Helps you with debugging, provides an extensible UIP schema, and provides a shared view for common transition points.

Several QuickStarts are included in the download of this block. You can use the QuickStarts to familiarize yourself with the functionality of the block. For more information about QuickStarts, see the section, “QuickStarts for the UIP Application Block,” in Chapter 3, “Developing Applications with the UIP Application Block.”

Chapter Overview

The remainder of this guide consists of the following chapters.

- **Chapter 2, Design of the UIP Application Block** — To make the most of the UIP Application Block, you need to understand its design. Chapter 2 discusses the architecture of the block.
- **Chapter 3, Developing Applications with the UIP Application Block** — After you understand the design of the UIP Application Block, you can start developing applications with it. Chapter 3 explains the process for building the block and modifying your application to use the block. The chapter also discusses the QuickStarts provided with the block.
- **Chapter 4, UIP Application Block Deployment and Operations** — After you create applications that use the UIP Application Block, you will need to deploy them. Chapter 4 discusses the special considerations for deploying these applications, including prerequisites for deployment and security threats and countermeasures.
- **Appendix A, Extending the UIP Application Block** — Using the UIP Application Block in a default configuration works well in most situations. However, there are times when you may want to customize certain parts of the block to be more specific to your needs. Appendix A provides guidance on extending the functionality of the block to develop your own state management class, custom state persistence providers, custom view implementations, and custom task implementations.
- **Appendix B, UIP Application Block Terminology** — This appendix will help you understand many of the concepts and terminology used in the UIP Application Block and in this guide.

If you are interested in evaluating the UIP Application Block, Chapters 1 and 2 and Appendix B are most useful. If you have decided to incorporate UIP in your application or if you are already using the previous version of the block and want to upgrade to this version, Chapters 2, 3, and 4 are valuable. If you already have a good understanding of developing with UIP and want to extend the functionality of the block, Appendix A will help you.

Summary

The User Interface Process (UIP) Application Block, version 2, is based on the model-view-controller (MVC) pattern, and provides an extensible framework you can use to simplify the process of separating business logic code from the user interface. The block will help you write complex user interface navigation and workflow processes that can be reused in multiple scenarios, and extended as your applications evolve. The remainder of this guide describes the design and features of the block and demonstrates how you can develop your applications to take advantage of the functionality of the block.

2

Design of the UIP Application Block

The UIP Application Block provides you with an infrastructure that allows you to write complex user interface navigation and workflow processes. Additionally, you can reuse and extend these processes as your application evolves. This chapter examines the design of the UIP Application Block to help you understand how to use the block in your environment.

UIP Application Block Architecture

As mentioned in Chapter 1, the UIP Application Block is based on the model-view-controller (MVC) pattern. Each of the MVC objects is represented in the UIP Application Block in the following ways:

- Model — Implemented in the **State** class. This stores both user information and control information within the user interface process.
- View — Implemented in the **WebFormView**, **WindowsFormView**, and **WindowsFormControlView** classes. These are used to create views in your application.
- Controller — Implemented in classes that derive from the **ControllerBase** abstract class. These classes are responsible for the starting of, navigation within, and ending of a user interface process.

The UIP Application Block contains several classes and interfaces that combine to provide an infrastructure you can use with your own applications. The following diagram illustrates the design of the UIP Application Block, showing the main components of the block, and how they interact with components provided by your organization.

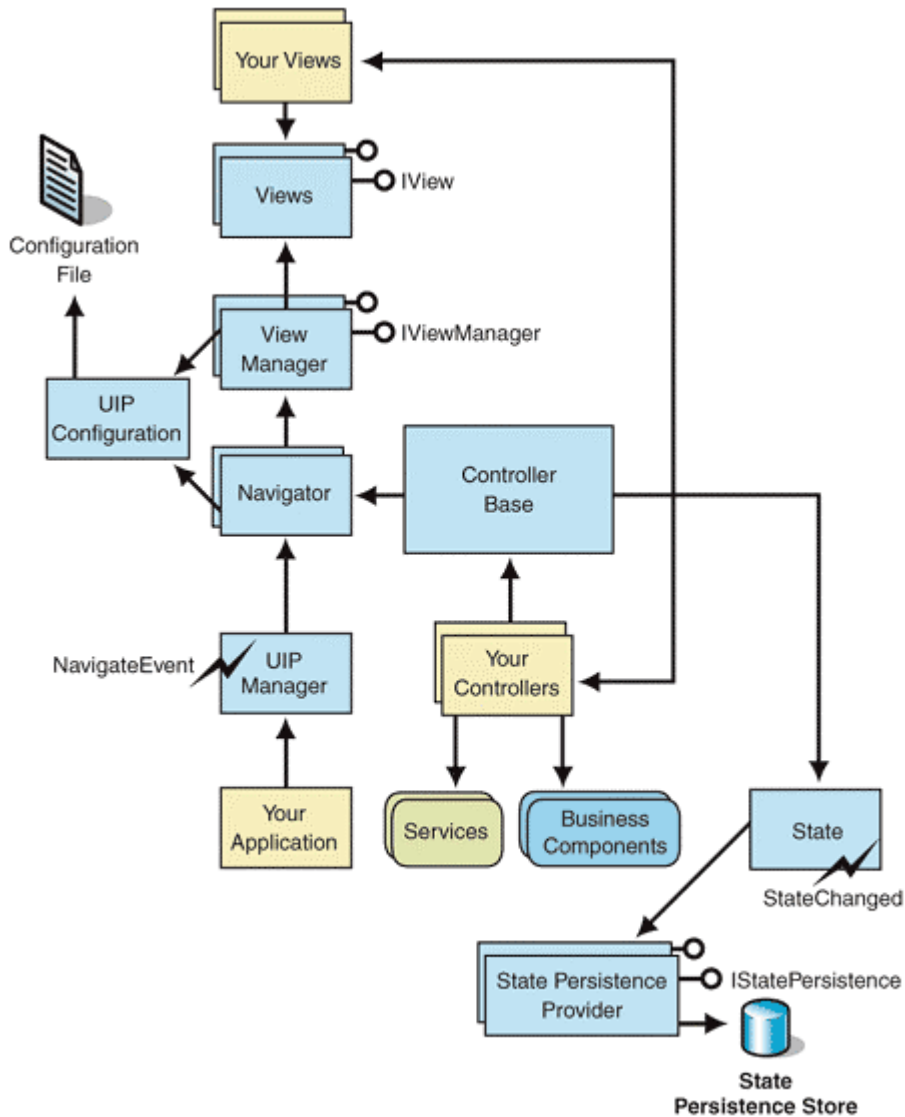


Figure 2.1
UIP Application Block design

Table 2.1 describes the elements shown in Figure 2.1.

Table 2.1: UIP Application Block Elements

Block element	Provided by	Description
Your Application	Your organization	Contains code for calling the appropriate start task method on the UIP manager.
Configuration File	Your organization	Contains application configuration information.
Your Controllers, Controller Base	Your organization (UIP Application Block provides the ControllerBase abstract class you can use to create the controller).	Controls the navigation between views and acts as a façade between the user interface layer and the business layer. Provides access to the business logic of your application.
Services, Business Components	Your organization	The services (such as Web or data services) and business logic of your application.
UIP Manager	UIP Application Block	Provides entry points to the UIP Application Block for starting or loading a task with a variety of navigators.
Navigator	UIP Application Block	Manages transitions between views and determines the appropriate view, asking the view manager to activate it.
UIP Configuration	UIP Application Block	Retrieves, verifies, and stores the information contained in the application configuration file.
Your Views, Views	Your organization (UIP Application Block provides view classes you can inherit from to create your application's views.)	User interface elements for user interaction.
View Manager	UIP Application Block provides some implementations.	Creates and activates views as requested by the navigator.
State	UIP Application Block provides an implementation.	Maintains user process state and maintains the current view in the task.
StatePersistence Provider	UIP Application Block provides some implementations.	Captures the state to be stored in the state persistence store.
State Persistence Store	Your organization	Preserves the state.

The UIP Application Block abstracts the creation of the view to a view manager, which ensures that you can create and configure views that appear consistent to UIP. This allows you to use the same state management and configuration capabilities for both Microsoft® Windows® operating system and Web environments. For more information about the elements in Table 2.1, see “Elements of the UIP Application Block,” later in this chapter.

Note: In many cases, you must write separate presentation layer code for Windows-based applications and Web applications. This is because the nature of user interaction can differ substantially between a Windows-based application and a Web application.

At a high level, the UIP Application Block displays views as follows:

1. Your application calls the appropriate **StartNavigationTask**, **StartOpenNavigationTask**, or **StartUserControlsTask** method on the UIP manager. After a task is started, creation of views, navigation, and state management are performed through UIP.
2. The UIP manager creates the appropriate navigator (for example, the graph navigator, user controls navigator, or wizard navigator) and passes the necessary information to the navigator.
3. The navigator calls the **UIPConfigSettings** class to retrieve configuration information parsed from the configuration file.
4. The navigator creates the appropriate view manager, based on the type of views being used (as specified in the configuration file).
5. The navigator creates or loads the task.
6. The navigator determines which view to activate.
7. The navigator issues a **NavigateEvent** event through the UIP manager. This provides the current and next view and can be used by applications external to the UIP Application Block.
8. The navigator calls the **Save** method on the **State** object; the **State** object then asks the appropriate state persistence provider to save the state.
9. The navigator calls the **ActivateView** method on the appropriate view manager and passes the view name to the view manager. The **ActivateView** method is responsible for creating the controller and displaying the view. Exactly how this occurs varies depending on whether the application is Windows Forms or Web-based.
10. After the view displays, the user is in control and UIP does nothing further until the user performs an action on the user interface (for example, clicking a button).

11. The user performs an action, and the event causes a method to be called on the view, which in turn calls a method on the controller. Depending on the user action, the controller does one of the following:
- Perform or delegate a function related to business logic (such as saving information to the database) that causes the view to refresh and change state. If the function causes a modification to the current state, the **State** class will raise the **StateChanged** event, to notify any interested listeners of changes that have occurred to the state.
 - Call the **Navigate** method on the navigator to transition to the next view or user control, and update the **State** object.
 - Call the **SuspendTask** method to clear the **State** object from memory but not clear it from the state persistence storage.
 - Call the **CompleteTask** method to clear the **State** object from both the memory and state persistence storage.
 - Call the **OnstartTask** method to chain or link tasks.

Elements of the UIP Application Block

To understand the design of the UIP Application Block, it is important to examine in more detail each of the elements that make up the block. The following paragraphs discuss these elements.

UIP Manager

To use the functionality of the UIP Application Block, your application must call the UIP manager to start a UIP task and start the appropriate navigator. The **UIPManager** class contains static methods you can use to start tasks with a variety of navigators, as shown in Table 2.2.

Table 2.2: UIPManager Public Methods That Start a Navigator and Task

Methods	Description
StartNavigationTask	Starts a graph navigator-based task. The name of the graph navigator in the configuration file must be provided.
StartOpenNavigationTask	Starts an open navigator-based task with no restrictions on transitions between views. An initial view name must be specified.
StartUserControlsTask	Starts a user controlled navigator-based task. The name of the user controls navigator in the configuration file must be provided.

These methods are overloaded so you can call the same method with different parameters. In some cases you will create a new task and in others you will load a pre-existing task. The UIP manager creates an instance of the appropriate navigator depending on the parameters that are sent to the **StartNavigationTask**, **StartOpenNavigationTask**, or **StartUserControlsTask** method. For more information about how to start a task from your application, see “Starting and Resuming Tasks” in Chapter 3, “Developing Applications with the UIP Application Block.”

The **UIPManager** class also allows you to listen for navigation changes through the **NavigationEvent** event. A **NavigationEvent** event is raised after the **Navigate** method is called on the **Controller** class, but before actual navigation takes place. This gives you one last opportunity to perform a function before the user is taken to the next view. For more information, see “Raising Navigation Events” in Chapter 3, “Developing Applications with the UIP Application Block.”

UIP Configuration Classes

The UIP Application Block uses the **UIPConfigHandler**, **UIPConfigSettings**, and **UIPConfiguration** classes to retrieve and maintain information from the configuration file. The **UIPConfigHandler** class implements the .NET Framework **IConfigSectionHandler** interface and is used to validate and parse the **<uipConfiguration>** section of an application configuration file. It retrieves and validates the configuration information against the XML schema and creates an instance of an XML node that represents the configuration information for the block. The **UIPConfigSettings** class parses the XML node retrieved from the **UIPConfigHandler** and loads the objects that an application using the block needs to function. There is one instance of the **UIPConfigSettings** object per application. The **UIPConfiguration** class exposes an instance of the **UIPConfigSettings** object to the application.

Navigator

The **Navigator** class is an abstract class that coordinates the navigation between views and provides functionality common to all of the navigators. The **Navigator** class contains five properties, as shown in Table 2.3.

Table 2.3: Navigator Class Properties

Property	Description
ViewManager	Provides access to the view manager associated with the navigator.
Name	Provides the name of the navigator.
CurrentState	Provides the current state of the navigator.
CacheExpirationMode	Specifies the expiration mode of the State cache for this task.
CacheExpirationInterval	Specifies the interval used to expire entries in the State cache.

Two important methods of the **Navigator** class are the **Navigate** method and the **OnStartTask** method. The controller tells the navigator which node or view to transition to by calling the **Navigate** method, and then passing the node or view name as a parameter. The **OnStartTask** method handles any **StartTask** events started by the **Controller** and is used to link tasks together. For more information about linking tasks, see “Linking Tasks” in Chapter 3, “Developing Applications with the UIP Application Block.”

The **Navigator** class behaves slightly differently, depending on the parameters that are passed to the appropriate **StartNavigationTask**, **StartOpenNavigationTask**, or **StartUserControlsTask** method:

- If no task ID or task object has been created, the navigator assumes it is creating a new task. The navigator creates a task ID, calls the **Create** method on the state factory to create the **State** object, calls the **Create** method on the state persistence factory to create the **StatePersistence** object, and sets the task ID on the task.
- If the task object has been created but there is no task ID, the navigator creates a task ID and passes that back to the task. The navigator then calls the **Create** method on the state factory to create the **State** object, calls the **Create** method on the state persistence factory to create the **StatePersistence** object and sets the task ID on the task.
- If there is already a valid task object and corresponding task ID, the navigator calls the **Load** method on the **State** factory class to retrieve the task from the appropriate state persistence provider, and passes the task ID to the **State** object.

For more information about starting tasks, see “Starting and Resuming Tasks” in Chapter 3, “Developing Applications with the UIP Application Block.”

Note: You should only be concerned with how to maintain task IDs if you want to persist and load the state at a later date. Your task IDs can be maintained in a table or a list, or if you are using SQL server state persistence, you can use a SQL database.

The **Navigator** class also behaves differently, depending on which view it activates:

- If your application is showing its first view, the view is specified in the configuration file, or, in the case of open navigation, as a parameter in the **StartOpenNavigationTask** method.
- If your application is restoring a view from a saved state, the navigator determines the current view from that saved state and sets it as the start view.
- If your application is showing any view other than the first view, the navigator retrieves the configuration information for the view from the **UIPConfigSettings** object, sets the current view property on the state object to be the name of the view, clears the **navigateValue** property, saves the state, and calls the **ActivateView** method on the appropriate view manager.

For Web applications, the navigator also uses conditional logic to test whether the user is allowed to navigate to a view by specifying a URL or by clicking on the back or forward buttons. Before the navigator creates the controller for the view in the **GetController** method, the code performs the following steps:

1. The navigator calls the **ActivateViewInStateIfNecessary** internal method.
2. This method calls the **IsRequestCurrentView** on the view manager to determine if the requested view is the same as the view on the current state.
3. If the views differ, the navigator verifies that the **AllowBackButton** attribute in the configuration file is enabled.
4. If **AllowBackButton** is not enabled, the navigator reloads the view in the current state, presenting the user with the same view as before.
5. If **AllowBackButton** is enabled, the navigator performs an additional check for graph navigators to ensure the user is not navigating outside of the navigation graph. If the view does not exist in the navigation graph, the navigator reloads the view in the current state, presenting the user with the same view as before.
6. If the view does exist in the navigation graph or if a navigation graph is not being used, the navigator loads the requested view.

Four navigators are supplied with the UIP Application Block, each derived from the **Navigator** class:

- **GraphNavigator** — Allows you to define the process of views and permitted transitions for a given task. This applies to Windows-based applications and Web applications.
- **OpenNavigator** — Allows you to freely navigate between views where there are no restrictions on transitions between views. This applies to Windows-based applications and Web applications.
- **UserControlsNavigator** — Allows you to define the views and permitted transitions between various controls on a Windows Form. This only applies to Windows-based applications.
- **WizardNavigator** — Allows you to define the views and allowed transitions in a wizard. This only applies to Windows-based applications.

Note: The **WizardNavigator** class derives from the **GraphNavigator** class.

You can only specify one navigator per task. If you want to transition between navigators, you need to create a new task to do so. Each navigator is designed slightly differently, so it is useful to look at each at each in turn.

Graph Navigator

The **GraphNavigator** class is derived from the **Navigator** class and coordinates the navigation between views by using a navigation graph. Navigation graphs encapsulate workflow functionality through a list of views and allowed transitions. They allow you to determine which views exist in a user interface process and the transitions that are valid from each node. Navigation graphs contain a start node, an end node, and zero or more nodes in between. The path between the start and end can be a simple linear path, or include branching and circular routes. This can be seen in the sample navigation graph shown in Figure 2.2.

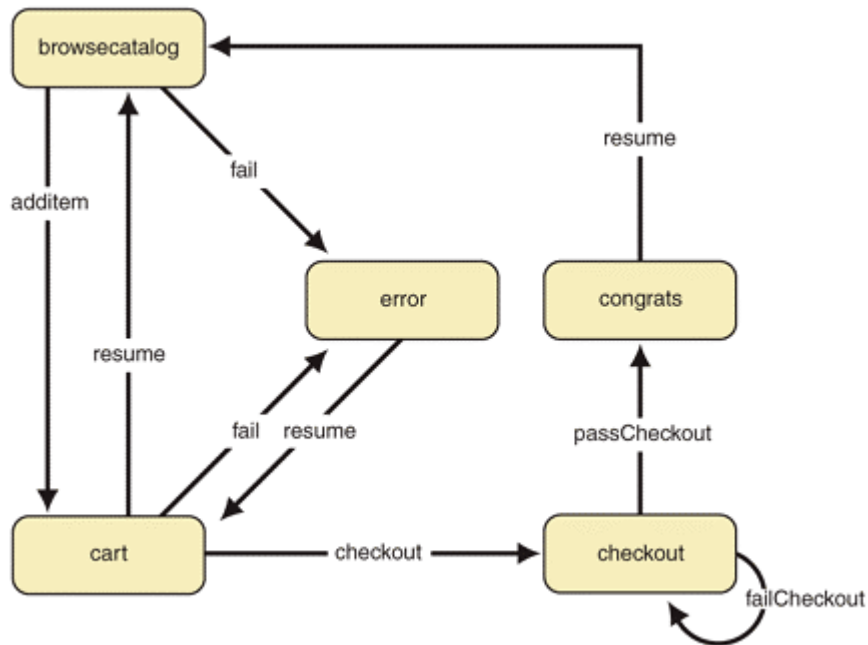


Figure 2.2

Sample navigation graph

You can create and begin a graph navigator by calling the **StartNavigationTask** method on the UIP manager and passing the name of the navigation graph in the configuration file as a parameter, along with task information if the navigator reloads a task. For more information, see “Starting and Resuming Tasks with the Graph Navigator” in Chapter 3, “Developing Applications with the UIP Application Block.” When the **StartNavigationTask** method is called, the graph navigator ensures that the task, state object, and state persistence provider are created. The controller tells the graph navigator which node it should transition to by calling the **Navigate** method and passing the node name as a parameter. The navigator checks the navigation graph in the **UIPConfigSettings** class to ensure that the transition is valid, and if it is not, generates an exception. A valid node is either listed as a child of the current node or specified in a shared transition.

When using the UIP Application Block, you define navigation graphs in the **<navigationGraph>** section of your XML configuration file. Each graph contains a node element for each view, and the node element contains **<navigateTo>** child elements, which define the different paths that can be taken from that view. The following code shows the XML required to follow the paths shown in Figure 2.2, earlier in this section.

```
<navigationGraph
  iViewManager="WinFormViewManager"
  name="Shopping"
  state="State"
  statePersist="SqlServerPersistState"
  startView="cart">
  <node view="cart">
    <navigateTo navigateValue="resume" view="browsecatalog" />
    <navigateTo navigateValue="checkout" view="checkout" />
    <navigateTo navigateValue="fail" view="error" />
  </node>
  <node view="browsecatalog">
    <navigateTo navigateValue="addItem" view="cart"/>
    <navigateTo navigateValue="fail" view="error" />
  </node>
  <node view="error">
    <navigateTo navigateValue="resume" view="cart" />
  </node>
  <node view="checkout">
    <navigateTo navigateValue="passCheckout" view="congrats" />
    <navigateTo navigateValue="failCheckout" view="checkout" />
  </node>
  <node view="congrats">
    <navigateTo navigateValue="resume" view="browsecatalog" />
  </node>
</navigationGraph>
```

Shared Transitions

An application may have views that are a common transition point for many (or all) nodes within the navigation graph. These views are named *shared transitions*. This functionality is useful if, for example, you want errors that are generated anywhere in the navigation graph to transition to a common view or if you want to display a common help page. This has the same effect as repeating the **navigateTo** element in the shared transitions for each node in the navigation graph. Any node-specific transitions override the global shared transitions.

Open Navigator

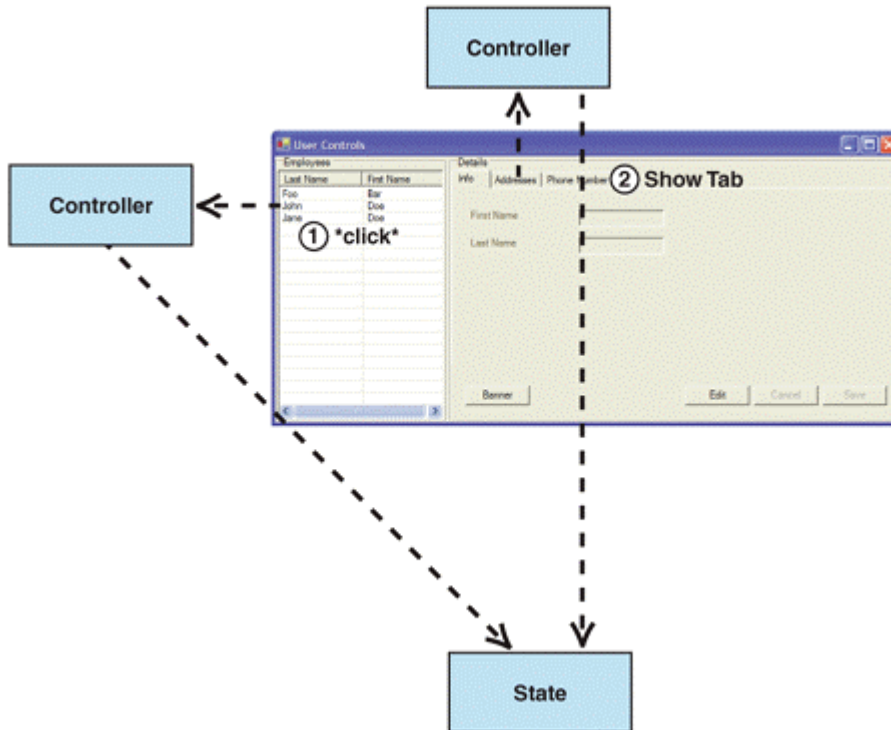
In some cases, you will want to use tasks and persist state without predefining a navigation graph. By using the **OpenNavigator** class, you give controllers the ability to navigate to any view by name. The open navigator validates that the requested view name exists and asks the view manager to activate it.

The **OpenNavigator** class derives from the **Navigator** class to coordinate the navigation between views and provide functionality for open navigators. You can create and begin an open navigator by calling the **StartOpenNavigationTask** method on the UIP manager, and passing the name of the open navigator in the configuration file as a parameter, as well as the first view and any task information if the navigator is reloading a task. For more information, see “Starting and Resuming Tasks with the Open Navigator” in Chapter 3, “Developing Applications with the UIP Application Block.” Controllers can then navigate to other views by using the **Navigate** method and passing the view name. The navigator is responsible for validating that the view exists. As with the graph navigator, the open navigator loads information from the UIP configuration file. It uses the default state type, default view manager, and provided start view information to activate the first view.

User Controls Navigator

Windows Forms applications can have user controls that appear as child controls on a form or other user control. The UIP Application Block uses the **UserControlsNavigator** class to provide a navigation mechanism that shifts focus from one control to another within a form, instead of transitioning between forms. This class allows you to populate a form with user controls that have their own model and controller, and it allows you to switch between the controls by name.

To give you a better idea of how a user controls navigator operates, suppose an application has a list view on the left and a tab control with multiple tabs on the right. Each form of the tab control contains one or more controls. As the user clicks an item in the left view, it may trigger one of the tabs to become the active page. Figure 2.3 shows a form that contains user controls, and demonstrates the relationship between the views, controllers, and state. The list view on the left and the tabs on the right each have their own controller, but they communicate with the same **State** object.

**Figure 2.3***Windows Form with user controls*

User controls de-emphasize the notion of process across time and emphasize the idea of many views collaborating to expose or retrieve information from the user without any specific sequencing. The views or controllers may raise events that are listened to by other controls. All of the views have access to the shared state of the navigator.

The **UserControlsNavigator** class derives from the **Navigator** class to coordinate the navigation between views and provide functionality for user controls navigators. You can create and begin a user controls navigator by calling the **StartUserControlsTask** method on the UIP manager and passing the name of the user controls navigator in the configuration file and any task information if the navigator is reloading a task. For more information, see “Starting and Resuming Tasks with the User Controls Navigator” in Chapter 3, “Developing Applications with the UIP Application Block.” The controller then tells the user controls navigator which view it should transition to by calling the **Navigate** method and passing the name of the control or view as a parameter. All collaborating views are declared in the **<userControls>** section of the configuration file. This section contains a list of the forms that participate in the task. Each form can have a list of child views.

Wizard Navigator

Wizards are commonly used in Windows-based applications, and navigation occurs between the forms in a predictable way. The UIP Application Block uses the wizard navigator to provide a navigation mechanism between the forms of a wizard. You can use the wizard navigator to transition between views defined by the **Cancel**, **Back**, **Next**, and **Finish** transitions, or you can use a navigation graph to create more elaborate transitions.

Figure 2.4 shows an example wizard and demonstrates the relationship between the views, the controllers, and state. Each view has its own controller and each controller communicates with the same **State** object.

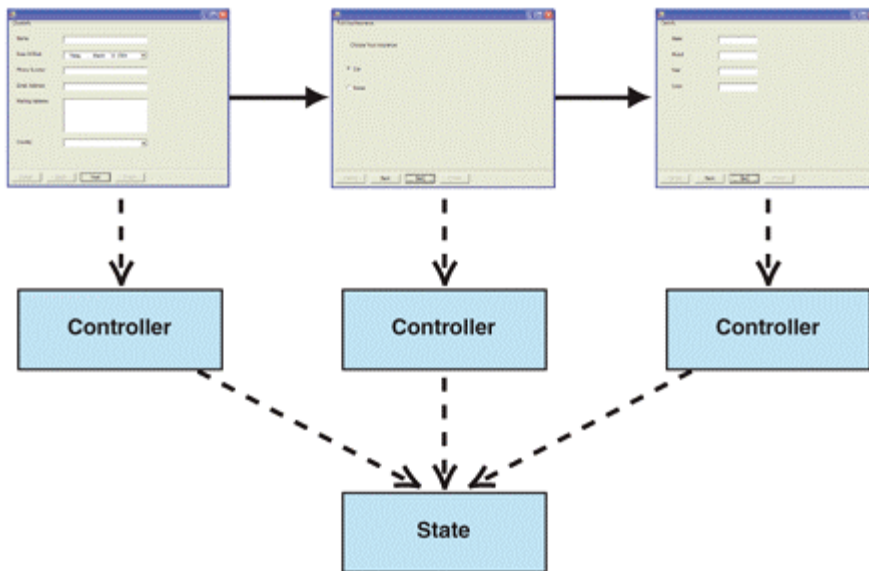


Figure 2.4

Wizard example

The **WizardNavigator** class derives from the **GraphNavigator** class to coordinate the navigation between views and provide functionality for wizard navigators.

You can create and begin a wizard navigator by calling the **StartNavigationTask** method on the UIP manager and passing the name of the wizard in the configuration file. The wizard navigator loads information from either the **<uipWizard>** element or the **<navigationGraph>** element in the UIP configuration file, passed as a parameter to the **StartNavigationTask** method. To define a wizard with no branching capabilities, you define the **<uipWizard>** section of the configuration file. To define a wizard with branching capabilities, you define the wizard navigator in the **<navigationGraph>** section and set the **runInWizardMode** attribute to **true**. For more information, see “Creating a **<uipWizard>** Section” and “Creating the **<navigationGraph>** Section” in Chapter 3, “Developing Applications with the UIP Application Block.” To support custom functionality when the **Cancel** or **Finish** buttons are clicked, implement the **IWizardViewTransition** interface. For more information, see the section, “IWizardViewTransition,” in this chapter.

View Manager

View managers are responsible for creating and activating views as requested by the navigator. In some cases, you will want to create applications that have more than one user interface. In particular, you may want to create Windows-based and Web versions of the same application. The navigator coordinates the interactions of views and controllers by creating view managers and using them to execute code specific to the application type. The management of these views generally differs depending on their type; for example, the code to display a Windows Form is distinctly different from that used to display a Web page. Because of this, different managers are needed for each type of application supported.

Each view manager implements the **IViewManager** interface, and you can create your own if you want, but three view managers are supplied with the block:

- **WebFormViewManager** — Used for managing Web views.
- **WindowsFormViewManager** — Used for managing Windows Forms views and user controls views.
- **WizardViewManager** — Used for managing wizards.

The **IViewManager** interface defines the six methods listed in Table 2.4.

Table 2.4: IViewManager Interface Methods

Methods	Description
ActivateView	Activates a specified view.
GetCurrentTasks	Returns all tasks currently being executed.
StoreProperty	Stores a property of the view by task ID.
IsRequestCurrentView	Checks that the current view matches the requested view.
GetViewNameForCurrentRequest	Checks the URL requested and gets the view name that corresponds with the URL.
GetActiveViewCount	Gets the number of active views in the manager.

Each of the view managers is discussed in more detail in the following sections.

WebFormViewManager

When a Web application is navigating between views, it must save state and display pages. The code to do this is specific to Web Forms and cannot be shared with other application types. The **WebFormViewManager** is used to provide the methods needed to manipulate views based on the **WebFormView** and to save state in the process. The **WebFormViewManager** implements four of the six methods defined in the **IViewManager** interface, as shown in Table 2.5.

Table 2.5: WebFormViewManager Methods

Methods	Description
ActivateView	Activates the view identified in the view parameter.
GetCurrentTasks	Returns an array of task GUIDs currently being executed.
IsRequestCurrentView	Checks to see that the current view as stored in the state is the view being requested
GetViewNameForCurrentRequest	Checks the URL requested and gets the view name that corresponds with the URL

The **ActivateView** method performs a number of steps necessary to activate your Web view:

1. The view manager loads the view settings (which tell the view manager which URL to navigate to), along with any additional information contained in the extensible configuration schema.
2. The view manager creates a **sessionMoniker** object that holds a reference to the navigator. The object is stored in the ASP.NET session state with a key of the task ID.

3. The view manager calls the **Response.Redirect** method from ASP.NET, which creates an instance of the Web Form.
4. ASP.NET starts a load event, which is captured by a load event handler in UIP. The handler uses the task ID (specified as a parameter on the **Response.Redirect** method) to identify the session moniker object. It then asks the navigator specified in this object for the view manager and tells the view manager to continue setting up the view.
5. The view manager creates an instance of the controller, sets it on the view, and calls the **Initialize** method on the view. When the method returns, ASP.NET displays the view. At this point, the user is in control rather than the application block or ASP.NET.

WindowsFormViewManager

When a Windows-based application is navigating between views, it must save state, display forms, and close forms. The code to do this is specific to Windows Forms and cannot be shared with other application types. The **WindowsFormViewManager** is used to provide the methods and events needed to manipulate views and manage the display of Windows Forms, and to save state in the process.

The **WindowsFormViewManager** implements four of the six methods defined in the **IViewManager** interface, as shown in Table 2.6.

Table 2.6: WindowsFormViewManager Methods

Methods	Description
ActivateView	Activates the view identified in the view parameter.
GetCurrentTasks	Returns an array of task GUIDs currently being executed.
StoreProperty	Stores a property of the form in the internal hash table.
GetActiveViewCount	Gets the number of active views in the manager.

The **ActivateView** method performs a number of steps to activate a Windows view:

1. The view manager loads the view settings and any additional information contained in the extensible configuration schema. This gives the view manager the class name of the view and the controller to use.
2. The view manager creates an instance of the controller and passes an instance of the navigator (and by extension, the state) to the controller.
3. The view manager creates an instance of the appropriate view and sets the controller on the view.

4. The view manager calls the **Initialize** method on the view. Any additional information from the view settings is passed to the **Initialize** method.
 - If the view is a multiple document interface (MDI) child and the **IsMDIChild** flag is set to **true**, the view manager adds the view to the MDI container.
 - If the view is a floating window, the view is added to the floating window parent. The parent and the floating window are specified in the configuration file.
 - If the view has child views, the view manager finds the child views and initializes them.
 - If the view has a layout manager, the view manager calls the **LayoutControls** method to position the controls on the view.
5. The view manager calls the **Show** method on the **WindowsFormView** or **WindowsFormControlView** class to display the view. If the view is specified as modal, the view manager calls the **ShowModal** method on the **WindowsFormView** class to display the modal view.

The view manager is responsible for making decisions about what to do with previous views. By default, the view manager closes the previous view. If the view is marked as **stayOpen** and the user navigates to another view, the previous view will remain open. The user will then need to manually close the view marked as **stayOpen**.

If a view is specified as modal (where the parent view cannot be modified until the modal window is closed) or floating (where the parent view can be modified), the previous view becomes the parent view. It remains open but behind the child view. For more information, see “Creating the <views> Section” in Chapter 3, “Developing Applications with the UIP Application Block.”

WizardViewManager

The **WizardViewManager** is used to provide the methods and events needed to manipulate views and manage the display of forms in a wizard, and to save state in the process. The view manager hosts and displays controls according to the wizard stage. It displays the first view as a user control, attaching the buttons to the delegates in the **WizardControllerBase** class. The **WizardViewManager** implements two of the four methods defined in the **IViewManager** interface, as shown in Table 2.7.

Table 2.7: WizardViewManager Methods

Methods	Description
ActivateView	Activates the view identified in the viewName parameter.
GetCurrentTasks	Returns an array of task GUIDs currently being executed.

The following steps activate your wizard:

1. The view manager retrieves the current task from the **WizardNavigator**.
2. The view manager then gets a reference to the **WizardContainer** for the task. The **WizardContainer** is responsible for controlling the flow between the different views that make up the wizard sequence.
3. The view manager calls the **ActivateView** method on the **WizardContainer** object and passes the name of the view that is to be activated as a parameter to the **WizardContainer** object.
4. The **WizardContainer** object ensures that the current view is pushed onto the history stack for the wizard (this allows for the transition to the previous views in a wizard). After the current view has been pushed onto the stack, the container retrieves the view that is being requested from the views that are contained within its private views collection. After the view is found, it is added into the wizard panel so that it can be prepared for the user to view it.
5. The **WizardContainer** object checks if the **IWizardViewTransition** interface is implemented for the view loaded in the panel. If so, it will do the following:
 - It will call the methods to customize the behavior of various transitions, such as **DoCancel** and **DoFinish**.
 - It will call the **WizardActivated** method to notify the view that it is being activated by the **WizardContainer**. This method allows you to perform any processing that should occur whenever the view is activated in a wizard.

Views

Views are central components of the UIP Application Block and represent the view portion of the MVC pattern. When you use the UIP Application Block, your views must include block-specific functionality so you can control them within a process. However, this functionality should not be defined in an interface specific to a view type, because the block is designed to support multiple view types.

Views are initialized by the view manager calling the virtual **Initialize** method on the view. Any additional information from the **<view settings>** section of the configuration file is passed to the **Initialize** method; therefore, you must ensure that the view knows what to do with the information. The UIP Application Block gives you the ability to have view-specific attributes, such as modality and background color, stored with the view configuration information. Attributes are supported for each view, and interpreted by the view manager. The UIP Application Block will pass the **<view settings>** element, including any additional attributes or child elements, to the view during execution.

Note: After the view is initialized, the view is aware of the state. This allows you to add code that makes use of the existence of state, such as pulling a dataset out of the state and binding it to a control on a form.

As with view managers, the UIP Application Block provides an interface that you can implement to create views. The **IView** interface is a generic interface that contains required properties only and can be implemented in classes that represent the specific view types in an application. The **IView** interface defines the properties that any view must support to interact with the management classes of the UIP Application Block.

Three implementations of the **IView** interface are provided with the block:

- **WebFormView** — Provides common functionality to implement views derived from the ASP.NET **Page** class in the **System.Web.UI** namespace for Web applications.
- **WindowsFormView** — Provides common functionality to implement views derived from the **Form** class in the **System.Windows.Forms** namespace.
- **WindowsFormControlView** — Provides common functionality to implement views derived from the **UserControl** class in the **System.Windows.Forms** namespace for handling nested views in Windows-based applications.

The **IView** interface defines the properties and methods shown in Table 2.8 and Table 2.9.

Table 2.8: IView Interface Properties

Property	Description
Controller	Returns the controller being used by the view.
ViewName	Returns the name of the view.
NavigationGraph	Returns the graph navigator in which the view is defined.
Navigator	Returns the navigator in which the view is defined.
TaskId	Returns the task ID related to the view.

Table 2.9: IView Interface Methods

Method	Description
Enable	Used to determine when a WinFormControlView object gains focus or loses focus.
Initialize	Used to pass optional arguments to the view in addition to the view settings; called after the view manager instantiates the view but before calling the Show method.

In most situations, it is best for Windows-based applications and Web applications to use the **WebFormView**, **WindowsFormView**, and **WindowsFormControlView** classes as base classes from which the actual views (Windows Forms, user controls, and ASP.NET pages) are derived. This approach reduces the code in your views, because much of the standard view manipulation code is included within the **WebFormView**, **WindowsFormView**, and **WindowsFormControlView** classes.

The **Enable** method is used to determine when a **WinFormControlView** object gains or loses focus. Suppose View A's **stayOpen** attribute is set to **true**. After the user transitions to another view, View A will remain open until it is closed by the user. You can use the **Enable** method to determine which view is in focus.

The **Initialize** method is used when implementing views with custom elements or attributes, such as background color. This method passes a **ViewSettings** object that can be used by the view to retrieve additional information. It is called after the view manager instantiates the view, but before calling the **Show** method. You implement the **Initialize** method when deriving from a **WebFormView**, **WindowsFormView**, or **WindowsFormControlView** class.

The **ViewSettings** class captures the configuration information for a specific view that implements the **IView** interface. The **ViewSettings** contains a number of properties including the view's controller, layout manager, and modal, floating, or stay open attributes. It also captures any custom attributes or elements defined for the view.

If you extend the view by using only additional attributes, you can access these attributes by using the **CustomAttributes** property. If you define additional elements that provide view configuration information, you can use the **Navigator** property. This returns an **XPathNavigator** object, which you can use to query for the appropriate information. For an example, see "Adding Extensible Configuration Schema," in Chapter 3, "Developing Applications with the UIP Application Block."

Note: If you want to create your own custom view types, you will need to implement the **IView** interface directly. For more details on how to do this, see Appendix A, "Extending the UIP Application Block."

It is useful to examine in turn each of the three implementations of the **IView** interface that are provided with the block. The next sections describe these implementations.

WebFormView

You can use the **WebFormView** class as a base class for views in a Web application. It implements the **IView** interface defined in the UIP Application Block and derives from the **Page** class in the .NET Framework. The constructor is used to link the **Load** event of the Web page to a method defined within the class. You can use the **WebFormView** class to create your Web page views. However, if you are already inheriting a base class in your view class, you cannot also derive from **WebFormView**. In this situation, you must directly implement the **IView** interface in your class.

The **WebFormView** class implements the four read-only properties defined in the **IView** interface, as shown in Table 2.10.

Table 2.10: WebFormView Class Properties

Property	Description
Controller	Returns the controller being used by the page.
ViewName	Returns the name of the page.
Navigator	Returns the name of the navigator in which the page is defined.
TaskId	Returns the task ID related to the page.

Unlike Windows Forms, Web pages cannot be instantiated by UIP directly; therefore, the controller cannot be set on the view during construction. Instead, the load event handler responds to the load event started by ASP.NET, and retrieves the navigator stored in the session object. You then use the navigator to find the controller for this view, and finish initializing the view.

WindowsFormView

You can use the **WindowsFormView** class as a base class for views in a Windows Forms application. It implements the **IView** interface defined in the UIP Application Block and derives from the **Form** class in the .NET Framework. The constructor is used to link the **Load** event of the Windows Form to a method within the class. Using the **WindowsFormView** class allows you to create Windows Forms views, but if you are already inheriting a base class in your view class, you cannot also derive from **WindowsFormView**. In this situation, you must directly implement the **IView** interface in your class.

The **WindowsFormView** class implements the four read-only properties defined in the **IView** interface, as shown in Table 2.11.

Table 2.11: WindowsFormView Class Properties

Property	Description
Controller	Returns the controller being used by the form.
ViewName	Returns the name of the form.
Navigator	Returns the name of the navigator in which the form is defined.
TaskId	Returns the task ID related to the form.

WindowsFormControlView

You can use the **WindowsFormControlView** class as a base class for views that frequently reuse common controls in a Windows Forms application. It implements the **IView** interface defined in the UIP Application Block and derives from the **UserControl** class in the .NET Framework. The **UserControl** class provides the functionality of a container control and also provides the **Load** event. The constructor is used to link the **Load** event of the **UserControl** class to a method within the class. Using the **WindowsFormControlView** allows you to create user control or wizard views; however, if you are already inheriting a base class in your view class, you cannot also derive from **WindowsFormControlView**. In this situation, you must directly implement the **IView** interface in your class.

The **WindowsFormControlView** class implements the four read-only properties defined in the **IView** interface, as shown in Table 2.12.

Table 2.12: WindowsFormControlView Class Properties

Property	Description
Controller	Returns the controller being used by the form.
ViewName	Returns the name of the form.
Navigator	Returns the name of the navigator in which the form is defined.
TaskId	Returns the task ID related to the form.

Controller Base

A central part of the UIP Application Block is the controller base. It is equivalent to the controller portion of the MVC pattern, and its primary purpose is to control the navigation between views and act as a façade between the user interface layer and the business layer. It has access to the state for a particular task and has the ability to move through the views defined by the appropriate navigator. It also provides access to the applications underlying business components.

ControllerBase is an abstract class that you must inherit from so you can coordinate tasks and create your own controller base. However, the UIP Application Block provides only the base functionality for the controller base. You must add methods specific to your implementation that provide access to the business layer of your application and that respond to user actions by calling the appropriate methods on the navigator. Code in the controller must understand the difference between the open navigator and the graph navigator so that it can determine if it is navigating based on view names or node names. The **NavigateValue** passed to the **Navigate** method defines the next view or node to which the navigator will transition.

The **ControllerBase** class contains two read-only properties, as shown in Table 2.13.

Table 2.13: ControllerBase Class Properties

Property	Description
State	Provides access to the state associated with this controller.
Navigator	Provides the navigator for the current controller.

The **ControllerBase** class contains four methods, as shown in Table 2.14.

Table 2.14: ControllerBase Class Methods

Methods	Description
EnterTask	Enables controllers to access state passed between tasks.
Navigate	Calls the Navigate method on the appropriate navigator, and passes the NavigateValue as a parameter.
SuspendTask	Suspends the current task; clears the state of a suspended task from memory.
OnStartTask	Navigates to another task.
OnCompleteTask	Ends the current task; clears the state of a completed task from memory and from the state persistence provider.

Note: A task ended with the **OnCompleteTask** method cannot be restarted because this method removes the state from the state persistence provider. If you want to reload a task, use the **SuspendTask** method.

State

State represents the model portion of the MVC pattern. You use state to store the status of a task throughout a user process. State also contains methods that allow you to retrieve specific items within the state information, store the state in a persistent location, and notify views that the state has changed. The controller and view manager must have some way of knowing where the user is within the process and where the user should go next, as well as having somewhere to store ephemeral application information, for example, shopping cart contents. Also, if the user suspends a task and then restarts it, the application must have some way of retrieving their restart location and any other information it requires.

Dealing with the state related to a task presents a number of problems:

- You must be able to access state for a task throughout the duration of that task. This state is unique to a particular task, and as such, should only be accessible to that task and the classes working with it.
- You must be able to persist state to a permanent data store. The code that does this will vary according to the type of permanent store being used; however, the state class must be able to communicate with any type of permanent store.
- You must be able to serialize state before persisting it, as some storage mediums require it.
- You must be able to notify views that the state has changed, because the views in an application present the state to the user.

Each of these problems is solved in the design of the **State** class:

- Instances of the **State** class cannot be shared across multiple sessions in a Web application. This ensures that only the task to which the state belongs can access it.
- The **State** class accepts an instance of a state persistence class. This means that the state can then call methods in the particular state persistence class to store itself in a durable medium.
- The **State** class implements the **ISerializable** interface, which facilitates serialization.
- The **State** class has an event called **StateChanged**, to which views can subscribe to be notified of changes to the state.

Note: It is difficult to ensure that the **StateChanged** event is always executed. For example, if you are storing information in a struct within the **State** object's internal hash table, your **State** class will not be aware of the contents of that struct changing, so it will not raise the event. To avoid this issue, you can derive your own strongly typed state class and raise the event whenever any state item changes. For more information, see Appendix A, "Extending the UIP Application Block."

The class contains a number of properties that allow you direct access to the state stored within it, as shown in Table 2.15.

Table 2.15: State Class Properties

Property	Description
CurrentView	Returns the navigator's current view.
NavigateValue	Returns the next view in the navigator.
Navigator	Returns the navigator.
TaskId	Returns the current task ID.
SyncRoot	Used to synchronize access to the inner hash table.

The **State** class contains the methods shown in Table 2.16.

Table 2.16: State Class Methods

Methods	Description
Add	Adds information to an existing instance.
Remove	Removes information from an existing instance.
Contains	Checks whether an item exists.
Accept	Passes an instance of the state persistence provider to the existing State instance.
Save	Calls the Save method of the relevant state persistence provider class.
CopyTo	Copies the contents of the State object to an array.

Your views may need to be notified when state changes; therefore, the **State** class includes a **StateChanged** event to which views can subscribe. This is a feature that notifies registered listeners when changes to the **State** object occur. This is useful, for example, in a view that populates its controls based on information stored in the **State** object. The view can listen to the **StateChanged** event and update the contents of its controls based on changes to values stored in the **State** object. For an example of an application that uses the **StateChanged** event, please see the AdvancedHostDemo QuickStart. For more information about using the **StateChanged** event, see the section, "Responding to Changes in State," in Chapter 3, "Developing Applications with the UIP Application Block."

State Persistence Providers

It is not only important to store state information; you must also make sure that the information is preserved where necessary. This is important for two reasons:

- The user may suspend and restart the process, which means that the in-memory version of the data will be lost.
- Items in memory can become invalid due to expiration schedules that you define.

To store state, it would be possible to add methods to the **State** class to persist the information using a particular storage mechanism, but this would limit the flexibility of the class, and therefore the application block. Alternatively, multiple **State** classes could be developed for various storage mechanisms, but this would result in code duplication and maintenance issues. The solution to the problem is to use specific classes to persist the state in a durable mechanism and to link the persistence class to the **State** class.

The UIP Application Block provides the **IStatePersistence** interface to help you create your own state persistence implementation. It defines how **State** and **Task** objects can be dehydrated and rehydrated to and from any type of storage. A state persistence provider is associated with each user interface process in the XML configuration file.

The UIP Application Block contains six implementations of the **IStatePersistence** interface that you can use in your applications:

- **IsolatedStoragePersistence** — For tasks that need to be persisted across multiple interactions or users and stored in isolated storage.
- **SecureIsolatedStoragePersistence** — For tasks that need to be persisted across multiple interactions or users and stored in isolated storage, and that contain sensitive information.
- **SqlServerPersistState** — For tasks that need to be persisted across multiple interactions or users in a database.
- **SecureSqlServerPersistState** — For tasks that need to be persisted across multiple interactions or users in a database, and that contain sensitive information.
- **SessionStatePersistence** — For transient tasks in Web applications that have lifetime equal to or shorter than a single user's Web session.
- **MemoryStatePersistence** — For transient tasks in Windows-based applications, when the task will have a lifetime shorter than the running application and is only applicable to a single user.

You can also customize the state persistence mechanisms according to your needs. You may decide to only persist the state when it changes, as opposed to storing it whenever a user navigates to a different view. This results in improved performance because data access is reduced. The most scalable option is to persist data on every round trip, but the best performing option is to only persist data when the task is ending. The default block implementation always persists the data when a user changes view.

The **IStatePersistence** interface defines three methods that must be implemented as shown in Table 2.17.

Table 2.17: IStatePersistence Interface Methods

Methods	Description
Init	Passes any required parameters to the class.
Save	Saves the state to a particular medium.
Load	Loads the state from a particular medium.

It is useful to look at each implementation of the **IStatePersistenceInterface** interface in more detail.

IsolatedStoragePersistence

For desktop applications, you may want to use isolated storage to store and retrieve the state. The **IsolatedStoragePersistence** class is used to store references to the task state in the isolated storage. The storage is isolated by user, application, and application domain. This allows you to store state by user and application, and it means that many applications based on UIP can run at the same time and not collide. Data stored by one user on a common machine cannot be seen by another user on the same machine. User, application domain, and application were chosen to provide as much isolation as possible on multiuser machines that may have more than one application that uses UIP.

The **IsolatedStoragePersistence** class implements two of the methods defined in the **IStatePersistence** interface, as shown in Table 2.18.

Table 2.18: IsolatedStoragePersistence Class Methods

Methods	Description
Save	Serializes the State object and writes it to a file in isolated storage.
Load	Returns the appropriate item from the State object.

SecureIsolatedStoragePersistence

The **SecureIsolatedStoragePersistence** class is used to store state in, and retrieve state from, isolated storage. It differs from the **IsolatedStoragePersistence** class in that it encrypts the data before storing it, and decrypts the data upon retrieval.

The **SecureIsolatedStoragePersistence** class implements the three methods defined in the **IStatePersistence** interface, as shown in Table 2.19.

Table 2.19: SecureIsolatedStoragePersistence Class Methods

Methods	Description
Init	Receives a NameValueCollection parameter containing the connection string to be used to connect to the SQL Server database.
Save	Serializes and encrypts the contents of the State object, and then uses the Data Access Application Block to save the information to a SQL Server database.
Load	Uses the Data Access Application Block to load the information from the SQL Server database, decrypts it, and then deserializes it into the State object.

The class also uses the internal **CryptHelper** class that contains two methods, **Encrypt** and **Decrypt**. These methods use the Triple Data Encryption Standard algorithms to encrypt state before it is stored in the database, and to decrypt it when it is retrieved.

SqlServerPersistState

The **SqlServerPersistState** class is used to store state in and retrieve state from a SQL Server database. The **SqlServerPersistState** class implements the three methods defined in the **IStatePersistence** interface, as shown in Table 2.20.

Table 2.20: SqlServerPersistState Class Methods

Methods	Description
Init	Receives a NameValueCollection parameter containing the connection string to be used to connect to the SQL Server database.
Save	Serializes the contents of the State object, and then uses the Data Access Application Block to save the information to a SQL Server database.
Load	Uses the Data Access Application Block to load the information from the SQL Server database, and then deserializes it into the State object.

SecureSqlServerPersistState

The **SecureSqlServerPersistState** class is used to store state in and retrieve state from a SQL Server database. It differs from the **SqlServerPersistState** class in that it encrypts the data before storing it in the database, and decrypts the data upon retrieval. The **SecureSqlServerPersistState** class stores the state object in a table with three columns: the **taskId** (a GUID), the **DateAdded** (a date), and **State** (a blob). The **taskId** column is the key. The **State** object is serialized before it is persisted as a blob. The **SecureSqlServerPersistState** class implements the three methods defined in the **IStatePersistence** interface as shown in Table 2.21.

Table 2.21: SecureSqlServerPersistState Class Methods

Methods	Description
Init	Receives a NameValueCollection parameter containing the connection string to be used to connect to the SQL Server database.
Save	Serializes and encrypts the contents of the State object, and then uses the Data Access Application Block to save the information to a SQL Server database.
Load	Uses the Data Access Application Block to load the information from the SQL Server database, decrypt the information, and then deserialize it into the State object.

The class also uses the internal **CryptHelper** class that contains two methods, **Encrypt** and **Decrypt**. These methods use the Triple Data Encryption Standard algorithms to encrypt state before it is stored in the database, and to decrypt it when it is retrieved.

Note: The **SqlServerPersistState** and **SecureSqlServerPersistState** classes both use the Data Access Application Block to communicate with the SQL Server database. The UIP Application Block ships with and has been tested with version 2.0 of the Data Access Application Block; however, future versions of this block may be released and they are not guaranteed to work with this version of the UIP Application Block. For more information about the Data Access Application Block, see “Data Access Application Block for .NET” available at <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/daab-rm.asp>. For more guidance on data access, see the “.NET Data Access Architecture Guide” available at <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/daag.asp>.

MemoryStatePersistence

The **MemoryStatePersistence** class is used to store references to the task state in memory. There are times that it is inappropriate or undesirable to persist the state with any permanence. The **MemoryStatePersistence** class allows you to not persist state outside of the timeline of the current process. Typically, UIP assumes that a **State** object exists and that it can be persisted. By providing an in-memory persistence provider, there are no artifacts left behind after the program is shut down but UIP can still function as if the state is being persisted. The **MemoryStatePersistence** class stores each **State** object in a hash table keyed by the task ID.

Note: **MemoryStatePersistence** should only be used for processes in Windows-based applications that do not need to span application exits or multiple users. This is because the locking mechanisms may cause bottlenecks in busy Web applications.

The **MemoryStatePersistence** class implements two of the methods defined in the **IStatePersistence** interface, as shown in Table 2.22.

Table 2.22: MemoryStatePersistence Class Methods

Methods	Description
Save	Stores a reference to the State object in a HybridDictionary using the TaskId as a key. It synchronizes access to the dictionary to provide multiple reader single writer access.
Load	Returns the appropriate item from the HybridDictionary .

Note: Unlike the **SqlServerPersistState** and **SecureSqlServerPersistState** implementations, the **MemoryStatePersistence** implementation does not serialize the **State** object, but instead stores a reference to the object. If you have written code to store complex object interrelationships in your state object, you may observe different behavior between this state persistence provider and the SQL Server state persistence providers.

SessionStatePersistence

The **SessionStatePersistence** class is used to store references to the task state in the Web session state. The exact physical location of the session state is determined by setting the **Mode** attribute of the **Session** object. The valid values for this attribute are **InProc**, **SqlServer**, and **StateServer**. The **InProc** setting should be used for processes in Web applications that do not need to span application exits or multiple users. For more information about the **Mode** attribute, see the .NET Framework documentation.

Note: The **SessionStatePersistence** class should only be used in Web applications.

The **SessionStatePersistence** class implements two of the methods defined in the **IStatePersistence** interface, as shown in Table 2.23.

Table 2.23: SessionStatePersistence Class Methods.

Methods	Description
Save	Stores a reference to the state object in the Session object of the current HTTP context, using the taskId property as a key.
Load	Returns the appropriate item from the Session object.

Note: You should test your custom **State** types with this state persistence provider to make sure the serialization works as expected.

Additional Interfaces

This section describes additional interfaces supplied with the UIP Application Block that you may choose to implement.

ITask

When you start a task in the UIP Application Block, you have two choices. The most straightforward approach is to let the UIP Application Block manage and maintain your task IDs. This makes use of the overloaded methods on the UIP manager to start a task and the navigator will create and manage the task IDs. The disadvantage to this approach is that the block does not give you easy access to the task IDs.

If you want to suspend a task associated with the user and have the task restart with the correct state when the user is ready to resume, you will need to keep track of the task ID. To do this, you will need to implement the **ITask** interface from a **Task** class that you create. The navigator will pass the **taskId** property to your **Task** class through the **Create** method. You can then store the task ID; for example, by user name. The main advantage of predefining a task in this way is that you can correlate a task with a user. The Store QuickStart shows how to suspend and resume a task.

A user may work on multiple tasks at any one time; for example, a user could have two browser windows open, log in twice, and work on two different tasks. It is the responsibility of the calling application to detect this scenario and handle assigning a unique task ID to each task.

The **ITask** interface defines two methods as shown in Table 2.24.

Table 2.24: ITask Interface Methods

Methods	Description
Create	Creates and stores a task/user correlation with a known Task ID.
Get	Returns the Task ID.

ILayoutManager

The **ILayoutManager** interface gives you the ability to add custom layout logic to views in your applications. You can define generic layouts such as horizontal or vertical layouts in which your controls appear from left to right or top to bottom respectively. You can also customize a layout for each of your views by specifying the position of each control. You create a class that implements the **ILayoutManager** interface and defines the **LayoutControls** method as shown in Table 2.25.

Table 2.25: ILayoutManager Interface Methods

Methods	Description
LayoutControls	Performs layout functions on any child controls of the parent control. It accepts a parameter named parentControl , which is the container on which it will perform the layout.

The layout manager and the views using the layout manager must also be specified in the configuration file. For more information, see “Customizing View Layouts” in Chapter 3, “Developing Applications with the UIP Application Block.”

IShutdownUIP

In Windows-based applications, a user interface process is completed when the user closes the forms involved. Even though the process is completed, control does not automatically return to the originating form. It is possible to explicitly exit the application from UIP, but if the application has additional tasks to complete, this option is not acceptable. If control is to be returned to the originating form (or to another part of the application that is not UIP-related), the classes that need to be notified when UIP is complete must implement the **IShutdownUIP** interface. The **IShutdownUIP** interface has one public method named **Shutdown** as shown in Table 2.26.

Table 2.26: IShutdownUIP Interface Methods

Methods	Description
Shutdown	Called when UIP shuts down.

The classes implementing the **IShutdownUIP** interface must also be registered with the **UIPManager** class. For more details, see “UIP Shutdown” in Chapter 3, “Developing Applications with the UIP Application Block.”

IWizardViewTransition

Implement the **IWizardViewTransition** interface if you want to customize the functionality of your wizard views. This interface provides seven methods for wizard transitions, as shown in Table 2.27.

Table 2.27: IWizardViewTransition Interface Methods

Method	Description
DoNext	Performs custom processing before navigating to the next view.
DoBack	Performs custom processing before navigating to previous view.
DoCancel	Performs custom processing canceling the wizard task.
DoFinish	Performs custom processing before finishing the wizard task.
SupportsCancel	Indicates the particular wizard view supports the cancel behavior.
SupportsFinish	Indicates the particular wizard view supports the finish behavior.
WizardActivated	Called when a wizard view is activated. Allows you to perform any processing that should occur whenever the view is activated in a wizard.

Summary

This chapter examined the architecture of the UIP Application Block. It explained how views are displayed, transitions occur, and state is maintained. It also examined in detail each of the elements that make up the block. By understanding the design of the UIP Application Block, you can determine how to use it in your own environment.

3

Developing Applications with the UIP Application Block

You can use the UIP Application Block to control presentation layer business logic in both Windows-based applications and Web applications. This chapter includes information about developing applications using the provided classes of the UIP Application Block. It shows how to develop the views (Windows Forms or Web pages) that your users interact with, define the path of flow between the views, and develop a controller to coordinate the navigation between views.

This chapter also introduces several QuickStarts that come with the block. The QuickStarts allow you to gain a deeper understanding of the block.

Important: The QuickStarts are intended to aid you in understanding the block; they are not intended to be production-ready code samples and may not be indicative of the way you should develop your application.

The UIP Application Block is designed to be flexible: It is useful without modification, and also gives you the ability to extend its functionality. You can write custom classes in your application to replace standard classes in the block. Appendix A, “Extending the UIP Application Block,” provides guidance on developing your own state management class, developing custom state persistence providers, and developing custom view implementations.

Preparing to Build the UIP Application Block

To work with the UIP Application Block, you need to ensure that your system meets the following minimum software requirements:

- Microsoft® Windows® XP or Windows Server™ 2003 operating system
- Microsoft .NET Framework version 1.1
- Microsoft Visual Studio® .NET 2003 Enterprise Architect, Enterprise Developer, or .NET Professional edition development system

These are the minimum requirements for using the block. However, if you are using the block for Web applications, you need Internet Information Services (IIS) version 5.0 or later, and if you are planning to use the SQL Server persistence provider, you need Microsoft SQL Server™ 2000.

Note: Although the UIP Application Block has not been fully tested on Windows 2000, the QuickStarts run on Windows 2000.

Migrating Applications That Use UIP Application Block, Version 1

If you are already using version 1 of the UIP Application Block in an application, you must make a few modifications in your code to ensure that your application works with version 2 of the UIP Application Block. The main changes are in the calls to the **StartTask** method on the **UIPManager** class and in the constructor on the **ControllerBase** class.

► To migrate an application from version 1 to version 2

1. Replace calls to the **StartTask** method with one of the overloaded **StartNavigationTask**, **StartOpenNavigationTask**, or **StartUserControlsTask** methods on the **UIPManager** class in UIP version 2. For more information, see “Starting and Resuming Tasks,” later in this chapter.
2. Replace the constructor for your class that inherits from the **ControllerBase** class with a constructor that accepts the navigator you are using as a parameter. For more information, see “Creating the Controller,” later in this chapter.

Developing an Application with the UIP Application Block

The main elements you need to develop if you are using the UIP Application Block are:

- The custom controller class — You create this by inheriting from the **ControllerBase** class supplied with the block. Use the base class to define the custom methods for the functionality and navigation within your application.
- The views — You create the views for your specific application type, inheriting from the predefined view types in the block.
- The configuration file — You define the classes to use for state management and process management, the controller class, the views in the process, and the navigation paths between them.
- The **State** class — You may need to add information related to your business logic that you want to be stored in and retrieved from the **State** class.

At a high level, to create applications using the UIP Application Block, you need to do the following:

- Separate the application into distinct user interface processes.
- Select a navigator for each user interface process.
- Determine the state persistence provider.
- Build the UIP Application Block, and refer to it in your application.
- Create the controller(s).
- Construct your process flow by starting, resuming, linking, and nesting tasks.
- Create the views.
- Create the configuration file.
- Add additional functionality.

The topics that follow detail these requirements.

Separating the Application into Distinct User Interface Processes

Before you can begin writing code for your application, you need to consider the requirements of your user interface processes. Applications are usually composed of many user interface processes. You should define user interface processes by looking at your business requirements, not your technological requirements. By separating these correctly, you can reuse your code more efficiently because the user interface processes are encapsulated into distinct functional units.

You should identify each user interface process in the application. Remember that you can link user interface processes by starting and resuming tasks programmatically. For example, in a travel booking application, you will have user interface processes to enter personal settings, purchase plane tickets, make hotel reservations, and make payments. Note that many of these are dependent on each other, but are still separate processes.

Selecting a Navigator for Each User Interface Process

Each user interface process has a control flow. You define the control flow for each user interface process through a navigator. There are four navigators you can choose from: the graph navigator, the wizard navigator, the user controls navigator, and the open navigator. Each navigator controls the transitions between views differently, and may or may not lock the user into a predefined series of views.

Graph Navigator

Use the graph navigator when you have well-defined and fixed transitions between views. You should know which transitions are possible from each view, because the user is locked into following a certain process. Outputs from each view lead to either another view or the termination of the task. The graph navigator can be used with either Windows-based applications or Web applications.

You should draw the user interface process as a flow between the views with the outputs of each view leading to other views. For an example of this, see the sample graph navigator in “Graph Navigator” in Chapter 2, “Design of the UIP Application Block.” Use your diagram to help you configure the `<startView>`, `<node>`, and `<navigateTo>` elements of the `<navigationGraph>` section of the application configuration file.

Open Navigator

The open navigator provides you with the greatest degree of flexibility. Use it when you do not have well-defined transitions and when you have your own external navigation functionality. In the latter case, the external navigation figures out what view is next and uses the open navigator to display the view. Open navigation is most typically used in Web applications or with Windows Forms that do not have user controls.

User Controls Navigator

A user controls navigator allows the user flexibility in transitioning between views and is best used with complex forms that contain user controls. With this navigator, you can allow the user to freely switch between controls, or you can control the user’s navigation to some degree. If you create controls that are reused on many Windows Forms or multiple times on a single Windows Form, then the user controls navigator is a good choice. The user controls navigator can only be used in Windows-based applications.

Wizard Navigator

The wizard navigator provides a way to transition between views in a wizard. The wizard navigator can only be used in Windows-based applications.

Note: For more information about the design of these navigators, see Chapter 2, “Design of the UIP Application Block.”

Determining the State Persistence Provider

Each user interface process has a different task lifetime. For example, you may want a user to be able to resume a task that he or she started earlier in a different Web session or after closing a Windows application. Some tasks should be associated with a single user (for example, a checkout task in a retail site) or multiple users (for example, a support call being escalated through members of a helpdesk team). You can use the information in Table 3.1 to determine the best state persistence provider for your user interface process and application type.

Table 3.1: Attributes of State Persistence Providers

Persistence Provider Class Name	Task is Assignable to Different Users	Task Can Span Sessions	Windows-based Applications Support	Web-based Applications Support	Persisted Data is Encrypted
SqlServerPersistState	✓	✓	✓	✓	
SecureSqlServerPersistState	✓	✓	✓	✓	✓
SessionStatePersistence				✓	
MemoryStatePersistence			✓		
IsolatedStoragePersistence	✓	✓	✓		
SecureIsolatedStoragePersistence	✓	✓	✓		✓

Note: If you use the **SecureSqlServerPersistState** or **SecureIsolatedStoragePersistence** implementations, you must ensure that the encryption key exists in the expected path in the registry. For more information, see “Creating the <objectTypes> Section” in “Creating the Configuration File Section” later in this chapter.

There are many ways to approach state management, and each has its own advantages and disadvantages. The implementations included in the UIP Application Block are scalable, but may have performance implications. You should look at the features of each persistence provider, and determine which is most appropriate. Bear in mind that if you use the **SecureSqlServerPersistState** class or the **SecureIsolatedStoragePersistence** class, the performance of your application may be degraded. This is due to the cost of decrypting and encrypting your data on each read and write operation. For more information about each of the persistence providers, see “State Persistence Providers” in Chapter 2, “Design of the UIP Application Block.”

Note: If the supplied providers do not meet your requirements, you can implement your own provider. For more details about how to do so, see “Customizing State Persistence,” in Appendix A, “Extending the UIP Application Block.”

Building and Referencing the UIP Application Block

Before you can use the UIP Application Block, you must build the block and then reference it in your application.

► To build the UIP Application Block

1. Open the **UIProcess_cs.sln** solution (found in the *<installation location>* folder).
2. Build the solution. Doing so generates two assemblies:
 - **Microsoft.ApplicationBlocks.Data.dll**
 - **Microsoft.ApplicationBlocks.UIProcess.dll**

► To reference the UIP Application Block in your application

1. Set a reference to the UIP Application Block assembly **Microsoft.ApplicationBlocks.UIProcess.dll**.
2. Set a reference to the Data Access Application Block assembly **Microsoft.ApplicationBlocks.Data.dll**.
3. Add an **Imports** (Visual Basic) or **using** (C#) statement at the top of each source file that uses classes from the block to reference the **Microsoft.ApplicationBlocks.UIProcess** namespace. All UIP Application Block types are located within this namespace.

The following code examples show how to reference the block in your source files.

```
[Visual Basic]
Imports Microsoft.ApplicationBlocks.UIProcess
```

```
[C#]
using Microsoft.ApplicationBlocks.UIProcess;
```

Creating the Controller

You need to create controller classes with logic to interact with the business layers of your application. Controller classes expose methods that the views can use to get data or invoke specific functions. You should create methods that the views will invoke to get the data they need to render, and methods to pass data to the controller class. The views can access state held by the controller, and can modify the data, but only the controller can perform operations with the data, such as sending it to a Web service.

You can create your controller by inheriting from the **ControllerBase** class provided in the UIP Application Block.

Note: For more information about the design of the **ControllerBase** class, see Chapter 2, “Design of the UIP Application Block.”

To create a controller for use with your application, you need to:

- Reference and inherit from the **ControllerBase** class.
- Create the constructor for your class.
- Add custom code.

Inheriting the ControllerBase Class

The following code shows how to reference and inherit from the **ControllerBase** class.

```
[Visual Basic]
Imports Microsoft.ApplicationBlocks.UIProcess
```

```
Public Class MyController
    Inherits ControllerBase
End Class
```

```
[C#]
using Microsoft.ApplicationBlocks.UIProcess;

public class MyController : ControllerBase
{
}
```

Creating the Constructor

The constructor for your class must accept the navigator you are using as a parameter. You must also call the constructor of the base class. The following code examples show how to do this.

```
[Visual Basic]
Public Sub New(ByVal navigator As Navigator)
    MyBase.New(navigator)
    . . .
End Sub

[C#]
public MyController(Navigator navigator) : base(navigator){}
```

Adding Custom Code

You must add code in your controller class to allow the user to navigate through your workflow process. You need to write one method for each action a user needs to undertake. Each method should execute relevant business logic and call the **Navigate** method with the **NavigateValue** passed as a parameter. The **NavigateValue** corresponds to the next view or node you specify. The following code examples show how to do this.

```
[Visual Basic]
Me.Navigate("AddedData")

[C#]
Navigate("AddedData");
```

Whenever you call this method, you must pass a valid **NavigateValue**; otherwise, UIP will throw an exception because the view is unknown. For backward compatibility with the first version of UIP, you can set the **NavigateValue** property on the **State** object, and then call the **Navigate** method, as follows.

```
[Visual Basic]
Public Sub AddDataToDatabase(ByVal SomeData As Object)
    ' call custom data access code to insert data into database
    Me.State.NavigateValue = "AddedData"
    Me.Navigate
End Sub

[C#]
public void AddDataToDatabase(object SomeData)
{
    // call custom data access code to insert data into database
    this.State.NavigateValue = "AddedData";
    this.Navigate();
}
```


You can also add code to your controller to link tasks or navigators together. For more information, see “Linking Tasks,” later in this chapter.

Starting and Resuming Tasks

Before you can guide your users through the workflow you have defined for them, you must use the **UIPManager** class to start a task. The **StartNavigationTask**, **StartOpenNavigationTask**, or **StartUserControlsTask** methods have several public overloaded methods you can use to start a task. Table 3.2 explains the parameters that you can pass to the **StartNavigationTask**, **StartOpenNavigationTask**, or **StartUserControlsTask** methods.

Table 3.2: StartNavigationTask, StartOpenNavigationTask, or StartUserControlsTask Parameters

Parameter	Description
navigator	The name of the navigator defined in the application configuration file. It may refer to the name of the graph navigator, wizard navigator, open navigator, or user controls navigator.
task	An existing task whose state should be retrieved and used.
taskId	The ID of a previously saved task.
firstViewName	Name of the first view using the open navigator.
taskArguments	Any information that is required when starting or resuming an existing task.

Note: The **task** and **taskId** parameters are required only when you start a previously suspended task. These parameters are relevant only if you have implemented the **ITask** interface to provide the ability to suspend and restart tasks. For more information about implementing the **ITask** interface, see “Implementing the **ITask** Interface,” in Appendix A, “Extending the UIP Application Block.”

The navigator you are using, and whether or not you are starting a new task or starting a task that was previously suspended, determines which overloaded method you should use and which parameters you should pass to the method. Each of the methods is described in the sections that follow, organized by navigator type.

Starting and Resuming Tasks with the Graph Navigator

If you are using the graph navigator and starting a new task, you must pass the name of the graph navigator as a parameter, as shown in the following code examples.

```
[Visual Basic]
UIPManager.StartNavigationTask("MyNavGraph")
```

```
[C#]
UIPManager.StartNavigationTask("MyNavGraph");
```

If you are using the graph navigator and starting a task that was previously suspended, you need to pass the name of the graph navigator and either the task or the task ID. The following code examples show the methods you can use to resume a task with a graph navigator.

```
[Visual Basic]
UIPManager.StartNavigationTask("MyNavGraph", MyTask)
UIPManager.StartNavigationTask("MyNavGraph", MyTaskID)
```

```
[C#]
UIPManager.StartNavigationTask("MyNavGraph", MyTask);
UIPManager.StartNavigationTask("MyNavGraph", MyTaskID);
```

Starting and Resuming Tasks with the Open Navigator

If you are using the open navigator and starting a new task, you need to pass the name of the open navigator and the name of the first view as a parameter, as shown in the following code examples.

```
[Visual Basic]
UIPManager.StartOpenNavigationTask("MyOpenNav", "MyFirstViewName")
```

```
[C#]
UIPManager.StartOpenNavigationTask("MyOpenNav", "MyFirstViewName");
```

If you are using the open navigator and starting a task that was previously suspended, you must pass the name of the open navigator and either the **task** object or the task ID. You may also specify the name of the first view when passing a **task** object. This name is used if the current view is not already stored in the **State** object. The following code examples show the methods you can use to resume a task with an open navigator.

```
[Visual Basic]
UIPManager.StartOpenNavigationTask("MyOpenNav", MyTask)
UIPManager.StartOpenNavigationTask("MyOpenNav", MyTaskID)
UIPManager.StartOpenNavigationTask("MyOpenNav", "MyFirstViewName", MyTask)
```

```
[C#]
UIPManager.StartOpenNavigationTask("MyOpenNav", MyTask);
UIPManager.StartOpenNavigationTask("MyOpenNav", MyTaskID);
UIPManager.StartOpenNavigationTask("MyOpenNav", "MyFirstViewName", MyTask);
```

Starting and Resuming Tasks with the User Controls Navigator

If you are using the user controls navigator and starting a new task, you must pass the name of the user controls navigator as a parameter, as shown in the following code examples.

```
[Visual Basic]
UIPManager.StartUserControlsTask("MyUserControlsNav")
```

```
[C#]
UIPManager.StartUserControlsTask("MyUserControlsNav");
```

If you are using the user controls navigator and starting a task that was previously suspended, you must pass the name of the user controls navigator and either the **task** object or the task ID. You can pass data to the tasks by using the overloaded method containing the **TaskArgumentsHolder** parameter. You can use the methods shown in the following code examples to resume a task with a user controls navigator.

```
[Visual Basic]
UIPManager.StartUserControlsTask("MyUserControlsNav", MyTask)
UIPManager.StartUserControlsTask("MyUserControlsNav", MyTaskID)
UIPManager.StartUserControlsTask("MyUserControlsNav", MyTask, MyTaskArgs)
UIPManager.StartUserControlsTask("MyUserControlsNav", MyTaskID, MyTaskArgs)
```

```
[C#]
UIPManager.StartUserControlsTask("MyUserControlsNav", MyTask);
UIPManager.StartUserControlsTask("MyUserControlsNav", MyTaskID);
UIPManager.StartUserControlsTask("MyUserControlsNav", MyTask, MyTaskArgs);
UIPManager.StartUserControlsTask("MyUserControlsNav", MyTaskID, MyTaskArgs);
```

Note: If you pass a **task** object that is null, you are actually starting (and not resuming) a task. You may also use any of the overloaded methods to start a task by passing a null **task** object.

Linking Tasks

As a user navigates through an application, he or she often needs to use more than one user interface process to complete the job. Sometimes the user interface processes follow a linear path, and at other times one user interface process branches to another process and then returns to the original. For example, when shopping online, a user typically works through a Shopping user interface process and then moves to a Checkout user interface process. However, during the Checkout user interface process, the user may want to enter an alternative shipping address, meaning that the user must complete an Add Ship Address user interface process, and then return to and complete the Checkout user interface process. These tasks — including the optional additional tasks — are *linked*.

There are four concepts involved in linking tasks:

- Starting tasks — You can start a task by calling any of the navigation start task methods or by calling the **OnStartTask** on your controller.
- Ending tasks — You can end a task by calling either the **SuspendTask** or the **OnCompleteTask** method on your controller. Both methods allow the user interface process to clear the state for the current task and erase the current view from memory. However, the **SuspendTask** method does not remove the task from the state persistence provider. To clear the **State** object from both memory and from the state persistence provider, you must call the **OnCompleteTask** method.
- Resuming tasks — You can resume a task that has been suspended by calling a navigation start task method (passing the task identifier or the **task** object that you want to resume), or by calling the **OnStartTask** method of an existing controller (passing the task identifier, the **task** object, or task arguments).
- Passing data between tasks — You can pass data between tasks by using the **OnStartTask** overload with the **TaskArgumentsHolder** parameter. You can use this parameter to pass any data you want, and if the controller of the current view implements the **EnterTask** method, it can handle that data appropriately.

The **TaskArgumentsHolder** parameter contains a property of type object, meaning that you can pass any type of data by using this property. You should not pass an instance of the controller itself, nor should you pass the controller's **State** object. This is because each task should have its own **State** object of the type specified in the graph navigator. Instead, you should pass your own types or common data classes. If you design your own structs or types, remember that you need to reference them from both the caller and the callee.

Note: You can link tasks that have different types of navigators. The name of the navigator that is passed to the **OnStartTask** method determines what type of navigator is appropriate for your task.

The UIP Application Block allows you to link tasks in two ways: by *chaining*, or by *nesting*.

Chaining Tasks

You should chain tasks when you want one task to begin when another one ends, as shown in Figure 3.1.

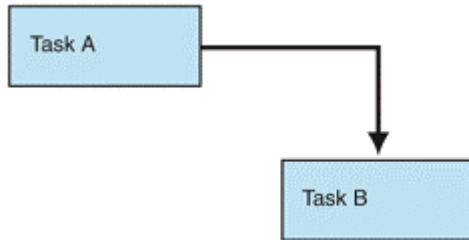


Figure 3.1

Chaining tasks

When you chain two tasks, you can pass data between them before ending the first task. For example, a user may be booking a flight and hotel accommodations using a Purchase Plane Ticket process followed by a Book Hotel process. If both of these bookings require the same Booking ID, then you must pass that ID between the tasks.

The following steps use this example to describe how to chain tasks in your controller:

1. In the method of your controller class for the Purchase Plane Ticket process, store the Booking ID from the task state into a local variable.
2. Invoke the **SuspendTask** method to suspend the Purchase Plane Ticket task.
3. Invoke the **OnStartTask** method, passing the name of the next navigator and the event Booking ID as part of the **TaskArgumentsHolder**.
4. In the controller for the first view of the Book Hotel process, implement the **EnterTask** method, retrieve the Booking ID from the **TaskArgumentsHolder**, and place it in the current task's **State** object. The first view of the Book Hotel process is displayed and the task continues.

For a complete example of how to chain tasks, see the MultiNavGraph QuickStart included with the UIP Application Block.

Nesting Tasks

Nesting tasks simply extends the concepts of chaining, but requires you to pass data back and forth, and to re-enter a running task. One task flows control into another, which is executed and ended, and control returns to the original task. This process is illustrated in Figure 3.2.

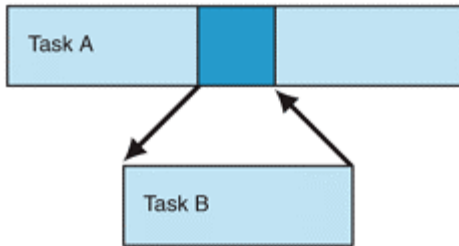


Figure 3.2

Nesting tasks

For example, you may need to allow a user to run an Add New Shipping Address process while he or she is partially through the Purchase Product process. Once the new address is saved, the Add New Shipping Address process needs to end and the Purchase Product process needs to continue, using the new address.

The following steps use this example to describe how to nest tasks in your controller:

1. In the method of your controller class for the Purchase Product process, store the **CustomerId** and **TaskId** from the task state into a local structure or collection. You must pass the **TaskId**, so the nested process knows the task to which it must return control.
2. Invoke the **OnStartTask** method, passing the name of the next navigator, the customer information as part of the **TaskArgumentsHolder**, and the **TaskId**.
3. In the controller for the first view of the Add New Shipping Address process, implement the **EnterTask** method, retrieve the **CustomerId** and **TaskId** from the **TaskArgumentsHolder**, and place them in the current task's **State** object. The first view of the Add New Shipping Address process is displayed and the task continues.
4. When the Add New Shipping Address process is ready to return, retrieve the **TaskId** for the Purchase Product process, which is stored in the current task's state, and invoke the **OnStartTask** method, passing the name of the navigator, the new address, and the original **TaskId**. If the task state for the specified task is found, the current view for that task is displayed.

Creating Views

You need to create or reuse the user interface components for the views in your application. For each node in the process, you should design a view that displays data to, or gets data from, the user. The views should invoke methods on the controllers to set data, but can retrieve the state of the controller directly.

You can create views in your application by performing the following steps:

1. Inherit from the appropriate view class. There are three view classes in the UIP Application Block: **WebFormView**, **WindowsFormView**, and **WindowsFormControlView**.
2. Implement the **Initialize** method to access custom view information, such as background color. For more information see “Adding Extensible Configuration Schema” later in this chapter.
3. Write code to access the controller for each view in the user interface process.
4. Add navigation code to your views by calling the methods defined in your controller class.

The AdvancedHostDemo QuickStart demonstrates how to use the **WindowsFormControlView** class to create user control views. You can also define your own custom views by implementing the **IView** interface. For more information, see “Customizing Views and View Management” in Appendix A, “Extending the UIP Application Block.”

Inheriting from the Base Class

All views that are to be controlled by the UIP Application Block must inherit from the relevant view base class or must implement the **IView** interface. If you create your own implementation of **IView**, you must create the associated view manager as well. The first view in the application does not need to inherit from the view base class; it is not until after this view is created that the controlling objects are created.

Inheriting the WebFormView Class

The following example code shows how to inherit from the **WebFormView** class:

```
[Visual Basic]
Public Class WebForm2
    Inherits WebFormView
End Class

[C#]
public class WebForm2 : WebFormView
```

The **WebFormView** class is inherited from the **System.Web.UI.Page** class; therefore, you do not need to explicitly inherit from that class.

Inheriting the **WindowsFormView** Class

The following example code shows how to inherit from the **WindowsFormView** class.

```
[Visual Basic]
Public Class Form2
    Inherits WindowsFormView
End Class

[C#]
public class WinForm2 : WindowsFormView
```

The **WindowsFormView** is inherited from the **System.Windows.Forms.Form** class; therefore, you do not need to explicitly inherit from that class.

Inheriting the **WindowsFormControlView** Class

The following example code shows how to inherit from the **WindowsFormControlView** class.

```
[Visual Basic]
Public Class Form2
    Inherits WindowsFormControlView
End Class

[C#]
public class WinForm2 : WindowsFormControlView
```

The **WindowsFormControlView** is inherited from the **System.Windows.User.UserControl** class; therefore, you do not need to explicitly inherit from that class.

Accessing the Controller

To use the controller navigation methods, you need to write code to access the controller for each view in the process. The following example code shows how this can be done.

```
[Visual Basic]
Private ReadOnly Property MyController() As MyController
    Get
        Return CType(Me.Controller, MyController)
    End Get
End Property

[C#]
private MyController MyController
{
    get{ return (MyController)this.Controller; }
}
```


Adding the Navigation Code

To add the navigation abilities to your views, you call the methods defined in your controller class. The following example code shows how to call the controller method shown earlier.

```
[Visual Basic]  
MyController.AddDataToDatabase(SomeData)
```

```
[C#]  
MyController.AddDataToDatabase(SomeData);
```

You should make any property changes to your **State** object within the controller methods, not directly from your views. This ensures that the controller can intercept these calls, and makes some of the custom state management techniques simpler to implement.

Note: If you receive exceptions stating that a view cannot be created, you may have errors in your configuration file, errors in the constructor of the view, or errors initializing the controller of the view.

Creating the Configuration File

The XML-based configuration file is used to define the classes that play the major roles in your application user interface process, the views that are used, and the navigators that define the workflow paths for the application.

The contents of the configuration file are similar for all application types. Generally, in Web applications, you work with Web.config, and in Windows-based applications, you work with the app.config file.

Defining the uipConfiguration Section

You must add a new configuration section to the configuration file to define your UIP configuration settings. This section must be declared in the <configSections> section of the file as shown in the following code.

```
<configuration>  
  <configSections>  
    <section name="uipConfiguration"  
      type="Microsoft.ApplicationBlocks.UIProcess.UIPConfigHandler,  
        Microsoft.ApplicationBlocks.UIProcess,  
        Version=1.0.1.0,Culture=neutral,PublicKeyToken=null" />  
  </configSections>  
</configuration>
```

Creating the `uipConfiguration` Section

You need to create the `<uipConfiguration>` section itself. This section is used to define the classes to be used by the application for view management, state management, and controllers, the views available to the user, and the navigators that define the workflow path through those views.

The structure of this section is defined in the XML Schema Definition (XSD) file called **UIPConfigSchema.xsd**, which is included in the block. This schema details which elements and attributes are required and which are optional, how many times each may occur, and in what order they should occur. If the required elements of the `<uipConfiguration>` section do not adhere to the schema, exceptions are thrown when the **UIPConfigHandler** class attempts to parse and validate the configuration section.

There are minor differences in the contents of this section depending on the type of application that you are developing. Examples for Web applications and Windows-based applications are included in this section.

The structure of the `<uipConfiguration>` section is shown in the following code. The sections in italics are optional.

```
<uipConfiguration enableStateCache="true">
  <objectTypes>
    . . .
  </objectTypes>
  <views>
    . . .
  </views>
  <sharedTransitions>
    . . .
  </sharedTransitions>
  <navigationGraph>
    . . .
  </navigationGraph>
  <uipWizard>
    . . .
  </uipWizard>
  <userControls>
    . . .
  </userControls>
</uipConfiguration>
```

Note: If you want to allow users to navigate to views by specifying a URL or by clicking on the back or forward buttons in a Web browser in your Web applications, you must set the **allowBackButton** attribute to **true**. For example, the **<uiConfiguration>** section may begin with the following line:

```
<uiConfiguration enableStateCache="true" allowBackButton="true">
```

If you enable the **allowBackButton** attribute, state is not rolled back when your user clicks the back button. For example, if a user has items in a shopping cart and he or she clicks the back button, the user still has the same items in his or her shopping cart. For more details about how the UIP Application Block uses conditional logic to determine how the user navigates, see the “Navigator” section in Chapter 2, “Design of the UIP Application Block.”

The sections are briefly described in Table 3.3.

Table 3.3: <uiConfiguration> Sections

Section Name	Description
objectType	Defines the classes to be used for managing and controlling the user interface process in an application.
views	Defines the details of the views to be used in an application.
sharedTransitions (optional)	Defines the common transition points for all views.
navigationGraph (optional)	Defines the details of the graph navigator. Required by each task that uses a graph navigator. Used to define wizards that support branching.
uiWizard (optional)	Defines the details of the wizard navigator. Used to define simple wizards that do not support branching.
userControls (optional)	Defines the details of the user controls navigator. Required by each task that uses a user controls navigator.

For each task that uses a wizard navigator, you must either define the wizard navigator in the **<uiWizard>** element or in the **<navigationGraph>** element. The wizard defined by the **<uiWizard>** element supports linear navigation, and the wizard defined by the **<navigationGraph>** element supports branching. If you define a wizard with the **<navigationGraph>** element, you must set the **runInWizardMode** attribute to **true**. Any of your wizard views that support branching must implement the **IWizardViewTransition** interface. For an example of this, see the InsurancePurchaseWizard QuickStart.

If you are using the open navigator, you do not need to define any of the optional sections because you call the views directly. However, all valid views need to be specified in the **<views>** section of the configuration file.

Note: If you are developing an application with both a Web and a Windows interface to the same underlying state, and you expect both interfaces to be used on the same computer, you should not cache the state. If you do cache the state, the cached version from the Web application may overwrite later changes that have been made using the Windows version. To disable caching of state, set the **enableStateCache** attribute of the **<uipConfiguration>** section to **false**.

The following sections describe how to define each of these configuration sections.

Creating the <objectTypes> Section

The <objectTypes> section defines the classes used for managing and controlling the user interface process in a Windows or Web application. The sections are shown in the following code; the <layoutManager> section is optional.

```
<objectTypes>
  <iViewManager . . . />
  <state . . . />
  <controller . . . />
  <layoutManager . . . />
  <statePersistenceProvider . . . />
</objectTypes>
```

The sections are briefly described in Table 3.4.

Table 3.4: <objectTypes> Sections

Section Name	Description
iViewManager	Specifies the view manager responsible for activating and deactivating views within the application.
state	Controls the state of your application. Can be extended to include application-specific state attributes.
controller	Controls the flow of the views and responds to requests from views.
layoutManager (optional)	Performs custom layout of controls on a view.
statePersistenceProvider	Specifies the location where the state should be saved between view transitions.

Each of these sections requires specific attribute settings to link it to the relevant class in the UIP Application Block. Table 3.5 defines the attributes required by each of the elements in the <objectTypes> section.

Table 3.5: Configuration Attributes of the <objectTypes> Elements

Attribute Name	Description
name	The name used to refer to this object within the configuration file and block code.
type	The type information for this object, which identifies the assembly where the object is defined and its class name.

If you use the **SecureSqlServerPersistState** or **SecureIsolatedStoragePersistence** implementations, you must ensure that the encryption key exists in the expected path in the registry. You can set this key in one of two ways:

- Run the **InstallDemos.vbs** file to create the registry key. This file is included with this block.
- Add the **registryPath** attribute to the <statePersistenceProvider> node of the <objectTypes> element in the configuration file, and set the value of the attribute to the registry path you want to use.

If you use the **SqlServerPersistState** or **SecureSqlServerPersistState** implementations, you also need to set a **connectionString** attribute that provides connection information for the SQL Server database.

The following code is an example of the <objectTypes> section.

```
<objectTypes>
  <iViewManager
    name="WebFormViewManager"
    type="Microsoft.ApplicationBlocks.UIProcess.WebFormViewManager,
    Microsoft.ApplicationBlocks.UIProcess,
    Version=1.0.0.0, Culture=neutral, PublicKeyToken=null"/>

  <state
    name="State"
    type="Microsoft.ApplicationBlocks.UIProcess.State,
    Microsoft.ApplicationBlocks.UIProcess,
    Version=1.0.0.0, Culture=neutral, PublicKeyToken=null"/>

  <controller
    name="MyController"
    type="Test.MyController, Test,
    Version=1.0.0.0, Culture=neutral, PublicKeyToken=null" />

  <layoutManager
    name="HorizontalLayoutManager"
    type="LayoutManager.HorizontalLayoutManager, LayoutManager,
    Version=1.0.1.0, Culture=neutral, PublicKeyToken=null" />
```

(continued)

(continued)

```
<layoutManager
  name="VerticalLayoutManager"
  type="LayoutManager.VerticalLayoutManager, LayoutManager,
  Version=1.0.1.0,Culture=neutral,PublicKeyToken=null" />

<stateProvider
  name="SqlServerPersistState"
  type="Microsoft.ApplicationBlocks.UIProcess.SqlServerPersistState,
  Microsoft.ApplicationBlocks.UIProcess, Version=1.0.0.0,
  Culture=neutral, PublicKeyToken=null"
  connectionString="Data Source=localhost;Initial Catalog=UIPState;
  Trusted_Connection=True"/>
</objectTypes>
```

Note: Exceptions thrown from the **GenericFactory** class when creating objects may be due either to errors in your configuration file (such as invalid assembly/class names) or to errors in the constructor of the object you are trying to create.

Creating the <views> Section

The <views> section defines the details of the views to be used in the Web or Windows application. Each element requires three attributes: the **name**, **type**, and **controller**. In Windows-based applications, you can use the optional attributes as described in Table 3.6. The **layoutManager** and **any** attributes are optional in both Web and Windows-based applications.

Table 3.6: Configuration Attributes of the <view> Section

Attribute Name	Description
name	The name of the view as referred to in the navigator.
type	The class name of the class that defines this view.
controller	The name of the controller as defined in the <objectTypes> section to be used for this view.
layoutManager (optional)	The name of layout manager.
stayOpen (optional)	Whether the view should remain open when the user navigates away from it.
openModal (optional)	Whether the view should be opened as a modal view. If openModal is set to true , then stayOpen should be set to false .
canHaveFloatingWindows (optional)	Whether the view can host floating windows.
floatable (optional)	Whether the view should float.
any (optional)	You can augment your view information with custom elements or attributes.

The following code is an example of the **<views>** section for a Web application. This example shows the definition of five views, all identifying ASP.NET Web pages.

```
<views>
  <view name="Page1" type="Page1.aspx" controller="MyController" />
  <view name="Page2" type="Page2.aspx" controller="MyController" />
  <view name="Page3" type="Page3.aspx" controller="MyController" />
  <view name="Page4" type="Page4.aspx" controller="MyController" />
  <view name="Error" type="Error.aspx" controller="MyController" />
</views>
```

The following code is an example of the **<views>** section for a Windows application. This example shows the definition of five views, all linked to Windows Forms.

```
<views>
  <view
    name="Form1"
    type="Demo.Form1, Demo, Version=1.0.0.0, Culture=neutral,
    PublicKeyToken=null"
    controller="MyController"
    layoutManager="VerticalLayoutManager"
    stayOpen="true"/>
  <view
    name="Form2"
    type="Demo.Form2, Demo, Version=1.0.0.0, Culture=neutral,
    PublicKeyToken=null"
    controller="MyController" />
  <view
    name="Form3"
    type="Demo.Form3, Demo, Version=1.0.0.0, Culture=neutral,
    PublicKeyToken=null"
    controller="MyController" />
  <view
    name="Form4"
    type="Demo.Form4, Demo, Version=1.0.0.0, Culture=neutral,
    PublicKeyToken=null"
    controller="MyController" />
  <view
    name="Error"
    type="Demo.Error, Demo, Version=1.0.0.0, Culture=neutral,
    PublicKeyToken=null"
    controller="MyController"
    stayOpen=false
    openModal=true />
</views>
```

If the **ShowModal** method is called on the next view and the view's **openModal** attribute is set to **true**, then the current view becomes the parent form and the view manager disables the parent form, but leaves it open when the view is activated. Once the user has closed the view, the view manager enables the parent form.

If the view's **floatable** attribute is set to **true**, the view manager determines whether the parent form's **canHaveFloatingWindows** attribute is set to **true**. If so, the view manager leaves the parent view enabled but behind the floating window view.

You can add customized settings for each of your views to this section of the configuration file using the **any** attribute. For more information, see “Adding Extensible Configuration Schema,” later in this chapter.

Creating the <sharedTransitions> Section

An application may have views that are a common transition point for nodes within a single graph navigator or for all graph navigators. This section is relevant only if you have the <navigationGraph> section defined. If the views have common transition points, they are global transition points. The following code is an example of a global transition point.

```
<sharedTransitions>
  <sharedTransition navigateValue="storehelp" navigateTo='storehelp' />
</sharedTransitions>
```

For information about how to add shared transitions a specific graph navigator, see “Adding Shared Transitions to the <navigationGraph> Section” later in this chapter.

Creating the <navigationGraph> Section

The <navigationGraph> section defines the details of the graph navigator. It also declares the objects to be used for navigation management. The <navigationGraph> section contains a number of attributes as described in Table 3.7.

Table 3.7: Configuration Attributes of the <navigationGraph> Section

Attribute Name	Description
startView	The name of the first view to be shown within the process. Note that this is not necessarily the first view that the user sees, but the first view referenced in the graph navigator.
iViewManager	The name of the view manager as defined in the <objectTypes> section.
name	The name of the graph navigator.
state	The name of the state type as defined in the <objectTypes> section.
statePersist	The name of the state persistence mechanism as defined in the <objectTypes> section.
cacheExpirationMode (optional)	The expiration method used to invalidate items in the cache.

(continued)

Table 3.7: Configuration Attributes of the <navigationGraph> Section (*continued*)

Attribute Name	Description
cacheExpirationInterval (optional)	The expiration interval used before invalidating items in the cache. For absolute expiration, the cacheExpirationInterval should be specified as the absolute time of day when the item should expire. For sliding expiration, the cacheExpirationInterval should be specified as the time, in milliseconds, before the item should expire.
runInWizardMode (optional)	Specifies that this is a wizard navigator for Windows-based applications. Used to support wizards with branching capabilities.

The <navigationGraph> section also contains a number of <node> elements, each of which define a node within the graph navigator. The **view** attribute of the node identifies the view as defined in the views section, and you can use the <navigateTo> elements to identify the navigational routes for this view. The <navigateTo> elements have two attributes as described in Table 3.8.

Table 3.8: Configuration Attributes of the <navigateTo> Elements

Attribute Name	Description
navigateValue	The value that is assigned to an action in the code to identify the route to be taken.
view	The view that should be navigated to when this navigateValue is used.

The following code is an example of the <navigationGraph> section for a Web application.

```
<navigationGraph
  startView="Page2"
  iViewManager="WebFormViewManager"
  name="MyNavigationGraph"
  state="State"
  statePersist="SqlServerPersistState"
  cacheExpirationMode="Sliding"
  cacheExpirationInterval="1000">

  <node view="Page2">
    <navigateTo navigateValue="MoreInfo" view="Page3"/>
    <navigateTo navigateValue="DataAdded" view="Page4" />
    <navigateTo navigateValue="fail" view="error" />
  </node>
  <node view="Page3">
    <navigateTo navigateValue="DataAdded" view="Page4" />
    <navigateTo navigateValue="fail" view="error" />
  </node>
</navigationGraph>
```

(*continued*)

(continued)

```

</node>
<node view="Page4">
  <navigateTo navigateValue="Restart" view="Page1" />
  <navigateTo navigateValue="fail" view="error" />
</node>
<node view="Error">
  <navigateTo navigateValue="Page1" view="Page1" />
  <navigateTo navigateValue="Page2" view="Page2" />
  <navigateTo navigateValue="Page3" view="Page3" />
  <navigateTo navigateValue="Page4" view="Page4" />
</node>
</navigationGraph>

```

The following code is an example of the `<navigationGraph>` section for a Windows application.

```

<navigationGraph
  startView="Form2"
  iViewManager="WindowsFormViewManager"
  name="MyNavigationGraph"
  state="State"
  statePersist="SqlServerPersistState" >

  <node view="Form2">
    <navigateTo navigateValue=" MoreInfo" view="Form3"/>
    <navigateTo navigateValue="DataAdded" view="Form4" />
    <navigateTo navigateValue="fail" view="error" />
  </node>
  <node view="Form3">
    <navigateTo navigateValue="DataAdded" view="Form4" />
    <navigateTo navigateValue="fail" view="error" />
  </node>
  <node view="Form4">
    <navigateTo navigateValue="Restart" view="Form1" />
    <navigateTo navigateValue="fail" view="error" />
  </node>
  <node view="Error">
    <navigateTo navigateValue="Form1" view="Form1" />
    <navigateTo navigateValue="Form2" view="Form2" />
    <navigateTo navigateValue="Form3" view="Form3" />
    <navigateTo navigateValue="Form4" view="Form4" />
  </node>
</navigationGraph>

```

The preceding examples show the previously declared views with their navigational routes now defined. Notice that each view has a `<navigateTo>` element with a fail value pointing to the error view. It is good practice to include a global error page that can be reached from each page, and then if the controller detects an irresolvable error, it can safely navigate to your error page.

Adding Shared Transitions to the <navigationGraph> Section

An application may have views that are a common transition point for nodes within a single graph navigator. To add shared transitions to the <navigationGraph> section, you should modify the configuration file, as shown in the following example.

```
<navigationGraph
  iViewManager="WindowsFormViewManager"
  name="Shopping"
  state="State"
  statePersist="SqlServerPersistState"
  startView="cart"
>
<sharedTransitions>
  <sharedTransition navigateValue="shoppinghelp" navigateTo='shoppinghelp' />
</sharedTransitions>
```

In the preceding example, a **navigateValue** of **shoppingHelp** issued from any of the forms would result in navigation to the **shoppinghelp** view. Note that a specific form can also have a specification of a view for **shoppingHelp**; that specification takes precedence over the shared transition specification.

Creating a <uiWizard> Section

The <uiWizard> section is a quick way to create a graph navigator that supports linear navigation. The next, previous, finish, and cancel transitions are provided for all nodes. The name of the wizard navigator and each view that appears in the wizard should be provided in the correct sequence. The following code is an example of a <uiWizard> section of a configuration file.

```
<uiWizard name="CarWizard">
  <sequence view="ClientInfo"/>
  <sequence view="CarInfo"/>
  <sequence view="Confirmation"/>
</uiWizard>
```

The InsurancePurchaseWizard QuickStart demonstrates the wizard functionality for UIP in a Windows Forms application.

Creating the <userControls> Section for a Windows Application

The <userControls> section defines the navigational details of a user control navigator for a Windows application. The <userControls> section contains a number of attributes as described in Table 3.9.

Table 3.9: Configuration Attributes of the <userControls> Section

Attribute Name	Description
startForm	The name of the first form to be shown within the process. Note that this is not necessarily the first view that the user will see, but the first view referenced in the user controls navigator.
iviewManager	The name of the view manager as defined in the <objectTypes> section.
name	The name of the navigator.
state	The name of the state type as defined in the <objectTypes> section.
statePersist	The name of the state persistence mechanism as defined in the <objectTypes> section.
cacheExpirationMode (optional)	The expiration method to use to invalidate items in the cache.
cacheExpirationInterval (optional)	The expiration interval to use before invalidating items in the cache. For absolute expiration, the cacheExpirationInterval should be specified as the absolute time of day when the item should expire. For sliding expiration, the cacheExpirationInterval should be specified as the time, in milliseconds, before the item should expire.

The <userControls> section also contains a number of <form> elements, each of which defines a form within the task as shown in Table 3.10. Each <form> element has a **name** attribute, which defines the view instance.

The <form> element may contain a number of <childView> elements, each of which defines a control on the form that derives from the **WindowsFormControlView** class.

Table 3.10: Configuration Attributes of the <UserControls> <childView> Elements

Attribute Name	Description
name	The name of the instance of the view on the form; for example, a billing or shipping address.
viewName	The name of the view type from the views section of the configuration file.

The following is an example of the `<userControls>` section of a configuration file.

```
<userControls name="Shop"
  startForm="StoreForm"
  iViewManager="WindowsFormViewManager"
  state="State"
  statePersist="SqlServerPersistState">
  <form name="StoreForm">
    <childView name="shoppingCart" viewName="cart"/>
    <childView name="catalog" viewName="browsecatalog"/>
  </form>
</userControls>
```

Adding Additional Functionality

If you have implemented the requirements specified earlier in this chapter, your applications will support the UIP Application Block. However, the block has some additional features that you can make use of, as this section describes.

Adding Extensible Configuration Schema

The UIP Application Block gives you the ability to have view-specific attributes, such as background color, stored with the view configuration information. The block validates configuration information against a supplied XML schema. Because these attributes are application-specific, the UIP schema has no knowledge of them and therefore cannot validate them. Your code is responsible for validating the schema and semantics of these extra attributes.

To add extensible configuration schema, add any custom attributes to the `<views>` section of the configuration file, as shown in the following code example.

```
<view name="customView"
  type="UIProcessQuickstarts_Store.WinUI.CustomView,UIProcessQuickstarts_Store.WinUI
  ,Version=1.0.1.0,Culture=neutral,PublicKeyToken=null"
  controller="StoreController" bgColor="Yellow">

  <Captions>
    <caption control="button1" text="Hello" />
    <caption control="button2" text="Goodbye" />
  </Captions>
</view>
```

Attributes are supported per view, and are interpreted by the view manager. The UIP Application Block passes unrecognized attributes or elements to the view manager and to the view (in the view's **Initialize** method) during execution. You must implement the **Initialize** method to access custom view information. The following example code shows how to access this information programmatically from the **Initialize** method. In this example, you must reference the **System.Xml.XPath** namespace because it uses an **XPathNodeIterator** type.

```

[Visual Basic]
Public Sub Initialize(ByVal args() As Object,ByVal settings As ViewSettings)
MyBase.Initialize(args, settings)

'retrieving a custom attribute
Me.BackColor = Color.FromName(settings.CustomAttributes.Item(0).Value)

'retrieving information from custom elements
Dim iterator as XPathNodeIterator =
settings.Navigator.Select("descendant::Captions/caption")

While (iterator.MoveNext())
    'reference the caption node
    Dim navigator As XPathNavigator = iterator.Current
    'reference controls and set properties using attributes on the current caption
    node
    Dim control as Control =
    FindControlByName(navigator.GetAttribute("control",navigator.NamespaceUR))

    Control.Text = navigator.GetAttribute("text",navigator.NamespaceURI)
End While
End Sub

[C#]
public override void Initialize(object[] args, ViewSettings settings)
{
    base.Initialize (args, settings);

    //retrieving a custom attribute
    this.BackColor = Color.FromName(settings.CustomAttributes.Item(0).Value);

    //retrieving information from custom elements
    XPathNodeIterator iterator =
    settings.Navigator.Select("descendant::Captions/caption");
    while (iterator.MoveNext())
    {
        // reference the caption node
        XPathNavigator navigator = iterator.Current;
        // reference controls and set properties using attributes on the
        // current caption node
        Control control =
        FindControlByName(navigator.GetAttribute("control",navigator.NamespaceURI));
        control.Text = navigator.GetAttribute("text",navigator.NamespaceURI);
    }
}

```

Raising Navigation Events

The UIP Application Block allows your application to listen for a navigate event. Navigate events are raised after the **Navigate** method is called on the controller, but before the actual navigation takes place. This gives you one last opportunity to modify the **NavigateValue** property on the **State** class — or to perform any other function — before the user is taken to the next view.

You need to define the **NavigateEvent** event in the **UIManager** class as shown in the following code example.

```
[Visual Basic]
AddHandler UIManager.NavigateEvent, AddressOf UIManager.NavigateEvent

[C#]
UIManager.NavigateEvent +=
New Microsoft.ApplicationBlocks.UIProcess.UIManager.NavigateEventHandler
(UIManager.NavigateEvent);
```

You also need to define the corresponding event handler, which might look something like this.

```
[Visual Basic]
Private Sub UIManager.NavigateEvent(ByVal sender As Object, ByVal e As
EventArgs)
    'Some logic that is performed on the navigate event
    'You can obtain the state from the NavigateEventArgs: e.State
    'You can change the navigate value like this:
    ' e.State.NavigateValue = someNewNavigateValue
End Sub

[C#]
private void UIManager.NavigateEvent(object sender, EventArgs e)
{
    // Some logic that is performed on the navigate event
    // You can obtain the state from the NavigateEventArgs: e.State
    // You can change the navigate value like this:
    // e.State.NavigateValue = someNewNavigateValue;
}
```

Raising navigation events is demonstrated in the **InsuranceClientManagement QuickStart**.

Responding to Changes in State

The UIP Application Block allows your application to listen for a **StateChanged** event. Your views can subscribe to this event so that they are notified when the state changes, and, if necessary, can update themselves. If you derive from the **State** class, you must ensure you raise this event whenever you change an item of state. For details, see “Creating a Strongly Typed Class” in Appendix A, “Extending the UIP Application Block.”

The following example handles the **StateChanged** event and uses it to display a message box.

```
[Visual Basic]
Private Sub Page2_Load(ByVal sender As Object, ByVal e As System.EventArgs)
    myState = MyController.State
    myState.StateChanged += New State.StateChangedEventHandler(myState_StateChanged)
End Sub

Private Shared Sub myState_StateChanged(ByVal sender As Object, _
                                       ByVal e As StateChangedEventArgs)
    Dim result As DialogResult
    result = MessageBox.Show( _
        "State has changed. Would you like to update your view?", _
        "State Changed", MessageBoxButtons.YesNo)
    If result = DialogResult.Yes Then
        ' Update view
    End If
End Sub

[C#]
private void Page2_Load(object sender, System.EventArgs e)
{
    myState = MyController.State;
    myState.StateChanged+=new State.StateChangedEventHandler(myState_StateChanged);
}

private static void myState_StateChanged(object sender, StateChangedEventArgs e)
{
    DialogResult result;
    result = MessageBox.Show(
        "State has changed. Would you like to update your view?",
        "State Changed", MessageBoxButtons.YesNo);
    if(result == DialogResult.Yes)
    {
        // Update view
    }
}
```

Raising state events is demonstrated in the AdvancedHostDemo QuickStart.

UIP Shutdown

Classes that should be notified when a user interface process is complete need to implement the **IShutdownUIP** interface. The classes that implement the shutdown interface must be registered with the **UIPManager** class. The **IShutdownUIP** interface has one public method called **Shutdown**. For more information about when you would implement this interface, see “IShutdownUIP Interface” in Chapter 2, “Design of the UIP Application Block.”

When the last UIP form closes, UIP detects that the windows have been closed and the registered classes’ **Shutdown** method is invoked. Normally this method is used by the bootstrap form (the form that initiates the user interface process). When the user interface process is started, the bootstrap form visibility is usually set to hidden. When the user interface process is complete, the bootstrap form can be set to visible.

The following example shows how to register a shutdown task and implement the **IShutdownUIP** interface. See the AdvancedHostDemo QuickStart for more information.

```
[Visual Basic]
public Class StartMeUp : System.Windows.Forms.Form, IShutdownUIP
    Private Sub button1_Click(ByVal sender as Object, ByVal e as
System.EventArgs)
        UIPManager.RegisterShutdown(Me)
        Me.Visible = false
        _startingTask = new TestTask
        UIPManager.StartUserControlsTask("demo", _startingTask)
    End Sub

    Public Sub Shutdown() Implements IShutdownUIP.Shutdown
        Me.Visible = true
    End Sub
End Class
```

```
[C#]
public class StartMeUp : System.Windows.Forms.Form, IShutdownUIP
{
    ....
    private void button1_Click(object sender, System.EventArgs e)
    {
        // register this class for shutdown
        UIPManager.RegisterShutdown(this);
        // hide this form
        this.Visible = false;
        _startingTask = new TestTask();
        // start UI process
        UIPManager.StartUserControlsTask("demo", _startingTask);
    }
}
```

(continued)

(continued)

```
#region IShutdownUIP Members
// method invoked when UI process is completed
public void Shutdown()
{
    // makes form visible again
    this.Visible = true;
}
#endregion
}
```

Customizing View Layouts

To lay out controls on your views in a customized manner, you need to create a class that implements the **ILayoutManager** interface and defines the **LayoutControls** method. The **ILayoutManager** interface has one public method called **LayoutControls**. For more information about the **ILayoutManager**, see “ILayoutManager Interface” in Chapter 2, “Design of the UIP Application Block.”

You also need to specify each of your custom layout managers in the **<objectTypes>** section of the configuration file, and specify the layout manager attribute for each view that uses the custom layout manager in the **<views>** section of the configuration file.

When the view is activated, if you have defined a layout manager and have specified the view to use the layout manager in the configuration file, the view manager calls the **LayoutControls** method on your layout manager and places the controls in the correct order. This method positions each control on the parent control that is passed as a parameter to this method (which is the form being activated).

QuickStarts for the UIP Application Block

To help you gain a better understanding of the UIP Application Block, several QuickStarts are included with the block:

- **AdvancedHostDemo QuickStart** — Demonstrates the use of user controls and transitioning between user controls in a Windows Forms application.
- **InsuranceClientManagement QuickStart** — Demonstrates some of the new functionality that has been added to UIP. Specifically, it demonstrates the shared transitions functionality and handling navigation events in a Windows Forms application.
- **InsurancePurchaseWizard QuickStart** — Demonstrates the wizard functionality for UIP in a Windows Forms application.

- NoNavGraph QuickStart — Demonstrates how you can use the open navigator in a Windows Forms application. This style of navigation allows you to make transitions without having to specify a navigator in the configuration file.
- MultiNavGraph QuickStart — Demonstrates how to chain and nest tasks with multiple graph navigators in a Windows Forms application.
- Store QuickStart — Demonstrates multiple aspects of the UIP Application Block for Windows-based applications and Web applications including user controls, random navigation, and shared transitions.

Important: The QuickStarts are intended to aid you in understanding the block; they are not intended to be production-ready code samples, and may not be indicative of the way you should develop your application.

To build the QuickStarts provided with the block, you need to ensure that your system meets the following minimum software requirements in addition to the minimum requirements for working with the UIP Application Block.

- SQL Server 2000 or Microsoft SQL Desktop Engine (MSDE)
- Internet Information Services (IIS) 5.0 or later

Building the QuickStarts is a relatively straightforward process. It consists of the following steps.

► **To build the UIP Application Block QuickStarts**

1. Locate the solution file for the QuickStart you want to build (located in the *<installation location>\ApplicationBlocks\...\QuickStarts\cs\<QuickStartName>* folder).

Note: This discussion refers to files with a .cs file name extension. If you are using the Visual Basic versions of the samples, you should use the .vb file name extension instead.

2. Build the solution.

The following sections discuss each QuickStart in more detail.

AdvancedHostDemo QuickStart

This QuickStart demonstrates the use of user controls and transitioning between user controls in a Windows Forms application. The user interface contains two user controls: the tree view on the left and the tab view on the right. When the user selects an item on the tree view, the information on the tab changes to reflect the information that corresponds to the selected item. When the user clicks the edit button, the user can edit the information in the tab view for that tree selection. This QuickStart also demonstrates handling the **StateChanged** event and demonstrates the extensible configuration schema functionality by adding custom attributes to the **<views>** section of the configuration file.

InsuranceClientManagement QuickStart

This QuickStart demonstrates some of the new functionality that has been added to UIP. Specifically, it demonstrates the shared transitions functionality and handling navigation events in a Windows Forms application. This Windows Forms application consists of a logon page that any user can view and a page that only authenticated users can view. The QuickStart illustrates how you can intercept navigation events to redirect unauthenticated users to the logon page.

InsurancePurchaseWizard QuickStart

This QuickStart demonstrates the wizard functionality for UIP in a Windows Forms application. The application consists of three wizards: InsurancePurchase Wizard, Car Wizard, and Home Wizard. The car wizard and home wizard behave in a straightforward manner with **Next**, **Back**, **Cancel**, and **Finish** transitions, and the sequence of views as defined in the application configuration file in the `<uiWizard>` section. The insurance purchase wizard gives the user more options than the straightforward wizard. Valid transitions are defined using a graph navigator, and the `runInWizardMode` attribute is set to **true**. This QuickStart also demonstrates how you can customize the behavior of views in the wizard by implementing the `IWizardViewTransition` interface.

NoNavGraph QuickStart

This QuickStart demonstrates how you can use the open navigator in a Windows Forms application. This style of navigation allows you to make transitions without having to specify a navigator in the application configuration file and allows you to navigate using view names. However, you must still specify the default values, including valid views, in the configuration file.

MultiNavGraph QuickStart

This QuickStart demonstrates how to link tasks that are defined in two separate graph navigators. It primarily shows how information and state can be passed between the two graph navigators. This application also contains a custom implementation of the `ITask` interface that records task entries into an XML file. Some of the features of the UIP Application Block that this application demonstrates are the **stayOpen** and **openModal** attributes. When a view has the **openModal** attribute set to **true**, the screen opens in a modal fashion and must be closed before the user can continue in other views that were open before it. When a view has the **stayOpen** attribute set to **true**, the screen stays open and visible even after transitions are made from it.

Store QuickStart

This QuickStart demonstrates multiple aspects of the UIP Application Block for Windows-based applications and Web applications, including the graph navigator, the user controls navigator, shared transitions, and preventing random navigation.

The random navigation functionality is only relevant in the Web application.

This application demonstrates how to prevent random navigation by setting the **allowBackButton** attribute to **false** in the application configuration file. When this functionality is disabled, it prevents the user from transitioning back to the previous view by using the back button. It also prevents users from navigating to views by typing the URL for a particular view.

Summary

The UIP Application Block provides you with a flexible infrastructure that you can use to control the user interface process within a Windows or Web application. UIP provides you with a framework for creating views, controlling navigation, and persisting state. This chapter discussed the steps you would perform to use the UIP Application Block with your application. The QuickStarts supplied with the User Interface Process Block give you a useful starting point in understanding how to use the block.

4

UIP Application Block Deployment and Operations

When you use the UIP Application Block with your applications, you need to ensure that the initial deployment of the block is planned and managed, and that subsequent updates are deployed with minimal impact to your existing environment.

This chapter describes how to deploy the UIP Application Block, how to update the block when future versions become available, and security considerations for the block.

Deployment Requirements

Before you can deploy the UIP Application Block on any computer, you need to install the Microsoft® .NET Framework version 1.1. In addition, if you want to use the **SecureSqlServerPersistState** class or the **SqlServerPersistState** class to persist state, you will also need Microsoft SQL Server™. If you are using the block for Web applications, you need to install Internet Information Services (IIS).

Deploying the UIP Application Block

The UIP Application Block is implemented as a single assembly named **Microsoft.ApplicationBlocks.UIProcess.dll**. The block ships with and uses the Data Access Application Block, **Microsoft.ApplicationBlocks.Data.dll**. When you deploy the UIP Application Block, you must also deploy the Data Access Application Block.

Note: The UIP Application Block ships with and has been tested with version 2.0 of the Data Access Application Block. Other versions of the Data Access Application Block are not guaranteed to work with the UIP Application Block, version 2.0.

For more information about the Data Access Application Block, see *Data Access Application Block for .NET*, available on MSDN® at <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/daab-rm.asp>. For more guidance on data access, see *.NET Data Access Architecture Guide*, available on MSDN at <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/daag.asp>.

You must deploy the UIP Application Block, your controller classes, and the Data Access Application Block on the same computer as the user interface components that they orchestrate. In a Web application, this means deploying the blocks and controllers on the Web servers. In a Microsoft® Windows® operating system-based application, you must deploy the blocks and controllers with the Windows Forms application.

Applications that use the UIP Application Block can be deployed in one of two configurations:

- As a private assembly in the application folder hierarchy
- As a shared assembly in any file system location or in the global assembly cache

Deploying the UIP Application Block as a Private Assembly

The UIP Application Block and the Data Access Application Block can be deployed in the application directory structure. This simplifies deployment because you can use the **xcopy** command to install the entire application (including the UIP Application Block and the Data Access Application Block) on the target computer.

If multiple applications on the same computer use the blocks, you can install a copy of the assembly in each application folder hierarchy. This allows each application's copy of the assembly to be updated independently.

Deploying the UIP Application Block as a Shared Assembly

You can deploy shared assemblies in the global assembly cache, which allows all applications on the computer to use the assembly without any further configuration. The global assembly cache also provides support for Component Object Model (COM) interoperability, component services (such as transaction management), and versioning management.

You can use one of the following tools to install an assembly in the global assembly cache:

- An installer utility, such as the Microsoft Windows Installer, version 2.0
- The Global Assembly Cache Tool command line utility (Gacutil.exe)
- The .NET Admin Microsoft Management Console (MMC) snap-in (Mscorcfg.msc)

To install the UIP Application Block and Data Access Application Block in the global assembly cache, you must first generate a strong name key, sign the assemblies, and then install them in the global assembly cache in the correct order.

Once you have installed the assemblies in the global assembly cache, you will need to make changes to your application configuration file. In the configuration file, the `<iViewManager>`, `<state>`, `<controller>`, `<layoutManager>`, and `<stateProvider>` in the `<objectTypes>` section each have a setting called **PublicKeyToken** in the **type** attribute. This is normally set to null, but if your UIP assembly is in the global assembly cache, you must set the **PublicKeyToken** equal to the public key token value associated with the UIP assembly in the global assembly cache.

► **To sign the assemblies and add them to the global assembly cache**

1. Use **Sn.exe** to generate a key for the **Microsoft.ApplicationBlocks.Data** assembly.
2. Edit the **AssemblyInfo.cs** file in the **Microsoft.ApplicationBlocks.Data** assembly to reference the key file.
3. Rebuild the **Microsoft.ApplicationBlocks.Data** assembly.
4. Use **Gacutil.exe** or the .NET Microsoft Management Console (MMC) snap-in to add the **Microsoft.ApplicationBlocks.Data** assembly to the global assembly cache.
5. Use **Sn.exe** to generate a key for the **Microsoft.ApplicationBlocks.UIProcess** assembly.
6. Remove and then add the reference to the **Microsoft.ApplicationBlocks.Data** assembly in the **Microsoft.ApplicationBlocks.UIProcess** assembly.
7. Edit the **AssemblyInfo.cs** file in the **Microsoft.ApplicationBlocks.UIProcess** assembly to reference the key file.
8. Rebuild the **Microsoft.ApplicationBlocks.UIProcess** assembly.
9. Use **Gacutil.exe** or the .NET Microsoft Management Console (MMC) snap-in to add the **Microsoft.ApplicationBlocks.UIProcess** to the global assembly cache.

For more information, see the following:

- “Deploying .NET Framework-based Applications,” available at <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/DALGRoadmap.asp>.
- “Deploying Applications,” from the *.NET Framework Developer’s Guide*, available at <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpcondeployingnetframeworkapplications.asp>.

Updating the UIP Application Block

If an upgraded version of the UIP Application Block becomes available, you will need to determine whether the new version introduces compatibility problems for some of your applications. Install the new version of the block where appropriate. Exactly what procedure you use will depend on whether the assembly is private or in the global assembly cache.

Updating Private Assemblies

If the UIP Application Block assembly has been deployed as a private assembly, you can deploy the upgrade by simply replacing the old version of the dynamic-link library (DLL) with the new one.

Note: You should keep a copy of the old version so that if you encounter any compatibility issues with the new assembly, you can revert to the earlier version.

Updating Shared Assemblies

If multiple applications use the UIP Application Block, install the updated assembly in the global assembly cache. By default, the .NET runtime attempts to load the assembly that has the latest build and revision numbers and the same major and minor revision numbers as the assembly with which the application was built. Therefore, if the major and minor revision numbers have not changed, adding a later version to the global assembly cache should automatically upgrade all applications that refer to the UIP Application Block assembly.

If the major or minor revision numbers have been incremented, or if the new version causes compatibility issues with existing applications, you can override the default version policy. To specify that an application should use a particular version of an assembly, edit the **BindingPolicy** section of the application configuration file (for individual applications) or the machine policy file. Alternatively, you can distribute the new version of the assembly with a publisher policy file to redirect assembly requests to the new version.

Note: For more information about versioning in .NET, see the MSDN article, “Simplifying Deployment and Solving DLL Hell with the .NET Framework,” at <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndotnet/html/dplywithnet.asp>.

See Also

For more information, refer to the following:

- “How the Runtime Locates Assemblies,” from the *.NET Framework Developer’s Guide*, available at <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconhowruntimelocatesassemblies.asp>
- “Redirecting Assembly Versions,” from the *.NET Framework Developer’s Guide*, available at <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconassemblyversionredirection.asp>
- “Specifying an Assembly’s Location,” from the *.NET Framework Developer’s Guide*, available at <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconspecifyingassemblylocation.asp>

Security Considerations

If you deploy applications that use the UIP Application Block, you should be aware of the security considerations that surround block deployment.

SQL Server Security

The UIP Application Block allows you to persist state in a SQL Server database. The connection information is declared in the application configuration file, meaning that you must ensure that the contents of this file do not expose your SQL Server computer to unauthorized users.

SQL Server provides two modes of authentication: Windows authentication mode integrates SQL Server with Microsoft® Windows® operating system security; whereas mixed mode allows both Windows accounts and SQL Server logins to be used. Where possible, you should run your SQL Server computer in Windows authentication mode. This means that no user-specific information needs to be stored in the configuration file because the SQL Server computer will use the Windows information for authentication.

The classes in the UIP Application Block use the Data Access Application Block to access the SQL Server database. For more information about security issues when using the Data Access Application Block, see *Data Access Application Block for .NET*, available at <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/daab-rm.asp>.

Configuration File Security

The application configuration file may contain sensitive information, such as the SQL Server name and the database name. You can secure application configuration files by using Windows NTFS file permissions. In an ASP.NET-based application, the runtime prevents access to the Web.config file, although you can enhance this protection with NTFS permissions.

For more information about securing applications, see “Securing Applications,” from the *.NET Framework Developer’s Guide*, available at <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconsecuringyourapplication.asp>.

Security Permissions

If you are not working in a full trust environment, you will need to grant specific permissions to several of the classes. Table 4.1 shows the UIP classes that require security permissions.

Table 4.1: UIP Security Permissions

Class	Permission Type	Comment
CryptHelper	RegistryPermission	
IsolatedStoragePersistence	IsolatedStorageFilePermission	
SqlServerPersistState	SqlClientPermission	
SecureSqlServerPersistState	SqlClientPermission	
State	SecurityPermissionAttribute(Demand, SerializationFormatter=True)	– demand, Serialization permissions
State	SecurityPermissionAttribute (LinkDemand, Flags=SecurityPermissionFlag. SerializationFormatter)	– immediate caller requires serialization permissions
GenericFactory	ReflectionPermission	

Note: For more information about code access security, see “Security Policy Best Practices,” from the *.NET Framework Developer’s Guide*, available at <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconsecuritypolicybestpractices.asp>.

Security Threats and Countermeasures

The UIP Application Block is designed to provide you with reusable code that meets the security requirements of customers. It is important to identify areas where threats exist, as well as appropriate countermeasures to those threats. You should use the results of your risk analysis to determine whether the countermeasures are necessary in your environment. This section addresses several security threats that could be introduced by the UIP Application Block, but does not comprehensively cover security issues for Web or Windows Forms applications that use the block.

The main asset that needs to be protected as part of the block is the data stored in the **State** object. You may store sensitive information, such as credit card information, that needs to be secured.

Table 4.2 identifies the threats to these assets, along with appropriate countermeasures.

Note: The countermeasures listed in Table 4.2 have not been implemented in the UIP Application Block. You should analyze the risk posed by these threats, and where appropriate, implement the countermeasures to mitigate the risk.

Table 4.2: User Interface Process Block Threats and Countermeasures

Threat: State object is exposed after being persisted

Threat target	The State object
Risk	The State object holds the business logic information and is persisted by the StatePersistenceProvider .
Attack techniques	Depends on the type of StatePersistenceProvider .
Countermeasures	Each StatePersistenceProvider needs to secure the data. The IsolatedStoragePersistState persistence provider includes certain security features, such as requiring the IsolatedStorageFilePermission for certain operations.

Threat: Malicious assemblies are loaded during runtime

Threat target	An application that uses the UIP Application Block
Risk	Because UIP relies heavily on the configuration file, the configuration file can be used as a break-in point for hackers. UIP uses the configuration information to dynamically load assemblies. Additionally, critical information, such as the SQL Server connection string, is stored in the configuration file. This information is at risk because it is in plain text format.
Attack techniques	The attacker deploys malicious assemblies on the client and alters the configuration file to force UIP to load the malicious assemblies. The attacker can then take control of the workflow.
Countermeasures	<ul style="list-style-type: none"> • Use the Configuration Management Application Block (CMAB) to increase the level of protection for the configuration file needs. • Specify another resource, such as the SQL Server computer, to provide the configuration information, instead of the traditional ExecutableAssemblyName.exe.config. • Permit UIP to read “Class” information from the configuration file only, instead of from the entire assembly information. The drawback to this approach is that all assemblies must be known at compile time. • Declare all assemblies that are part of the application and use SecurityAction.FullDemand.

Threat: Malicious assemblies use reflection to create class instances

Threat target	A system that runs the UIP Application Block and holds the user data.
Risk	Malicious code could compromise Factory classes in the block that use reflection, and create an instance of another class.
Attack techniques	<ul style="list-style-type: none">• Luring attacks• Malicious assemblies can acquire an instance of the UIP component class by calling classes in the block that use reflection, and thereby create another class instance.
Countermeasures	Demand ReflectionPermission on all code that uses reflection.

Summary

Once you have developed applications that use the UIP Application Block, you are in a position to deploy them. However, it is important to ensure that you think about deployment considerations during application development. You need to ensure that your environment has the prerequisites to install the block, and then determine how and where you will install the assemblies contained in the block. Any time you deploy new code in your environment, you should think about the security implications of the deployment, and take appropriate measures to minimize any threats that you face. By using the information contained in this chapter, you increase the likelihood that you can successfully deploy and run your applications.

Appendix A

Extending the UIP Application Block

In most cases you can use the UIP Application Block in its default configuration. However, the block is designed to be extensible, and you can customize certain parts of it to meet the needs of your application. You can extend the UIP Application Block, in the following ways:

- Customizing state — You can develop your own state management class derived from the **State** class that has your own custom properties. One use of this is to develop a strongly typed state class.
- Customizing state persistence — You can develop your own implementation of the state persistence interface to persist state wherever and whenever you want.
- Developing custom view implementations — You can implement the **IView** interface to create a base class for your views and the **IViewManager** interface to create a management class to control them.
- Developing custom task implementations — You can implement the **ITask** interface within your application to save and load task information in persistent storage. This enables a user to work on a task, suspend that task, and then later resume the task. Because the information can be stored in a persistent medium, the task can be suspended across computer reboots.

Customizing State

The existing state class is sufficient for most uses of the UIP Application Block. However, there are circumstances when you may need to create a custom state class. These include situations where you want to modify when state is persisted and when you want to create a strongly typed state class.

Creating a Custom State Class

To create and use a custom state class, you need to do the following:

1. Inherit the existing **State** class from within the block, and add any custom properties you require.
2. Create the constructors.
3. Modify the configuration information to use the class.

After you have a state derivation, you can customize it to include whatever functionality you require.

Note: This section describes how to override the **State** class itself and does not show how to extend the functionality of that base class. The examples shown simply call the functionality included in the base class, which is required of any custom derivation.

Inheriting the State Class

To create a custom state class, you must inherit from the existing **State** class in the block. You must also ensure that your class is serializable so that any state persistence implementations that serialize data to binary format for storage will work correctly. The following example code shows how to do this.

```
[Visual Basic]
Imports Microsoft.ApplicationBlocks.UIProcess
<Serializable(> Public Class MyState
    Inherits State
    . . .
End Class
```

```
[C#]
using Microsoft.ApplicationBlocks.UIProcess;
[Serializable] public class MyState : State
```

Creating the Constructors

Your custom state class must contain the constructor that is used by the block factories for creating the custom state object. You can achieve this by calling the code in the base **State** class, as shown in the following code example.

```
[Visual Basic]
Public Sub New(statePersistenceProvider As IStatePersistence)
    MyBase.New(statePersistenceProvider)
End Sub
```

```
[C#]
public MyState(IStatePersistence statePersistenceProvider)
    :base(statePersistenceProvider)
{
}
}
```

The class must also contain a constructor for the serialization code to use when deserializing state from the persisted store. Again, you can call the matching base class constructor, as shown in the following code example.

```
[Visual Basic]
<SecurityPermissionAttribute(SecurityAction.Demand, _
    SerializationFormatter := True), _
    SecurityPermissionAttribute(SecurityAction.LinkDemand, _
    Flags := SecurityPermissionFlag.SerializationFormatter)> _
    Protected Sub New(si As SerializationInfo, context As StreamingContext)
        MyBase.New(si, context)
    End Sub

[C#]
[SecurityPermissionAttribute(SecurityAction.Demand, SerializationFormatter = true)]
[SecurityPermissionAttribute(SecurityAction.LinkDemand,
    Flags=SecurityPermissionFlag.SerializationFormatter)]
protected MyState(SerializationInfo si, StreamingContext context)
    : base(si, context)
{
}
```

Finally, the class must contain the standard constructors, which once again can call the matching base class constructors, as shown in the following code example.

```
[Visual Basic]
Public Sub New()
    MyBase.New()
End Sub

Public Sub New(taskId As Guid)
    MyBase.New(taskId)
End Sub

Public Sub New(taskId As Guid, navGraph As String)
    MyBase.New(taskId, navGraph)
End Sub

Public Sub New(taskId As Guid, navGraph As String, currentView As String)
    MyBase.New(taskId, navGraph, currentView)
End Sub

Public Sub New(taskId As Guid, navigationGraph As String, currentView As String, _
    navigateValue As String, statePersistence As IStatePersistence)
    MyBase.New(taskId, navigationGraph, currentView, navigateValue, _
    statePersistence)
End Sub
```

(continued)

(continued)

```
[C#]
public MyState() : base(){}

public MyState(Guid taskId) : base(taskId){}

public MyState(Guid taskId, string navGraph) : base(taskId, navGraph){}

public MyState(Guid taskId, string navGraph, string currentView) : base(taskId,
    navGraph, currentView){}

public MyState(Guid taskId, string navigationGraph, string currentView, string
    navigateValue, IStatePersistence statePersistence) : base(taskId,
    navigationGraph, currentView, navigateValue, statePersistence){}
```

Modifying Configuration Information to Use the Custom State Class

After you create your new state class, you must ensure that you modify the configuration information to use this class. To do this, modify the `<state>` element in the `<objectTypes>` section and the `state` attribute of the `<navigationGraph>` element in the configuration file, as shown in the following code example.

```
<state
  name="MyState"
  type="MyNamespace.MyState, MyNamespace/>

<navigationGraph
  iViewManager="WebFormViewManager"
  name="MyGraph"
  state="MyState"
  statePersist="MyStatePersistence"
  cacheExpirationMode="Sliding"
  cacheExpirationInterval="1000"
  startView="View1">
  . . .
</navigationGraph>
```

Creating a Strongly Typed Class

One reason for creating a custom state class is so that you can use a strongly typed class. You can create a strongly typed class by inheriting from the existing state class in the block, and then modifying it to use strongly typed properties.

Adding Strongly Typed Properties

To enable strong typing of your custom state class, you can add strongly typed properties that store state information specific to your application. When setting any properties, you should call the **StateChanged** event to notify any event listeners that a state item has changed.

The following example code shows how to include a custom string property and custom structure property as part of the **State** object.

```
[Visual Basic]
Private _userName As String
Public Property UserName() As String
    Get
        Return _userName
    End Get
    Set(ByVal Value As String)
        _userName = value
        RaiseEvent StateChanged(Me, New StateChangedEventArgs("UserName"))
    End Set
End Property

Private _address As AddressStruct
Public Property Address() As AddressStruct
    Get
        Return _address
    End Get
    Set(ByVal Value As AddressStruct)
        _address = value
        RaiseEvent StateChanged(Me, New StateChangedEventArgs("Address"))
    End Set
End Property
```

```
[C#]
private string _userName;
public string UserName
{
    get{return _userName;}
    set
    {
        _userName = value;
        if (null != StateChanged)
        {
            StateChanged(this, new StateChangedEventArgs("UserName"));
        }
    }
}

private AddressStruct _address;
public AddressStruct Address
{
    get{return _address;}
    set
    {
        _address=value;
        if ( null != StateChanged )
        {
            StateChanged(this, new StateChangedEventArgs("Address"));
        }
    }
}
```

Declaring the Events

Because your custom state class calls the **StateChanged** event, it must include declarations of the **StateChangedEventHandler**. This handler allows views to be notified if the internal state changes. The following example code shows how to declare this.

```
[Visual Basic]
Shadows Delegate Sub StateChangedEventHandler(ByVal sender As Object, _
                                              ByVal e As StateChangedEventArgs)
Public Shadows Event StateChanged As StateChangedEventHandler

[C#]
public new delegate void StateChangedEventHandler(object sender,
                                              StateChangedEventArgs e);
public new event StateChangedEventHandler StateChanged;
```

Creating the Constructors

Your class constructors must contain code that builds the custom state, in addition to calling the base constructor. The following example code shows how to modify the constructors to initialize your custom properties.

```
[Visual Basic]
Public Sub New()
    MyBase.New()
    ' Initialize custom properties
    _userName = Nothing
    _address = New AddressStruct(0, Nothing, Nothing, Nothing)
End Sub

Public Sub New(taskId As Guid)
    MyBase.New(taskId)
    ' Initialize custom properties
    _userName = Nothing
    _address = New AddressStruct(0, Nothing, Nothing, Nothing)
End Sub

Public Sub New(taskId As Guid, navGraph As String)
    MyBase.New(taskId, navGraph)
    ' Initialize custom properties
    _userName = Nothing
    _address = New AddressStruct(0, Nothing, Nothing, Nothing)
End Sub

Public Sub New(taskId As Guid, navGraph As String, currentView As String)
    MyBase.New(taskId, navGraph, currentView)
    ' Initialize custom properties
    _userName = Nothing
    _address = New AddressStruct(0, Nothing, Nothing, Nothing)
End Sub
```

(continued)

(continued)

```
Public Sub New(taskId As Guid, navigationGraph As String, currentView As String, _
    navigateValue As String, statePersistence As IStatePersistence)
    MyBase.New(taskId, navigationGraph, currentView, navigateValue, _
        statePersistence)
    ' Initialize custom properties
    _userName = Nothing
    _address = New AddressStruct(0, Nothing, Nothing, Nothing)
End Sub

[C#]
public MyState() : base()
{
    // Initialize custom properties
    _userName = null;
    _address = new AddressStruct(0, null, null, null);
}

public MyState(Guid taskId) : base(taskId)
{
    // Initialize custom properties
    _userName = null;
    _address = new AddressStruct(0, null, null, null);
}

public MyState(Guid taskId, string navGraph) : base(taskId, navGraph)
{
    // Initialize custom properties
    _userName = null;
    _address = new AddressStruct(0, null, null, null);
}

public MyState(Guid taskId, string navGraph, string currentView) : base(taskId,
    navGraph, currentView)
{
    // Initialize custom properties
    _userName = null;
    _address = new AddressStruct(0, null, null, null);
}

public MyState(Guid taskId, string navigationGraph, string currentView, string
    navigateValue, IStatePersistence statePersistence) : base(taskId,
    navigationGraph, currentView, navigateValue, statePersistence){}
{
    // Initialize custom properties
    _userName = null;
    _address = new AddressStruct(0, null, null, null);
}
```

The constructor that is used by the serialization code to deserialize state from the persisted store must also create your custom properties. The following example code shows how to store the deserialized data in an instance of the custom state object.

```
[Visual Basic]
<SecurityPermissionAttribute(SecurityAction.Demand, _
    SerializationFormatter:=True), _
    SecurityPermissionAttribute(SecurityAction.LinkDemand, _
    Flags:=SecurityPermissionFlag.SerializationFormatter)> _
    Protected Sub New(si As SerializationInfo, context As StreamingContext)
        MyBase.New(si, context)
        ' Deserialize custom properties
        _userName = si.GetString(NameUserName)
        _address = CType(si.GetValue(NameAddressStruct, GetType(AddressStruct)), _
            AddressStruct)
    End Sub

[C#]
[SecurityPermissionAttribute(SecurityAction.Demand,SerializationFormatter =
true)]
[SecurityPermissionAttribute(SecurityAction.LinkDemand,
Flags=SecurityPermissionFlag.SerializationFormatter)]
protected MyState(SerializationInfo si, StreamingContext context)
    : base(si, context)
{
    // Deserialize custom properties
    _userName = si.GetString(NameUserName);
    _address =
        (AddressStruct)si.GetValue(NameAddressStruct, typeof(AddressStruct));
}
```

Implementing ISerializable

To ensure that your custom properties are serialized by the state persistence mechanism, you must also modify the **ISerializable.GetObjectData** method. The following example code shows how to modify this method.

```
[Visual Basic]
<SecurityPermissionAttribute(SecurityAction.Demand, _
    SerializationFormatter:=True), _
    SecurityPermissionAttribute(SecurityAction.LinkDemand, _
    Flags:=SecurityPermissionFlag.SerializationFormatter)> _
    Public Overrides Sub GetObjectData(info As SerializationInfo, _
        context As StreamingContext)
        MyBase.GetObjectData(info, context)

        ' Add custom properties
        info.AddValue(NameUserName, _userName)
        info.AddValue(NameAddressStruct, _address, GetType(AddressStruct))
    End Sub
```

(continued)

(continued)

```
[C#]
[SecurityPermissionAttribute(SecurityAction.Demand,SerializationFormatter = true)]
[SecurityPermissionAttribute(SecurityAction.LinkDemand,
    Flags=SecurityPermissionFlag.SerializationFormatter)]
public override void GetObjectData(SerializationInfo info,
    StreamingContext context)
{
    base.GetObjectData(info, context);

    // Add custom properties
    info.AddValue(NameUserName, _userName);
    info.AddValue(NameAddressStruct, _address, typeof(AddressStruct));
}
```

Customizing State Persistence

Another way of extending the capability of the UIP Application Block is to modify how state is persisted. In Web applications, scalability is generally an important requirement, and by optimizing your state persistence mechanism, you can increase the scalability of the overall application. Applications based on the Microsoft® Windows® operating system generally require high performance, and careful planning of your state persistence provider can help meet this requirement.

You can customize state persistence in one of two ways:

- Create and use a custom state persistence provider.
- Modify when state is persisted.

Creating a Custom State Persistence Provider

You may decide against using the **IStatePersistence** implementations supplied in the block, and instead create your own provider. You might do this so that you can persist your state to a different type of database, a file, or any other type of data storage. For more information about caching data, see the “Caching Architecture Guide for .NET Framework Applications” at <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/cachingarch.asp>.

Note: You cannot easily persist state using the .NET Framework **XmlSerializer** class. This is because the **XmlSerializer** treats classes that implement **IDictionary** in a special way. It is possible to wrap the **State** class using a **Hashtable** class, which will hide the **IDictionary** implementation from the **XmlSerializer** class. However, this can inhibit performance and is not generally recommended.

To create your own custom state persistence provider, you need to create a class that inherits from the **IStatePersistence** interface, as shown in the following code example.

```
[Visual Basic]
Public Class MyStatePersistence
    Implements IStatePersistence
End Class

[C#]
using Microsoft.ApplicationBlocks.UIProcess;

public class MyStatePersistence : IStatePersistence
```

The interface defines three classes: **Init**, **Load**, and **Save**. Your custom class must override these three methods.

The following code example shows how to override the **Init** method to pass user information to a custom database state persistence provider.

```
[Visual Basic]
Public Sub Init(ByVal statePersistenceParameters As _
    Specialized.NameValueCollection) _
    Implements IStatePersistence.Init
    connectionInfo = statePersistenceParameters(conInfo)
End Sub

[C#]
public void Init(NameValueCollection statePersistenceParameters)
{
    connectionInfo = statePersistenceParameters[conInfo];
}
```

The following code snippet shows how to override the **Load** method.

```
[Visual Basic]
Public Function Load(ByVal taskId As System.Guid) As State _
    Implements IStatePersistence.Load
    ' Database specific code to access the stored state
    Return storedState
End Function

[C#]
public State Load(Guid taskId)
{
    // Database specific code to access the stored state
    return storedState;
}
```

The following code snippet shows how to override the **Save** method.

```
[Visual Basic]
Public Sub Save(ByVal state As State) Implements IPersistence.Save
    ' Database specific code to save the passed state
End Sub

[C#]
public void Save(State state)
{
    // Database specific code to save the passed state
}
```

Modifying Configuration Information to Use the Custom State Persistence Class

After you implement your new state persistence provider, you must ensure that you modify the configuration information to use this class. To do this, you must modify the **<stateProvider>** element in the **<objectTypes>** section and the **statePersist** attribute of the **<navigationGraph>** element in the configuration file, as shown in the following example code.

```
<stateProvider
  name="MyStatePersistence"
  type="MyNamespace.MyStatePersistence, MyNamespace"/>

<navigationGraph
  iViewManager="WebFormViewManager"
  name="MyGraph"
  state="State"
  statePersist="MyStatePersistence"
  cacheExpirationMode="Sliding"
  cacheExpirationInterval="1000"
  startView="View1">
  .
  .
  .
</navigationGraph>
```

Customizing When State Is Persisted

By default, the state in a process is persisted to the main store each time a new view is displayed. While this is the safest option and ensures that no state is ever lost, it can result in performance issues due to the time taken to serialize the data and store it elsewhere.

You may decide against using the **State** object at all for application data. If you are using high-scale online transaction processing that has been designed to use a stateless middle tier, storing application data in the **State** object will conflict with the original middle tier design. Alternatively, you may choose to cache state in memory for the lifetime of the application, and only persist it when the process is about to end.

However, if you want to persist state more frequently and still improve performance, you can customize the block to ensure that state is only persisted if it has changed since last stored. You can do this by implementing an **IsDirty** property in a custom state class. Then, you must create a custom state persistence provider by implementing the **IStatePersistence** interface. After you develop these custom classes, you must modify them to implement and use an **IsDirty** check.

Note: For information about creating a custom state class and a custom state persistence provider, see “Creating a Custom State Class” and “Creating a Custom State Persistence Provider” earlier in this Appendix.

Adding the IsDirty Property to the Custom State Class

The following code example shows how to add the **IsDirty** property to your custom state class.

```
[Visual Basic]
Private _isDirty As Boolean = False

Public Property IsDirty() As Boolean
    Get
        Return _isDirty
    End Get
    Set(ByVal Value As Boolean)
        _isDirty = value
    End Set
End Property
```

```
[C#]
private bool _isDirty = false;

public bool IsDirty
{
    get{return _isDirty;}
    set{_isDirty = value;}
}
```

Initializing the IsDirty Property

You should initialize the **IsDirty** property to **false** on creation. This is because the **State** object gets recreated multiple times during the lifetime of an application, and the flag should only be **true** when state needs to be saved. However, the **IsDirty** flag should be set to **true** when the task starts so that the initial values can be persisted.

To do this, you should set the flag to **true** in the constructor used by the custom state persistence provider, as shown in the following example code.

```
[Visual Basic]
Public Sub New(statePersistenceProvider As IStatePersistence)
    MyBase.New(statePersistenceProvider)

    ' Initialize _isDirty to ensure that initial values are persisted
    _isDirty = True
End Sub

[C#]
public MyState(IStatePersistence statePersistenceProvider)
    :base(statePersistenceProvider)
{
    // Initialize _isDirty to ensure that initial values are persisted
    _isDirty = true;
}
```

Setting the IsDirty Property to True

Each time you change a member of your custom state class, you need to set the **IsDirty** property to **true**. This may include the contents of the internal hash table, the custom properties within your custom state class, or both.

To set the **IsDirty** flag in your custom properties, you should add the following code to the **Set** accessor of each property.

```
[Visual Basic]
IsDirty = True

[C#]
IsDirty = true;
```

To set the **IsDirty** flag when the contents of the hash table change, you must override the accessors for the hash table from the base state class. These include the **Add**, **Remove**, and **this[]** methods.

The following code shows how to override these methods.

```
[Visual Basic]
Public Overrides Sub Add(key As String, value As Object)
    MyBase.Add(key, value)
    IsDirty = True
End Sub

Public Overrides Sub Remove(key As String)
    MyBase.Remove(key)
    IsDirty = True
End Sub
```

(continued)

(continued)

```
Default Public Overrides Property Item(key As String) As Object
    Get
        Return MyBase.Item(key)
    End Get
    Set(ByVal Value As Object)
        MyBase.Item(key) = Value
        IsDirty = True
    End Set
End Property

[C#]
public override void Add(string key, object value)
{
    base.Add(key, value);
    IsDirty = true;
}

public override void Remove(string key)
{
    base.Remove(key);
    IsDirty = true;
}

public override object this[string key]
{
    get{ return base[key]; }
    set
    {
        base[key] = value;
        IsDirty = true;
    }
}
```

Note: If you decide not to persist the changes made to the hash table members, you must still persist the creation of the members or the original values will be lost. In this case, you must ensure that you use the **Add** method (rather than the **this[]** indexer) to add members to the hash table, and that you set the **IsDirty** flag in the **Add** method.

If you are particularly concerned with performance, you can check whether the incoming state value is different from the current value before setting the **IsDirty** property. This will ensure that the flag is only set when the state has actually changed, as opposed to when it is passed to the state class.

Using IsDirty in a Custom IStatePersistence Class

After you define and set the **IsDirty** property in the custom state class, you are able to use it in a custom implementation of the **IStatePersistence** interface. You must add code to the **Save** method that persists the data if the custom state object is dirty and that resets the flag before exiting, as shown in the following code example.

```
[Visual Basic]
<SqlClientPermission(System.Security.Permissions.SecurityAction.Demand)> _
Public Sub Save(ByVal state As State) Implements IStatePersistence.Save
    ' cast base state object into custom state object
    Dim myState As MyState = CType(state, MyState)
    If myState.IsDirty = True Then
        ' code to serialize and persist data
        ' . . .
        myState.IsDirty = False
    End If
End Sub

[C#]
[SqlClientPermission(System.Security.Permissions.SecurityAction.Demand)]
public void Save(State state)
{
    // cast base state object into custom state object
    StateChangedCheck myState = (StateChangedCheck)state;
    if (myState.IsDirty == true)
    {
        // code to serialize and persist data
        // . . .
        myState.IsDirty = false;
    }
}
```

Customizing Views and View Management

The UIP Application Block contains view and view management implementations for use in Windows-based applications and in ASP.NET Web applications. If you want to use other types of user interfaces for your views, you need to customize the view system. However, creating your own view classes involves more than just implementing the **IView** interface in a class and deriving views from that class. You must also implement your own view management class (using **IViewManager**), and write custom code for the creation and manipulation of your views.

Creating a custom view and view management system involves the following steps:

1. Create a custom **IView** implementation. This may derive from whatever other classes you require.
2. Create the views to be used in your application using the base class you created.
3. Create a custom **IViewManager** implementation.
4. Add code to execute the creation of the views, the retrieval of configuration information for the views, and the management of views.

Note: The .NET Framework allows you to derive a class from many interfaces, but only one class. If your views must derive from a standard class, they cannot also derive from **WebFormView**, **WindowsFormView**, or **WindowsFormControlView**. In this situation, you must implement the **IView** interface in a base view, and then derive your individual views from it.

Creating a Custom IView Implementation

You should only create a custom **IView** implementation if the versions supplied in the UIP Application Block do not meet your requirements. To create a custom **IView** implementation, you need to:

1. Reference the **IView** interface.
2. Code the custom **View** class.

These procedures are discussed in the following subsections.

Referencing the IView Interface

To create a custom view, you must first inherit from the **IView** interface defined in the block, as shown in the following code examples.

```
[Visual Basic]
Imports Microsoft.ApplicationBlocks.UIProcess
Public Class NewBaseView
    Implements IView
```

```
[C#]
using Microsoft.ApplicationBlocks.UIProcess;
public class NewBaseView : IView
```

Coding the Custom View

Your custom view must include code to do the following:

- Declare private member variables to hold the data required by the block code.
- Implement internal properties to allow the block to set the member variables.
- Implement code to initialize and access the view's controller.
- Implement the methods defined in the interface.

The following code examples show the declaration of private member variables to hold data required by the block code.

```
[Visual Basic]
Private _taskId As Guid
Private _navigationGraph As String
Private _viewName As String
```

```
[C#]
private Guid _taskId;
private string _navigationGraph;
private string _viewName;
```

The following code examples implement internal properties to allow the block to set the member variables.

```
[Visual Basic]
Friend WriteOnly Property InternalTaskId() As Guid
    Set
        _taskId = value
    End Set
End Property

Friend WriteOnly Property InternalNavigationGraph() As String
    Set
        _navigationGraph = value
    End Set
End Property

Friend WriteOnly Property InternalViewName() As String
    Set
        _viewName = value
    End Set
End Property
```

```
[C#]
internal Guid InternalTaskId
{
    set { _taskId = value; }
}

internal string InternalNavigationGraph
{
    set{ _navigationGraph = value; }
}

internal string InternalViewName
{
    set{ _viewName = value; }
}
```

The following code examples show how to initialize, cache, and access the controller.

[Visual Basic]

```
Private _controller As ControllerBase

Public ReadOnly Property Controller() As ControllerBase _
    Implements IView.Controller
    Get
        Return _controller
    End Get
End Property

Public Sub myBaseView_Load ([source] As Object, e As EventArgs)
    ' This event must be linked to the Load event of the derived class in
    ' the class constructor
    _controller = UIPManager.InitializeController(Me)
End Sub

' Constructor
Public Sub New()
    AddHandler Me.Load, AddressOf myBaseView_Load
End Sub
```

[C#]

```
private ControllerBase _controller;

public ControllerBase Controller
{
    get { return _controller; }
}

public void myBaseView_Load(object source, EventArgs e)
{
    // This event must be linked to the Load event of the derived class in
    // the class constructor
    _controller = UIPManager.InitializeController( this );
}

// Constructor
public MyBaseView()
{
    this.Load += new EventHandler(myBaseView_Load);
}
```

Note: If you are developing views based on Windows Forms, you must verify that the form is not in design mode before initializing the controller to avoid throwing a design time exception.

The following code examples show how to implement the methods defined in the **IView** interface.

```
[Visual Basic]
ReadOnly Property ViewName() As String Implements IView.ViewName
    Get
        Return _viewName
    End Get
End Property

ReadOnly Property TaskId() As Guid Implements IView.TaskId
    Get
        Return _taskId
    End Get
End Property

ReadOnly Property NavigationGraph() As String Implements IView.NavigationGraph
    Get
        Return _navigationGraph
    End Get
End Property

[C#]
string IView.ViewName
{
    get{ return _viewName; }
}

Guid IView.TaskId
{
    get{ return _taskId; }
}

string IView.NavigationGraph
{
    get{ return _navigationGraph; }
}
```

Create the Application Views

After you implement your custom view class, you can derive the application views from it, reference them in the application configuration file, and use them in the same way that you have used derivations of the **WebFormView**, **WindowsFormView**, or **WindowsFormControlView** classes.

Creating a Custom IViewManager Implementation

When implementing custom views, you must also implement a custom view manager and other view creation and management code. You should only create your own custom **IViewManager** implementation if the versions supplied in the UIP Application Block (**WebFormViewManager**, **WindowsFormViewManager**, and **WizardViewManager**) do not meet your requirements. To create a custom **IViewManager** implementation, you must do the following:

1. Reference the **IViewManager** interface.
2. Code the custom **View Manager** class.

Each of these procedures is discussed in the following subsections.

Referencing the IViewManager Interface

To create a custom view, you must first inherit from the **IViewManager** interface defined in the block. The following example code shows how to do this.

```
[Visual Basic]
Imports Microsoft.ApplicationBlocks.UIProcess
Public Class NewViewManager
    Implements IViewManager
```

```
[C#]
using Microsoft.ApplicationBlocks.UIProcess;
public class NewViewManager : IViewManager
```

Coding the Custom View Manager

Your custom code must implement each of the methods defined in the interface. For the methods that you do not use in your code, simply return **true**.

The following code shows an example of a custom Windows Form view manager, including the declarations for the member variables used. The following example code shows how to keep track of which forms are already active, enables reuse of that form if appropriate, and closes the previous form when a new one is loaded.

```
[Visual Basic]
'Stores active forms
Private Shared ActiveForms As New Hashtable()
'Stores active views
Private Shared ActiveViews As New Hashtable()
'Stores active views
Private Shared Properties As New Hashtable()
```

(continued)

(continued)

```
Public Sub StoreProperty(taskId As Guid, name As String, value As Object) _
    Implements IViewManager.StoreProperty
    If Properties(taskId) Is Nothing Then
        Properties(taskId) = New Hashtable()
    End If
    CType(Properties(taskId), Hashtable)(name) = value
End Sub

Public Function IsRequestCurrentView(view As IView, stateViewName As String) _
    As Boolean _
    Implements IViewManager.IsRequestCurrentView
    ' Not needed for Windows Forms
    Return True
End Function

Public Sub ActivateView(previousView As String, taskId As Guid, _
    navigationGraph As String, view As String) _
    Implements IViewManager.ActivateView
    If ActiveForms(taskId) Is Nothing Then
        ActiveForms(taskId) = New Hashtable
        ActiveViews(taskId) = New Hashtable
    End If

    Dim taskActiveForms As Hashtable = CType(ActiveForms(taskId), Hashtable)
    Dim taskActiveViews As Hashtable = CType(ActiveViews(taskId), Hashtable)
    Dim newBaseView As NewBaseView = CType(taskActiveForms(view), NewBaseView)

    If Not (newBaseView Is Nothing) Then
        'Use the existing instance
        newBaseView.Activate()
    Else
        ' Locate form type info from config file
        ' . . .
        ' Create a new instance of the form
        ' . . .
        newBaseView.InternalTaskId = taskId
        newBaseView.InternalNavigationGraph = navigationGraph
        newBaseView.InternalViewName = view

        taskActiveForms(view) = newBaseView
        taskActiveViews(newBaseView) = view

        AddHandler newBaseView.Activated, AddressOf Form_Activated
        AddHandler newBaseView.Closed, AddressOf Form_Closed

        newBaseView.Show()
    End If
```

(continued)

(continued)

```
If Not (previousView Is Nothing) Then
    ' Get the previousView form details from config file
    ' . . .
    If Not (previousForm Is Nothing) Then
        previousForm.Close()
    End If
End If
End Sub

Public Function GetCurrentTasks() As Guid() _
    Implements IViewManager.GetCurrentTasks

    Dim tasks As New ArrayList
    Dim key As Guid
    For Each key In ActiveViews.Keys
        tasks.Add(key)
    Next key

    Return CType(tasks.ToArray(GetType(Guid)), Guid())
End Function

[C#]
//Stores active forms
private static Hashtable ActiveForms = new Hashtable();
//Stores active views
private static Hashtable ActiveViews = new Hashtable();
//Stores active views
private static Hashtable Properties = new Hashtable();

public void StoreProperty(Guid taskId, string name, object value)
{
    if( Properties [taskId] == null )
        Properties [taskId] = new Hashtable();
    ((Hashtable)Properties [taskId])[name] = value;
}

public bool IsRequestCurrentView(IView view, string stateViewName)
{
    // Not needed for Windows Forms
    return true;
}
```

(continued)

(continued)

```

public void ActivateView(string previousView, Guid taskId, string navigationGraph,
                        string view)
{
    if( ActiveForms[ taskId ] == null )
    {
        ActiveForms[ taskId ] = new Hashtable();
        ActiveViews[ taskId ] = new Hashtable();
    }
    Hashtable taskActiveForms = (Hashtable)ActiveForms[ taskId ];
    Hashtable taskActiveViews = (Hashtable)ActiveViews[ taskId ];
    NewBaseView newBaseView = (NewBaseView)taskActiveForms[ view ];

    if (newBaseView != null)
    {
        //Use the existing instance
        newBaseView.Activate();
    }
    else
    {
        //Locate form type info from config file
        . . .
        //Create a new instance of the form
        . . .
        newBaseView.InternalTaskId = taskId;
        newBaseView.InternalNavigationGraph = navigationGraph;
        newBaseView.InternalViewName = view;

        taskActiveForms[ view ] = newBaseView;
        taskActiveViews[ newBaseView ] = view;

        newBaseView.Activated += new EventHandler(Form_Activated);
        newBaseView.Closed += new EventHandler(Form_Closed);

        newBaseView.Show();
    }

    if( previousView != null )
    {
        // Get the previousView form details from config file
        // . . .
        NewBaseView previousForm =
            (NewBaseView)taskActiveForms[previousView];

        if (previousForm != null )
        {
            previousForm.Close();
        }
    }
}

```

(continued)

(continued)

```
public Guid[] GetCurrentTasks()
{
    ArrayList tasks = new ArrayList();
    foreach( Guid key in ActiveViews.Keys )
    {
        tasks.Add( key );
    }
    return (Guid[]) tasks.ToArray( typeof(Guid) );
}
```

To keep track of which forms are loaded, you must add event handlers for the **Activated** and **Closed** event of each form.

```
[Visual Basic]
Private Sub Form_Activated([source] As Object, e As EventArgs)
    Dim newBaseView As NewBaseView = CType([source], NewBaseView)
    Dim state As State = newBaseView.Controller.State

    Dim taskActiveViews As Hashtable = CType(ActiveViews(state.TaskId), Hashtable)

    Dim currentView As String = CStr(taskActiveViews([source]))

    If Not (currentView Is Nothing) Then
        state.CurrentView = currentView
    End If
End Sub

Private Sub Form_Closed([source] As Object, e As EventArgs)
    Dim newBaseView As IView = CType([source], IView)

    Dim taskActiveViews As Hashtable = CType(ActiveViews(newBaseView.TaskId), _
                                             Hashtable)
    Dim taskActiveForms As Hashtable = CType(ActiveForms(newBaseView.TaskId), _
                                             Hashtable)

    Dim currentView As String = CStr(taskActiveViews([source]))
    If Not (currentView Is Nothing) Then
        taskActiveForms.Remove(currentView)
        taskActiveViews.Remove([source])
    End If
End Sub
```

(continued)

(continued)

```
[C#]
private void Form_Activated( object source, EventArgs e)
{
    NewBaseView newBaseView = (NewBaseView)source;
    State state = newBaseView.Controller.State;

    Hashtable taskActiveViews = (Hashtable)ActiveViews[state.TaskId];

    string currentView = (string)taskActiveViews[source];

    if( currentView != null )
        state.CurrentView = currentView;
}

private void Form_Closed( object source, EventArgs e)
{
    IView newBaseView = (IView)source;

    Hashtable taskActiveViews = (Hashtable)ActiveViews[newBaseView.TaskId];
    Hashtable taskActiveForms = (Hashtable)ActiveForms[newBaseView.TaskId];

    string currentView = (string)taskActiveViews[source];
    if( currentView != null )
    {
        // Unhook event handlers
        // Code omitted for brevity

        taskActiveForms.Remove( currentView );
        taskActiveViews.Remove( source );
    }
}
```

Note: After you implement a custom view manager, you need to update the application configuration file with the type information for that class.

Custom View Management Code

After you create the custom view class, the application views, and the custom view manager, you need to add code to execute the creation of the views, the retrieval of configuration information for the views, and the management of views. You can do this either by writing your own versions of the **UIPConfiguration** classes and the **Factory** classes, or by customizing the code in the UIP Application Block to allow external classes to access the relevant code blocks.

Persisting Task Information Between User Sessions

One of the goals of the UIP Application Block is to provide the ability for users to suspend a session and then be able to restart it from the same point without losing any information. You can do this by creating a class that implements the **ITask** interface within your application. The class is then used to save and load a user/task correlation in persistent storage (for example, a SQL Server database). You can then call the **StartNavigationTask**, **StartOpenNavigationTask**, or **StartUserControlsTask** overloaded methods on the **UIPManager** with the task ID or task object, which the manager can then use to load the existing state for that task.

It is the responsibility of the client application to implement the user/task correlation and look up information, and also to ensure that a user does not try to work on the same task in two instances of the application at the same time. The block does not support multiple instances of the same task and this may result in corruption of state.

Implementing the ITask Interface

To implement the **ITask** interface, you must first create a class that references the UIPv2 Application Block and add **ITask** to the base class list. This is shown in the following code example.

```
[Visual Basic]
Imports System
Imports Microsoft.ApplicationBlocks.UIProcess
Public Class MyTask
    Implements ITask
End Class
```

```
[C#]
using System;
using Microsoft.ApplicationBlocks.UIProcess;
public class MyTask : ITask
```

Your class should include member variables that can be used to store the user information and task ID of a particular task. In the following code example, a user name and a task ID are used.

```
[Visual Basic]
Private _userName As String = ""
Private _taskId As Guid = Guid.Empty
```

```
[C#]
private string _userName = "";
private Guid _taskId = Guid.Empty;
```

Your class should contain constructors that allow for whatever information is required to correlate the user to the specific task. This could include an overload that takes a user information parameter to look up the task ID for that user. Another option is to pass user information and a human readable task identifier that can be used to look up a task ID when a single user may be working on multiple tasks. The following code contains examples of both of these overloads.

```
[Visual Basic]
Public Sub New(ByVal userName As String)
    _userName = userName
    _taskId = MyController.GetUserTaskId(_userName)
End Sub

Public Sub New(ByVal userName As String, ByVal taskName As String)
    _userName = userName
    _taskId = MyController.GetTaskIdFromTaskName(taskName)
End Sub

[C#]
public MyTask(string userName)
{
    _userName = userName;
    _taskId = MyController.GetUserTaskId( _userName );
}

public MyTask(string userName, string taskName)
{
    _userName = userName;
    _taskId = MyController.GetTaskIdFromTaskName( taskName );
}
```

The class must also implement the two methods defined in the **ITask** interface:

- **Create** method — This method is used by the appropriate **StartNavigationTask**, **StartOpenNavigationTask**, or **StartUserControlsTask** method on the UIP manager to write the task information to the persistent storage when a new task is created. The following code is an example of an implementation of this method.

```
[Visual Basic]
Public Sub Create(ByVal taskId As Guid) Implements ITask.Create
    _taskId = taskId
    MyController.WriteTaskInfoToDB(_userName, _taskId)
End Sub

[C#]
public void Create(Guid taskId)
{
    _taskId = taskId;
    MyController.WriteTaskInfoToDB( _userName, _taskId )
}
```


- **Get method** — This method is used by the appropriate **StartNavigationTask**, **StartOpenNavigationTask**, or **StartUserControlsTask** method on the UIP Manager to retrieve the task id of an existing task. The following code shows how to implement this method.

```
[Visual Basic]
Public Function [Get]() As Guid Implements ITask.Get
    Return _taskId
End Function

[C#]
public Guid Get()
{
    return _taskId;
}
```

Starting the Task

Each **StartNavigationTask**, **StartOpenNavigationTask**, or **StartUserControlsTask** method on the UIP manager has an overload that takes the name of the navigator and a task object or a task ID. This is the version that you should use when you need the ability to suspend and restart tasks. The following code is an example of how to use this overload. For more information see the section, “Starting and Resuming Tasks,” in Chapter 3, “Developing Applications with the UIP Application Block.”

```
[Visual Basic]
Private Sub Start_Click(sender As Object, e As System.EventArgs)
    UIPManager.StartNavigationTask("MyNavGraph", New MyTask(userName.Text))
End Sub

[C#]
private void Start_Click(object sender, System.EventArgs e)
{
    UIPManager.StartNaviagationTask("MyNavGraph", new MyTask(userName.Text));
}
```

Appendix B

UIP Application Block Terminology

Many of the concepts and terms used in the UIP Application Block, version 2, and its documentation may be new to you. In addition, the UIP Application Block contains new classes that you can use to control the interaction within your application. This appendix will help you understand these new concepts, terms, and new programming elements.

Configuration File

The UIP Application Block uses application configuration information to determine which object types it should know about, which user interface processes exist for an application, which navigators are used, which shared transitions exist, and which views exist. The user interface process configuration is encapsulated in a section of the application configuration called **<uiConfiguration>**.

Controller

Each application that uses the UIP Application Block contains one or more controllers, which are used as the overall control mechanism between the user and the application. Controllers contain navigation methods that are called by the user interface components that determine the views that will be shown to the user; functional methods that are called by the user interface components; and state methods that store information in the persistent storage device. The controller is a facade to the application's underlying business components, mediating communication between the user interface layer and the business components, while also controlling the user interface workflow. The controller to be used in an application is specified in the application configuration file.

The controller is implemented in a class that inherits from **ControllerBase**, which defines methods to achieve the following:

- Read and write the state of an ongoing task using a **State** object.
- Control navigation by setting navigate values.
- Receive arguments (data) when a task is started or resumed.

ControllerBase

The **ControllerBase** class is a base class included in the UIP Application Block that you can use as the basis for your application-specific controller classes.

Graph Navigator

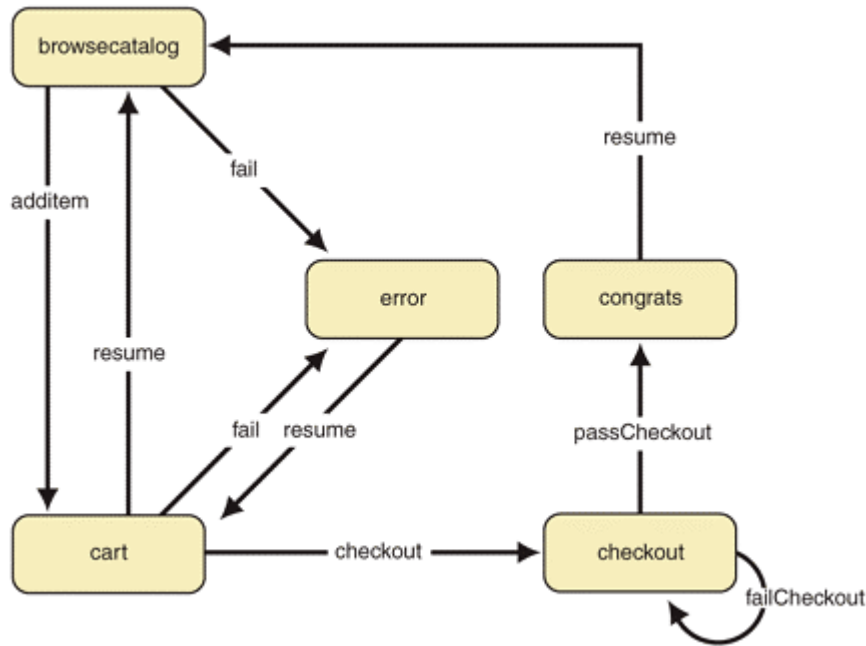
The graph navigator is a type of navigator that uses navigation graphs to specify the user interface control flow from one view to another.

Model-View-Controller (MVC)

Model-View-Controller (MVC) is a design pattern for user interface development that the UIP Application Block encourages you to use. Controllers and views in the UIP Application Block terminology are mapped to controllers and views in the MVC pattern. For more details, see Chapter 1, “Introduction to the UIP Application Block,” and Chapter 2, “Design of the UIP Application Block,” in this guide, or see “Model-View-Controller” at <http://msdn.microsoft.com/practices/type/patterns/enterprise/desmvc/>.

Navigation Graph

Navigation graphs contain a start node, an end node, and zero or more nodes in between. The path between the start and end can be a simple linear path, or it can include branching and circular routes. This can be seen in the sample navigation graph shown in Figure A.1. You define a navigation graph by specifying the next view in the user interface process for each node or view, depending on the different values set by the controller.

**Figure A.1***Sample navigation graph*

When you use the UIP Application Block, you define navigation graphs in your XML configuration file. Each graph contains a node element for each view, and the node element contains `<navigateTo>` child elements, which define the different paths that can be taken from that view. The following code shows the XML required to follow the paths shown in Figure A-1.

```

<navigationGraph
  iViewManager="WinFormViewManager"
  name="Shopping"
  state="State"
  statePersist="SqlServerPersistState"
  startView="cart">
  <node view="cart">
    <navigateTo navigateValue="resume" view="browsecatalog" />
    <navigateTo navigateValue="checkout" view="checkout" />
    <navigateTo navigateValue="fail" view="error" />
  </node>
  <node view="browsecatalog">
    <navigateTo navigateValue="addItem" view="cart"/>
    <navigateTo navigateValue="fail" view="error" />
  </node>
</navigationGraph>

```

(continued)

(continued)

```
<node view="error">
  <navigateTo navigateValue="resume" view="cart" />
</node>
<node view="checkout">
  <navigateTo navigateValue="passCheckout" view="congrats" />
  <navigateTo navigateValue="failCheckout" view="checkout" />
</node>
<node view="congrats">
  <navigateTo navigateValue="resume" view="browsecatalog" />
</node>
</navigationGraph>
```

Navigator

The navigator coordinates the interactions of views and controllers by creating view managers and using them to execute code specific to the application type. UIP provides four types of navigators: graph navigator, open navigator, user controls navigator, and wizard navigator. For more details, see Chapter 2, “Design of the UIP Application Block.”

Open Navigator

The open navigator is a type of navigator that does not require you to specify allowed transitions within the configuration file. Instead, the transitions are referred to in the code of the application.

Shared Transitions

Shared transitions are common transition points for many or all nodes within a graph navigator or between graph navigators.

State Persistence Provider

A state persistence provider is a component that implements the **IStatePersistence** interface that encapsulates loading and saving state for tasks from a specific state location. A state persistence provider is associated with each user interface process in the XML configuration file. The UIP Application Block ships with the following state persistence providers: Isolated Storage, Secure Isolated Storage, SQL Server, Encrypted SQL Server, Session, and Memory. For more details, see Chapter 2, “Design of the UIP Application Block.” You can also create your own classes for other storage mediums. For more details, see Appendix A, “Extending the UIP Application Block.”

State Type

All of the controllers and views interacting in a task share the task state. This state may be specified as a particular type, so that the views and controllers can benefit from type checking at design time, Microsoft® Visual Studio® IntelliSense® technology, and a stronger contract or agreement across the views and controllers. The UIP Application Block contains a state type that acts as a dictionary, but you can specify your own state type for a process if you require these benefits. State objects can raise events to views in order to notify them of changes so they can update themselves accordingly.

Task

A task is a running instance of a user interface process; for example, a user can start a task for the “Create New Customer” user interface process. A task encapsulates all conversation state between the application and the user and is identified by a **TaskId**.

A user can run multiple tasks simultaneously. For example, one task can be for browsing a catalog and another for checking out.

Task State

Task state is the current state of a task. The state is divided in two parts: the state the application uses (for example, credit card details being added while running the “Add Payment Method” task) and the state that represents where this task is within the user interface process. A controller class can access task state by using the **State** property in its base class. This state is stored in a location specified for a process, and the location determines the lifetime of the state.

Task State Location

The task state location specifies where the task state for the running tasks of a user interface process is stored. This can be any valid information store, but when deciding the task state location for an application, consider such issues as the following:

- The target platform of the user interface
- Whether state needs to persist across sessions
- Whether state needs to persist across application restarts
- Security requirements

The saving and loading of state from different locations is implemented in state persistence providers.

TaskId

TaskId identifies a specific task that is running in the application. Your application can use a **TaskId** to resume a running task or to associate a task with a user.

UIP Manager

The **UIPManager** class is used as an entry point to the UI process. If you use the UIP Application Block, your applications must contain code that calls the UIP Manager to start the task and launch the appropriate navigator.

User

A user is any person who interacts with an application that is executing tasks. If a task requires the interactions of more than one user to complete, the application must associate the task with the appropriate user at the appropriate time.

User Controls Navigator

Windows Forms applications often have user controls that appear as child controls on a form or on another user control. The user controls navigator provides a navigation mechanism that shifts focus from one control and gives the user flexibility to transition between controls.

User Interface Process

A user interface process is a set of user-related activities that will achieve a goal. For example, browsing a catalog, checking out, and registering a new credit card are typical user interface processes in a retail application.

User interface processes consist of the following:

- Name
- Navigator
- State location
- State type

View

Views are the actual user interface components of your application. For an application based on the Microsoft® Windows® operating system or for a device application, these are Windows Forms. For an ASP.NET application, these are Web pages. The UIP Application Block provides an interface (**IView**) that views must implement to interact with the block. It also contains base types for Windows Forms (including forms that contain user controls) and ASP.NET Web pages that encapsulate common implementations of this interface called **WindowsFormView**, **WindowsFormControlView**, and **WebFormView**. In most situations, it is best for Windows-based applications and Web applications to use these classes as base classes from which the actual views are derived. For details, see Chapter 2, “Design of the UIP Application Block.” You can also create your own view classes. For more details, see Appendix A, “Extending the UIP Application Block.”

Note: A view is not necessarily a visual element; for example, a view can be based on speech, using the Microsoft .NET Speech SDK. For more information about speech-enabled applications, see Microsoft Speech Technologies at <http://www.microsoft.com/speech>.

View Activation

View managers activate views and pass control to them. The specifics of how a view is activated depend on the application type; for example, a Web application may use a **Response.Redirect** method, whereas a Windows-based application may make a form visible using a **Show** method. Some application types allow more than one view to be active at a time; for example, in a Windows-based application, many forms may be visible.

View Manager

A view manager is a component that encapsulates all of the work relevant to activating views and associating views with specific tasks. The view manager implements the **IViewManager** interface in the UIP Application Block.

The block ships with view manager implementations for Web pages and Windows Forms (including a view manager especially designed for wizards). For details, see Chapter 2, “Design of the UIP Application Block.” You can also build your own view manager implementations for other types of views. For details, see Appendix A, “Extending the UIP Application Block.”

Wizard Navigator

A wizard navigator is a navigator that provides a navigation mechanism between the forms of a wizard. The wizard navigator provides a way to define the transitions between views by using cancel, back, next, and finish transitions.

patterns & practices

proven practices for predictable results

About Microsoft *patterns & practices*

Microsoft *patterns & practices* guides contain specific recommendations illustrating how to design, build, deploy, and operate architecturally sound solutions to challenging business and technical scenarios. They offer deep technical guidance based on real-world experience that goes far beyond white papers to help enterprise IT professionals, information workers, and developers quickly deliver sound solutions.

IT Professionals, information workers, and developers can choose from four types of *patterns & practices*:

- **Patterns**—Patterns are a consistent way of documenting solutions to commonly occurring problems. Patterns are available that address specific architecture, design, and implementation problems. Each pattern also has an associated GotDotNet Community.
- **Reference Architectures**—Reference Architectures are IT system-level architectures that address the business requirements, LifeCycle requirements, and technical constraints for commonly occurring scenarios. Reference Architectures focus on planning the architecture of IT systems.
- **Reference Building Blocks and IT Services**—References Building Blocks and IT Services are re-usable sub-system designs that address common technical challenges across a wide range of scenarios. Many include tested reference implementations to accelerate development. Reference Building Blocks and IT Services focus on the design and implementation of sub-systems.
- **Lifecycle Practices**—Lifecycle Practices provide guidance for tasks outside the scope of architecture and design such as deployment and operations in a production environment.

Patterns & practices guides are reviewed and approved by Microsoft engineering teams, consultants, Product Support Services, and by partners and customers. *Patterns & practices* guides are:

- **Proven**—They are based on field experience.
- **Authoritative**—They offer the best advice available.
- **Accurate**—They are technically validated and tested.
- **Actionable**—They provide the steps to success.
- **Relevant**—They address real-world problems based on customer scenarios.

Patterns & practices guides are designed to help IT professionals, information workers, and developers:

Reduce project cost

- Exploit the Microsoft engineering efforts to save time and money on your projects.
- Follow the Microsoft recommendations to lower your project risk and achieve predictable outcomes.

Increase confidence in solutions

- Build your solutions on proven Microsoft recommendations so you can have total confidence in your results.
- Rely on thoroughly tested and supported guidance, but production quality recommendations and code, not just samples.

Deliver strategic IT advantage






- Solve your problems today and take advantage of future Microsoft technologies with practical advice.

patterns & practices: Current Titles

October 2003




Title	Link to Online Version	Book
Patterns		
Enterprise Solution Patterns using Microsoft .NET	http://msdn.microsoft.com/practices/type/Patterns/Enterprise/default.asp	
Microsoft Data Patterns	http://msdn.microsoft.com/practices/type/Patterns/Data/default.asp	
Reference Architectures		
Application Architecture for .NET: Designing Applications and Services	http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/distapp.asp	
Enterprise Notification Reference Architecture for Exchange 2000 Server	http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnentdevgen/html/enraelp.asp	
Improving Web Application Security: Threats and Countermeasures	http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnnetsec/html/ThreatCounter.asp	
Microsoft Accelerator for Six Sigma	http://www.microsoft.com/technet/treeview/default.asp?url=/technet/itsolutions/mso/sixsigma/default.asp	
Microsoft Active Directory Branch Office Guide: Volume 1: Planning	http://www.microsoft.com/technet/treeview/default.asp?url=/technet/prodtechnol/ad/windows2000/deploy/adguide/default.asp	
Microsoft Active Directory Branch Office Series Volume 2: Deployment and Operations	http://www.microsoft.com/technet/treeview/default.asp?url=/technet/prodtechnol/ad/windows2000/deploy/adguide/default.asp	
Microsoft Content Integration Pack for Content Management Server 2001 and SharePoint Portal Server 2001	http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncip/html/cip.asp	
Microsoft Exchange 2000 Server Hosting Series Volume 1: Planning	Online Version not available	
Microsoft Exchange 2000 Server Hosting Series Volume 2: Deployment	Online Version not available	

To learn more about *patterns & practices* visit: <http://msdn.microsoft.com/practices>
 To purchase *patterns & practices* guides visit: <http://shop.microsoft.com/practices>

Title	Link to Online Version	Book
Microsoft Exchange 2000 Server Upgrade Series Volume 1: Planning	http://www.microsoft.com/technet/treeview/default.asp?url=/technet/itsolutions/guide/default.asp	
Microsoft Exchange 2000 Server Upgrade Series Volume 2: Deployment	http://www.microsoft.com/technet/treeview/default.asp?url=/technet/itsolutions/guide/default.asp	
Microsoft Solution for Intranets	http://www.microsoft.com/technet/treeview/default.asp?url=/technet/itsolutions/mso/msi/Default.asp	
Microsoft Solution for Securing Wireless LANs	http://www.microsoft.com/downloads/details.aspx?FamilyId=CDB639B3-010B-47E7-B234-A27CDA291DAD&displaylang=en	
Microsoft Systems Architecture—Enterprise Data Center	http://www.microsoft.com/technet/treeview/default.asp?url=/technet/itsolutions/edc/Default.asp	
Microsoft Systems Architecture—Internet Data Center	http://www.microsoft.com/technet/treeview/default.asp?url=/technet/itsolutions/idc/default.asp	
The Enterprise Project Management Solution	http://www.microsoft.com/technet/treeview/default.asp?url=/technet/itsolutions/mso/epm/default.asp	
UNIX Application Migration Guide	http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnucmg/html/ucmglp.asp	
Reference Building Blocks and IT Services		
.NET Data Access Architecture Guide	http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/daag.asp	
Application Updater Application Block	http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/updater.asp	
Asynchronous Invocation Application Block	http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/paiblock.asp	
Authentication in ASP.NET: .NET Security Guidance	http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/authaspdotnet.asp	
Building Interoperable Web Services: WS-I Basic Profile 1.0	http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnsvcenter/html/wsi-bp_msdn_landingpage.asp	
Building Secure ASP.NET Applications: Authentication, Authorization, and Secure Communication	http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnnetsec/html/secnetlpMSDN.asp	

To learn more about *patterns & practices* visit: <http://msdn.microsoft.com/practices>
To purchase *patterns & practices* guides visit: <http://shop.microsoft.com/practices>

Title	Link to Online Version	Book
Caching Application Block	http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/Cachingblock.asp	
Caching Architecture Guide for .Net Framework Applications	http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/CachingArch.asp?frame=true	
Configuration Management Application Block	http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/cmab.asp	
Data Access Application Block for .NET	http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/daab-rm.asp	
Designing Application-Managed Authorization	http://msdn.microsoft.com/library/?url=/library/en-us/dnbda/html/damaz.asp	
Designing Data Tier Components and Passing Data Through Tiers	http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/BOAGag.asp	
Exception Management Application Block for .NET	http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/emab-rm.asp	
Exception Management Architecture Guide	http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/exceptdotnet.asp	
Microsoft .NET/COM Migration and Interoperability	http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/cominterop.asp	
Microsoft Windows Server 2003 Security Guide	http://www.microsoft.com/downloads/details.aspx?FamilyId=8A2643C1-0685-4D89-B655-521EA6C7B4DB&displaylang=en	
Monitoring in .NET Distributed Application Design	http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/monitordotnet.asp	
New Application Installation using Systems Management Server	http://www.microsoft.com/business/reducecosts/efficiency/manageability/application.mspix	
Patch Management using Microsoft Systems Management Server - Operations Guide	http://www.microsoft.com/technet/treeview/default.asp?url=/technet/itsolutions/msm/swdist/pmsms/pmsmsog.asp	
Patch Management Using Microsoft Software Update Services - Operations Guide	http://www.microsoft.com/technet/treeview/default.asp?url=/technet/itsolutions/msm/swdist/pmsus/pmsusog.asp	
Service Aggregation Application Block	http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/serviceagg.asp	
Service Monitoring and Control using Microsoft Operations Manager	http://www.microsoft.com/business/reducecosts/efficiency/manageability/monitoring.mspix	

Title	Link to Online Version	Book
User Interface Process Application Block	http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/uip.asp	
Web Service Façade for Legacy Applications	http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/wsfacadelegacyapp.asp	
Lifecycle Practices		
Backup and Restore for Internet Data Center	http://www.microsoft.com/technet/treeview/default.asp?url=/technet/ittasks/maintain/backuprest/Default.asp	
Deploying .NET Applications: Lifecycle Guide	http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/DALGRoadmap.asp	
Microsoft Exchange 2000 Server Operations Guide	http://www.microsoft.com/technet/treeview/default.asp?url=/technet/prodtechnol/exchange/exchange2000/maintain/operate/opsguide/default.asp	
Microsoft SQL Server 2000 High Availability Series: Volume 1: Planning	http://www.microsoft.com/technet/treeview/default.asp?url=/technet/prodtechnol/sql/deploy/confeat/sqlha/SQLHALP.asp	
Microsoft SQL Server 2000 High Availability Series: Volume 2: Deployment	http://www.microsoft.com/technet/treeview/default.asp?url=/technet/prodtechnol/sql/deploy/confeat/sqlha/SQLHALP.asp	
Microsoft SQL Server 2000 Operations Guide	http://www.microsoft.com/technet/treeview/default.asp?url=/technet/prodtechnol/sql/maintain/operate/opsguide/default.asp	
Operating .NET-Based Applications	http://www.microsoft.com/technet/treeview/default.asp?url=/technet/itsolutions/net/maintain/opnetapp/default.asp	
Production Debugging for .NET-Connected Applications	http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/DBGrm.asp	
Security Operations for Microsoft Windows 2000 Server	http://www.microsoft.com/technet/treeview/default.asp?url=/technet/security/prodtech/win2000/secwin2k/default.asp	
Security Operations Guide for Exchange 2000 Server	http://www.microsoft.com/technet/treeview/default.asp?url=/technet/security/prodtech/mailexch/opsguide/default.asp	
Team Development with Visual Studio .NET and Visual SourceSafe	http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/tdlg_rm.asp	



This title is available as a Book

To learn more about *patterns & practices* visit: <http://msdn.microsoft.com/practices>
To purchase *patterns & practices* guides visit: <http://shop.microsoft.com/practices>