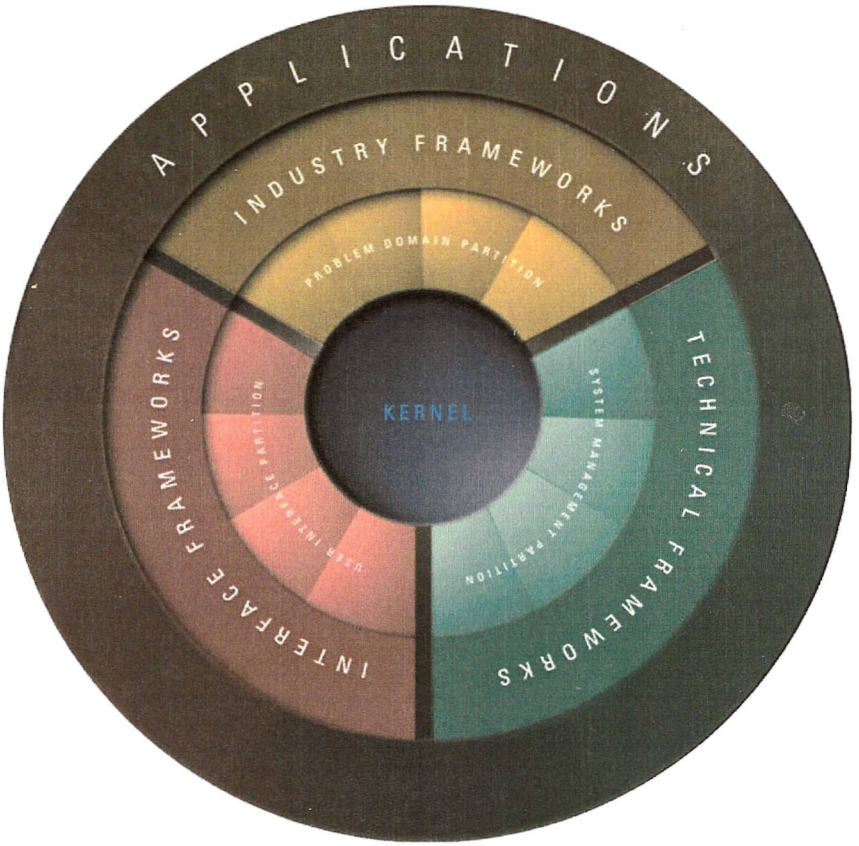
AdAPT

Advanced Application Partitioning Toolkit

Application Developer's Guide



ASF
AUSTIN SOFTWARE FOUNDRY

AUSTIN SOFTWARE FOUNDRY

AdAPT

Application Developer's Guide

Version 1.2, August 1997

Printed: August 1997

Austin Software Foundry
Capital of Texas Highway, North
Building 8, Suite 250
Austin, Texas 78746
(512) 329 6697
www.foundry.com

TABLE OF CONTENTS

Introduction I
AdAPT Overview I
About This Manual III
How the Manual Is Organized III
Users III
Comments and Information III
PART ONE: BASE CLASSES 1
The Kernel 3
Adapters 5
Introduction 5
Adapter Patterns 5
Switching Adapters 6
Dynamic Adapter Detection 11
Data Source Adapters 12
Object Factories 17
Introduction 17
Creating Object Factories 17
Using Object Factories 18
Facades 20
The Problem Domain Partition 22
Introduction 24
Business Contexts 26
Business Entities 28
The System Management Partition 34
Managers 36
Introduction 36
Creating Managers 36
Deciding to Use Named or Unnamed Object Managers 36
Walking the Manager 37
Using the Sole Use Only Attribute 38
Using the Persistent Attribute 39
Component Managers 39
Service Managers and Services 40
Application Manager 42
Transaction Manager 45
Resource 46
Introduction 46
Using a Windows .INI File As a Resource 46
Using the Windows Registry As a Resource 48
Using a Database Table As a Resource 49
Using Multiple Resources 50
Controllers 54

The User Interface Partition 56
Introduction 58
Windows 60
Menus 62
Controls 64
PART TWO: COMPONENTS 65
Logging 67
Introduction 67
Log Component Usage 67
Logging Standard Data Types 68
Logging Object Attributes 69
Logging Event Arguments 69
Developing a Custom Log Window 70
Developing a Custom Log Adapter 71
Messaging 73
Introduction 73
Using the Message Parsing Component 74
Creating a New Parser Adapter 77
CGI Parser - A Case Study 78
Query 81
The Ad hoc Query Component 81
Using the Ad hoc Query Component 82
Using the Query Object 85
Query Statement Generator Capabilities 87
Adding a New Language Adapter 91
Modifying SQL/OQL Generators 94
PART THREE: KITS 97
System Management Kits 99
Publish & Subscribe 101
Introduction 101
Publish & Subscribe: Example 102
User Interface Kits 107
Model-View-Controller 109
Introduction 109
Data Item Validation 113
Coordinating the Closing of Business Contexts and Windows 117
Appendix A: Building an Application Shell 119
Introduction 121
Communications 125
Meta Control 129
Creating a Business Component 129
Destroying a Business Component 133
Exiting the Application 134
Windows 135

Renaming Instructions 139
Renaming the Generic Application Shell Insulation Layers 139
Renaming Conventions 139
General Instructions 139
Problem Classes/Windows 139
Special Instructions for the Problem Classes/Windows 139
Appendix B: Automating the Publishing of Technical Documentation 143
Overview 145
Installation 145

INTRODUCTION

AdAPT Overview

ASF has put the best object-oriented theory and practice into a robust and continually growing library which supports all key characteristics of the next generation of client/server development:

- Application partitoning
- Distributed deployment
- Adapters to existing or legacy systems
- Plug-and-play component technology
- Highly reusable business objects
- Pattern-based design

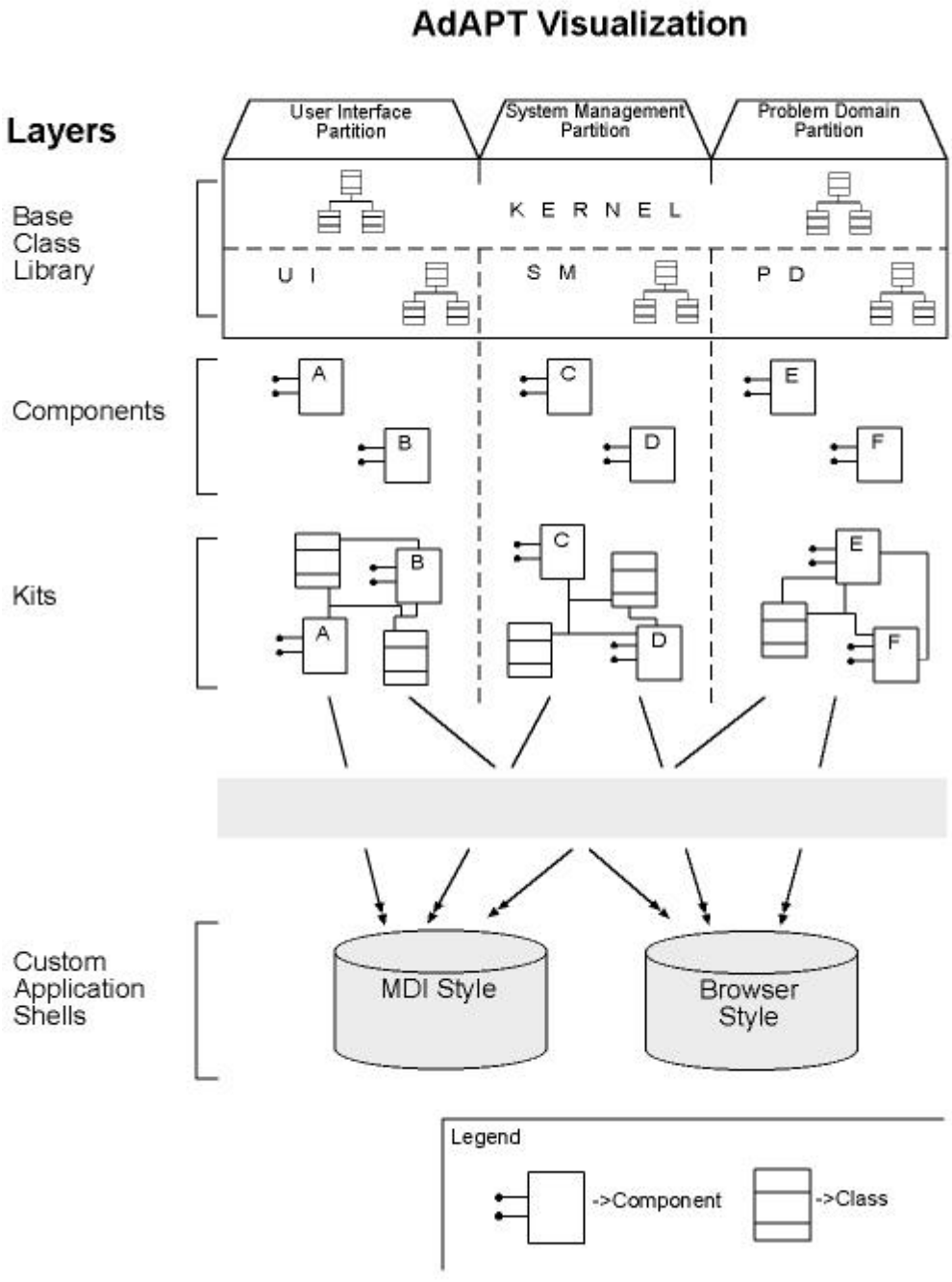
By dividing applications into three discrete partitions (system management, user interface, and business problem domain), AdAPT supports modularity, flexibility, and reuse.

AdAPT's business architecture takes a revolutionary step forward in creating reusable business components. With business entities, business context objects, and business rules each in its own object, we provide an architecture which provides maximum reuse.

AdAPT's adapter strategy provides an insulating layer between components and existing applications which makes it quicker and easier for developers to leverage the value of existing software assets, while being prepared for future development needs.

With AdAPT, development organizations can put the true power of a reusable component library to work and begin to enjoy faster development times, more flexible components and applications, and a significantly larger return on investment in existing software applications.

The picture on the following page presents the logical view of AdAPT. This manual is layered to be consistent with this view.


AdAPT Visualization
Layers
User Interface Partition
System Management Partition
Problem Domain Partition
Base Class Library
KERNEL
UI
SM
PD
Components
A
B
C
D
E
F
Kits
A
B
C
D
E
F
Custom Application Shells
MDI Style
Browser Style
Legend
->Component
->Class

About This Manual

This manual provides information that is crucial for learning about the concepts and tools for the Advanced Application Partitioning Toolkit (AdAPT). This manual contains a conceptual foundation for application and framework developers on the AdAPT product and its library of object-oriented classes. Much of the information presented here will give the developer insight into using the AdAPT Application Developer's Technical Reference. It is therefore suggested that this manual and the AdAPT Application Developer's Technical Reference be used in tandem.

How the Manual Is Organized

This manual is divided into five parts. The first part describe concepts pertaining to each part of ASF's Application Partitioning model. The second part addresses AdAPT Components and the third part, describes the AdAPT Kits.Information concerning other beneficial techniques outside of the AdAPT product is provided in appendices.

Part One: Base Classes - ***"The Kernel,"*** describes the set of utilities and data structures that are used repeatedly, commonly referred to as the kernel. Topics covered in this section include: Adapters and Facades. ***"The System Management Partition,"*** describes concepts pertaining to application services and connections to external systems. Topics covered in this section include: Managers, Resource, and Controllers. ***"The User Interface Partition,"*** describes concepts pertaining to the means by which the user manipulates the system. Topics covered in this chapter include: Windows, Menus, and Controls. ***"The Problem Domain Partition,"*** describes the business events and rules included in the application. Topics covered in this section include descriptions of AdAPT's ASFBusinessContext, ASFBusinessEntity, and ASFBusinessRule classes.

Part Two: Components - describes the use of the following AdAPT Components: the Log Component, the Message-Parser Component, and the Ad-hoc Query Component.

Part Three: Kits - discusses the specifics of the Model-View_Controller (MVC) triad of classes and the Publish & Subscribe pattern implementation.

Appendix A: ***"Building an MDI-Style Application Starter Shell,"*** describes issues involved in beginning to create a library of application shells.

Appendix B: ***"Automating the Publishing of Technical Documentation,"*** describes how technical documentation can be generated in Rational Rose to a Microsoft Word document.

Users

The users of the AdAPT Class Library are PowerBuilder developers with at least a cursory understanding of object-oriented development. Though some users may have experience with an existing framework, they should understand the differences between a component library and a traditional/first-generation PowerBuilder framework.

Comments and Information

If you have comments or would like more information about this document or about the AdAPT product line, please contact the Austin Software Foundry at (512) 329-6697.

PART ONE: BASE CLASSES

THE KERNEL

In all programming, there are a set of utilities and data structures that are used repeatedly. In the worst case, these utilities are not captured and must be recreated for each application, which violates good reuse practices. In object-oriented design and development a set of common kernel classes can provide the flexibility that developers require.

Kernel classes comprise components and classes that are not partition-specific. Classes in this layer are either root classes or abstract data types. These classes are commonly inherited from and are rarely instantiated themselves.

This part discusses the concepts on the following Kernel components:

- Adapters
- Object Factories
- Facades

Adapters

Introduction

An adapter is an object that provides a common public interface in an application to some system or component. Essentially, an adapter is a two-way translation device. As shown in Figure 1, it takes commands and events from the application (the client) and translates them to function calls to the server system. It also translates information from the server system so the client can properly interpret the data.

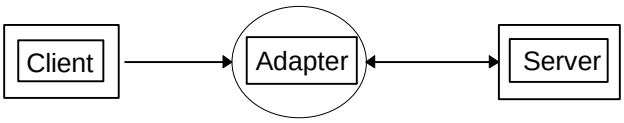


Figure 1

Adapters can be used to provide a common public interface to any number of systems providing similar functionality, such as security systems. Adapters can also be used to "wrap" legacy systems. Legacy functions can then be replaced with newer functions simply by replacing the adapter with one that "talks" to the newer ones. The code inside the applications remains the same.

Adapter Patterns

An adapter pattern is a design pattern that maximizes the utility of adapters. The figure below shows the basic adapter pattern used in AdAPT.

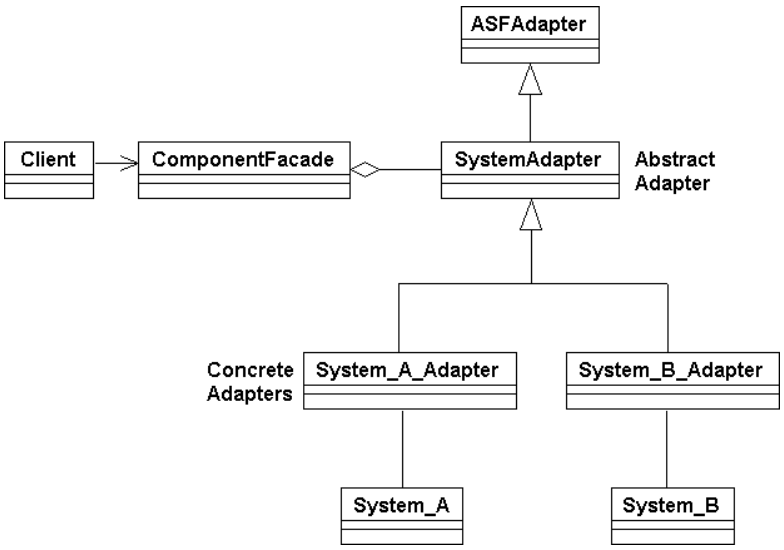


Figure 2

SystemA and SystemB are two different systems that provide similar functionality, but not in the same exact format. As examples, SystemA and SystemB might be two different security systems or two different finance systems. The ComponentFacade class provides a stable public interface to the application. All calls from the application for system functionality are made to ComponentFacade objects.

The SystemAdapter class is an abstract class that provides a stable interface from the ComponentFacade class to system functionality. This class does not implement methods, but rather defines virtual methods as place holders so that ComponentFacade objects can call concrete adapter methods.

A concrete adapter is one that implements the virtual methods defined in the abstract adapter class. The implementations are specific to the system with which the adapter is associated.

This pattern is especially useful when systems change. Suppose that there is a new system called SystemC. Create a new concrete adapter class, System_C_Adapter, that inherits from the abstract class SystemAdapter. Implement the SystemAdapter virtual methods in the System_C_Adapter class. Simply replace the old concrete adapter class with the System_C_Adapter.

Switching Adapters

Resource Management Technique

A common development task is switching the usage of an external system with another. This example will show how a Personal Information Manager (PIM) application, which used an adapter to access the application's .INI file, is modified to use a different adapter to access the Window registry. The simple change switches from using the ASFResourceIniAdapter to the ASFResourceRegistryAdapter. Without the use of adapters, all references to ProfileString( ) throughout the application would have to be changed to RegistryGet( ).

During the initialization of the application, the resource facade is created. The type of adapter is then specified. Only two statements were changed in the switch from the application's .INI file to the Windows registry. The resource facade provides the common user interface to all types of resources. The adapter contains the code specific to the type of resource.

```
STRING ls_registration_name
ANY lany_object
ANY lany_requestor
BOOLEAN lb_sole_use_only
BOOLEAN lb_persistent
ASFResourceFacade lnv_resource

lany_requestor = THIS

ipronv_app_manager = CREATE ASFApplicationManager

/* Register INI file as Resource object */
lnv_resource = CREATE ASFResourceFacade
/*************************'
* Original code which interfaced with the application's
* .ini file
*  lnv_resource.TRIGGER STATIC FUNCTION &
*     SetAdapter( "ASFResourceIniAdapter" )
*  lnv_resource.TRIGGER STATIC FUNCTION &
*     OpenResource( "c:\pim\pim.ini" )
*************************/
/* The next two statements were modified to interface
 * with the Windows Registry */
lnv_resource.TRIGGER STATIC FUNCTION &
      SetAdapter( "ASFResourceRegistryAdapter" )
```

```
lnv_resource.TRIGGER STATIC FUNCTION &
     OpenResource( "HKEY_LOCAL_MACHINE\SOFTWARE\pim" )
lany_object = lnv_resource
ls_registration_name = "PIM Resource"
lb_sole_use_only = FALSE
lb_persistent = TRUE
ipronv_app_manager.TRIGGER STATIC FUNCTION &
     Register( lany_object, ls_registration_name, &
     lany_requestor, lb_sole_use_only, lb_persistent )
```

In the main body of the application, calls to the facade object retrieve and update values from the resource through its adapter. Note that these calls to retrieve and update the resource values are not dependent on the type of resource being used.

```
ANY lany_this
STRING ls_value
STRING ls_resource_name
ASFApplicationManager lnv_app_mgr
ASFResourceFacade lnv_resource

lany_this = THIS
ls_resource_name = "PIM Resource"

lnv_app_mgr = GetApplication( ).TRIGGER DYNAMIC FUNCTION &
     af_get_application_manager( )
/* Get the resource object */
lnv_resource = lnv_app_mgr.TRIGGER STATIC FUNCTION &
     RequestComponent( ls_resource_name, lany_this )
IF ( IsNull( lnv_resource ) ) THEN
     IF ( NOT IsValid( lnv_resource ) ) THEN
        RETURN
     END IF
END IF

/* Retrieve the values from the Resource */
lnv_resource.TRIGGER STATIC FUNCTION SetSection( "Database" )
SQLCA.DBMS = lnv_resource.TRIGGER STATIC FUNCTION &
     RetrieveValue( "DBMS" )
```

Library Management Technique

This section is an example of how to switch adapters using library (PBL) management. This example calls for a single customer business class. However, a legacy system with customer information already exists. Also, the company is switching to SAP, which will be a time-consuming conversion process. Adapters will help to complete the application in a timely manner and make the conversion from the legacy system to SAP transparent.

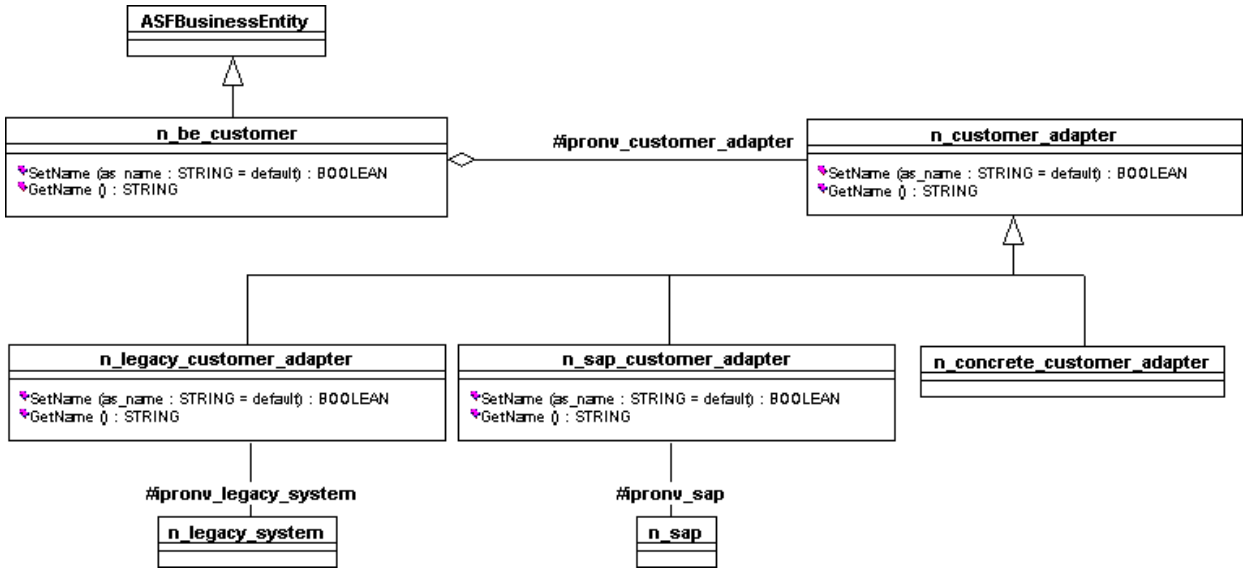


Figure 3

In Figure 3 above, the n_be_customer class provides the public interface to the application for the customer business entity. The n_be_customer class has an adapter instance variable, ipronv_customer_adapter, which represents the aggregation of the n_customer_adapter class. The following code is for the methods GetName() and SetName() in the n_be_customer class:

```
/* n_be_customer.GetName() */
STRING ls_return

ls_return = ipronv_customer_adapter.TRIGGER STATIC FUNCTION
GetName()
RETURN ls_return

. . .

/* n_be_customer.SetName() */
/* Parameter:  STRING as_name */
BOOLEAN lb_return

lb_return = ipronv_customer_adapter.TRIGGER STATIC FUNCTION &
      SetName( as_name )
RETURN lb_return
```

The following code is for the methods GetName() and SetName() in the n_customer_adapter class. This is an abstract class and these are virtual methods. Since PowerBuilder does not have a virtual method capability, these methods will simply return and they will be implemented in descendent classes.

```
/* n_customer_adapter.GetName() */
STRING  ls_return

ls_return = ""
RETURN ls_return
```

```
. . .

/* n_customer_adapter.SetName() */
/* Parameter:  STRING as_name */
BOOLEAN lb_return

lb_return = FALSE
RETURN lb_return
```

The class n_concrete_customer_adapter is simply a place holder. It has no methods or instance variables. It will be replaced with the actual adapters on an as-needed basis.

The classes n_legacy_customer_adapter and n_sap_customer_adapter will implement the methods GetName() and SetName(). Since the legacy system (n_legacy_system) and the SAP system (n_sap) access customer data in different ways, the GetName() and SetName() methods will be different in the two adapters. The following code is for the GetName() and Set Name() for the n_legacy_customer_adapter:

```
/* n_legacy_customer Instance Variables */
N_LEGACY_SYSTEM   ipronv_legacy_system
. . .
/* n_legacy_customer_adapter.GetName() */
STRING   ls_return

ls_return = ipronv_legacy_system.TRIGGER STATIC FUNCTION &
     GetField( "customer_name" )

RETURN ls_return

. . .

/* n_legacy_customer_customer_adapter.SetName() */
/* Parameter:  STRING as_name */
BOOLEAN  lb_return

lb_return = ipronv_legacy_system.TRIGGER STATIC FUNCTION &
     SetField( "customer_name", as_name )

RETURN lb_return
```

The following code is for GetName() and SetName() for the n_sap_customer_adapter:

```
/* n_sap_customer Instance Variables */
N_SAP   ipronv_sap
. . .
/* n_sap_customer_adapter.GetName() */
STRING   ls_return

ls_return = ipronv_sap.TRIGGER STATIC FUNCTION &
     sap_customer_name()

RETURN ls_return

. . .
```

```
/* n_sap_customer_customer_adapter.SetName() */
/* Parameter:  STRING as_name */
BOOLEAN  lb_return

lb_return = ipronv_sap.TRIGGER STATIC FUNCTION &
     SetField( as_name )

RETURN lb_return
```

Properly maintaining libraries can make switching from one system to another a straightforward process and transparent to the application. The following table is an example of how the libraries might be organized:

customer.pbl (in library list)	n_be_customer n_concrete_customer_adapter
extsys.pbl (in library list)	n_legacy_system
legacy.pbl	n_legacy_system n_legacy_customer_adapter
sap.pbl	n_sap n_sap_customer_adapter

The PBLs customer.pbl and extsys.pbl are included in the application library list. Initially, the application is adapted to the legacy system. As parts of the SAP system come on line, it is necessary to switch adapters.

To create a new set of PBLs for distribution:

1. Copy customer.pbl and extsys.pbl into a distribution directory. All changes are made to the PBLs in the distribution directory.

2. Save n_sap_customer_adapter (legacy.pbl) as n_concrete_customer_adapter.

3. Move the new n_concrete_customer_adapter to customer.pbl, an action overriding the old n_concrete_customer_adapter.

4. Delete n_legacy_system from extsys.pbl and copy n_sap from sap.pbl to extsys.pbl.

Distribute the new customer.pbl and extsys.pbl.

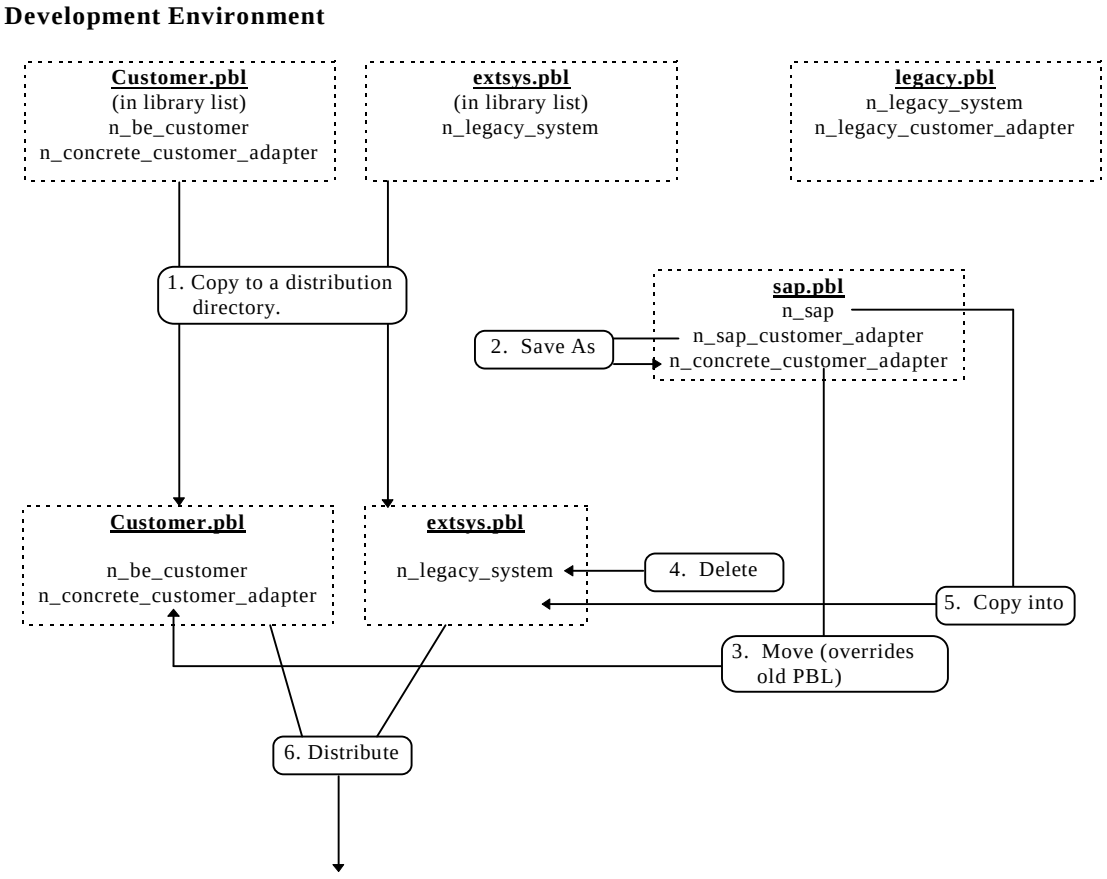


Figure 4

In Figure 4, it is possible to name each of the adapters the same thing, in this case, n_concrete_customer_adapter. Since legacy.pbl and sap.pbl are not included in the application library list, there should be no name conflicts. Problems might crop up during adapter development and maintenance. Since the names are the same, the developer may get confused as to which adapter is being worked on.

Dynamic Adapter Detection

The scenario includes an application that must interface with different external systems, i.e. accounting programs. The application is to be used by different locations, which used different accounting programs. During the application development, an adapter is written for each type of accounting program. An adapter would distributed in acctadpt.pbl to each site according to the accounting program being used. Each adapter name begins with the pattern 'AcctAdapter', i.e. AcctAdapterAbc. If the accounting program is switched at a location, the acctadpt.pbl with the corresponding adapter is then distributed.

This example shows how an application determines at run time which adapter to use.

```
STRING ls_entries
LONG ll_start, ll_end
```

```
STRING ls_adapter_name
AccountingAdapter lnv_adapter

// Retrieve a list of all user objects in the PBL
ls_entries = LibraryDirectory( "acctadpt.pbl", DirUserObject! )

// The adapter to be used begins with the pattern 'acctadapter'
ll_start = Pos( ls_entries, "acctadapter" )
ll_end = Pos( ls_entries, "~t", ll_start )
ls_adapter_name = Mid( ls_entries, ll_start, ll_end - ll_start )

// Create the adapter to be used with this application
lnv_adapter = CREATE USING ls_adapter_name
```

Data Source Adapters

Introduction

A data source adapter is an object that provides a common application interface to a data source. A data source is an object or interface to a repository of data (e.g., relational database, object-oriented database, etc.). A data source adapter allows the client application to retrieve, add, delete, modify, and update data in diverse data sources using a common interface.

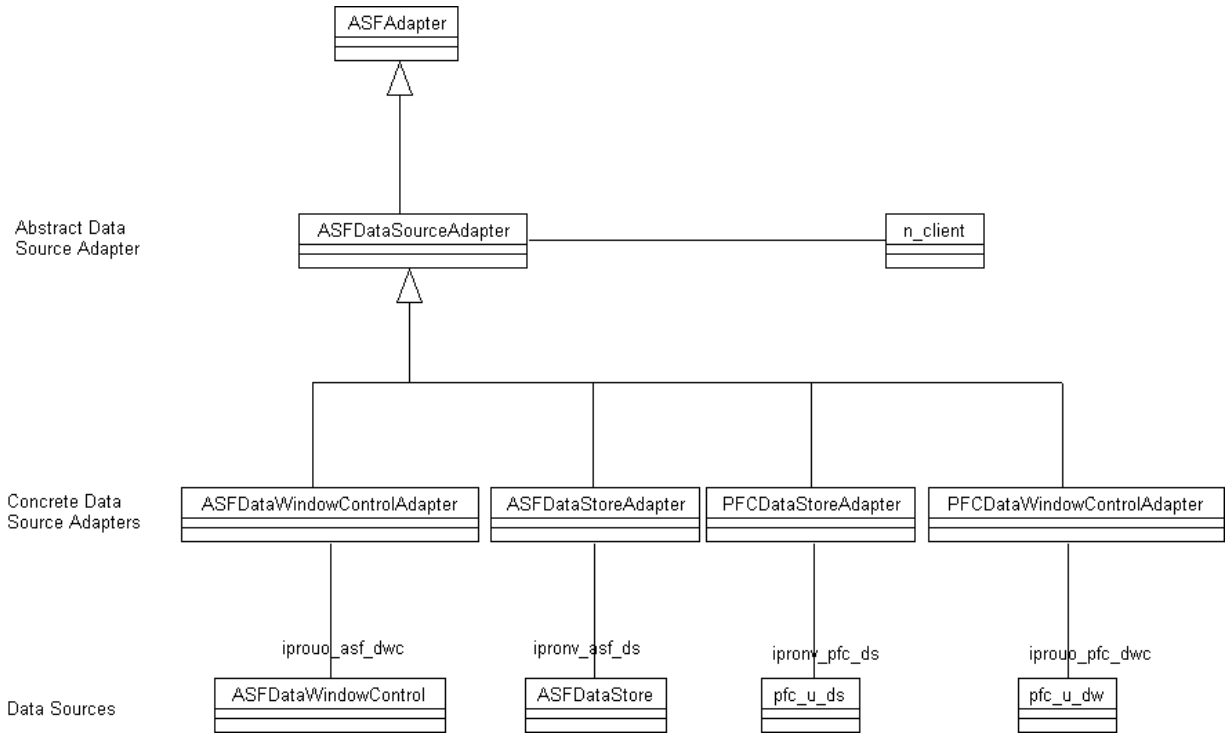


Figure 5

The abstract data source adapter, ASFDataSourceAdapter, provides the interface to some client. Although each data source operates differently, the client only sees a single interface. As long as

the concrete data source adapters maintain a constant public interface, changing data sources does not mean changing the client. The concrete adapters provide the implementations to adapt to specific data sources.[1]

Creating Data Source Adapters

Creating a new data source adapter is a straightforward process:

- Specialize (inherit from) ASFDataSourceAdapter.
- Implement ASFDataSourceAdapter virtual methods in the new class.
- Connect the data source adapter to the data source.

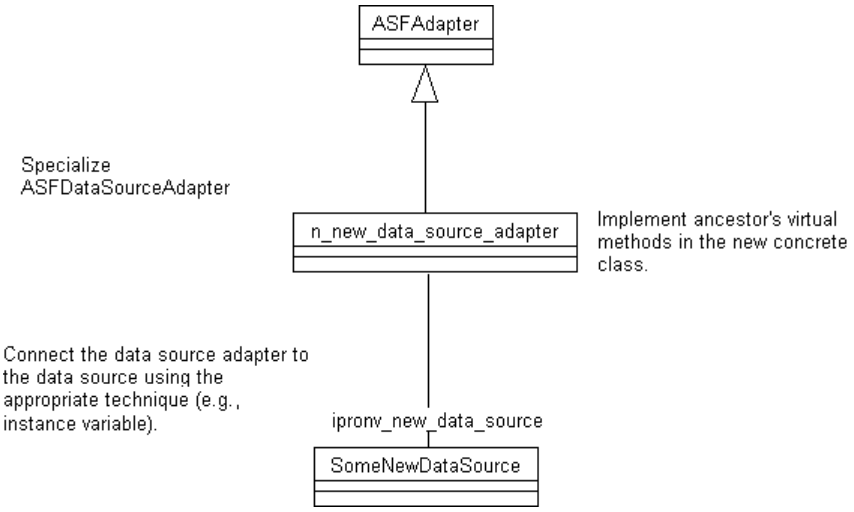


Figure 6

When implementing the virtual methods, it is not necessary to implement all of the methods. Only implement those that will be used in the new data source. Also, the view of the adapter from the client looks object-oriented. It is necessary to map the object-oriented methods to the data source itself. For example, when adapting to a DataWindow control, the adapter DeleteObject() method should delete the current row in the DataWindow control. For tables & views is a row represents an object and a column represents an attribute in the object. The reason for this object-oriented view is to add flexibility for varying designs, which is discussed in the next section.

Using Data Source Adapters

Data source adapters present an object-oriented view of the data source to the clients in order to add flexibility for varying designs. The following approaches to business entity-data source connections demonstrate the flexibility available.

---

1. AdAPT 1.1 provides the ASFDataWindowControlAdapter class. Future versions of AdAPT will contain the ASFDataStoreAdapter, PFCDataStoreAdapter, and PFCDataWindowControlAdapter classes.

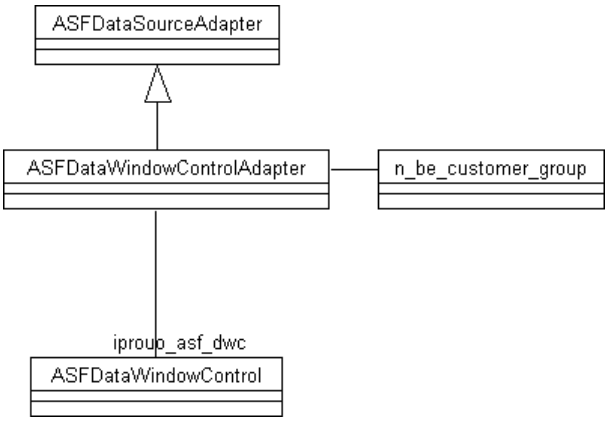


Figure 7

In this first approach, the n_be_customer_group class provides access to both individual customers and aggregate, or group, functions (e.g., customer count). This pattern is used when customer accounts are always treated the same. However, there are times when customer accounts should be treated in different ways. In this case, a separation of individual and aggregate functions is needed.

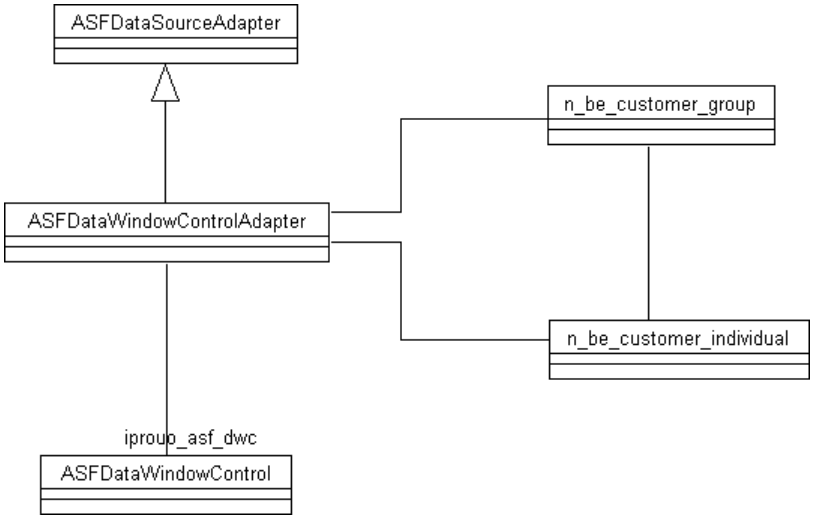


Figure 8

In this second approach, the n_be_customer_group class performs aggregate functions and the objects of type n_be_customer_individual perform the individual functions (e.g., modify customer). By using the adapter, the object of type n_be_customer_group creates an object of type n_be_customer_individual for each row in the DataWindow control.

These are not the only possible design variants possible when using data source adapters. The two designs above take a data-source-centric approach and assume that DataWindow controls are treated the same. However, there are groups of data sources that provide the same type of data, although their operations are different.

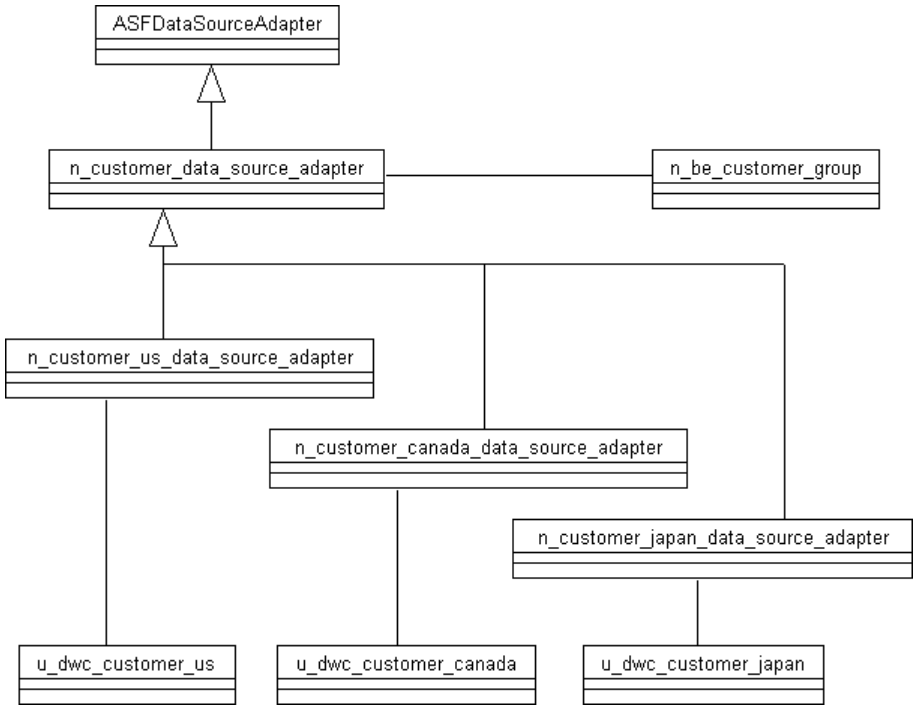


Figure 9

The data source adapter adds a great deal of flexibility to designs. Selecting the appropriate design for a given application, or set of applications, can greatly enhance object reuse.

Object Factories

Introduction


ASFObject


ASFKernel


ASFCreation


ASFFactory


ASFObjectFactory

An object factory is responsible for creating objects. Specific object factories create related or dependent objects. Object factories allow the developer to specify objects to be created while maintaining a loose coupling between the requesting object (client) and the created object.

AdAPT provides the abstract object factory ASFFactory and the general factory ASFObjectFactory.

Creating Object Factories

The ASFObjectFactory class provides a CreateObject() method that can create any type of non-visual object. The CreateObject() method uses the PowerBuilder CREATE USING statement. Although a powerful method, the method can be costly in terms of performance. The CREATE USING statement requires PowerBuilder to check its list of object types at run time to find the correct one to implement. This search activity can bog the system down.

A more efficient approach is to specialize (inherit from) the ASFObjectFactory class and override the CreateObject() method. In the new method, use the SELECT CASE statement to find the appropriate object type to instantiate. For example, suppose an object factory is needed to generate the business entity objects n_be_customer, n_be_invoice, and n_be_part. First, create a new object factory (let's call it n_object_factory_be) inherited from ASFObjectFactory. Then write the CreateObject() method in the class n_object_factory_be:

```
/* PUBLIC FUNCTION ANY CreateObject(  STRING as_object_type ) */

ANY   lany_object
SetNull( lany_object )

/* check the parameter */
IF ( IsNull( as_object_type ) ) THEN
     RETURN lany_object
END IF

/* Find and create the appropriate object */
CHOOSE CASE Upper( as_object_type )

     CASE "CUSTOMER"
        lany_object = CREATE n_be_customer

     CASE "INVOICE"
```

```
            lany_object = CREATE n_be_invoice

         CASE "PART"
            lany_object = CREATE n_be_part

   END CASE

   RETURN lany_object
```

The first thing to note is that this object factory only creates three types of objects, a much smaller number than the hundreds of objects searched through by PowerBuilder when CREATE USING is used. Changing the type of object created is simply a matter of changing the CREATE statement. Also note that the object type does not necessarily have to be the PowerBuilder class type, which is the case when using the ASFObjectFactory class CreateObject() method.

Using Object Factories

An object factory is used whenever there is a need to separate a client object from the created object. One potential place for an object factory is with the Application Manager.

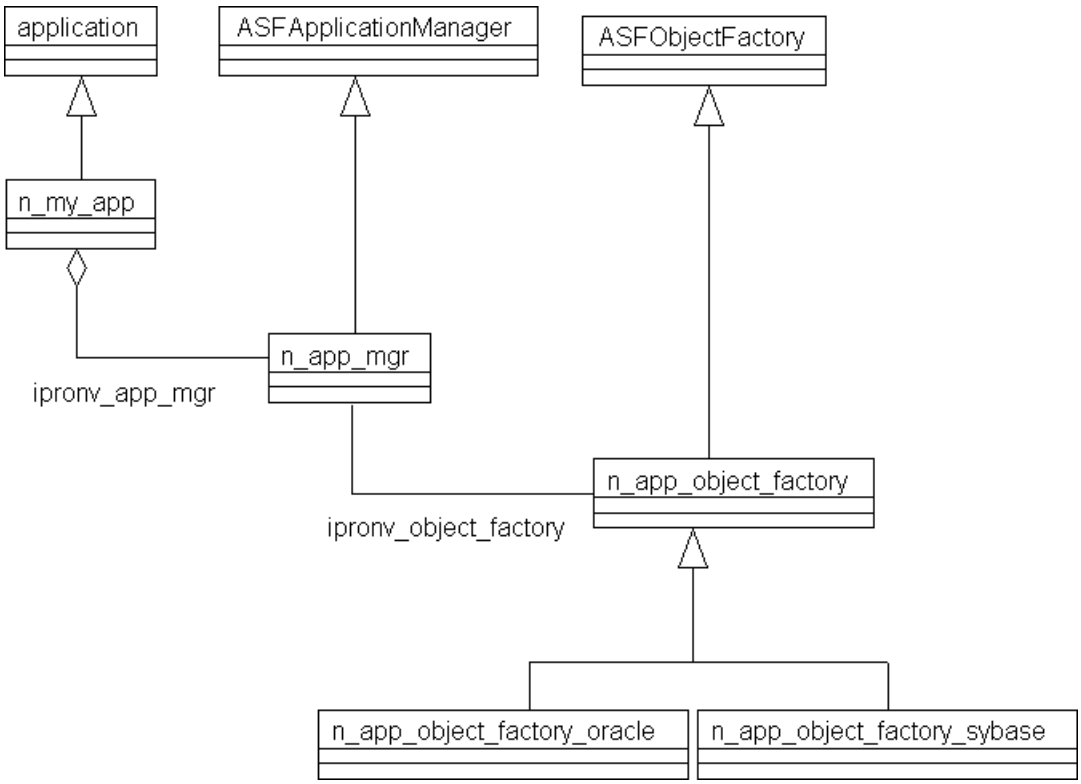


Figure 10

The classes n_app_object_factory_oracle and n_app_object_factory_sybase each contain the method CreateObject() that creates a security component, a transaction object manager, a log component, etc., each factory class specific to a database management system. By changing factory class is instantiated, the type of objects created are changed. The type of factory class can be easily read in using a resource component (i.e., INI files, Windows registry, or database).

Some other uses for object factories might include the following:

- Creating business entities for business contexts.
- Creating data source adapters for business entities.
- Creating windows for controllers.

The above list is certainly not exhaustive.

Facades

The Facade pattern is utilized throughout AdAPT. A facade is a class that acts as the public interface to a complex subsystem of classes. A facade class also routes messages from objects external to the subsystem (clients) to classes within the subsystem. Facade classes simplify the interface between a subsystem and clients of the subsystem.

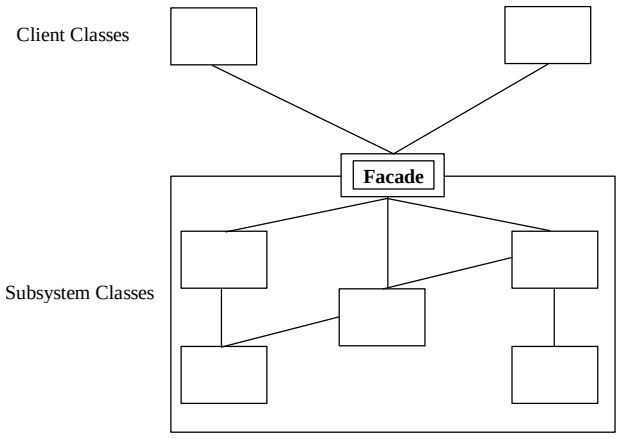
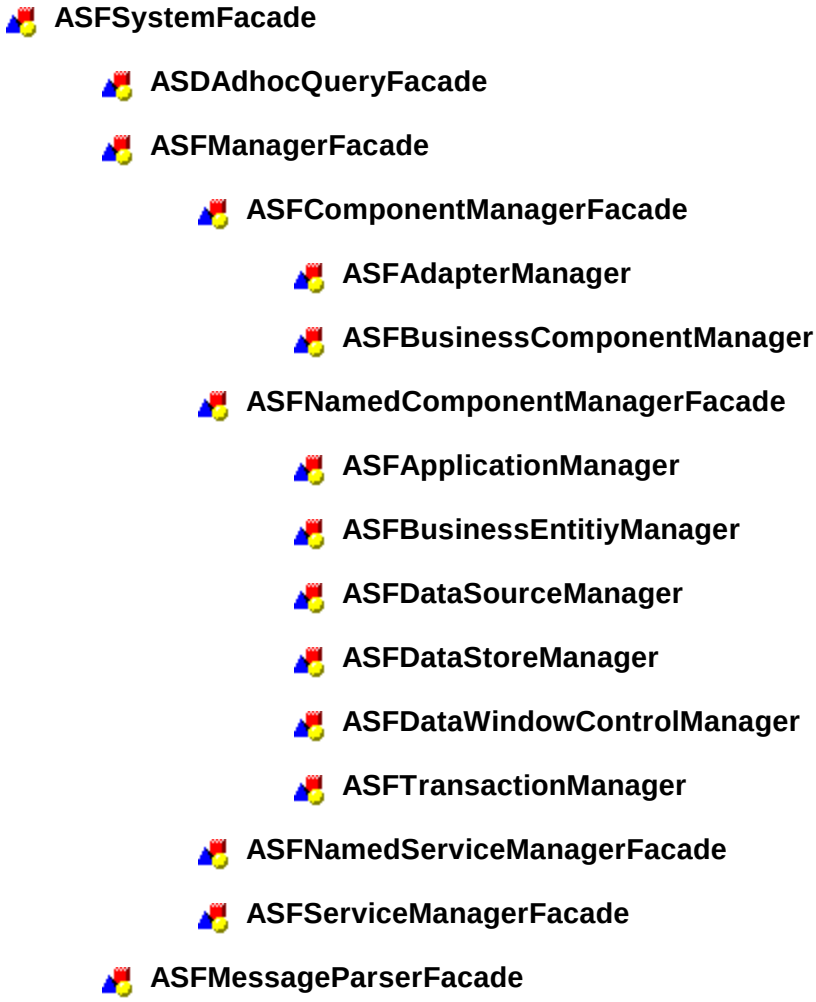


Figure 11

Facades should be used to provide a simple interface to a complex subsystem. The facade decouples the subsystem classes from the client classes and this makes the subsystem more reusable. Also, changes to the subsystem are transparent to the client classes.

The AdAPT facade class hierarchy uses a separate branch to organize facades. This organization helps reduce the tendency to put subsystem functionality into a facade class and increases the ability to reuse subsystem classes.

- **ASFObject**
  - **ASFFacade**
    - **ASFDomainFacade**
    - **ASFKernelFacade**
      - **ASFContainerFacade**
        - **ASFListFacade**
          - **ASFDoubleLinkedListFacade**
          - **ASFSingleLinkedListFacade**
      - **ASFLogBaseFacade**
        - **ASFLogFacade**
      - **ASFResourceFacade**



The ancestor classes ASFKernelFacade, ASFDomainFacade, and ASFSystemFacade are organized to partition facade classes in the same way other classes are partitioned. Creating a new facade class is simply an issue of selecting the appropriate facade class from which to inherit. For example, suppose the security system used by applications has many classes with complex relationships. A facade class n_security would inherit from ASFSystemFacade, since security is part of the System Management Partition.

THE PROBLEM DOMAIN PARTITION

The Problem Domain Partition focuses on business models. This is the area in which business events and rules are included in application. Business models handle the requests captured in the user interface and coordinate other system components to satisfy those requests. By providing easy-to-use tools in the User Interface Partition and the System Management Partition, the developer can spend more time focusing on solving business problems, not system problems. Furthermore, because developers can build reusable business components, they will spend less time building business models for future applications. This part discusses the following concepts:

- Business entities
- Contexts
- Rules

Introduction

AdAPT provides three base business classes for deriving specific business class from:

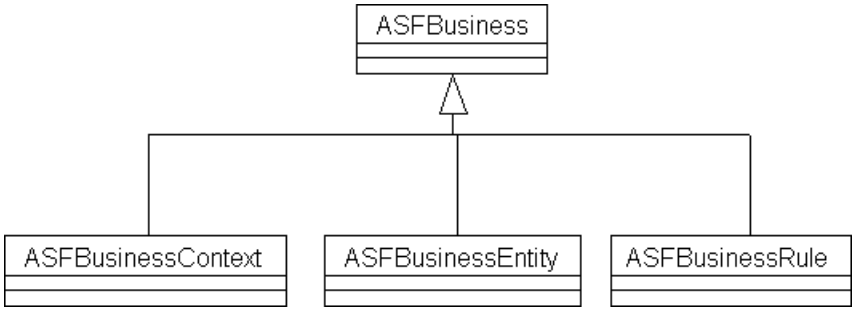


Figure 12

The ASFBusinessContext class provides the basic mechanisms needed to orchestrate multiple business entities to solve context-specific problems. The business context also provides the public interface to the rest of the business-related objects. This decoupling of business classes from other application classes enhances the reusability of business-related classes.

The ASFBusinessEntity embodies real business objects for the application. The same business entity can be used in more than one business context. For example, a customer entity may be used in a booking context and a marketing context (e.g., marketing newsletter distribution).

The ASFBusinessRule class provides flexible behavior to both the ASFBusinessContext and ASFBusinessEntity classes. A business rule can be context-specific or business-entity-specific. For example, a context-specific rule may be that a customer may pay for orders in 30, 60, or 90 days depending upon some function of the amount of past sales to that customer (billing context). A business-entity-specific rule may be that a customer must have a customer ID number.

Business Contexts

nonvisualobject

ASFObject

ASFDomain

ASFBusiness

ASFBusinessContext

The ASFBusinessContext class is used for two interrelated purposes. First, it provides the basic public interface for business entities to the rest of the application. As such, the ASFBusinessContext class has attributes similar to facade classes. Second, it orchestrates the interactions between business entities. In this role, the ASFBusinessContext class acts like a controller class for business entities.

The public interface of business context classes derived from ASFBusinessContext can use methods or some sort of messaging system. Using the method approach, every public method must be known to objects communicating with the business context object. This means a tight binding between the business context object and other objects. The ability to reuse the business context and the other objects is significantly reduced. However, application performance is optimized by using direct method calls.

A messaging system can decouple the business context and objects external to the context. The business context object is responsible for decoding the message and responding accordingly. This means that the basic messaging mechanism can be placed in an ancestor to the business context class and the class is then responsible for the specific decoding implementation.

The ASFBusinessContext class provides a simple messaging system.

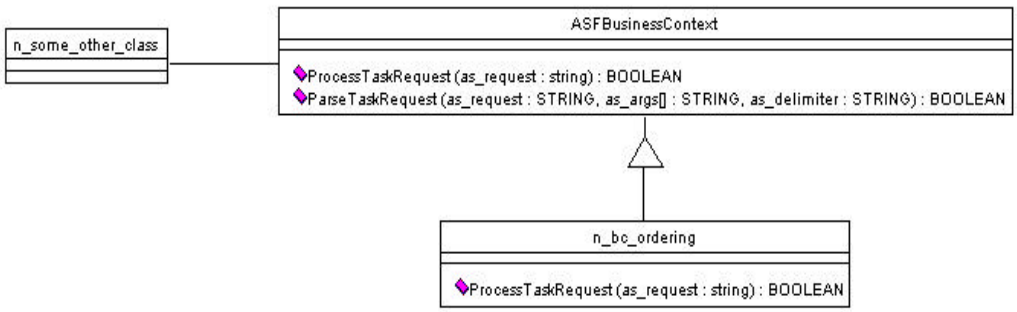


Figure 13

The ProcessTaskRequest() method in ASFBusinesContext is a virtual method, that is, the method is intended to be overridden in descendent classes, in this case, n_bc_ordering. This technique allows classes external to the context (i.e., n_some_other_class) to access the context through method calls to ASFBusinessContext. The technique also obviates the need to use the DYNAMIC keyword in the method call.

For more information about ProcessTaskRequest(), see the AdAPT Technical Reference, ASFBusinessContext.

Business Entities

nonvisualobject

ASFObject

ASFDomain

ASFBusiness

ASFBusinessEntity

A business entity embodies a real business object in applications. For example, a customer business entity represents one or more customers in the applications. In AdAPT, the class ASFBusinessEntity is the base class for deriving specific business classes.[2] The ASFBusinessEntity class provides a data source manager upon construction and destroys the manager when the class is destroyed.

Common sources of data for business entities are DataWindow controls (ASFDataWindowControl) and DataStores (ASFDataStore). These object types are utilized to provide common access to a wide variety of database systems. The ASFBusinessEntity contains an adapter manager (ipronv_data_source_manager of type ASFDataSourceManager) to handle business entity connections to multiple data sources. Any number of data sources can be associated with the business entity. The data source adapters are named, so the business entity only needs the adapter name to access the data source.

2. Most of the methods in the AdAPT 1.0 ASFBusinessEntity are subsumed by the AdAPT 1.1 ASFDataSourceAdapter. The earlier methods are now obsolete and will be removed in a future version of AdAPT. These methods are clearly labeled as obsolete in the AdAPT Application Developers Technical Reference, the models, and in the code.

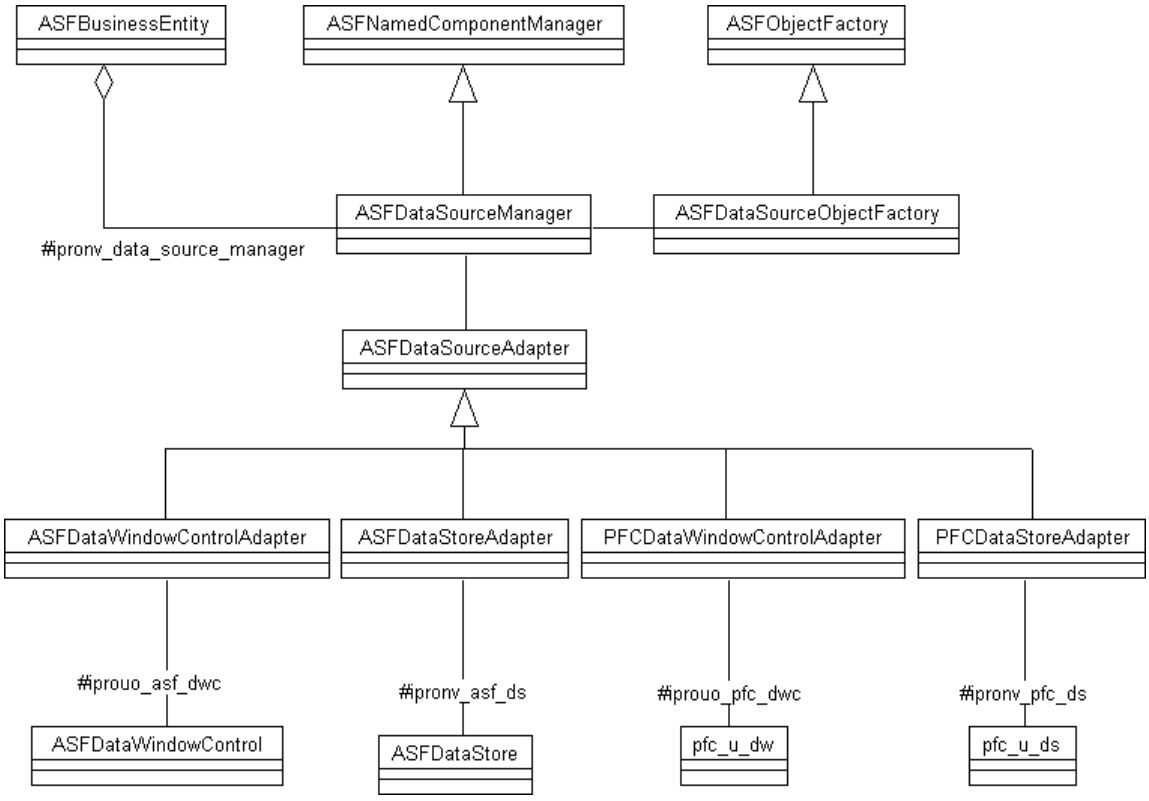


Figure 14

The ASFDataSourceAdapter is a virtual class. This means that it contains a placeholder for almost all of the class methods. To create an adapter for a new data source, inherit from the ASFDataSourceAdapter class and implement the virtual methods.[3]

When the business entity is being initialized, it registers whatever data source adapters it needs with the Data Source Manager. The Data Source Manager uses the ASFDataSourceAdapterFactory class to actually generate the adapter. Although it is not necessary, it is recommended that the ASFDataSourceAdapterFactory class be inherited from and the CreateObject() method be overridden. The CreateObject() method in the ASFDataSourceAdapterFactory class uses the PowerBuilder CREATE USING statement. This is a rather inefficient way of creating objects since PowerBuilder must search through its list of class to find the correct one to implement. A more efficient method is to use the SELECT CASE statement. This method not only restricts the search to a few class types, but also allows the actual type created to be easily changed. For example, suppose the business entity object knows about an adapter called "CustomerAdapter". The adapter registration might look like the following:

```
ANY   lany_this
lany_this = THIS
. . .
ipronv_data_source_adapter.TRIGGER STATIC FUNCTION &
    RegisterAdapter( "customers", "CustomerAdapter", lany_this ,
TRUE, FALSE )
```

3. AdAPT 1.1 contains a data source adapter for the ASFDataWindowControl. AdAPT 1.2 will contain data source adapters for the ASFDataStore, the PFC DataWindow control and the PFC DataStore.

RegisterAdapter() method calls CreateObject() method in the Object Factory (inherited from ASFDataSourceObjectFactory). The CreateObject() method overrides the ancestor method and uses a SELECT CASE statement to convert "CustomerAdapter" to the correct type:

```
ANY   aany_object
. . .
CHOOSE CASE Upper(  )
   . . .
   CASE Upper( "CustomerAdapter" )
      aany_object = CREATE u_dwc_adapter_customer
   . . .
END CASE
. . .
RETURN aany_object
```

In this example, the adapter type u_dwc_adapter_customer is created. In a different application, a different adapter may be needed. Simply replace the u_dwc_adapter_customer with the new adapter type.

It is not possible to identify the specific Object Factory used by the Data Source Manager within AdAPT. One approach is to create the Object Factory and then connect it to the Data Source Manager. An alternative approach is to inherit from the ASFDataSourceManager class and create the Object Factory in the constructor of the manager class.

Data Window Control Example

The ASFBusinessEntity class has the flexibility to handle DataWindow controls, DataStores, and other data sources in a variety of ways. One technique is to hard code instance variables for each of the objects of interest. For example, suppose the DataWindow control u_dwc_customer is associated with n_my_business_entity, a class inherited from ASFBusinessEntity:

```
/* n_my_business_entity Instance Variables */
U_DWC_CUSTOMER   iprouo_dwc_customer
```

The primary disadvantage to this technique is that n_my_business_entity is bound to u_dwc_customer. Changing the business entity class to use a different DataWindow control means changing all references to u_dwc_customer.

An alternative approach is to use the Data Source Manager, ipronv_data_source_manager, which is a part of ASFBusinessEntity. DataWindow controls are then accessed using data source adapters.

1. Register Adapter

```
 /* Constructor: register the data source adapter */
ANY   lany_object lany_object = THIS
. . .
ipronv_data_source_manager.TRIGGER STATIC FUNCTION &
   RegisterAdapter( "customers", "CustomerAdapter", lany_object,
TRUE, FALSE )
```

2. Register DataWindow Control

```
/* n_my_business_entity receives reference to DataWindow control */
```

```
BOOLEAN                lb_return
ANY                    lany_this
ASFDATASOURCEADAPTER   lnv_adapter

lany_this = THIS
. . .
/* get reference to the data source adapter */
lnv_adapter = ipronv_data_source_manager.TRIGGER STATIC FUNCTION &
   RequestComponent( "customers", lany_this )
. . .
/* DataWindow control is passed by reference to this method */
/* Register the DataWindow control using
      RegisterDataWindowControl()which is specific to
      ASFDataWindowControlAdapter */
lb_return = lnv_adapter.TRIGGER DYNAMIC FUNCTION &
   RegisterDataWindowControl( auo_dwc )
. . .
```

3. Get

```
 /* In some other scope */
BOOLEAN                lb_return
ASFDATASOURCEADAPTER   lnv_adapter
. . .
lnv_adapter = ipronv_data_source_manager.TRIGGER STATIC FUNCTION &
    RequestComponent( "customers", lany_this )
```

4. Use

```
lb_return = lnv_adapter.TRIGGER STATIC FUNCTION &
   RetrieveData()
```

The judicious use of the DYNAMIC keyword when calling functions can also be extremely valuable. Using the example, suppose that the data source adapter "CustomerAdapter", named "customers," has a method called uf_find_customer(). The method can be called without special type casting:

```
lb_return = luo_dwc.TRIGGER DYNAMIC FUNCTION &
   uf_find_customer( "customer_name = 'Krusteaz'" )
```

The DYNAMIC keyword should be used sparingly and with caution. If DYNAMIC is used and the method does not exist in the object at run-time, a run-time system error will occur. Also, over using DYNAMIC can cause significant performance problems because the method call is resolved at runtime. This means that PowerBuilder must search through the object's method list to find a match to the method call.

There is no mechanism in AdAPT to pass a DataWindow control reference to a business entity and then to the data source adapter. DataWindow control registration from windows to adapters is framework dependent and the AdAPT Base Class Library does not force a particular framework onto the developer. Some potential methods for connecting a DataWindow control to a data source adapter include the following:

- Pass the DataWindow control reference in a message.

- Pass the DataWindow control reference in a method.
- Pass a location where the DataWindow control reference exists.

THE SYSTEM MANAGEMENT PARTITION

The System Management Partition focuses application services and connections to external systems. Application services are those components that are used throughout the application. External systems include database management systems and external server applications.

This part discusses concepts on the following System Management components:

- Managers
- Resource
- Controllers

Managers

Introduction

A manager is a class that provides the ability to maintain a list of objects by registration. A registration is simply a means to identify an object. An object can be registered with a manager either with or without a registration name. If the object is registered by name, then the registration name is used to access the object in the manager. If the object is registered without a name, then a registration ID is returned and the registration ID is used to access the object in the manager. The manager gives to requesting objects a reference to a registered object.

Creating Managers

A new manager class is created by inheriting from one of the following base manager classes:

ASFServiceManager
ASFComponentManager
ASFNamedServiceManager
ASFNamedComponentManager

A service manager handles objects descendent from ASFService (see "Service Managers and Services"). A component manager handles objects of any type. A manager can handle objects that are either named or unnamed, but not both. Unnamed object managers, which descend from ASFServiceManager or ASFComponentManager, return a registration ID upon registration. This registration ID is used to reference the object. Named object managers, which descend from ASFNamedServiceManager or ASFNamedComponentManager, require a registration name to be supplied when an object is registered with the manager. The registration name is then used to reference the object.

Deciding to Use Named or Unnamed Object Managers

The decision to use a named or unnamed object manager depends on whether the manager is being used to service many different objects or only one object. The ASFApplicationManager (descendent from ASFNamedComponentManager) is a good example of where the registered objects in the manager are needed by many different objects throughout the application. Suppose, for example, that the application manager (n_app_mgr) has registered a security object called n_security under the name "Security". Any object that gains access to n_app_mgr can get a reference to n_security simply by referring to its name:

```
N_SECURITY lnv_security
ANY        lany_this
APPLICATIONlapp
N_APP_MGR  lnv_app_mgr
. . .
lapp = GetApplication()
lnv_app_mgr = lapp.TRIGGER DYNAMIC FUNCTION af_get_app_mgr()
lany_this = THIS
lnv_security = lnv_app_mgr.TRIGGER STATIC FUNCTION &
     RequestComponent( "Security",  lany_this )
. . .
```

This example assumes that an application manager is an instance variable in the application object.

An object memory manager is an example of an unnamed-object manager. Object memory man-

agers are responsible for creating and destroying other objects. Since more than one instance of any given class may be instantiated at any given time, it makes more sense to use the registration ID from the manager than it does to create unique names for each object. Suppose there is an object memory manager of class type n_object_memory_manager which is descendent from ASFComponentManager.

```
 /* local variables */
STRING                    ls_object_type
ANY                       lany_this
BOOLEAN                   lb_sole_use_only
BOOLEAN                   lb_persistent
N_OBJECT_MEMORY_MANAGER   lnv_obj_mgr
. . .
/* Code to get a pointer to the object memory manager and assign
     it to lnv_obj_mgr goes here */
. . .
/* set arguments and register object */
ls_object_type = "n_object"
lany_this = THIS
lb_sole_use_only = TRUE
lb_persistent = FALSE

ipros_registration_id = lnv_obj_mgr.TRIGGER STATIC FUNCTION &
     RegisterComponent( ls_object_type, aany_this,
lb_sole_use_only, lb_persistent )
```

Walking the Manager

There are times when it is necessary to walk the manager, that is, get the first registered object in the manager, then the next registered object, and so on. This technique is useful when it's necessary to apply an operation to all of the business entities registered with a business context.

The following example is a code segment in a class descendent from ASFBusinessContext. The business entity manager is attached to the business context using an instance variable, ipronv_business_entity_manager.

```
BOOLEAN    lb_found
ASFBUSINESSENTITYlnv_be
...

/* Move to the first object in the manager */
lb_found = ipronv_business_entity_manager.TRIGGER STATIC FUNCTION
     & MoveToFirst()

DO WHILE ( lb_found )

     /* Get the business object */
     lnv_be = ipronv_business_entity_manager.TRIGGER STATIC
        FUNCTION & GetComponent()

     /* Perform the operation */
     IF ( NOT IsNull( lnv_be ) ) THEN
        IF ( IsValid( lnv_be ) ) THEN
           lnv_be.TRIGGER STATIC FUNCTION UpdateData(
```

```
            END IF
         END IF

         /* Get the next object */
         lb_found = ipronv_business_entity_manager.TRIGGER STATIC
FUNCTION &
         MoveToNext(
) LOOP
...
```

In the example above, the same method (UpdateData()) is applied to each business entity. Alternatively, the business context can query each business entity about its state and exit the loop, as needed, as shown in the example below.

```
BOOLEAN       lb_found
BOOLEAN       lb_modified
ASFBUSINESSENTITYlnv_be

/* Move to the first object in the manager */
lb_found = ipronv_business_entity_manager.TRIGGER STATIC FUNCTION
      & MoveToFirst()

lb_modified = FALSE
DO WHILE ( lb_found )

      /* Get the business object */
      lnv_be = ipronv_business_entity_manager.TRIGGER STATIC
         FUNCTION & GetComponent()

      /* Perform the operation */
      IF ( NOT IsNull( lnv_be ) ) THEN
         IF ( IsValid( lnv_be ) ) THEN
            lb_modified = lnv_be.TRIGGER STATIC FUNCTION &
               nf_is_modified()
            IF ( lb_modified ) THEN
               lb_modified = TRUE
               EXIT
            END IF
         END IF
      END IF

      /* Get the next object */
      lb_found = ipronv_business_entity_manager.TRIGGER STATIC
         FUNCTION &
         MoveToNext()
LOOP
RETURN lb_modified
```

Using the Sole Use Only Attribute

"Sole use only" means that the object that requested registration of another object with the manager is the only object that can access the registered object. No other object would be allowed access to the registered object by the manager. If sole use only is turned off, then any object can access the registered object.

Use the "sole use only" attribute when it is inappropriate for other objects to utilize the registered object. A business context may register a transaction object with the Transaction Manager for sole use only. The following example shows how to register an object so that no other object in the application could gain accidental access to the registered transaction object.

```
/* Instance Variables */
ASFTransactionManager ipronv_trans_mgr
CONSTANT STRINGicpros_trans_obj_name = "Customer"
CONSTANT STRINGicpros_trans_obj_type = "n_trans"
. . .
/* code segment */
BOOLEAN   lb_return
BOOLEAN   lb_sole_use_only
BOOLEAN   lb_persistent
ANY       lany_this
. . .
/* register the transaction object */
aany_this = THIS
lb_sole_use_only = TRUE
lb_persistent = TRUE
lb_return = ipronv_trans_mgr.TRIGGER STATIC FUNCTION &
     Register( icpros_trans_obj_type, icpros_obj_name, aany_this,
        & lb_sole_use_only, lb_persistent )
```

Using the Persistent Attribute

When an object is registered with a manager, the object's registration count is set to one. Assuming that other objects can also attempt to register the object ("sole use only" is FALSE), each registration of the same object increments the registration count for the object by one. If an object is registered three times, then its registration count is three.

When an object is unregistered, its registration count is decremented by one. When the registration count for that object reaches zero and it is flagged as not persistent, then the object is automatically destroyed. If the object is flagged as persistent and the registration count reaches zero, then the registration is removed and the object is not destroyed. It is up to the developer to properly destroy unregistered, non-persistent objects.

In any case, when the manager is destroyed, all registered objects are destroyed whether they are persistent or not.

Component Managers

A component manager is a manager that can register any type of object. Both ASFComponentManager and ASFNamedComponentManager provide the base classes for component managers. They are not specialized for a specific set of tasks. These managers provide the basic methods of registration (Register(), UnRegistered()) and access (RequestComponent()).

When inheriting from ASFComponentManager or ASFnamedComponentManager, it is recommended that a specialized request method be implemented. An example of how to develop a request method is found by examining the ASFDataWindowManager class. The RequestDataWindow() method takes two arguments:

```
STRING   as_registration_name  // object registration name
ANY      aany_requestor        // requesting object
```

This method returns a reference pointer to an object of type ASFDataWindowControl.

```
/* RequestDataWindow Code */
DATAWINDOWCONTROLldwc

ldwc = this.Trigger Static Function &
      RequestComponent( as_registration_name, aany_requestor )

RETURN ldwc
/* End Function */
```

RequestDataWindow() method simply performs the type casting needed to use objects returned by managers and makes code using this method easier to understand.

If the manager handles a variety of objects of known types, then it is feasible to provide a request method for each type. For example, suppose a named-object manager handles objects of the following types:

n_datawindow
n_datastore
n_business_entity

Then the following request methods could be developed for the manager:

RequestDataWindow()
RequestDataStore()
RequestBusinessEntity()

Each of these methods would take the following arguments:

```
STRING   as_registration_name// object registration name
ANY      aany_requestor  // requesting object
```

If the manager is an unnamed-object manager, then the argument as_registration_name would be change to pass the registration ID:

```
STRING   as_registration_name  // object registration name
ANY      aany_requestor        // requesting object
```

Service Managers and Services

A service is an object that performs a single function for objects of a particular type. Any number of objects can access the service object, as needed. The ASFService class is the base class for all services using the AdAPT libraries. The ASFService class has one virtual method:

```
PUBLIC PerformService( aany_requestor : ANY : by ref ) : BOOLEAN
```

To provide a service, inherit from ASFService and override the PerformService() method. In the following example, the class n_service_window_resize provides a window resizing service. The class n_service_window_resize is inherited from ASFService.

```
/* PerformService() code which overrides
   the virtual method in the ancestor */
```

```
BOOLEAN  lb_return
STRING   ls_requestor_type
W_WINDOW lw_window

lb_return = FALSE

/* check if the requestor is of the right type */
ls_requestor_type = aany_requestor.TRIGGER STATIC FUNCTION &
   ClassName()

IF ( Upper( ls_requestor_type ) <> "W_WINDOW" ) THEN
   RETURN lb_return
END IF

/* Assign the argument to the local variable */
lw_window = aany_requestor

/* Resize the window */
/* <window resizing code> */

/* return */
RETURN lb_return
```

The PerformService() method should return TRUE, if it's successful; FALSE, otherwise.

Although service objects can be accessed directly, they are more useful when registered with a service manager and accessed through the manager. Direct access to the service object means that the requesting object must know the class type of the service object in order to call the PerformService() method. By using a service manager, the requesting object simply needs to know the registered name of the service object in order to have the service performed. This decoupling of the service object and the requesting object allows both to be reused in different contexts.

Suppose that there is a named service manager of type n_window_service_mgr, which is inherited from ASFNamedServiceManagerFacade. The following code may be in a window requiring it be resized:

```
/* Instance Variables */
N_WINDOW_SERVICE_MGR   ipronv_w_service_mgr

CONSTANT STRING icpros_serivce_name = "Resize Window"
CONSTANT STRING icpros_service_type = "n_service_window_resize"

. . .

/* Register the service */
STRING    ls_registration_id
ANY       lany_this
STRING    ls_service_name
STRING    ls_service_type

ls_registration_id = ipronv_w_service_mgr.TRIGGER STATIC FUNCTION
      & Register( icpros_service_name, icpros_service_type,
         lany_this & TRUE, FALSE )

IF ( ls_registration_id = "" ) THEN
```

```
   /*<some error code>*/
END IF
. . .

/* Request the service */
lb_return = ipronv_w_service_mgr.TRIGGER STATIC FUNCTION &
   RequestService
```

Application Manager

An Application Manager is a named component manager for handling application-wide components. The Application Manager provides a single object from which to get other components needed in the application. The ASFApplicationManager makes no assumptions about the type of components a particular application might need. For example, one application may require a certain type of security component, whereas another application requires a different security component. In either case, a security component specific to the application can be registered without making changes to the Application Manager code. Since application components are registered by name in the Application Manager, different security components can be registered using the same name in different applications. Alternatively, different security components, each for a specific type of security, can be registered in the Application Manager in one application using different names.

Another advantage to the ASFApplicationManager class is that any type of object may be registered at any time. This means that objects of many different types are managed by the Application Manager, for example, security, transactions, logging, etc.

The Application Manager can be instantiated anywhere that is available to other objects. One technique is to make the Application Manager a global variable. The rest of this section assumes an Application Manager class called n_app_mgr that is descendent from ASFApplicationManager.

```
GLOBAL N_APP_MGR gnv_app_mgr
```

A recommended approach is to make the Application Manager an instance variable of the Application object: This helps protect the Application Manager from misuse because access is through the Application object:

```
/* Instance Variables */
PROTECTED:
N_APP_MGR  ipronv_app_mgr
```

Making the Application Manager a part of the Application object implies three very useful methods:

```
af_create_app_components():BOOLEAN
af_destroy_app_components():BOOLEAN
af_get_application_manager():N_APP_MGR
```

The method af_create_app_components() is called in the Application object's Open event. This method is used to create all of the application components needed for a specific application:

```
/* Instance Variables in the Application Object */
PROTECTED:
N_APP_MGR  ipronv_app_mgr

CONSTANT STRING icpros_transaction_mgr_type = "N_TRANSACTION_MGR"
```

```
CONSTANT STRING icpros_security_type = "N_SECURITY"
CONSTANT STRING icpros_log_type = "N_LOG"

CONSTANT STRING icpros_transaction_mgr_name = "Transactions"
CONSTANT STRING icpros_security_name = "Security"
CONSTANT STRING icpros_log_name = "Log"
. . .
/* af_create_app_components() */

ANY                 lany_this
STRING              ls_return
BOOLEAN             lb_return

l_return = FALSE

/* Create the application Manager
ipronv_app_mgr = CREATE N_APP_MGR

/* Create the Transaction Manager */
lany_this = THIS
ls_return = ipronv_app_mgr.TRIGGER STATIC FUNCTION &
   Register( icpros_transaction_mgr_name,
      icpros_transaction_mgr_type, &
       TRUE, FALSE )
IF ( ls_return = "" ) THEN
   RETURN lb_return
END IF

/* Create the Security Component */
lany_this = THIS
ls_return = ipronv_app_mgr.TRIGGER STATIC FUNCTION &
   Register( icpros_security_name, icpros_security_type, &
      TRUE, FALSE
IF ( ls_return = "" ) THEN
   RETURN lb_return
END IF

/* Create the Log Component */
lany_this = THIS
ls_return = ipronv_app_mgr.TRIGGER STATIC FUNCTION &
   Register( icpros_log_name, icpros_log_type, &
      TRUE, FALSE )
IF ( ls_return = "" ) THEN
   RETURN lb_return
END IF

lb_return = TRUE
RETURN lb_return
```

The Transaction Manager, Security Component, and Log Component are now registered with the Application Manager and available to the rest of the application.

The method af_destroy_app_components() is called in the Application object's Close event. The registered components are destroyed simply by destroying the Application Manager:

```
/* af_destroy_app_components() */
IF ( NOT IsNull( ipronv_app_mgr ) ) THEN
   IF ( IsValid( ipronv_app_mgr ) ) THEN
      DESTROY ipronv_app_mgr
   END IF
END IF
```

Note: The purpose of checking the validity of the Application Manager is to ensure that there will not be a run-time error. It is a good idea to check the validity of any object before destroying it.

Access to the Application Manager is provided by the method af_get_application_manager():

```
/* af_get_application_manager():N_APP_MGR */
N_APP_MGR   lnv_app_mgr

/* check the validity of the Application Manager */
IF ( NOT IsNull( ipronv_app_mgr  ) ) THEN
   IF ( IsValid( ipronv_app_mgr ) ) THEN
      /* return the Application Manager Reference Pointer */
      RETURN ipronv_app_mgr
   END IF
END IF

/* return NULL object pointer, indicating an error */
SetNull(lnv_app_mgr)
RETURN lnv_app_mgr
```

The local variable lnv_app_mgr is used to return a NULL reference pointer if the Application Manager object is not valid.

Suppose that the application is called my_app. Requesting a reference pointer to a desired component is now a simple task:

```
/* Instance Variables in some class */
CONSTANT STRING icpros_security_name = "Security"
. . .
/* code segment in some class */
N_APP_MGR   lnv_app_mgr
N_SECURITY  lnv_security
ANY         lany_this
. . .
lnv_app_mgr = my_app.TRIGGER STATIC FUNCTION &
   af_get_application_manager()

lany_this = THIS
lnv_security = lnv_app_mgr.TRIGGER STATIC FUNCTION &
   RequestComponent( icpros_security_name, lany_this )
```

Alternatively, if the name of the application object is not known, then the PowerScript function GetApplication() can be used. In this case, it is necessary to use DYNAMIC function calls:

```
/* Instance Variables in some class */
CONSTANT STRING icpros_security_name = "Security"
. . .
/* code segment in some class */
```

```
N_APP_MGR   lnv_app_mgr
N_SECURITY  lnv_security
ANY         lany_this
. . .
lnv_app_mgr = GetApplication().TRIGGER DYNAMIC FUNCTION &
   af_get_application_manager()

lany_this = THIS
lnv_security = lnv_app_mgr.TRIGGER STATIC FUNCTION &
   RequestComponent( icpros_security_name, lany_this )
```

Care must be taken when using the DYNAMIC function call. If the function, in this case the method af_get_application_manager, does not exist in the Application object, a run-time error will occur.

Transaction Manager

The ASFTransactionManager class[4] is the base class for managing application transaction objects derived from ASFTransaction. The ASFTransactionManager class provides for connecting and disconnecting to the database:

```
PUBLIC ConnectToDB( as_trans_obj_name : STRING : by val,
     aany_requestor : ANY : by ref  ) : INTEGER

PUBLIC DisconnectFromDB( as_trans_obj_name : STRING : by val,
     aany_requestor : ANY : by ref ) : INTEGER
```

When an object of a class type descendent from ASFTransaction is registered with a transaction manager, a transaction registration name is returned. This registration name is used to access the transaction object to connect and disconnect. The registration name is also used to get a reference pointer to the transaction object itself:

```
PUBLIC RequestTransaction( as_trans_obj_name : STRING : by val,
     aany_requestor : ANY : by ref ) : ANY
```

For more information about ASFTransaction, see the AdAPT Technical Reference.

4. The ASFTransactionManager and ASFTransaction classes are undergoing changes to significantly enhance their utility. Obsolete functions will remain for several versions of AdAPT after the new classes have been release to allow developers time to switch.

Resource

Introduction

The AdAPT Resource class provides a common interface to any of the three most common methods of providing information during program execution - a Windows .INI file, the Windows registry or a database table.

Overview

The ASFResourceFacade provides a common public interface to the three different sources. Related resource information is stored in sections or collections. For example, environment information would be stored in the section 'Environment' which would be separate from the 'Printer' information. Each section includes one or more entries. Each entry consists of an entry name and entry value.

The formats of the Windows .INI file and the Windows Registry allow for the information to be mapped to the section, entry name and entry value fields. When using a database as a resource the developer must create a PowerBuilder DataWIndow object to map the information to these fields. The DataWIndow object can select the three columns of information from any table(s) from any database or external file supported by PowerBuilder. The only limitation is that the column order must be section name, entry name and entry value (the column name is ignored).

Advantages

- The application could easily be modified if it was determined, for example, that the Registry should be used instead of an .INI file.
- The application could determine during initialization which source to use, depending on availability.
- If the source is a database table, the type of database or the table name be changed by merely changing the PowerBuilder DataWIndow object used by the ASFResourceDatabaseAdapter.

Using a Windows .INI File As a Resource

The Windows operating system introduced the concept of an initialization (. INI) file to store information an application could retrieve and update during execution. It is an ASCII file in which related information is grouped in sections.

Format

```
[Section] entry=value
```

Example

```
[Microsoft Excel]
Comment=The open=/f lines load custom functions into the Paste
Function list.
Maximized=3
Basics=1
DefaultPath=D:\WINAPPS\EXCEL
```

Application Initialization

The following code will create the Resource object and register it with the Application Manager during the application initialization.

```
STRING ls_registration_name
ANY lany_object
ANY lany_requestor
BOOLEAN lb_sole_use_only
BOOLEAN lb_persistent
ASFResourceFacade lnv_resource

lany_requestor = THIS

ipronv_app_manager = CREATE ASFApplicationManager

/* Register INI file as Resource object */
lnv_resource = CREATE ASFResourceFacade
lnv_resource.TRIGGER STATIC FUNCTION &
     SetAdapter( "ASFResourceIniAdapter" )
lnv_resource.TRIGGER STATIC FUNCTION &
     OpenResource( "c:\apps\app.ini" )
lany_object = lnv_resource
ls_registration_name = "Application INI file"
lb_sole_use_only = FALSE
lb_persistent = TRUE
ipronv_app_manager.TRIGGER STATIC FUNCTION &
     Register( lany_object, ls_registration_name, &
     lany_requestor, lb_sole_use_only, lb_persistent )
```

Application Usage

The following code shows how database information could be stored in an . INI file and retrieved during program execution. If the database information changes, only the .INI file needs to be updated. It is not necessary to change the application.

```
ANY lany_this
STRING ls_value
STRING ls_resource_name
ASFApplicationManager lnv_app_mgr
ASFResourceFacade lnv_resource

lany_this = THIS
ls_resource_name = "Application INI file"

lnv_app_mgr = GetApplication( ).TRIGGER DYNAMIC FUNCTION &
     af_get_application_manager( )

/* Get the resource object */
lnv_resource = lnv_app_mgr.TRIGGER STATIC FUNCTION &
     RequestComponent( ls_resource_name, lany_this )
IF ( IsNull( lnv_resource ) ) THEN
     IF ( NOT IsValid( lnv_resource ) ) THEN
        RETURN
```

```
      END IF
END IF

/* Retrieve the values from the Resource */
lnv_resource.TRIGGER STATIC FUNCTION SetSection( "Database" )
ls_resource_name = "DBMS"
ls_value = lnv_resource.TRIGGER STATIC FUNCTION &
      RetrieveValue( ls_resource_name )
```

Using the Windows Registry As a Resource

The Windows operating system includes a Registry to store information that an application can retrieve and update during execution. The registry contains folders which may contain one or more keys and its value.

Application Initialization

The following code creates the Resource object and register it with the Application Manager during the application initialization.

```
STRING ls_registration_name
ANY lany_object
ANY lany_requestor
BOOLEAN lb_sole_use_only
BOOLEAN lb_persistent
ASFResourceFacade lnv_resource

lany_requestor = THIS

ipronv_app_manager = CREATE ASFApplicationManager

/* Register Windows Registry as Resource object */
lnv_resource = CREATE ASFResourceFacade
lnv_resource.TRIGGER STATIC FUNCTION &
      SetAdapter( "ASFResourceRegistryAdapter" )
lnv_resource.TRIGGER STATIC FUNCTION &
      OpenResource( "HKEY_LOCAL_MACHINE\SOFTWARE\app" )
lany_object = lnv_resource
ls_registration_name = "Application Registry"
lb_sole_use_only = FALSE
lb_persistent = TRUE
ipronv_app_manager.TRIGGER STATIC FUNCTION &
      Register( lany_object, ls_registration_name, &
      lany_requestor, lb_sole_use_only, lb_persistent )
```

Application Usage

The following code shows how database information could be stored in the registry and retrieved during program execution. If the database information changes, only the registry needs to be updated. Thus no changes are required to the application.

```
ANY lany_this
STRING ls_value
STRING ls_resource_name
```

```
ASFApplicationManager lnv_app_mgr
ASFResourceFacade lnv_resource

lany_this = THIS
ls_resource_name = "Application Registry"

lnv_app_mgr = GetApplication( ).TRIGGER DYNAMIC FUNCTION &
     af_get_application_manager( )

/* Get the resource object */
lnv_resource = lnv_app_mgr.TRIGGER STATIC FUNCTION &
     RequestComponent( ls_resource_name, lany_this )
IF ( IsNull( lnv_resource ) ) THEN
     IF ( NOT IsValid( lnv_resource ) ) THEN
        RETURN
     END IF
END IF

/* Retrieve the values from the Resource */
lnv_resource.TRIGGER STATIC FUNCTION SetSection( "Database" )
SQLCA.DBMS = lnv_resource.TRIGGER STATIC FUNCTION &
     RetrieveValue( "DBMS" )
```

Using a Database Table As a Resource

A relational table in a database supported by PowerBuilder may be used to store information that an application can retrieve and update during execution. The resource object uses a PowerBuilder DataWindow object to select the information from the database table. The order of the columns selected is important, but the column names are ignored. The first column is the section or group identifier. The second column contains the entry or key name. The third column contains the value.

Application Initialization

The following code will create the Resource object and register it with the Application Manager during the application initialization.

```
STRING ls_registration_name
ANY lany_object
ANY lany_requestor
BOOLEAN lb_sole_use_only
BOOLEAN lb_persistent
ASFResourceFacade lnv_resource

lany_requestor = THIS

ipronv_app_manager = CREATE ASFApplicationManager

/* Register INI file as Resource object */
lnv_resource = CREATE ASFResourceFacade
lnv_resource.TRIGGER STATIC FUNCTION &
     SetAdapter( "ASFResourceDatabaseAdapter" )
lnv_resource.TRIGGER STATIC FUNCTION &
     SetTransactionObject( SQLCA )
lnv_resource.TRIGGER STATIC FUNCTION &
```

```
      OpenResource( "d_resource" )
lany_object = lnv_resource
ls_registration_name = "Application Database"
lb_sole_use_only = FALSE
lb_persistent = TRUE
ipronv_app_manager.TRIGGER STATIC FUNCTION &
      Register( lany_object, ls_registration_name, &
      lany_requestor, lb_sole_use_only, lb_persistent )
```

Application Usage

The following code shows how printer information could be stored in a database table and retrieved during program execution. If the printer information changes, only the table needs to be updated. It is not necessary to change the application.

```
ANY lany_this
STRING ls_value
STRING ls_resource_name
ASFApplicationManager lnv_app_mgr
ASFResourceFacade lnv_resource

lany_this = THIS
ls_resource_name = "Application Database"

lnv_app_mgr = GetApplication( ).TRIGGER DYNAMIC FUNCTION &
      af_get_application_manager( )

/* Get the resource object */
lnv_resource = lnv_app_mgr.TRIGGER STATIC FUNCTION &
      RequestComponent( ls_resource_name, lany_this )
IF ( IsNull( lnv_resource ) ) THEN
      IF ( NOT IsValid( lnv_resource ) ) THEN
         RETURN
      END IF
   END IF

/* Retrieve the values from the Resource */
lnv_resource.TRIGGER STATIC FUNCTION SetSection( "Printer" )
ls_value = lnv_resource.TRIGGER STATIC FUNCTION &
      RetrieveValue( "Name" )
```

Using Multiple Resources

Introduction

Applications use resources to store and retrieve default values, user preferences and application state information. Resources can be divided into two groups - user community resources and user specific resources.

User community resources usually include application default values. These resources usually have the following characteristics:

- Must be available to all users using the application.

- Require easy maintenance by an administrator.
- Require no redistribution after modification.
- Values are modified infrequently.
- Resource is usually read-only.

A Windows .INI file located on a server drive or a database located on a server are best suited to include user community resource values.

User specific resources usually include user preferences and application state information. These resources usually have the following characteristics:

- Available to a single user or machine.
- Changes require redistribution to all users.
- Resource is updateable.

A Windows .INI file located on a local drive, the Windows registry, or a database located on a local drive are suited to include user specific resource values.

Example

This example illustrates how an application might use an application manager to manage a user community resource and a user specific resource. The user community resource is a database table while the user specific resource is a Windows .INI file. The type of resource can be easily changed given the characteristics mentioned in the Introduction.

A PowerBuilder DataWindow object is created to retrieve resource entries from a database table that contains application default values. This table would contain any resource values that are required by the application when used by any user.

The SELECT statement should include the columns for the section name, entry name, and entry value. The column order is important, however the column names are ignored.

```
SELECT section, entry, value from dba.resource WHERE app = 'pim';
```

A Windows .INI file is created that would hold values specific to the user, such as personal preferences, how to access the user community resource, etc.

```
[User community resource]
DBMS=ODBC
DBPARM=Connectstring='DSN=PIM Resource'
...
[User Preferences]
...
```

The following code is coded in an application function, af_create_app_components( ), which is called during the application's Open event. This code will create the two resource objects and register them with the application manager.

```
STRING ls_registration_name
ANY lany_object
ANY lany_requestor
```

```
BOOLEAN lb_sole_use_only
BOOLEAN lb_persistent
ASFResourceFacade lnv_resource

lany_requestor = THIS

ipronv_app_manager = CREATE ASFApplicationManager

/* Register INI file as Resource object */
lnv_resource = CREATE ASFResourceFacade
lnv_resource.TRIGGER STATIC FUNCTION &
     SetAdapter( "ASFResourceIniAdapter" )
lnv_resource.TRIGGER STATIC FUNCTION &
     OpenResource( "c:\pim\pim.ini" )
lany_object = lnv_resource
ls_registration_name = "PIM user specific resource"
lb_sole_use_only = FALSE
lb_persistent = TRUE
ipronv_app_manager.TRIGGER STATIC FUNCTION &
     Register( lany_object, ls_registration_name, &
     lany_requestor, lb_sole_use_only, lb_persistent )

/* Register Database table as Resource object */
lnv_resource = CREATE ASFResourceFacade
lnv_resource.TRIGGER STATIC FUNCTION &
     SetAdapter( "ASFResourceDatabaseAdapter" )
lnv_resource.TRIGGER STATIC FUNCTION &
     SetTransactionObject( SQLCA )
lnv_resource.TRIGGER STATIC FUNCTION &
     OpenResource( "d_resource" )
lany_object = lnv_resource
ls_registration_name = "PIM user community resource"
lb_sole_use_only = FALSE
lb_persistent = TRUE
ipronv_app_manager.TRIGGER STATIC FUNCTION &
     Register( lany_object, ls_registration_name, &
     lany_requestor, lb_sole_use_only, lb_persistent )
```

Throughout the application, the resources may be accessed by requesting them from the application manager using their registration names.

```
ANY lany_this
STRING ls_style, ls_size
ASFApplicationManager lnv_app_mgr
ASFResourceFacade lnv_resource

lany_this = THIS

/* Get the application manager */
lnv_app_mgr = GetApplication( ).TRIGGER DYNAMIC FUNCTION &
     af_get_application_manager( )

/* Request the resource object from the application manager */
lnv_resource = lnv_app_mgr.TRIGGER STATIC FUNCTION &
```

```
      RequestComponent( "PIM user community resource", lany_this )
IF ( IsNull( lnv_resource ) ) THEN
      RETURN
END IF
IF ( NOT IsValid( lnv_resource ) ) THEN
      RETURN
END IF

/* Retrieve the Font resource values */
lnv_resource.TRIGGER STATIC FUNCTION &
      SetSection( "Font" )
ls_style = lnv_resource.TRIGGER STATIC FUNCTION &
      RetrieveValue( "Style" )
ls_size = lnv_resource.TRIGGER STATIC FUNCTION &
      RetrieveValue( "Size" )
```

Controllers

nonvisualobject

ASFObject

ASFSystem

ASFController

The ASFController class provides the base class for application controllers. A controller is a class that is a control mechanism for some set of classes. Although a controller provides a public interface to the class set, it is not a facade class. Where a facade encapsulates some subsystem of classes, a controller takes messages and interprets them. The controller then directs objects to perform certain tasks. Controllers do not necessarily require external messages to perform tasks. In fact, a controller may never communicate with classes external to the set of controlled classes throughout the life of the controller.

A messaging system decouples the controller and objects external to the controller context. The controller object is responsible for decoding the message and responding accordingly. This means that the basic messaging mechanism can be placed in an ancestor to the controller class and the class is responsible for the specific decoding implementation.

The ASFController class provides a simple messaging system similar to that of the ASFBusinessContext class (See Business Contexts; ASFBusinessContext class, AdAPT Technical Reference)

THE USER INTERFACE PARTITION

The User Interface Partition provides the means by which the user manipulates the system. PowerBuilder, and many third-party vendors, provide a wide range of UI mechanisms and controls. Unfortunately, they do not offer well-encapsulated controls. In fact, the class structures often encourage developing business rules within UI controls windows, which violates good object-oriented design and significantly limits object reuse.

This part focuses on the following User Interface tools:

- Windows
- Menus
- Controls

Introduction

PowerBuilder and other third party libraries provide many different GUI controls. The AdAPT GUI classes do not replace these other controls, just enhance their object oriented features. The window, menu and control classes in the AdAPT User Interface Partition are inherited from the standard PowerBuilder classes. Each class should be encapsulated.

Like all classes in the AdAPT library, these classes should follow ASF's partitioning guidelines. These guidelines stipulate that any business rules should stay in an object within the Problem Domain Partition and any code for external systems should stay within the System Management Partition. This provides the ability to switch out the style of user interface without having to remove the application's business logic from a menu's or control's events.

Any object within the User Interface Partition would use a standard method within the application to interface with the objects in the System Management and Problem Domain Partitions, such as the Model-View-Controller framework of Smalltalk or the Publisher/Subscriber pattern from Coad. See the topic Introduction to Model-View-Controller Framework for more information on using a Model-View-Controller framework.

Windows

- **window**
  - **ASFWindow**
    - **ASFContainerWindow**
    - **ASFViewWindow**

The two most widely used types of windows are the Container window and the View window. The Container window is a Container of child windows. The child windows are represented as icons or lists within the Container window. Examples of container windows are a MDI frame, a project window and a workspace window. The view windows are other windows within an application, such as user documents and forms.

The ASFContainerWindow is a base class which could be used to inherit a project style window. This new window class would manage the child windows according to project window guidelines.

The ASFViewWindow is a base class from which to inherit a window class that is the standard for all applications. It would include any methods that interface with the architectural framework, i.e. a Model-View-Controller framework. The methods in the window class would be the 'conduit' for all messages between the User Interface Partition and the System Management Partition or the Problem Domain Partition.

Menus

- **menu**
  - **ASFMenu**
    - **ASFContainerMenu**
    - **ASFViewMenu**

AdAPT library includes ASFContainerMenu and ASFViewMenu. These are base classes, from which menus are inherited to be used by the descendants of the ASFContainerWindow and ASFViewWindow, respectively.

Traditionally, the menu events contain code to actually perform the requested action. However in order to be able to reuse code and to achieve partitioning, the menu events should make the request to its parent window which forwards the message to the architectural framework to it can be processed.

Controls

The AdAPT control classes are inherited from the PowerBuilder controls. The goal is to encapsulate the PowerBuilder controls and provide a consistent interface, such as having the right mouse button behaves the same for all controls. Also, corporate wide control classes are easily created. A control component is a set of control objects that has a single public interface. Properly designed, control components can be easily reused. For example, a GroupBox control class might contain OK, Cancel, and Help button controls that have been standardized for all applications.

The steps in using the AdAPT control classes are:

1. Create a control inheriting from the AdAPT control class.

2. Add methods which implement any corporate standards.

3. Place the new control on the application's window.