Getting Started With iPlanet UDS

iPlanet™ Unified Development Server

Version 5.0

August 2001

Contents

List of Figures

List of Procedures

Preface

Welcome to *Getting Started With iPlanet UDS*. This manual presents a tutorial that walks you through the basic steps required to create an iPlanet UDS application. When you have completed this tutorial, you will be familiar with:

- the iPlanet UDS Workshops
- the architecture of an iPlanet UDS application
- the process of creating an iPlanet UDS application
- important and common features of iPlanet UDS programming, including the use of service object, events, and exception handling.

This preface contains the following sections:

Product Name Change

Forte 4GL has been renamed the iPlanet Unified Development Server. You will see full references to this name, as well as the abbreviations iPlanet UDS and UDS.

Audience for This Guide

Getting Started With iPlanet UDS is intended for application developers. We assume that you:

- have programming experience
- are familiar with your particular window system
- are familiar with SQL and your particular database management system

Organization of This Guide

The following table briefly describes the contents of each chapter:

Chapter	**Description**
Chapter 1, "Introduction"	Introduces the tutorial and discusses the architecture of the tutorial application.
Chapter 2, "Defining the Logical Application"	Describes the steps required to define the logical application and provides an overview of the iPlanet UDS Workshops.
Chapter 3, "Creating the Service Side"	Describes the steps required to create the service side of the application, including defining business classes, service objects, and working with events.
Chapter 4, "Creating the User Interface"	Explains how to create the user interface of the application using the Window Workshop.
Chapter 5, "Handling Exceptions"	Describes how to create the exception classes as well as the code to raise and handle the exception.
Chapter 6, "Testing and Running the Application"	Describes how to partition and then test run the application.

Text Conventions

This section provides information about the conventions used in this document.

Format	Description
italics	Italicized text is used to designate a document title, for emphasis, or for a word or phrase being introduced.
`monospace`	Monospace text represents example code, commands that you enter on the command line, directory, file, or path names, error message text, class names, method names (including all elements in the signature), package names, reserved words, and URLs.
ALL CAPS	Text in all capitals represents environment variables (FORTE_ROOT) or acronyms (UDS, JSP, iMQ). Uppercase text can also represent a constant. Type uppercase text exactly as shown.
Key+Key	Simultaneous keystrokes are joined with a plus sign: Ctrl+A means press both keys simultaneously.
Key-Key	Consecutive keystrokes are joined with a hyphen: Esc-S means press the Esc key, release it, then press the S key.

Other Documentation Resources

In addition to this guide, there are additional documentation resources, which are listed in the following sections. The documentation for all iPlanet UDS products (including Express, WebEnterprise, and WebEnterprise Designer) can be found on the iPlanet UDS Documentation CD. Be sure to read "Viewing and Searching PDF Files" on page 13 to learn how to view and search the documentation on the iPlanet UDS Documentation CD.

iPlanet UDS documentation can also be found online at http://docs.iplanet.com/docs/manuals/uds.html.

The titles of the iPlanet UDS documentation are listed in the following sections.

iPlanet UDS Documentation

- *A Guide to the iPlanet UDS Workshops*
- *Accessing Databases*
- *Building International Applications*
- *Escript and System Agent Reference Guide*
- *Fscript Reference Guide*
- *Getting Started With iPlanet UDS*
- *Integrating with External Systems*
- *iPlanet UDS Java Interoperability Guide*
- *iPlanet UDS Programming Guide*
- *iPlanet UDS System Installation Guide*
- *iPlanet UDS System Management Guide*
- *Programming with System Agents*
- *TOOL Reference Guide*
- *Using iPlanet UDS for OS/390*

Express Documentation

- *A Guide to Express*
- *Customizing Express Applications*
- *Express Installation Guide*

WebEnterprise and WebEnterprise Designer Documentation

- *A Guide to WebEnterprise*
- *Customizing WebEnterprise Designer Applications*
- *Getting Started with WebEnterprise Designer*
- *WebEnterprise Installation Guide*

Online Help

When you are using an iPlanet UDS development application, press the F1 key or use the Help menu to display online help. The help files are also available at the following location in your iPlanet UDS distribution: `FORTE_ROOT/userapp/forte/cln/*.hlp`.

When you are using a script utility, such as Fscript or Escript, type help from the script shell for a description of all commands, or help `<command>` for help on a specific command.

iPlanet UDS Example Programs

A set of example programs is shipped with the iPlanet UDS product. The examples are located in subdirectories under `$FORTE_ROOT/install/examples`. The files containing the examples have a `.pex` suffix. You can search for TOOL commands or anything of special interest with operating system commands. The `.pex` files are text files, so it is safe to edit them, though you should only change private copies of the files.

Viewing and Searching PDF Files

You can view and search iPlanet UDS documentation PDF files directly from the documentation CD-ROM, store them locally on your computer, or store them on a server for multiuser network access.

NOTE You need Acrobat Reader 4.0+ to view and print the files. Acrobat Reader with Search is recommended and is available as a free download from http://www.adobe.com. If you do not use Acrobat Reader with Search, you can only view and print files; you cannot search across the collection of files.

➤ **To copy the documentation to a client or server**

1. Copy the `doc` directory and its contents from the CD-ROM to the client or server hard disk.

   You can specify any convenient location for the `doc` directory; the location is not dependent on the iPlanet UDS distribution.

2. Set up a directory structure that keeps the `udsdoc.pdf` and the `uds` directory in the same relative location.

   The directory structure must be preserved to use the Acrobat search feature.

NOTE To uninstall the documentation, delete the `doc` directory.

➤ **To view and search the documentation**

1. Open the file `udsdoc.pdf`, located in the `doc` directory.
2. Click the Search button at the bottom of the page or select Edit > Search > Query.
3. Enter the word or text string you are looking for in the Find Results Containing Text field of the Adobe Acrobat Search dialog box, and click Search.

   A Search Results window displays the documents that contain the desired text. If more than one document from the collection contains the desired text, they are ranked for relevancy.

   NOTE For details on how to expand or limit a search query using wild-card characters and operators, see the Adobe Acrobat Help.

   ---

4. Click the document title with the highest relevance (usually the first one in the list or with a solid-filled icon) to display the document.

   All occurrences of the word or phrase on a page are highlighted.

5. Click the buttons on the Acrobat Reader toolbar or use shortcut keys to navigate through the search results, as shown in the following table:

Toolbar Button	Keyboard Command
Next Highlight	Ctrl+]
Previous Highlight	Ctrl+[
Next Document	Ctrl+Shift+]

   To return to the `udsdoc.pdf` file, click the Homepage bookmark at the top of the bookmarks list.

6. To revisit the query results, click the Results button at the bottom of the `udsdoc.pdf` home page or select Edit > Search > Results.

Chapter 1

Introduction

The sample program that you will create in the following tutorial is an extremely simple application. Its aim is to make you familiar with the iPlanet UDS Workshops and to introduce a number of features that are important in developing iPlanet UDS applications. These features include:

- the use of service objects
- the use of events
- exception handling
- the partitioning of an application

You will be writing the application in TOOL, the iPlanet UDS object-oriented, scripting language. You do not have to know TOOL to use this tutorial. For complete information about TOOL, see the *TOOL Reference Guide*.

This chapter introduces you to the tutorial and discusses the architecture of the tutorial application.

Getting Started

Before you can use this tutorial, you must have iPlanet UDS set up to run in distributed mode. This tutorial assumes that there is a repository you can connect to and that you specified its name in the iPlanet UDS Control Panel. It would also be helpful if you could clear your desktop so that your view of iPlanet UDS windows and dialogs is not cluttered.

Menu commands This tutorial represents menu commands in the following way: Choose File > New…. This means that you should select the menu indicated by the first bold-faced word, and then continue to select the next bold-faced word as an item from that menu. Thus, the general form is the following:

Choose Menu > MenuItem > MenuItem…

Comments in code There are several ways to enter comments in iPlanet UDS code. One way is to prefix two dashes before the comment. For example:

```
-- This is a comment.
self.open();
```

Sample code shown in the tutorial includes comments. You do not have to type these comments when you enter the code into your own application.

Stages of Application Development

There are four distinct stages in developing an iPlanet UDS application:

- designing the application

  During this stage, you model the application problem and decide how to divide the parts of a distributed application into separate interconnected services.

- defining the logical application

  During this stage, you create classes grouped into different projects. Classes represent the programming logic of the application. The application definition stage does not depend on the details of the target deployment environment.

- partitioning the application

  During this stage, you generate application distribution files. To do this you specify the target iPlanet UDS environment and partition the logical application in that environment. The application is not actually installed at this stage; you are, rather, creating an installation plan that iPlanet UDS uses to create installable files.

- deploying the application

  During this stage, the system manager deploys (installs) and manages the application in a deployment environment.

When you create an iPlanet UDS application, you are concerned with completing the first three stages of this process. These stages are covered in the following tutorial.

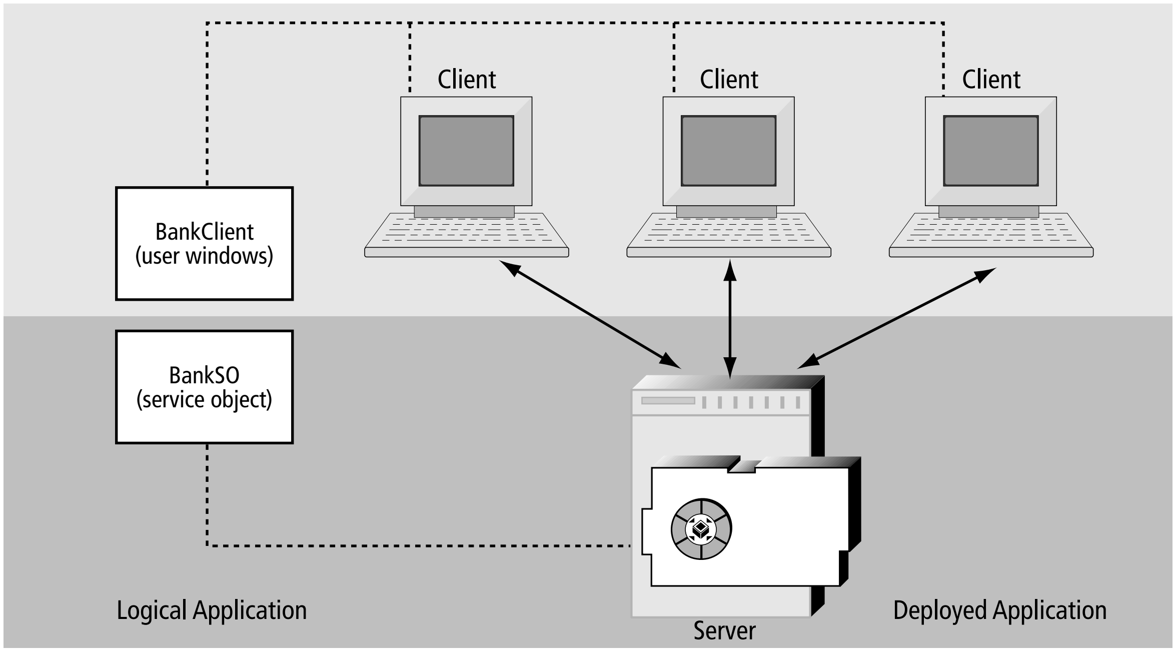
Designing the Application

An iPlanet UDS application is distributed across clients and servers by separating the logical application into discrete pieces called partitions. A *partition* is a separate operating system process running on client or server machines. All communication between partitions and across the nodes where they reside is handled by the underlying iPlanet UDS runtime system. While you do not have to do anything to make this communication happen, you do need to design the logical parts of your application to correspond to its eventual deployment on different machines.

The Architecture of an iPlanet UDS Application

Figure 1-1 shows how the logical parts of our sample application correspond to the partitions that are eventually installed on client and server nodes.

Figure 1-1 Logical and Deployed Applications



In a typical iPlanet UDS application, the client part of the application is responsible for display processing, while the server part is responsible for intensive business computation and for the enforcement of business rules. After the application is deployed, many clients can use the service provided by one or more server partitions. The typical application architecture consists of three types of classes:

Window classes Interact with the end user, display data to the user, allow the user to update data, and specify the logical flow of the user interface. Objects based on these classes normally reside on client partitions.

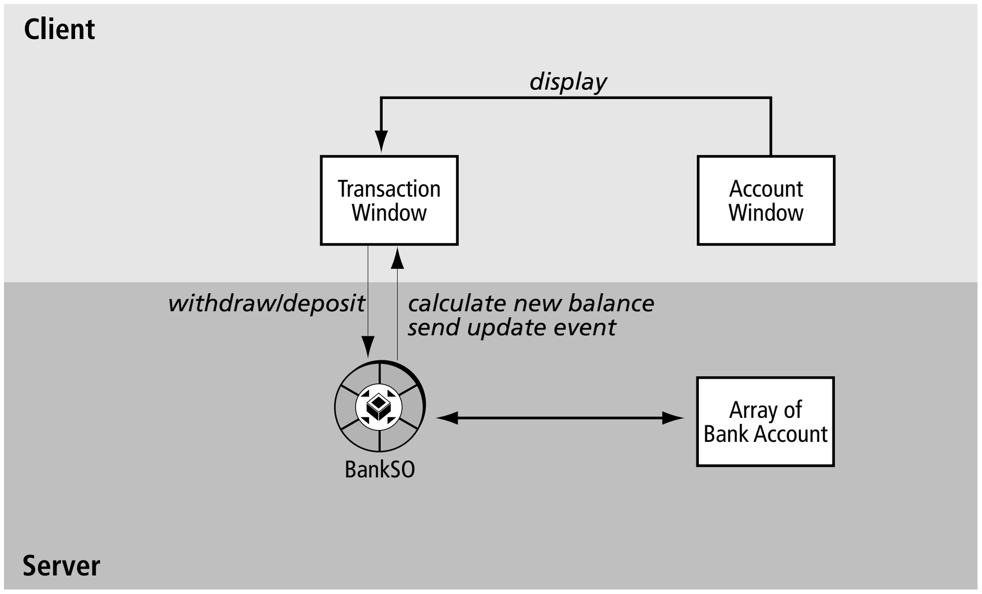
Service object classes Define the services provided to application clients. These classes define the business logic, access data stored in a database, and call out to external systems. Objects based on these classes normally reside on server partitions.

Business classes Define the data that is manipulated by the application. The data associated with these objects is often stored persistently in a database or on server partitions and is passed between the client and the server partitions.

The Architecture of the Sample Application

In accordance with the model just described, the application you will create is organized as a client and server as shown in Figure 1-2.

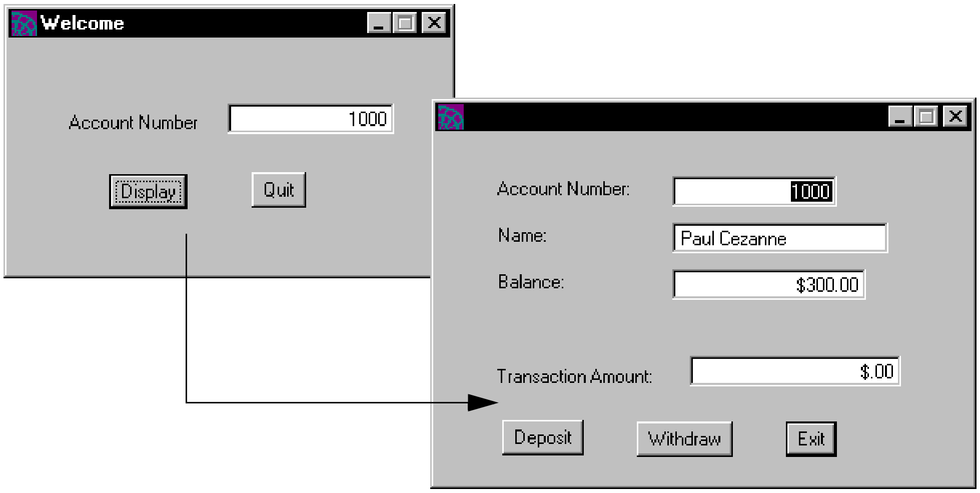
Figure 1-2 Architecture of the Sample Application



Client side The client side includes logic that displays windows, allowing the user to obtain information about her account, deposit and withdraw money, and view updated information about the account.

The user windows for your application will look like this.

Figure 1-3 User Windows for Sample Application



Server side The server side stores account data and defines methods that retrieve account information, calculate new account balances, and update the account information that is displayed to the user.

Service objects The server side uses a special kind of object, called a service object, to make these services available to the client. A *service object* is a named object that you can reference globally from any method in your application. In Figure 1-2, this object is shown as BankSO. The service object you will create in this application includes two methods:

- GetAccountData, which returns information for an individual account
- UpdateAcct, which is called (following a deposit or withdrawal) to calculate a new account balance and to update the account information shown in the user's window

This tutorial will discuss the special characteristics of service objects when you create one, later in this tutorial.

Chapter 2

Defining the Logical Application

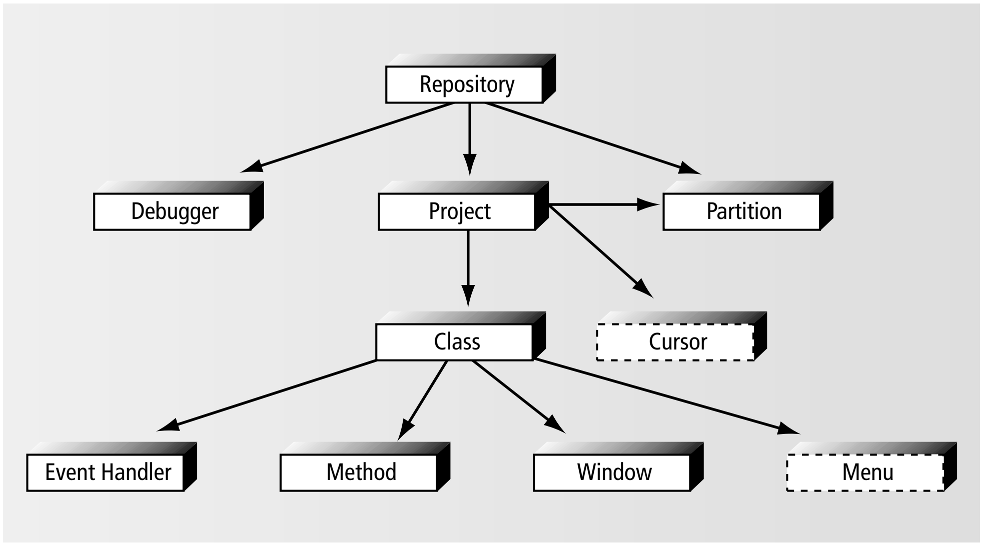
This chapter describes the steps required to define the logical application. It also provides an overview of the iPlanet UDS workshops, which you will then use to:

- open a workspace
- create your application's projects
- create service object classes and service objects
- create the user interface
- make a distribution

Working in the iPlanet UDS Workshops

You build an iPlanet UDS application using the iPlanet UDS Workshops, a development environment that you use to create application components and write TOOL code. Figure 2-1 shows the iPlanet UDS Workshops and their interrelation.

Figure 2-1 The iPlanet UDS Workshops

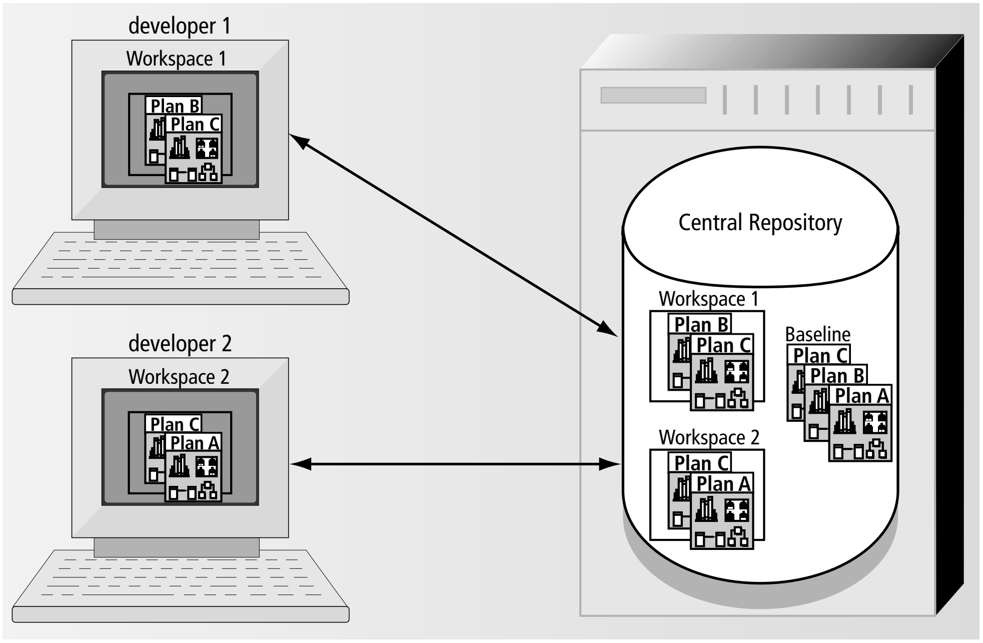


The arrows and the relative positioning of the workshops to one another indicates the path you must traverse to access a particular workshop. The Repository Workshop is the main entry point to the iPlanet UDS workshops. Thus, to open the Class Workshop, you must open the Repository Workshop, and then the Project Workshop. You will be working with all but the workshops with a dotted outline in this tutorial.

Creating a Workspace

iPlanet UDS provides a central development repository that allows a team of developers to synchronize their work, when each is creating a different part of the same application. The latest version of each project checked into the repository forms the *system baseline* for the repository. Each developer works in a *workspace*, which affords her a partial view of the total contents of the repository. Figure 2-2 shows how each developer views a subset of the files in a central repository.

Figure 2-2 Working with Repositories



This tutorial focuses on an individual development project, so it does not explore the subject of a shared repository. Instead, you will be creating your application in a private repository, which is a type of repository that is designed for independent development.

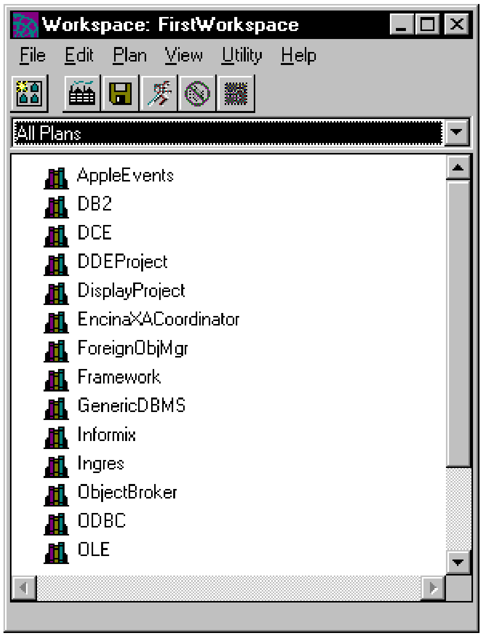
For information about using repositories and workspaces in a team environment, see *A Guide to the iPlanet UDS Workshops*. This tutorial assumes that there is a repository you can connect to and that you specified its name in the iPlanet UDS Control Panel.

In this part of the tutorial, you will open a workspace and create projects in it.

➤ To open a workspace

1. Launch iPlanet UDS distributed if you have not already done so.

   If you have not yet worked with the iPlanet UDS repositories, the application opens a window titled FirstWorkspace. Otherwise iPlanet UDS opens the last workspace in which you have been working.

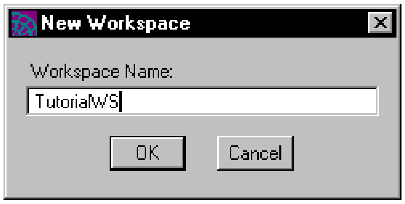


AppleEvents
DB2
DCE
DDEProject
DisplayProject
EncinaXACoordinator
ForeignObjMgr
Framework
GenericDBMS
Informix
Ingres
ObjectBroker
ODBC
OLE

FirstWorkspace contains the system libraries shipped with iPlanet UDS. Libraries are indicated by the books icon. It is recommended that you not use this default workspace to develop your iPlanet UDS application. You will be creating your own workspace in the next steps.

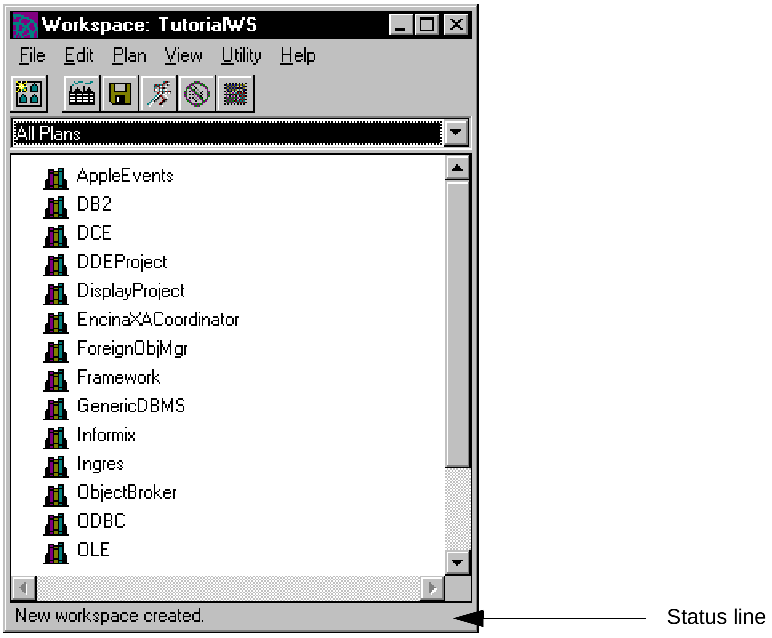
2. Choose the File > New Workspace… command.

iPlanet UDS displays a dialog that you use to specify the name of the workspace you want to create.



3. Enter `TutorialWS` for the workspace name and click OK.

iPlanet UDS displays the TutorialWS workspace in the Repository Workshop window (see next figure) and displays the message "New workspace created" on the status line of the Repository Workshop.



Your new workspace also contains the system libraries included in FirstWorkspace. These libraries are included by default for every iPlanet UDS workspace, and supply the basic functionality to create an iPlanet UDS application.

Three of the most commonly used system libraries are:

- Framework library, which contains foundation classes for building your application
- Display library, which contains classes for creating windows
- GenericDBMS library, which contains classes for accessing a database management system and its specific functions

iPlanet UDS also provides additional system libraries, for example CORBA and OLE, which allow you to integrate iPlanet UDS applications with external systems.

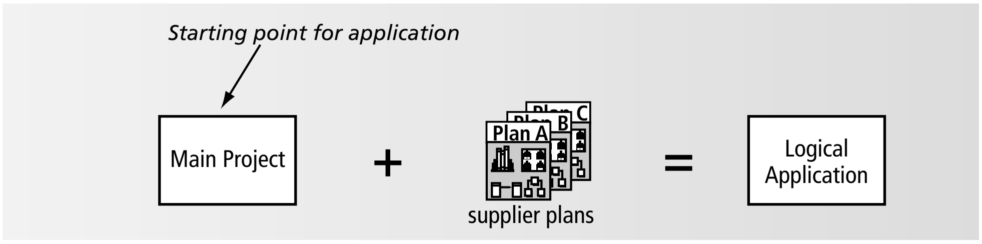
You can import additional libraries or plans into the workspace by choosing the Plan > Import command (or Plan > Include Public if the plan or library is already in the repository). You should not delete system libraries from your workspace, even if you do not plan on using them in your application.

Creating Projects

You use the Repository Workshop to develop an application in separate modules called plans. A *plan* is a project, a library, a business model, or an application model. In this tutorial, you will just be working with projects and libraries.

A *logical application* is a collection of plans. One plan is the main project; the rest of the plans are called supplier plans. The project that marks the starting point for your application is called the *main project*. Typically, this project contains the window classes that provide the user interface. A *supplier plan* is another project or library that you include as part of your project. All elements of supplier plans are available to your current project as though the elements were all part of one big project. Figure 2-3 shows the relationship between the main project and its supplier plans.

Figure 2-3 Main Project and Supplier Plans



An iPlanet UDS *project* is a collection of classes and other components. Classes include window classes or non-window classes. Components include service objects, constants, and cursors (used for database queries).

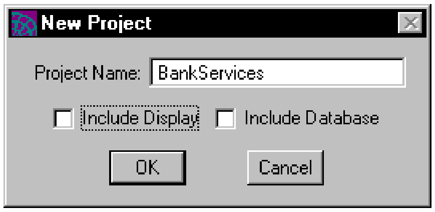
In this section of the tutorial you will create two projects for your application. The first project, called BankServices, will contain the classes used for retrieving and updating data. The second project, called BankClient, will contain the window classes for your application.

➤ **To create the BankServices project**

1. Choose the Plan > New Project… command.

   iPlanet UDS displays the New Project dialog, which allows you to specify the name of the new project and, optionally, to include the Display library or the GenericDatabase library. The Framework library is always included as a supplier plan.

2. Enter `BankServices` in the Project Name field.



New Project

Project Name: BankServices

☐ Include Display ☐ Include Database

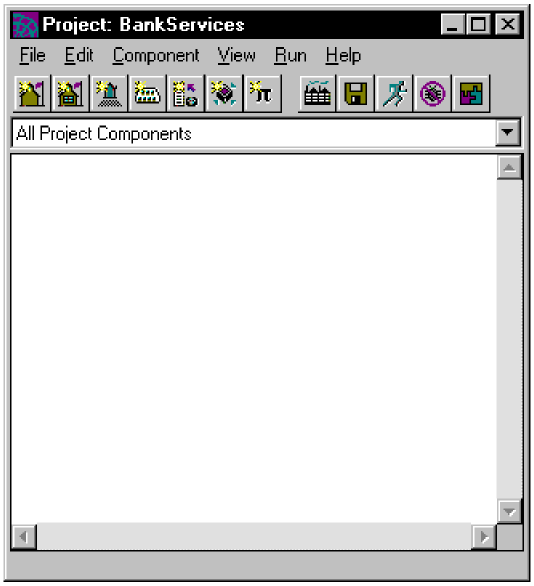
   By default all projects include both the Display and the GenericDatabase libraries. The BankServices project does not need either of these libraries.

3. Disable the Include Database and the Include Display toggles.

4. Click OK.

iPlanet UDS opens the Project Workshop for the BankServices project.

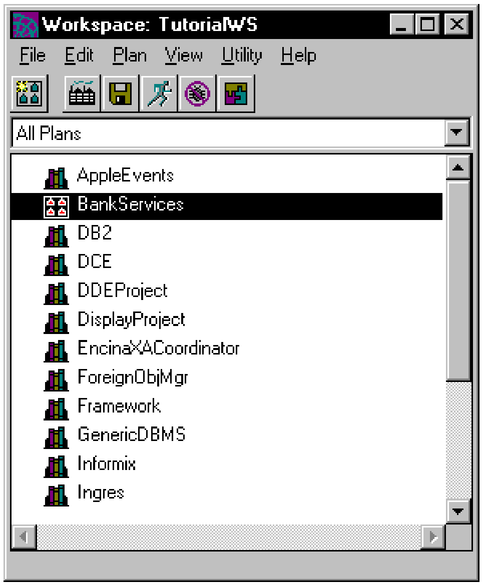
.



5. Close the BankServices project window using the appropriate control for your windowing system.

Take another look at your workspace window. The window, shown in Figure 2-4, now contains the BankServices project in addition to the libraries that were listed before.

Figure 2-4 BankServices Project



AppleEvents
BankServices
DB2
DCE
DDEProject
DisplayProject
EncinaXACoordinator
ForeignObjMgr
Framework
GenericDBMS
Informix
Ingres

➤ **To create the BankClient project**

1. Choose the Plan > New Project… command.
2. When iPlanet UDS displays the New Project dialog, enter `BankClient` in the Project Name data field.
3. Disable the Include Database toggle, but leave the Include Display toggle on.

   The BankClient project will include window classes and therefore needs the Display library as a supplier plan.
4. Examine the workspace window again. The BankClient and the BankServices projects should both be listed.

   iPlanet UDS has also opened the Project Workshop for the BankClient project.
5. Close the BankClient project window.

Including a Supplier Plan

When you created the BankClient and BankServices projects, you specified whether the Display and GenericDatabase libraries should be included as supplier plans. In addition to using system libraries as supplier plans, you also need to define the relationship between the projects you create. One project will always be the main project; the other projects will serve as suppliers to the main project or to one another. You will specify the supplier relationship between the BankClient and BankServices projects later in this tutorial.

Chapter 3

Creating the Service Side

In this part of the tutorial you create the service side of the application. You will learn how to:

- define business classes
- define service objects
- work with arrays
- create methods
- post and handle events

Overview

The service side contains *business classes*, which define the data you want to process, and *service object classes*, which process that data and define the services provided to the client. Because the service side is a key design component in an iPlanet UDS application, it is a good idea to work on developing the service side before tackling the client side.

For this application, the service side includes a business class that specifies the structure of account information for each client, and a service object class that returns account information and updates account information in response to the user making withdrawals and deposits. If an application uses a database, the service side would also be responsible for interacting with the database to obtain and update account information. In this case, however, the application will store account information in the service side.

Defining Business Classes

In this part of the tutorial, you will define a business class called BankAccount, which defines account information for each client. The class contains three attributes:

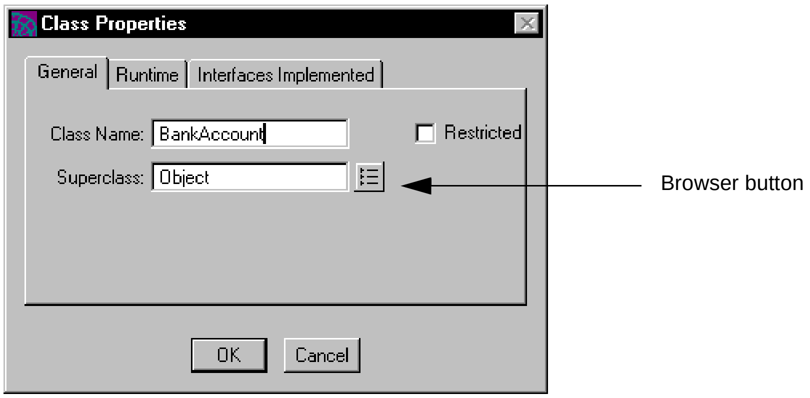
Attribute	Description
Balance	A floating-point number that specifies the current account balance.
Name	A string specifying the name of the client for this account.
Number	An integer that uniquely identifies the account.

➤ **To create a new class**

1. In the workspace window, double-click the BankServices project to open the Project workshop.
2. In the BankServices project, choose Component > New > Non-window Class.

   iPlanet UDS displays the Class Properties dialog. You use this dialog to specify the name of the class you want to create, the name of its superclass, and any runtime properties you want to define.

   All user-defined classes in iPlanet UDS must have a superclass. You can use the browser button in the Class Properties dialog to select the name of any class that can be a superclass from a list of supplier projects and their classes. The default superclass for non-window classes is Object.



Class Name: BankAccount

Superclass: Object

3. Enter `BankAccount` in the Class Name field.

4. Object is shown as the Superclass. Do not change this value.
5. Click OK.

iPlanet UDS displays the BankAccount class in the Class Workshop window. In the next part of the tutorial you will define the attributes of the BankAccount class.

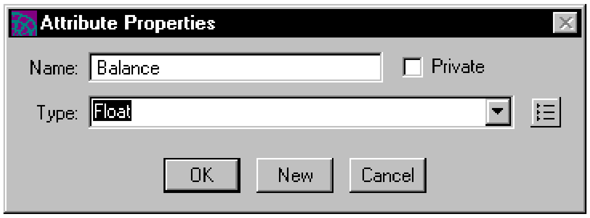
➤ **To create an attribute**

1. In the BankAccount class window, choose the Element > New Attribute… command.

   iPlanet UDS displays the Attribute Properties dialog, which you use to specify the name and type of an attribute.

2. Enter `Balance` in the Name field and `Float` in the Type field and click OK

.


Attribute Properties

Name: Balance

Private: [ ]

Type: Float

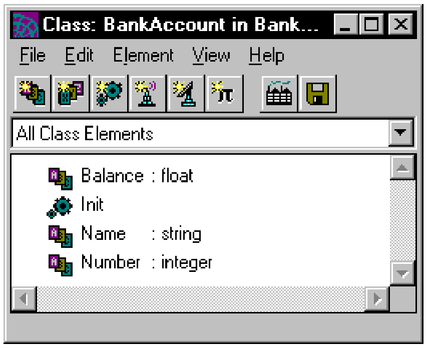
3. Click the New button.

   iPlanet UDS adds the Balance attribute to the BankAccount class and clears the Attribute Properties dialog so that you can enter information for additional attributes. (The New button provides a quicker way to add more than one attribute to a class. The alternate method is to click the OK button to save the current attribute and then to choose the Element > New Attribute command to bring up the Attribute Properties dialog again.)

4. Enter `Name` in the Name field and `String` in the Type field.
5. Click the New button.
6. Enter `Number` in the Name field and `Integer` in the Type field.

7. Click OK.

When you are done, the BankAccount class window should look like the following:



```
Balance : float
Init
Name    : string
Number  : integer
```

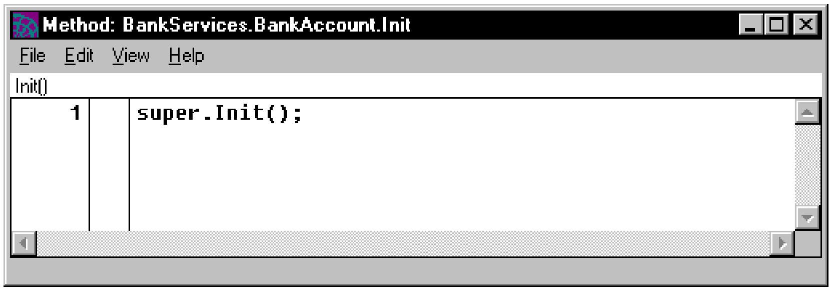
If you would like to change the definition of an attribute, because you want a different name or type, double-click the attribute name, make the desired changes in the Attribute Properties dialog, and then click OK to make your changes permanent.

If you want to delete an attribute, select the attribute in the class window and choose Edit > Delete.

As you can see, in addition to the attributes you have just defined, the class also contains an Init method, which is supplied in skeletal form for all newly created classes. You can examine the Init method by double-clicking it in the Class Workshop window.

iPlanet UDS opens the Method Workshop and displays the body of the Init method, as shown in Figure 3-1.

Figure 3-1 Init Method



```
super.Init();
```

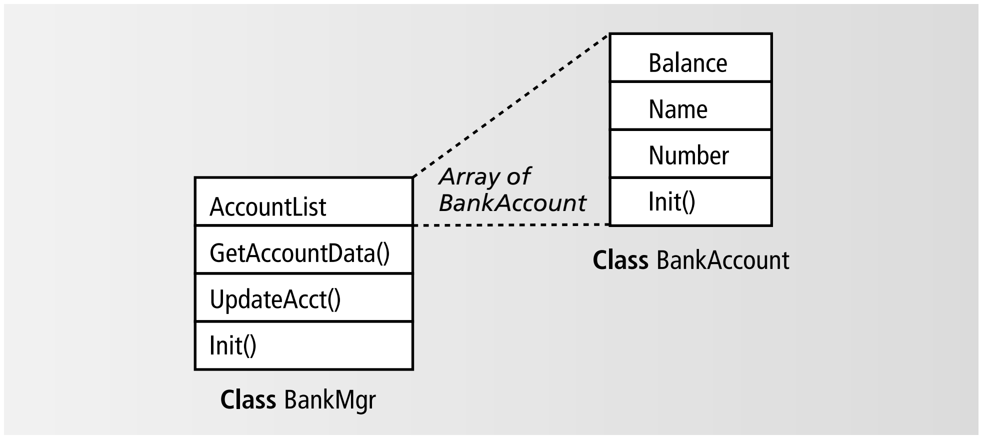
The super.Init statement, which is included by default, invokes the Init method of the Object class, which is the superclass of the BankAccount class. This Init method is called whenever an object of the BankAccount class is constructed. You do not need to modify this method, so you can just close the Method Workshop window, and then close the BankAccount class window.

You have just finished defining the business class for the sample application.

Defining Service Object Classes

To process client requests, the application will use the BankMgr class. This class includes an attribute that is an array of BankAccount objects and defines methods that manipulate this entire set of current accounts. Figure 3-2 shows the relationship between the BankAccount class, which you have just created, and the BankServices class, which you will create in the next part of the tutorial. Later in this tutorial, you will create a service object that is an instance of the BankMgr class.

Figure 3-2 Classes in the BankServices Project



The BankMgr class has an attribute defined for storing an array of BankAccount objects, and includes methods for getting and updating account information.

➤ **To create the BankMgr class**

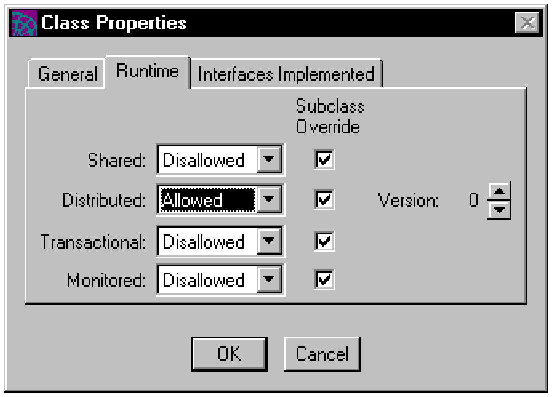
1. In the BankServices project workshop, choose the Component > New > Non-window Class… command.

   iPlanet UDS displays the Class Properties dialog.

2. Enter `BankMgr` for the Class Name field. Leave the Superclass property as Object.

3. Click the Runtime tab in the Class Properties dialog.

4. Set the Distributed property to Allowed.



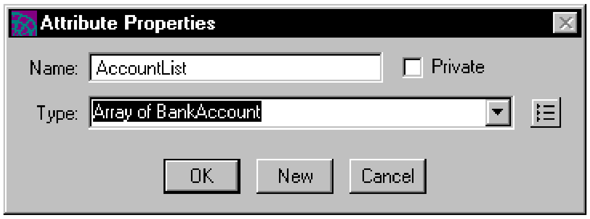
		Subclass Override		
Shared:	Disallowed	✓		
Distributed:	Allowed	✓	Version:	0
Transactional:	Disallowed	✓		
Monitored:	Disallowed	✓		

   The service object that instantiates the BankMgr class needs to be distributed so that the object can be visible to other partitions. To allow this to happen, you must set the class's Distributed property to Allowed.

5. Click OK.

   iPlanet UDS displays the BankMgr class window.

6. Use the procedure described in Step 1 and Step 2 (page 35) to create an attribute for the BankMgr class named AccountList as shown below:



Name: AccountList ☐ Private

Type: Array of BankAccount

Working with Arrays

A TOOL Array class is a special class for storing and manipulating a collection of objects. The objects in an array are either of the same class or they share the same superclass. Because an array is a single object that references each object item in the array, you can use the array to reference the set of objects as a unit or you can manipulate the individual objects. The array class also provides methods for manipulating the array and its elements, including methods for inserting rows, appending rows, and deleting rows. You will work with arrays when you define the Init method for the BankMgr class.

For information about arrays, see the *TOOL Reference Guide* and the Framework Library online Help.

In the next part of the tutorial you will initialize the AccountList array in the BankMgr class Init method.

➤ **To initialize the AccountList array**

1. In the BankMgr class window, double-click the Init method to open the Method Workshop.
2. Enter the code shown below in the Init method window, after the call to super.Init. Note that TOOL is not case sensitive.

```
-- declare local variable account
account : BankAccount;
-- instantiate the AccountList attribute
self.AccountList = new();
-- instantiate the account variable
-- and initialize some of its attributes
account = new(
    Number = 1000,
    Name = ’Paul Cezanne’,
    Balance = 300.00
    );
self.AccountList.AppendRow(account);

account = new(
    Number = 2000,
    Name = ’Vincent van Gogh’,
    Balance = 2.25
    );
self.AccountList.AppendRow(account);

account = new(
    Number = 2000,
    Name = ’Andy Warhol’,
    Balance = 45000000.00
    );
self.AccountList.AppendRow(account);
```

The statements that you added to the Init method instantiate the AccountList object, initialize three objects of type BankAccount, and append them to the AccountList array. Note that in a more realistic iPlanet UDS program, account data would be extracted from a database. In this tutorial, the data is placed in the Init method of the BankMgr class to keep the example simple.

The application now has the data it needs to operate on when the user interacts with the account. To respond to user actions, the application needs two more methods, one that gets account information and one that updates account information. You will be creating these in the next two sections.

3. In the Method workshop, choose File > Compile to compile the Init method. This step is not required, but it can help you check for errors in entering your code.
4. Close the Method Workshop.
5. Save the work you have done so far by choosing the File > Save All command in any open workshop window.

Creating a New Method

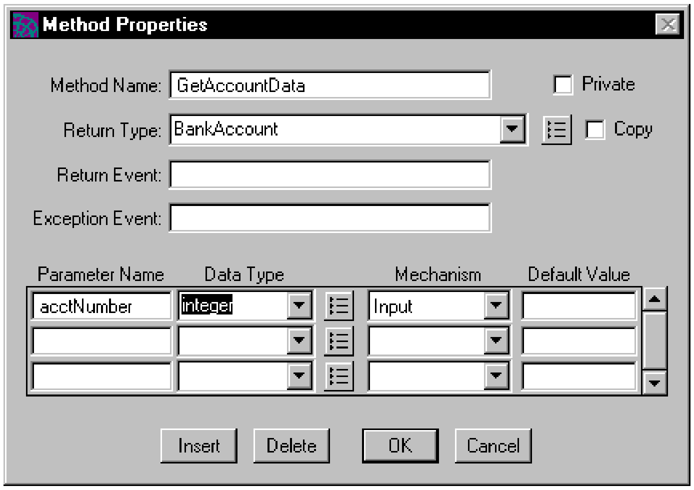
So far you have only examined or modified existing methods. In this part of the tutorial you will create a new method.

➤ **To create a new method**

1. In the BankMgr class window, choose Element > New Method.

   iPlanet UDS displays the Method Properties dialog. You use this dialog to define the method signature—its name, return type, and the names and types of its parameters.

2. Use the appropriate fields to enter the information shown below and then click OK.


Method Properties

Method Name: GetAccountData
Private: ☐
Return Type: BankAccount
Copy: ☐
Return Event:
Exception Event:

Parameter Name	Data Type	Mechanism	Default Value
acctNumber	integer	Input	

iPlanet UDS opens the Method Workshop.

3. Enter the following code to define the GetAccountData method:

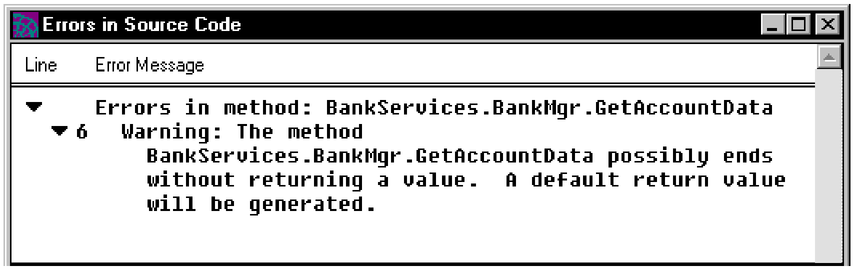
```
-- This method returns a copy of the BankAccount object
-- specified by the account number.
for a in self.AccountList do
  if acctNumber = a.Number then
      return a;
  end if;
end for;
```

The GetAccountData method attempts to find the Account object element in the AccountList array whose account number matches the one specified by the AcctNumber parameter. If a matching account is found, the method returns it to the caller.

For more information about working with arrays, see the *TOOL Reference Guide* and the Framework Library online Help.

4. In the Method Workshop, choose File > Compile to check for entry errors.

iPlanet UDS displays the following dialog:



```
▼    Errors in method: BankServices.BankMgr.GetAccountData
  ▼ 6  Warning: The method
         BankServices.BankMgr.GetAccountData possibly ends
         without returning a value.  A default return value
         will be generated.
```

You have made no provision for what the method should do if it does not find an account, and iPlanet UDS displays a warning about this situation. The section Chapter 5, "Handling Exceptions," will address that issue.

5. Close the Errors in Source Code window and the Method Workshop.

Let's review what you have done so far. You have defined two classes in the service side: BankAccount and BankMgr. In the BankMgr class you have defined the attribute AccountList, an array of Account type objects. You have also defined a method that returns an object containing information about an individual account. You still need to define a method to update account information when a client makes a deposit or withdrawal. To implement this method, you will use events, as described in the next section.

Posting and Handling Events

iPlanet UDS is an event-driven system. It triggers events representing user actions (mouse clicks, field entries, or menu activation) automatically. Typically, the Display method for a window includes an event loop that you use to handle events on the various controls (widgets) defined for the window. For example, if the window includes a Save button, you might handle it as follows:

```
self.Open();
event loop
  when <SaveButton>.Click do
    self.Save();
end event;
self.Close();
```

In the example just given, when the user pushes the Save button (which is an instance of the PushButton class), iPlanet UDS automatically posts the predefined Click event. The only thing you have to do is provide code in your event loop that is executed when the event occurs.

The custom classes that you create can also have custom events associated with them. Such events are not defined or triggered automatically. Instead, you must create the event using the Project Workshop and then trigger the event by using the `post` statement later in your TOOL code. Finally, you also have to write some code that handles the event.

In this part of the tutorial, you will create a custom event called AcctUpdated and post the event whenever the user makes a withdrawal or a deposit. The event you define will be added to the BankMgr class and displayed in its window.

➤ **To create an event**

1. In the BankMgr Class window, choose the Element > New Event… command.

   iPlanet UDS displays the Event Properties dialog.

2. Enter `AcctUpdated` in the Event Name field.

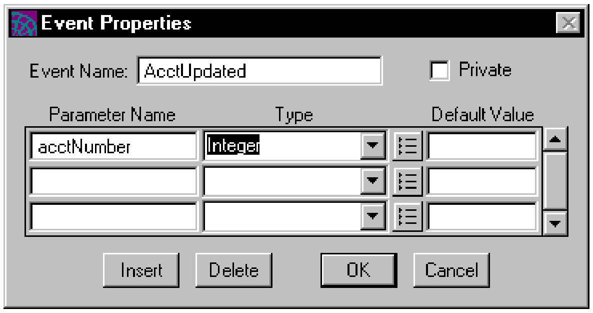
3. Enter `acctNumber` in the Parameter Name field.

   An event can have one or more parameters to provide additional information about the event. Why would you want to specify a parameter for an event?

   Remember that when the application is deployed, there might be many clients waiting for AcctUpdated events. Passing the account number as a parameter to the AcctUpdated event allows the client to check the value of the parameter to the event and to process only the event that has the client's account number.

4. Enter `Integer` in the Type field.

   When you are done, the Event Properties window should look like the following:



Event Properties

Event Name: AcctUpdated

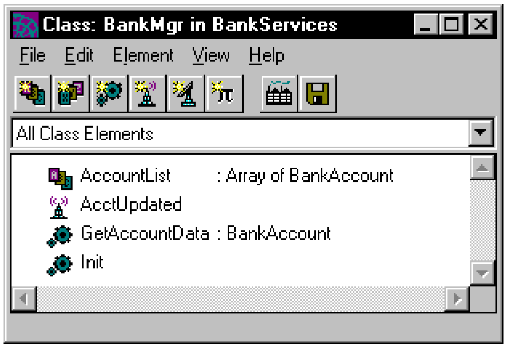
Private: ☐

Parameter Name	Type	Default Value
acctNumber	Integer	

5. Click OK to add the event to the BankMgr class.

Take a look at the BankMgr Class window. The AcctUpdated event has been added, as shown in Figure 3-3.

Figure 3-3 AcctUpdated Event



AccountList : Array of BankAccount
AcctUpdated
GetAccountData : BankAccount
Init

An event is identified by a “radio broadcasting” icon, which reflects the nature of posting, or “selectively broadcasting” an event to interested parties. Any number of clients or services can be tuned in, or registered, waiting for that particular signal.

After you have created an event for a particular class, a method must `post` the event. Posting an event causes that event to be broadcast, which notifies all interested parties that the event has occurred. An interested party is a class that contains a method that handles the event. In this example, the BankMgr class will post the event on itself in the UpdateAcct method.

➤ **To post an event**

1. In the BankMgr class window, choose the Element > New Method… command.

   iPlanet UDS displays the Method Properties dialog.

2. Enter UpdateAcct for the Method Name field.

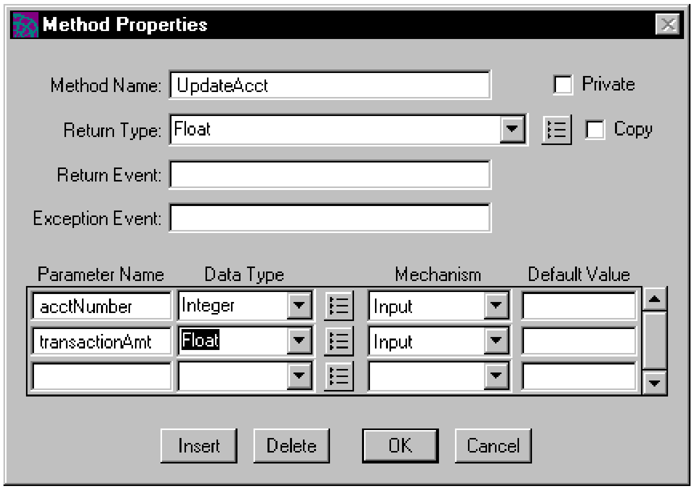
3. Enter Float for the Return Type field.

   The UpdateAcct method will return the new account balance.

4. Enter the following information for the Parameter Name and Data Type fields:

Parameter Name	Data Type
acctNumber	Integer
transactionAmt	Float

When you are done, the dialog should look like the following:



Method Name: UpdateAcct
Return Type: Float
Return Event:
Exception Event:

Parameter Name	Data Type	Mechanism	Default Value
acctNumber	Integer	Input	
transactionAmt	Float	Input	

5. Click OK to add the method to the BankMgr class.

iPlanet UDS opens the Method Workshop for the UpdateAcct method. This method will be called from the client side when the user makes a deposit or a withdrawal. The caller will specify the number of the account being updated, which will be passed in using the AcctNumber parameter. The caller will also specify the amount being added or subtracted, which will be passed in using the TransactionAmt parameter.

The AcctUpdate method will add the transaction amount to the current balance, post an AcctUpdated event to notify all parties that are registered to receive the event that some account information has changed, and return the new account balance.

6. Add the following code to the UpdateAcct method:

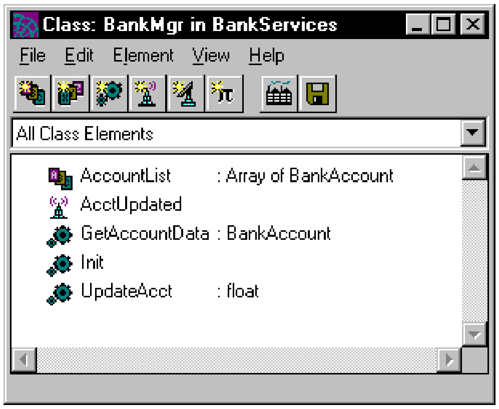
```
for a in self.AccountList do
  if acctNumber = a.Number then
    a.Balance =  a.Balance + transactionAmt;
    post self.AcctUpdated(acctNumber = acctNumber);
    return a.Balance;
  end if;
end for;
```

The code shown iterates through the array until it finds an account whose number matches the value of the acctNumber parameter passed to the UpdateAcct method. When it finds that account, the method adjusts the account balance by adding the transaction amount (passed in as the TransactionAmt parameter). The UpdateAcct method then posts the event and returns the new account balance. Note that this method does not include code for the case where the account number cannot be found. You will add this code in Chapter 5, "Handling Exceptions."

7. Choose File > Compile to compile the method.

   You will get a warning about the method possibly ending without returning a value. You can ignore this warning for now. Close the Errors in Source Code window and close the Method Workshop.

   You have now finished defining the BankMgr class. It should look like the following:



AccountList : Array of BankAccount
AcctUpdated
GetAccountData : BankAccount
Init
UpdateAcct : float

8. Close the BankMgr class window.

Creating Service Objects

So far, you have defined the BankMgr and BankAccount classes in the BankServices project. You have also created and initialized the BankAccount array object in the Init method of the BankMgr class. But where is the object that instantiates the BankMgr class? This is the object that interests us most because it contains all the logic required to return account information and update account information for all clients wanting to use the Banking application.

Instantiating a global object of type BankMgr that is accessible throughout the application is a special step in creating an iPlanet UDS application; this step is called creating a service object.

A *service object* is a named, shared, global object that provides a set of centralized operations. In a distributed application, the service object is used to implement business rules, provide access to persistent data such as databases, and provide access to external applications, such as a bar code reader or a telephone line. For this tutorial, the service object you will create (an instance of the BankMgr class) contains the methods that a client can call to return and update information about individual accounts.

Service objects are also used as the basis for partitioning an application. Partitioning is discussed in Chapter 6, "Testing and Running the Application."

When creating service objects and the classes upon which they are based, follow these guidelines:

- Make sure that the class for which you will be creating a service object has its Distributed runtime property set to IsDefault or Allowed. (You did this on page 38, Step 4.)
- Make the project that contains the service object a supplier to the project that calls service object methods. (You will do this in Chapter 4, "Creating the User Interface.")
- Do not instantiate the service object. Service objects are automatically instantiated by iPlanet UDS when the server partition containing the service object is started.

➤ To create a service object

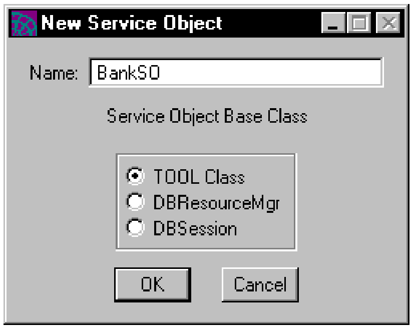
1. In the BankServices Project Workshop, choose the Component > New > Service Object… command.

   iPlanet UDS displays the New Service Object dialog.

2. Enter `BankSO` in the Name field.

3. Click the TOOL Class radio button.

   When you are done, the dialog should look like the following:



Name: BankSO

[✓ TOOL Class]

4. Click OK.

   iPlanet UDS displays the Service Object Properties dialog to get more information about the service object you want to create.

5. Enter `BankMgr` for the Class field.

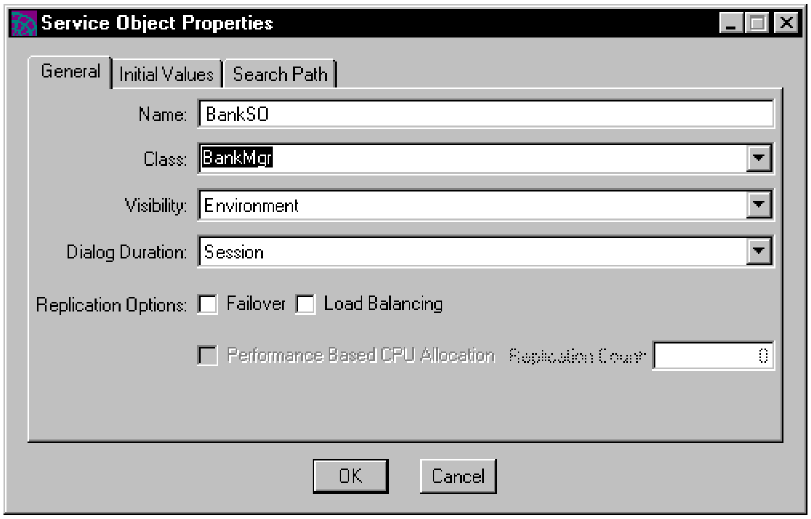
6. Specify `Environment` for the Visibility field.

   This makes the service object visible to all users in the environment.

7. Leave `Session` for the Dialog Duration field.

The Dialog Duration is the interval over which a service object retains its connection to a particular caller after the connection is started. Specifying *session* means that a caller making a request from the service object is bound to that object for the entire run of the application. For more information about dialog duration, see the *iPlanet UDS Programming Guide*.

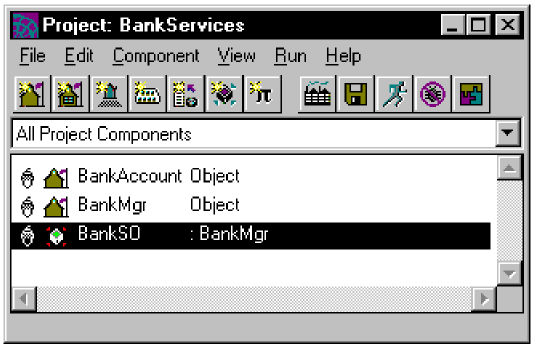
When you are done, the dialog window should look like the following:



Name: BankSO
Class: BankMgr
Visibility: Environment
Dialog Duration: Session
Replication Options: Failover Load Balancing
Performance Based CPU Allocation Replication Count 0

8. Click OK to add the service object to the BankServices project.

   You have finished creating the BankServices project. The BankServices Project Workshop should look like the following:



BankAccount Object
BankMgr Object
BankSO : BankMgr

   Note that the BankServices project contains two classes and a service object. The colon indicates that BankSO is a service object of type BankMgr.

9. Close the BankServices Project window.

For more information about service objects, see the *iPlanet UDS Programming Guide*.

In the next part of the tutorial, you will create the user interface for the application.

Chapter 4

Creating the User Interface

The user interface to the sample application consists of two windows that allow the user to obtain and modify information about her account. Account data and the methods used to return and manipulate that data reside in the objects you created in the last part of the tutorial.

In this chapter you will use the Window Workshop to create the tutorial application's user interface. You will learn how to:

- create a window
- add buttons and labels
- create data fields
- use grid fields
- code a window's logic
- test your window

About Window Classes

You use window classes to create the appearance and behavior of your application's user interface.

Window classes contain graphical user interface (GUI) elements called *widgets* such as push buttons, data fields, and so on. You use the Window Workshop to add these elements to a window. In addition to creating the graphical user interface, you must also add code to the window's skeletal Display method to respond to events (user actions) in the windows. At runtime, the application constructs a window object of a given window class and executes the methods defined for that class when the user manipulates the window's controls

Creating a Window Class

In creating the user windows for this application, you will be working with a small subset of the tools iPlanet UDS provides for building user interfaces. For more information about creating a user interface, see the *iPlanet UDS Programming Guide* chapters about creating user interfaces. For a complete reference to the iPlanet UDS Display library, see the Display Library online Help.

➤ **To create a window class**

1. Open the BankClient project by double-clicking its name in the Repository Workshop.

   In the next part of the tutorial you will be working in the Project Workshop to add two new window classes to the BankClient project. Each user window will be defined as a window class.

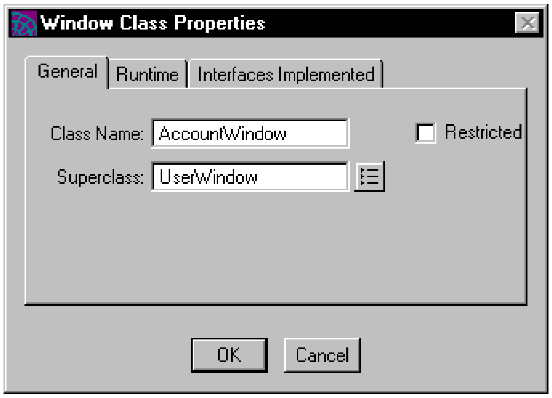
2. Choose the Component > New > Window Class… command.

   iPlanet UDS displays the Window Class Properties dialog. You use this dialog to enter the name of the window class you want to create and the name of its superclass.

3. Enter `AccountWindow` in the Class Name field.

   By default, UserWindow is the superclass for window classes. Creating a subclass of the UserWindow class automatically provides you with an empty window, which you can then format using the Window Workshop. The UserWindow class also provides the methods you need to manipulate your window; for example, opening it and closing it. For details, see the Display Library online Help.

When you are done, the Window Class Properties dialog should look like the following:



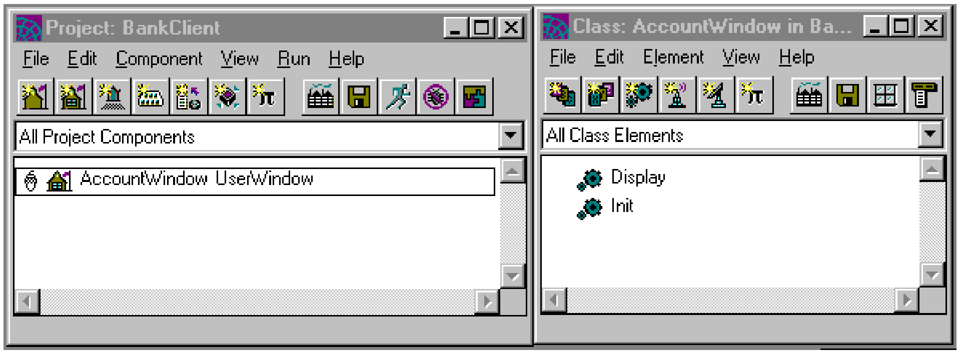
Class Name: AccountWindow

Superclass: UserWindow

4. Click OK to add the AccountWindow class to the BankClient project.

   iPlanet UDS displays the name of the class in the BankClient project window and opens the Class Workshop for this window class.

   Note that the class already includes two default methods: Init and Display.



5. Close the AccountWindow class window.

6. In the BankClient window, choose the Component > New > Window class… command.

   iPlanet UDS displays the Window Class Properties dialog.

7. Enter `TransactionWindow` in the Class Name field. The Superclass field is set to UserWindow. Do not change this value.

8. Click OK to add the TransactionWindow class to the BankClient project.

   iPlanet UDS displays the name of the class in the BankClient project window and opens the Class Workshop for this window class.

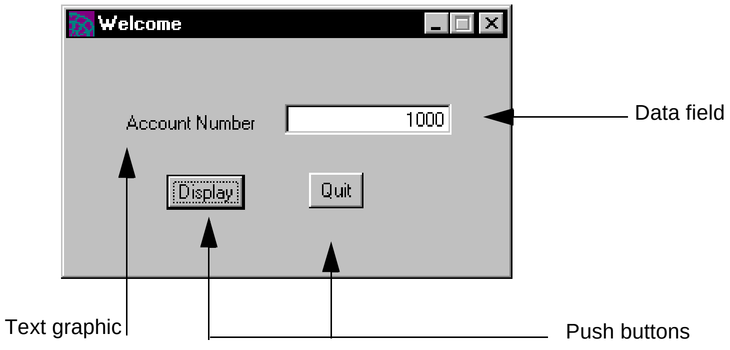
9. Close the TransactionWindow class window.

Using the Window Workshop

In this part of the tutorial, you will use the Window Workshop to create the user interface for the application.

To begin, you will create the first window the user sees after the application is launched. The window will look like the following:

Figure 4-1 User Start-up Window

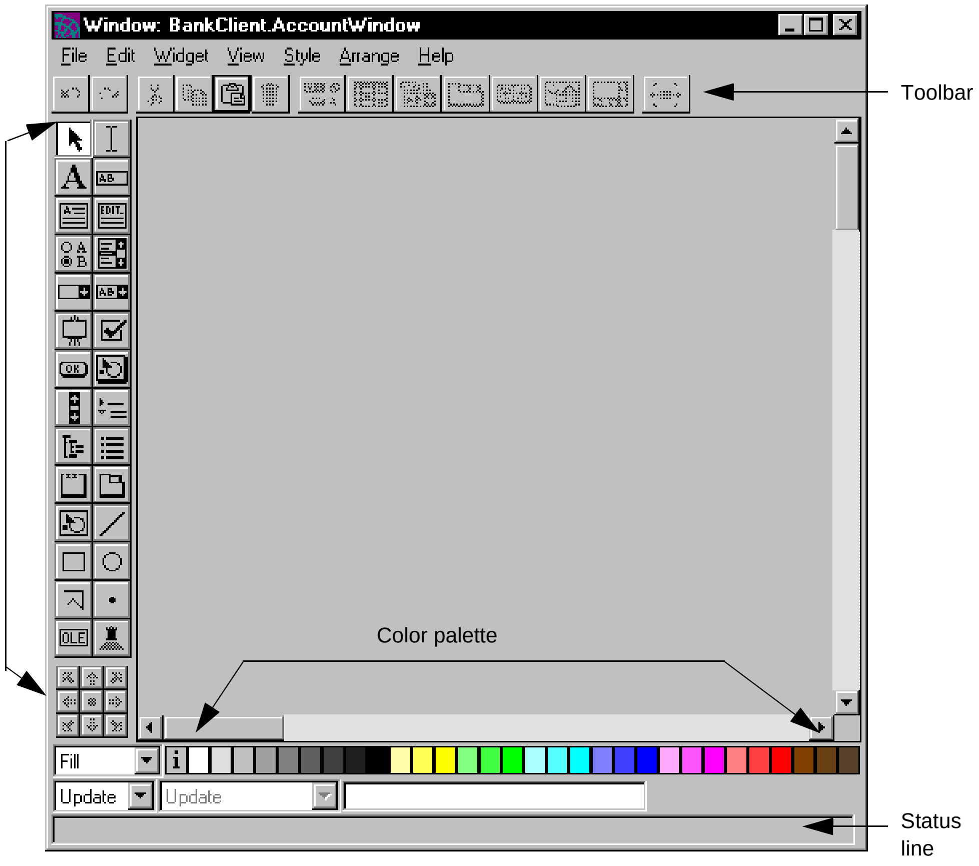


Note the default controls (zoom icon, window close icon, title bar) that are provided for the window in your windowing system. In the next part of the tutorial, you will add the three types of user-defined controls marked in Figure 4-1: a text graphic (used for labels), a data field, and two push buttons.

➤ **To open the Window Workshop and create the start-up window**

1. Double-click the AccountWindow class in the BankClient project window.
2. Choose File > Window to open the Window Workshop.

   iPlanet UDS opens the Window Workshop and displays a blank window, as shown below.



The Window Workshop provides the tools and widgets (controls) you need to create your window. You select the tools and widgets you need either from the appropriate palette or by choosing items from the menu.

Before you continue, take a little time to get acquainted with the Window Workshop. One way to do this is to move the cursor over the widgets and tools shown in the window. As you do, pause over each tool. iPlanet UDS displays float-over help text for each element, as well as descriptive text in the status line at the bottom of the window.

For information about the Window Workshop, see *A Guide to the iPlanet UDS Workshops*.

Adding a Label

A *text graphic* is a static series of characters that you enter directly onto the window form. You can use a text graphic to provide information, instructions, or labels for widgets.

➤ **To add a label to the window**

1. Click once on the text graphic icon, **A**.
2. Click again in the window at the approximate location where you want to enter text.

   iPlanet UDS displays a default text graphic like the following:

   Text Graphic

3. Type the following label text: `Account Number`:

   iPlanet UDS now displays this text in the window.

Adding a Data Field

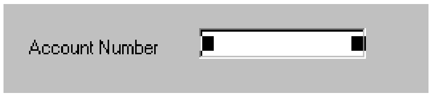
A *data field* displays a single line of data of a specific type, such as a date, money value, or ID number. Use a data field to display information or to provide a data entry field where the user can enter or modify information.

➤ **To add a data field for the account number**

1. Choose Widget > New > DataField.

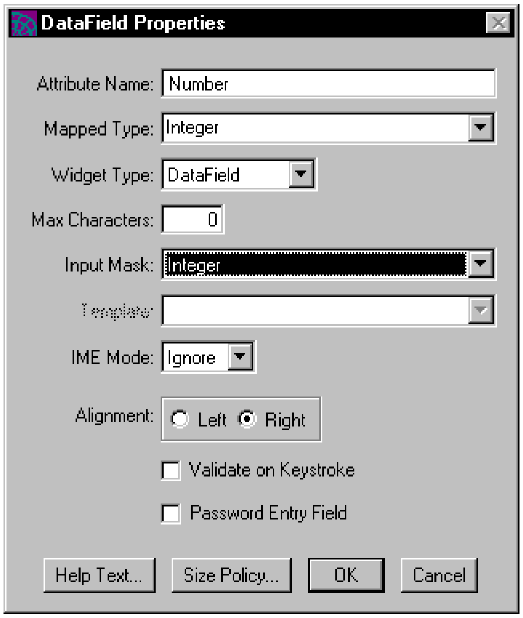
   iPlanet UDS displays a **+** cursor.

2. Click and drag the **+** cursor to form a rectangle that is roughly the size of the data field.



3. Double-click on the data field.

   iPlanet UDS displays the DataField Properties dialog.



DataField Properties

Attribute Name:	Number
Mapped Type:	Integer
Widget Type:	DataField
Max Characters:	0
Input Mask:	Integer
Template:	
IME Mode:	Ignore
Alignment:	Left [✓ Right]

☐ Validate on Keystroke

☐ Password Entry Field

4. Complete the fields in the dialog as shown in the figure above, then click OK.

Every widget has a properties dialog, such as this one, that defines the widget's attributes. In general these properties include the name of the widget and its mapped type.

You use the widget's attribute name to reference the widget in TOOL code. For example, if a push button's name is QuitButton, you could catch events on it as follows:

```
when <QuitButton>.Click do
    task.PostShutdown;
```

You specify a mapped type to define the type of data contained in the widget. This type can be a simple data type or a class data type.

DataField widgets can also have an input mask and a template to control the format of characters that are keyed in. If you attempt to enter an illegal format, the application gives a warning beep and does not accept the input.

Adding Push Buttons

A push button is a labeled button that the user can click to give a command or instruction. When you place a button on the form in the Window Workshop, iPlanet UDS provides a default label. You can change the label using the PushButton Properties dialog.

➤ **To add a push button**

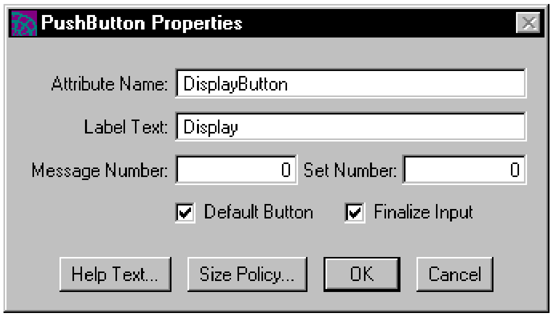
1. Choose Widget > New > PushButton.
2. Click in the window at the position where you want to place the button.

   iPlanet UDS displays a widget labeled “Button.”
3. Double-click the button.

   iPlanet UDS displays the PushButton Properties dialog. You use this dialog to specify the name of the PushButton attribute for the AccountWindow class and also to specify the name of the button as it will be displayed in the window.
4. Enter `DisplayButton` for the Attribute Name field.
5. Enter `Display` in the Label Text field.

6. Click the Default Button check box to set the Display button as the default button.

   A default button is displayed with an extra border around it. When the user presses Return, the application responds as if the default button were clicked. If a window contains a set of buttons, only one can be designated as the default button.


PushButton Properties

Attribute Name: DisplayButton

Label Text: Display

Message Number: 0 Set Number: 0

[✓] Default Button [✓] Finalize Input

7. Click OK.

   The newly created push button is now displayed in your window. To reposition the button, select the button and drag it to the desired location.

8. Choose Widget > New > Pushbutton to add a Quit button.

9. Click in the window at the position where you want to place the push button.

   iPlanet UDS displays a widget labeled “Button”.

10. Double-click the button.

    iPlanet UDS displays the PushButton Properties dialog.

11. Enter `QuitButton` in the Attribute Name field.

12. Enter `Quit` in the Label Text field and click OK.

    iPlanet UDS adds the button to your window. Click the button and drag to place it in the desired location.

You are now finished creating the start-up window for the application. Before you go on to create the window’s methods, take a moment to see what the window will actually look like and how its controls are represented in its class window.

Testing the Window

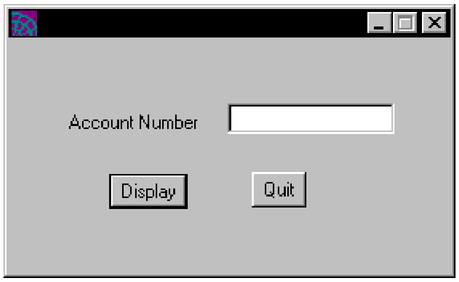
iPlanet UDS allows you to test a window without having to run the entire application. The window's methods will not execute, but you can examine the widget's layout and behavior.

➤ **To test a window**

1. In the Window Workshop, choose File > Test Window.

   iPlanet UDS displays the AccountWindow as it would when its Display method is invoked

.



   While the window is in test mode, you cannot use any of its controls. Note that the window has no title. You will add one in the next section.

2. Close the window (by using the window control, not the Quit button).

Defining a Window Title

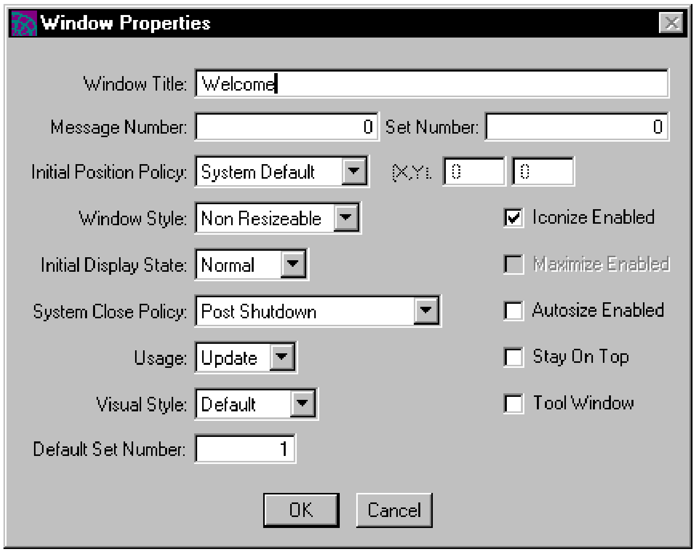
In the Window Workshop, you can also define text that is displayed in a window's title bar.

➤ To add title bar text

1. Choose File > Window Properties.

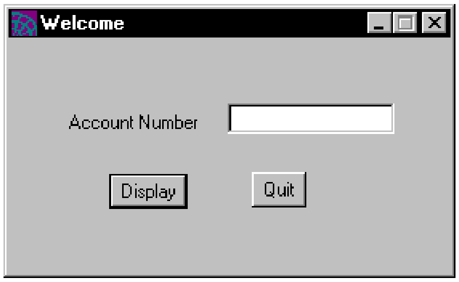
   iPlanet UDS displays the Window Properties dialog.

2. Enter `Welcome` for the Window Title field. Leave the other settings as they are, and click OK.


Window Properties

Window Title:	Welcome		
Message Number:	0	Set Number:	0
Initial Position Policy:	System Default	X,Y:	0 0
Window Style:	Non Resizeable	☑ Iconize Enabled	
Initial Display State:	Normal	☐ Maximize Enabled	
System Close Policy:	Post Shutdown	☐ Autosize Enabled	
Usage:	Update	☐ Stay On Top	
Visual Style:	Default	☐ Tool Window	
Default Set Number:	1		

3. Choose File > Test Window again and examine the window.



4. Close the Welcome window (by using the window control, not the Quit button).

5. Choose File > Save All from the Window Workshop to save the window.

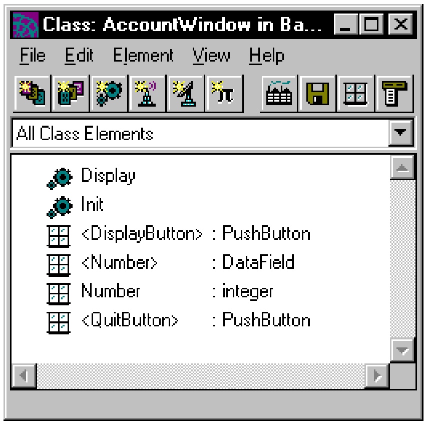
6. Close the Window Workshop.

The Representation of Window Attributes

Let’s take another look at the AccountWindow class window. You added widgets to control the appearance of the window. Widgets are graphic objects. When you use the Window Workshop to add widgets to a window, you are actually adding attributes to the window class.

How are these attributes represented in the window class? Take a look at Figure 4-2.

Figure 4-2 The AccountWindow Class



Display
Init
<DisplayButton> : PushButton
<Number> : DataField
Number : integer
<QuitButton> : PushButton

Both of the buttons you added are shown. Their class type is PushButton; their attribute names are displayed in angle brackets to indicate that they are widgets.

In the case of the Number widget, there are two entries: Number and <Number>. This convention for representing widgets is an important distinction in iPlanet UDS.

iPlanet UDS associates widgets that display data with two attributes:

- The attribute shown in angle brackets identifies the graphic object, its appearance and behavior.

  You use this attribute in code when you do something that affects the appearance of the widget. For example, the following statement changes the pen color so that the characters displayed in the Number field are red.

  ```
  <Number>.PenColor = C_BRIGHTRED;
  ```

- The attribute shown without angle brackets references the data displayed in the widget.

  You use this attribute in code when manipulating the value of the data; for example, the following statement calls the Display method for the transaction window, passing the value the user entered for the account number as a parameter.

  ```
  transwin.Display(currAcct = self.Number);
  ```

Creating a Window's Methods

In this part of the tutorial, you will create the methods for the AccountWindow class.

➤ **To create the AccountWindow Display method**

1. In the AccountWindow class window, double-click the Display method.

   As you can see, the Display method already contains some code, as shown below:

   ```
   self.Open();
     event loop
       when task.Shutdown do
         exit;
     end event;
   self.Close();
   ```

   If you do nothing else, the default Display method will open the window when the method is invoked and then close it when the task terminates.

2. Modify the existing code so that it matches the following:

```
self.Open();
  event loop
    when <DisplayButton>.Click do
    -- To display another window, first instantiate it
    -- then invoke its Display method.
    -- This will cause a synchronous modal window to appear.
      transWin: TransactionWindow = new();
      transWin.Display(currAcct = self.Number);
    when <QuitButton>.Click do
      task.PostShutdown();
    when task.Shutdown do
      exit;
  end event;
self.Close();
```

Do not compile the method yet. You need to modify the Display method of the TransactionWindow class before this method will compile.

The Display method you just created still opens the account window. Then, depending on whether the user clicks the Display button or the Quit button, it either creates and displays the transaction window or it quits the application, as described next.

For every task, (the sample application is defined and structured to run as one task), iPlanet UDS creates a TaskHandle object. You reference this object using the `task` keyword. In the code shown above, the PostShutdown method of the TaskHandle class is invoked when the user clicks the Quit button. This method posts a Shutdown event on the task. The following `when task.Shutdown` clause handles the event by calling `exit`. The `exit` statement closes the event loop statement block and passes control to the first statement that follows the block, if any.

3. Close the Method Workshop.

You do not need to change the Init method. If you want to see what it looks like, double-click its name in the AccountWindow class window. Close it when you are done.

4. Close the AccountWindow class window.

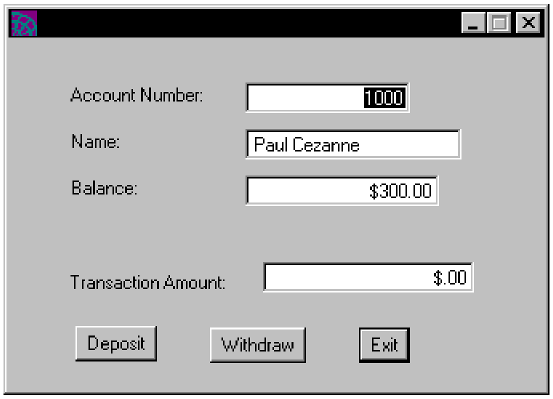
Creating the Transaction Window

In this part of the tutorial you will create the transaction window and, in the process, you will learn about a key feature of the iPlanet UDS user interface, the grid field.

A *grid field* is a compound widget that arranges widgets into rows and columns of cells. By using grid fields, you can specify the relative placement of widgets on a form. This assures that widgets in a grid field never overlap, even when the user interface has to be ported to other windowing systems. We will discuss other features of grid fields later in this tutorial.

The window you will create next is displayed to the user after she clicks the Display button in the Account window. The window is shown in Figure 4-3.

Figure 4-3 Transaction Window


Account Number: 1000
Name: Paul Cezanne
Balance: $300.00
Transaction Amount: $.00

Note that when the window is first displayed, all fields are filled in with account information except for the Transaction Amount field. The user would enter an amount in this field and then click the Deposit or Withdraw button. When the user is finished making transactions, she would click the Exit button.

As you create the window in the next part of the tutorial, do not worry about aligning labels and data fields neatly. You will be using grid fields to group and arrange widgets in a window.

➤ **To create the transaction window**

1. Open the TransactionWindow class.
2. Choose File > Window to open the Window Workshop.
3. Select the text graphic icon, **A**.

4. Click in the window where you want the label to be placed and enter the text: `Account Number:`

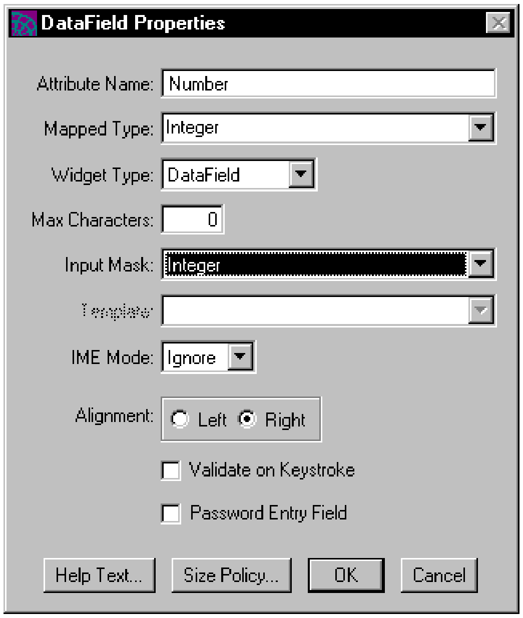
5. Choose Widget > New > DataField.

   iPlanet UDS displays a **+** cursor.

6. Click and drag the **+** cursor to form a rectangle that is roughly the size of the data field.

7. Double-click on the data field.

   iPlanet UDS displays the DataField Properties dialog.



DataField Properties

Attribute Name: Number

Mapped Type: Integer

Widget Type: DataField

Max Characters: 0

Input Mask: Integer

Template:

IME Mode: Ignore

Alignment: Left [✓ Right]

☐ Validate on Keystroke

☐ Password Entry Field

8. Complete the fields in the dialog as shown in the figure above and click OK.

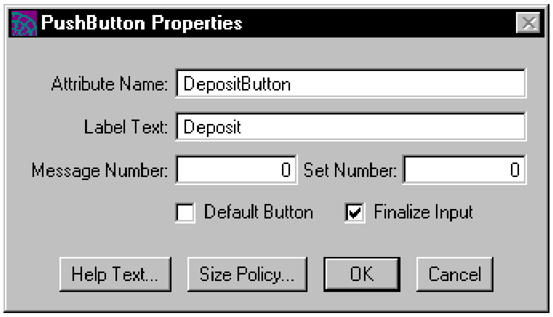
9. Use the procedure outlined in Step 3 through Step 7 to add the following data fields to the window form. Add the text graphic labels for each field, as shown in Figure 4-2 on page 64.

Attribute name	Mapped Type	Widget Type	Input Mask	Template
Name	String	DataField	None	None
Balance	Float	DataField	Template	CURRENCY
TransactionAmt	Float	DataField	Template	CURRENCY

➤ To add the push buttons

1. Choose Widget > New > Pushbutton and click again in the window at the approximate location where you want to place the button.
2. Double-click the button to set its properties.

   iPlanet UDS displays the PushButton Properties dialog.
3. Specify the Deposit button properties as shown below.


PushButton Properties

Attribute Name: DepositButton
Label Text: Deposit
Message Number: 0 Set Number: 0
☐ Default Button ☑ Finalize Input

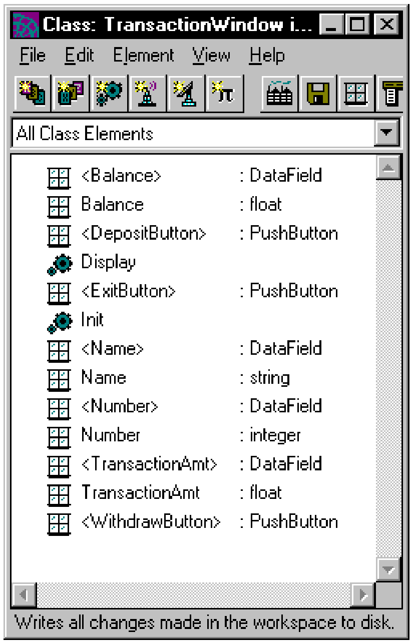
4. Create two more buttons and specify the following values in the PushButton Properties dialog for each button. Make the Exit button the default button.

Attribute Name	Label Text
WithdrawButton	Withdraw
ExitButton	Exit

5. Choose File > Save All from the Window Workshop to save the Transaction window.

You are now nearly done designing the Transaction window. Let's look at the TransactionWindow class.

Figure 4-4 Transaction Class Window



```
<Balance>          : DataField
Balance            : float
<DepositButton>    : PushButton
Display
<ExitButton>       : PushButton
Init
<Name>             : DataField
Name               : string
<Number>           : DataField
Number             : integer
<TransactionAmt>   : DataField
TransactionAmt     : float
<WithdrawButton>   : PushButton
```

Writes all changes made in the workspace to disk.

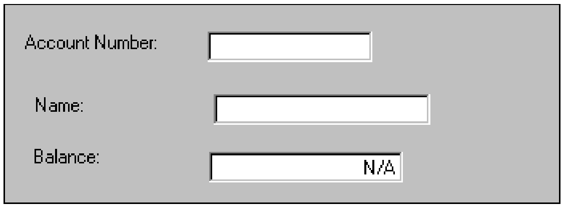
For each widget you added to the window form, iPlanet UDS has added an attribute to the TransactionWindow class. As with the AccountWindow class, widgets that are data fields have two attributes: one for the widget and the other for the data it contains.

Using Grid Fields

In this section you are going to create a grid field by grouping together some widgets in the transaction window. A grid field is a compound field that arranges its component fields in rows and columns.

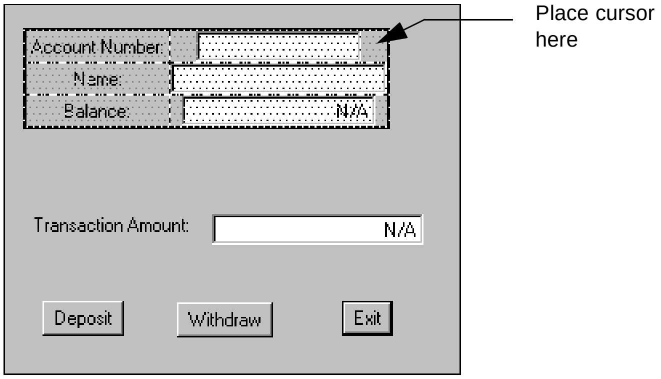
➤ **To create a grid field**

1. Open the Window Workshop for the TransactionWindow class if it is not already open.
2. Click the selection tool, and lasso the following elements in the window form:



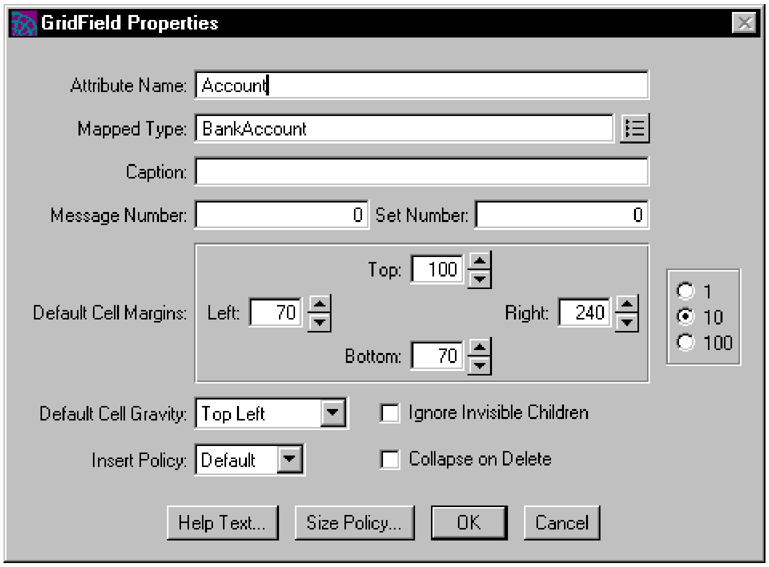
3. Choose Widget > Group into > GridField.
4. Choose View > Compound Field Lines.

   The window form should now look like this:



5. Position the cursor so that it is *not* over any of the data fields. (A good position is shown in the figure above.)
6. Double-click to bring up the GridField Properties dialog.

7. Change the values in your dialog so that it looks like the following.



Attribute Name: Account

Mapped Type: BankAccount

Caption:

Message Number: 0 Set Number: 0

Default Cell Margins: Top: 100 Left: 70 Right: 240 Bottom: 70

1 [✓ 10] 100

Default Cell Gravity: Top Left

Insert Policy: Default

☐ Ignore Invisible Children

☐ Collapse on Delete

The values you enter in the Default Cell Margin fields determine the space (in millimeters) between the widget and the cell margin. The value you enter in the Default Cell Gravity field determines the alignment of the widget within its cell.

One of the optional properties you can specify for a grid field is its mapped type. This is the name of a class to which the grid field maps. Mapping a grid field to a class can simplify the coding required to display a business object.

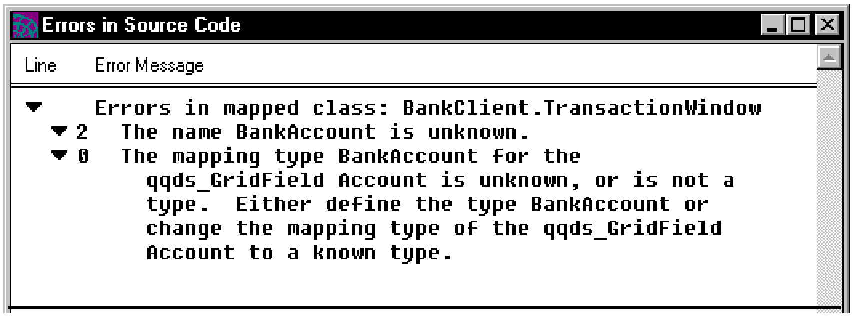
If you map a grid field, the named attributes of the widgets contained by the grid field must match the attribute names and types of the class to which you are mapping the grid field. When you want to display the business object in the window, instead of assigning each attribute to a widget, you simply assign the entire business object reference to the grid field data attribute. You will be using this technique in the Transaction window's Display method, by assigning a BankAccount object directly to the grid field.

8. Click OK.

Note the changed appearance of the Transaction window. We leave it as an optional exercise for you to grid the other widgets in the window. For more information about grids, see the *iPlanet UDS Programming Guide* and the Display Library online Help.

9. Choose File > Save All from the Window Workshop.

iPlanet UDS displays the following error message:



```
▼      Errors in mapped class: BankClient.TransactionWindow
  ▼ 2    The name BankAccount is unknown.
  ▼ 0    The mapping type BankAccount for the
           qqds_GridField Account is unknown, or is not a
           type.  Either define the type BankAccount or
           change the mapping type of the qqds_GridField
           Account to a known type.
```

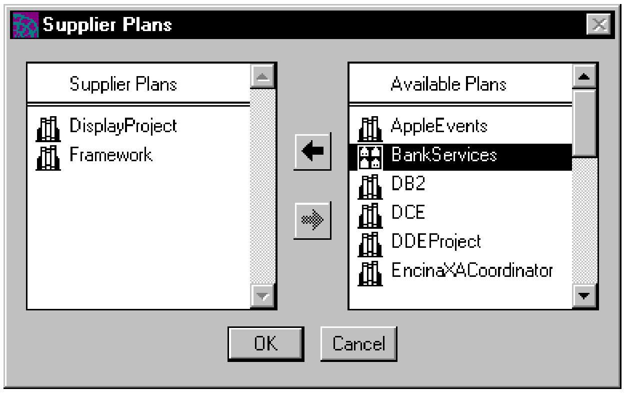
To make the BankAccount type known to the BankClient project, you must make the BankServices project (which defines the BankAccount class) a supplier plan to the BankClient project. Once you do this, classes defined in the BankClient project will be able to access classes and other elements defined in the BankServices project. (The concept of supplier plans was introduced in "Including a Supplier Plan" on page 32.)

➤ **To make a project a supplier plan to another project**

1. Open the BankClient project by double-clicking its icon in the workspace window.

2. Choose the File > Supplier Plans… command.

   iPlanet UDS displays the Supplier Plans window, which contains two scrolling lists: one labeled Supplier Plans and another labeled Available Plans. To make an available plan a supplier plan, you must select it and move it to the Supplier Plan scroll list.



3. Select the BankServices project in the Available Plans scroll list.

4. Click the left arrow to move the project to the Supplier Plans scroll list and click OK.

5. Choose File > Save All again from the Window Workshop.

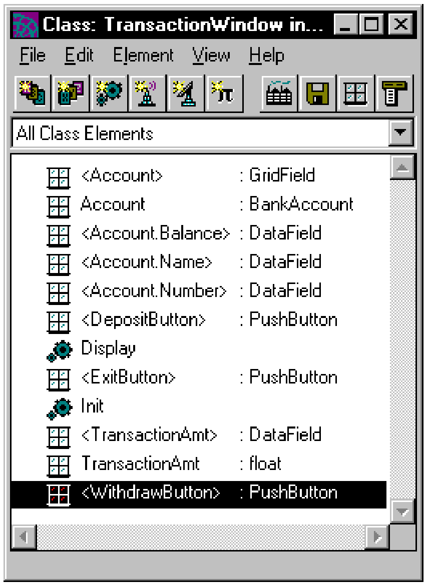
   This time, iPlanet UDS is able to compile and save your work.

6. Close the Window Workshop.

Window Classes Before and After Gridding

Let’s look at the TransactionWindow class again.

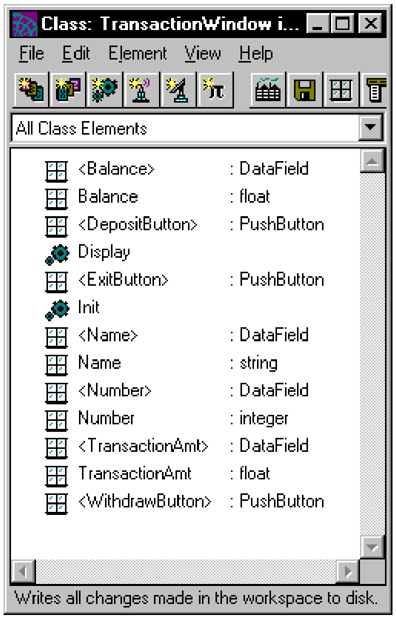
Figure 4-5 TransactionWindow Class After Gridding



```
<Account>          : GridField
Account            : BankAccount
<Account.Balance>  : DataField
<Account.Name>     : DataField
<Account.Number>   : DataField
<DepositButton>    : PushButton
Display
<ExitButton>       : PushButton
Init
<TransactionAmt>   : DataField
TransactionAmt     : float
<WithdrawButton>   : PushButton
```

Do you see the difference? By creating the grid field, we have added two new attributes to the TransactionWindow: Account (an attribute of type BankAccount) and <Account> (a widget of type GridField). In addition, the widgets that were named <Balance>, <Name>, <Number>, are now shown as children of the grid field. Figure 4-6 shows how the TransactionWindow class looked before creating the grid field.

Figure 4-6 TransactionWindow Class Before Gridding



```
<Balance>          : DataField
Balance            : float
<DepositButton>    : PushButton
Display
<ExitButton>       : PushButton
Init
<Name>             : DataField
Name               : string
<Number>           : DataField
Number             : integer
<TransactionAmt>   : DataField
TransactionAmt     : float
<WithdrawButton>   : PushButton
```

Writes all changes made in the workspace to disk.

Coding a Window's Methods

In the next part of the tutorial, you will be modifying the Display method for the TransactionWindow so that the window displays updated information as the user deposits or withdraws money.

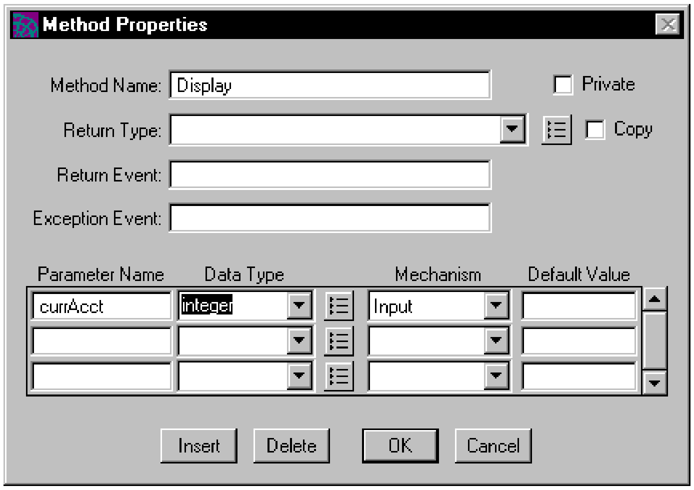
➤ **To modify the Display method for the TransactionWindow**

1. Double-click the Display method in the TransactionWindow class window.

   iPlanet UDS opens the Method Workshop and displays the contents of the default Display method. First, you need to modify the method signature so that the AccountWindow Display method can pass a parameter specifying the current account number when it calls this method.

2. Choose File > Properties from the Method Workshop.

   iPlanet UDS displays the Method Properties dialog.


Method Properties

Method Name: Display

Private: ☐

Return Type:

Copy: ☐

Return Event:

Exception Event:

Parameter Name	Data Type	Mechanism	Default Value
currAcct	integer	Input	

3. Enter `currAcct` for Parameter Name and `integer` for Data Type.

4. Click OK to close the dialog.

5. Modify the Display method so that it looks like the following:

```
Account = BankSO.GetAccountData (
      AcctNumber = currAcct);

self.Open();
event loop
    when <DepositButton>.Click do
      BankSO.UpdateAcct(acctNumber = currAcct,
          transactionAmt = TransactionAmt);
      self.TransactionAmt = 0;
    when <WithdrawButton>.Click do
      BankSO.UpdateAcct(acctNumber = currAcct,
        transactionAmt = -(TransactionAmt));
      self.TransactionAmt = 0;
    when <ExitButton>.Click do
      exit;
    when BankSO.AcctUpdated(AcctUpdated) do
      if AcctUpdated = currAcct then
        Account = BankSO.GetAccountData(currAcct);
      end if;
    when task.Shutdown do
      exit;
end event;
self.Close();
```

The Display method begins by invoking the GetAccountData method of the BankSO service object to get current account information that it can display in the Transaction window. Then, it opens the transaction window.

If the user clicks the Deposit button, the Display method invokes the UpdateAcct method of the BankSO service object, passing the TransactionAmt as a positive value. If the user clicks the Withdraw button, the Display method invokes the UpdateAcct method, passing the TransactionAmt as a negative value. In either case, it resets the TransactionAmt to zero.

When the user clicks the Exit button, the method exits.

The method also contains an event handler for the AcctUpdated event on the service object. When it receives such an event, the handler tests to see whether it concerns the current account. If it does, the handler invokes the GetAccountData method of the BankSO service object to update the account information being displayed in the TransactionWindow.

6. Close the Method Workshop.

You are now done creating the classes of the application.

Setting the Start Class and Method

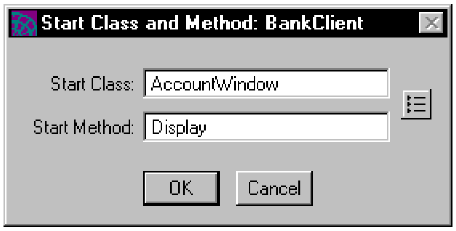
Before you can run the application, you need to specify the start class and method. These define the entry point to the application. iPlanet UDS begins execution of the application by creating an object of the start class and then invoking the start method on it.

➤ **To set a start method and class**

1. Open the BankClient project and choose the File > Start Class Method… command.

   iPlanet UDS displays the Start Class and Method dialog.

2. Enter `AccountWindow` for the StartClass property, and enter `Display` for the Start Method property as shown below.



Start Class and Method: BankClient

Start Class: AccountWindow

Start Method: Display

3. Click OK.

The start method for an application cannot have parameters and cannot be the Init method. iPlanet UDS will instantiate the object for you before calling the method you specified. In this case, for example, you specified that the Display method be invoked, but before this happens, the Init method is actually called first.

You can run your application now by choosing the Run > Test Run command. Remember that the only valid account numbers you can enter are 1000, 2000, 3000. When you run your application, you will still get a warning that there are methods that might not return a value. You will resolve this problem in the next section.

Chapter 5

Handling Exceptions

In this chapter you will create an exception class as well as the code for handling the exception.

About Exception Handling

Both the GetAccountData and the UpdateAcct methods of the BankMgr class search an array of accounts to find an account whose number is specified by the user. Neither method currently handles the case in which an account is not found. In this section of the tutorial, you will add an exception handler that will deal with this possibility. *Exception handling* means passing an exception to a special exception handler, outside the current block of code. This handler provides the code that handles the exception.

An *exception* is an abnormal condition. Such a condition might arise when the system detects an error; for example, division by zero, or a user input error. In the case of a system error, your application needs to handle the exception. In the case of an error that occurs in your code, you need to do the following:

- identify the possible source of an error and construct an exception object of a custom exception class
- use the `raise` statement to raise the exception in the method where the error might occur
- handle the exception in the method that calls the methods where the exception is raised

In the rest of this section you will create the AccountNotFound exception class, you will add code to the GetAccountData and UpdateAcct methods to raise this exception, and then you will add some code to the TransactionWindow.Display method to handle the exception.

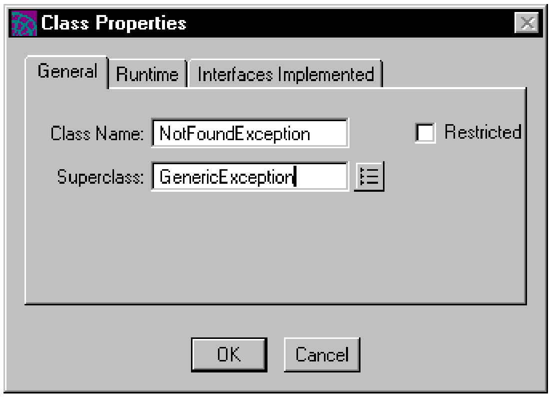
Creating an Exception

To create an exception, you need to create an exception class and make it a subclass of the GenericException class, which is a pre-defined iPlanet UDS class.

➤ **To create the exception class**

1. Open the BankServices project window.
2. Choose the Component > New > Nonwindow Class… command.

   The Class Properties dialog appears, as shown below.

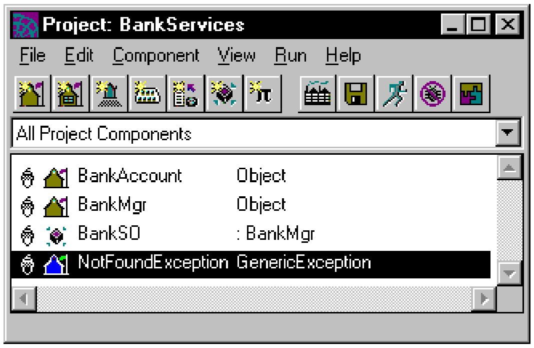


Class Name: NotFoundException
Restricted: [ ]
Superclass: GenericException

3. Enter `NotFoundException` as the Class Name.

4. Enter `GenericException` as the Superclass.

   iPlanet UDS provides a special class for exceptions, the GenericException class, which has attributes and methods defined that you can use when raising and handling exceptions. iPlanet UDS displays the name of the new class in the BankServices project window, as shown below, and opens the NotFoundException class window.



BankAccount	Object
BankMgr	Object
BankSO	: BankMgr
NotFoundException	GenericException

5. Close the NotFoundException class window.

Raising an Exception

Now you need to add some code to the GetAccountData method to raise the exception. Raising an exception generates the exception to be handled by an exception handler.

➤ **To raise the AccountNotFound exception**

1. Open the BankMgr class window.
2. Double-click the GetAccountData method to open the Method Workshop.

3. Add the following code to the method after the `end for` statement:

```
accountNotFound: NotFoundException = new();
accountNotFound.SetWithParams(severity = SP_ER_ERROR,
  message = ’The account number you entered does not exist’)
task.ErrMgr.AddError(accountNotFound);
raise accountNotFound;
```

   All subclasses of GenericException inherit the SetWithParams method from the superclass ErrorDesc. This method allows you to specify the Severity attribute for the class and a message that is displayed when the exception is handled. The code SP_ER_ERROR indicates an error in user code.

   Calling the ErrorMgr.AddError method allows iPlanet UDS to add a user-raised exception to the Error Stack, which is a listing of all exceptions that have been raised while the application has been executing.

   For more information about the GenericException, ErrorDesc, and ErrorMgr classes, see the Framework Library online Help.

4. Choose File > Compile to compile the method and close the Method Workshop.
5. Open the Method Workshop for the UpdateAcct method.
6. Add the same code to that method after the `for` loop and compile the method.
7. Close the Method Workshop and all windows in the BankServices project.

Handling an Exception

Finally, you need to add code to the Display method of the TransactionWindow class to handle the exception.

➤ **To handle the AccountNotFound exception**

1. Open the BankClient project and the TransactionWindow class.
2. Open the Method Workshop for the TransactionWindow Display method.
3. Add the following code to the Display method after the `self.Close` method call:

```
exception
  when accountNotFound : NotFoundException do
    self.window.MessageDialog(
     messageText = accountNotFound.message,
     messageType = accountNotFound.severity);
    self.Close();
```

   An exception handler begins with the word `exception`, followed by one or more `when` clauses. Each `when` clause contains the statement block to handle a given exception class. When an exception object that is raised is of the exception class defined in the `when` clause, the statement block for that clause is executed. In this case, the exception handler contains only one `when` clause, which executes when an exception of the NotFoundException class is raised.

   The code shown above also calls the UserWindow.MessageDialog method. This method displays an error message when an exception occurs. The Message attribute of the exception object is assigned to the MessageText parameter of the MessageDialog method, and the Severity attribute of the exception object is assigned to the MessageType parameter. For more information about the UserWindow.MessageDialog method, see the Display Library online Help.

4. Choose File > Compile to compile the method.
5. Close all windows except your Repository Workshop.

Checking Your Work

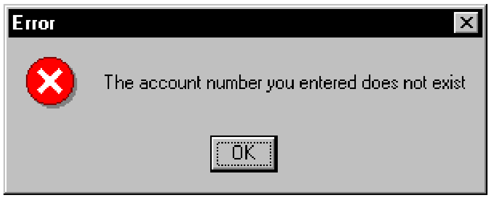
To check the exception handling code you entered, run the application and enter an account number you know is invalid.

➤ To run the application

1. Select the BankClient project.
2. Click the Run icon to run the application.
3. Enter any number in the Account number field other than 1000, 2000, or 3000.

   iPlanet UDS executes the exception handler you just added and displays the following dialog

.



The account number you entered does not exist

4. Click OK to close the dialog and exit the Welcome window.
5. Choose File > Save All.

For more information about exception handling, see the *TOOL Reference Guide*.

Chapter 6

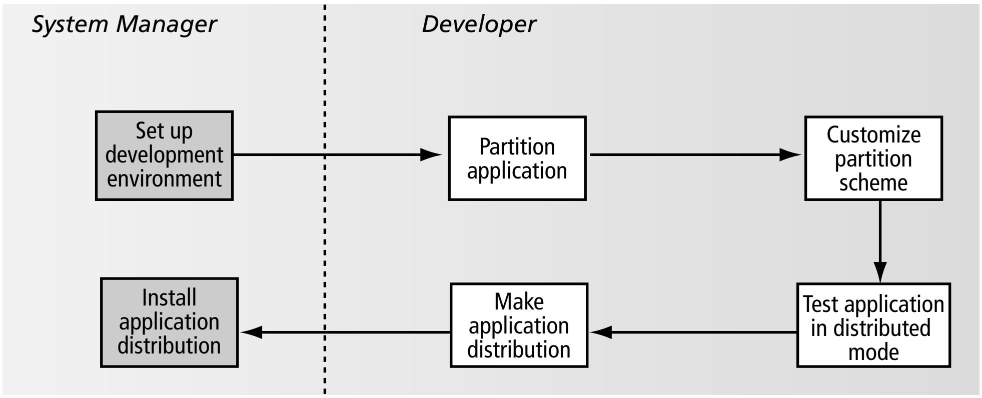
Testing and Running the Application

So far you have used the iPlanet UDS workshops to develop a logical application without worrying about the details of the environment in which the application will be deployed. In the last step of the development process, you are going to partition this logical application for a specific environment.

The Partitioning and Deployment Process

Figure 6-1 shows the steps required to complete the partitioning and deployment process. The rest of this section will provide a more detailed view of each step.

Figure 6-1 Making a Distribution and Installing the Application



System Manager
Developer
Set up development environment
Partition application
Customize partition scheme
Test application in distributed mode
Make application distribution
Install application distribution

In general, before you can partition the application, the system manager must set up the iPlanet UDS environment in which the application is partitioned. If you are running iPlanet UDS in distributed mode, you are already connected to an environment. You may or may not be able to duplicate the steps in this section of the tutorial depending on whether you are working on a machine that can be a server node. If you are running on a machine that is a pure client, you will be able to perform most of these steps, but you will not be able to install the application on your own machine.

For more information about partitioning and deployment, see the *iPlanet UDS Programming Guide*.

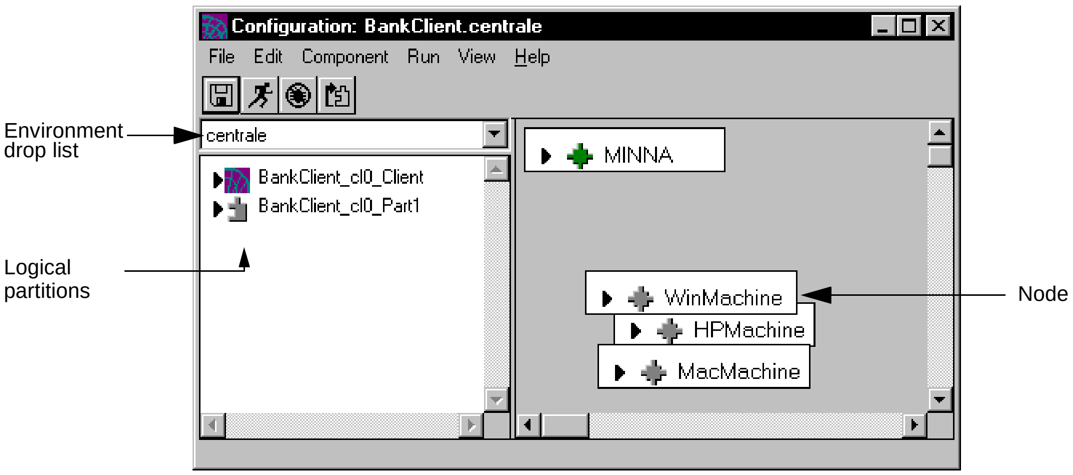
Partitioning the Application

When you use the Partition Workshop to partition your application, iPlanet UDS creates logical partitions and assigns them to various nodes in the environment based on the properties of each node. Each *partition* is an independent process that can run on its own machine. The result of assigning partitions to nodes in an environment is called a *configuration*.

➤ **To partition the application**

1. Open the BankClient project window.
2. Choose File > Configure as > Client.

   iPlanet UDS opens the Partition Workshop and displays the default partition configuration in the Configuration window, as shown below.



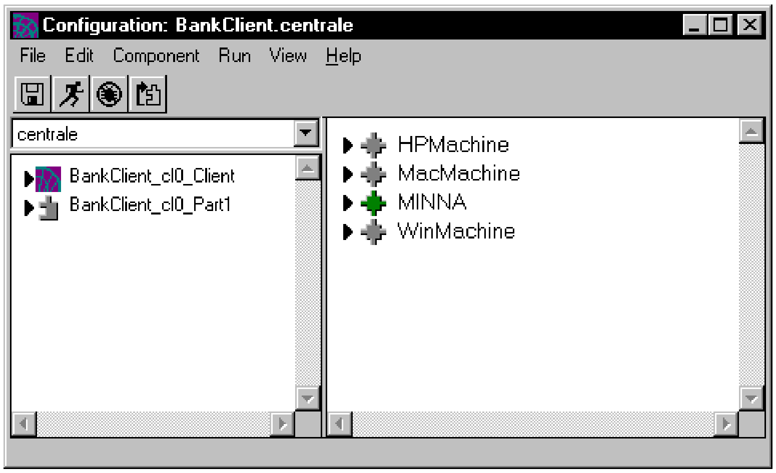
The left browser, called the Logical Partition browser, displays the application's logical partitions. The right browser, called the Node browser, displays the names of the nodes available in your environment. Just above the Logical Partition browser, there is a drop list that you can use to choose the environment in which to partition the application.

The node browser of your Partition Workshop window will look different, you might have only one node in your environment or you might have several, but their names and types will differ from the ones shown in the figure. You should be able to see the name of your own machine displayed in the node browser.

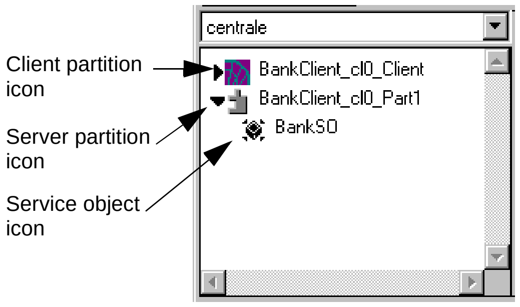
If the Logical Partition browser is not large enough to show the names of the logical partitions in their entirety, click the cursor on the vertical line dividing the browsers until it is changed into a cross-hair cursor and then drag to expand the browser to the right.

3. Choose View > Node Outline to change the view of the assigned nodes in the Node browser.

   iPlanet UDS changes the display to look like the one below.

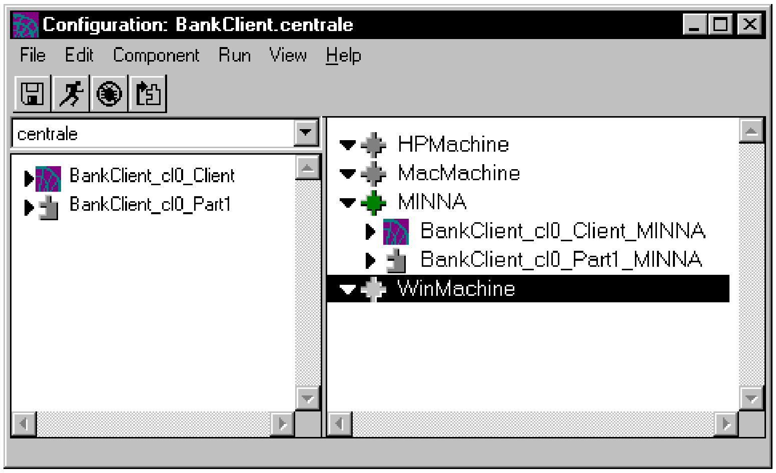


4. To see the contents of your logical partitions, click on the arrows to the left of the partition icon. Note the use of the icons to identify logical partition types.



5. To see how your logical partitions are assigned to the nodes in your environment, click on the arrows to the left of the node icons in the Node browser.

In the figure below, you can see that the BankClient client partition and the BankClient server partition are both assigned to the node named MINNA. If they were installed on this node, they would run as two separate processes on this machine.



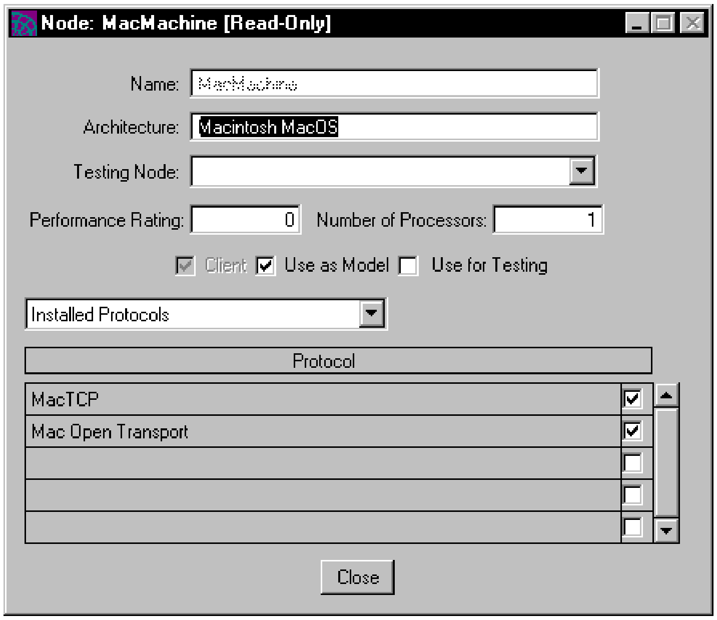
Getting Information About Nodes

You can get more information about the nodes in your environment to help you decide how to customize your configuration.

➤ To display information about individual nodes

1. Select the node and choose the Component > Properties… command.

   iPlanet UDS displays a Node dialog like the following:



Node: MacMachine [Read-Only]

Name: MacMachine

Architecture: Macintosh MacOS

Testing Node:

Performance Rating: 0 Number of Processors: 1

[✓] Client [✓] Use as Model [ ] Use for Testing

Installed Protocols

Protocol	
MacTCP	✓
Mac Open Transport	✓

Close

   The node properties that are displayed in this dialog were specified for the node when the environment was created. You cannot change them in the Partition Workshop.

2. Close the Node dialog.

Customizing the Partitioning Scheme

You can change the default partitioning configuration in the Partition Workshop by modifying the logical partitioning scheme or by modifying node assignments. After customizing the partitioning scheme, you can run the application in test mode. During this stage, iPlanet UDS creates the application partitions in the development environment and runs it as a distributed application.

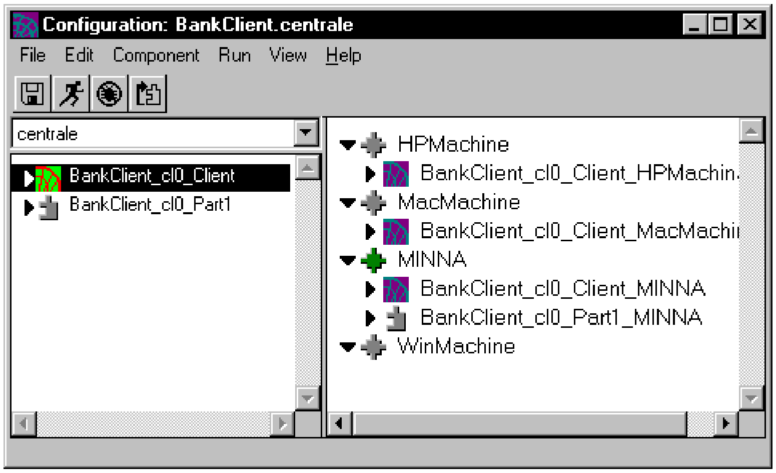
For complete information about customizing the partitioning scheme, see the *iPlanet UDS Programming Guide*.

Unless your machine is a pure client, the client and server partitions should be assigned on your machine. For this exercise, you want all partitions to be placed on your machine. This way you can be sure that you have read/write access to your own machine and complete the subsequent steps of the tutorial.

➤ **To move a client or server partition from another node to your own node**

1. Click the partition name and drag it to your node.

   The figure below shows one possible configuration for the application.


Configuration: BankClient.centrale
File Edit Component Run View Help
centrale
BankClient_cl0_Client
BankClient_cl0_Part1
HPMachine
BankClient_cl0_Client_HPMachin
MacMachine
BankClient_cl0_Client_MacMachi
MINNA
BankClient_cl0_Client_MINNA
BankClient_cl0_Part1_MINNA
WinMachine

2. Experiment with your own configuration to see what you can do.

3. When you are done, make sure that the client and server partitions are assigned to your machine.

Testing Your Configuration

When you have customized your configuration to your satisfaction, it is time to test the application as a distributed application using the configuration you just defined.

To run your application using the configuration that you have defined in the Partition Workshop, click the Run icon in the Partition Workshop. iPlanet UDS creates a process that instantiates the BankSO service object in memory on the node you have chosen, and you can test your application in distributed mode. Note, however, that no executable files are created and installed until you complete the next step, which is making a distribution.

Making a Distribution

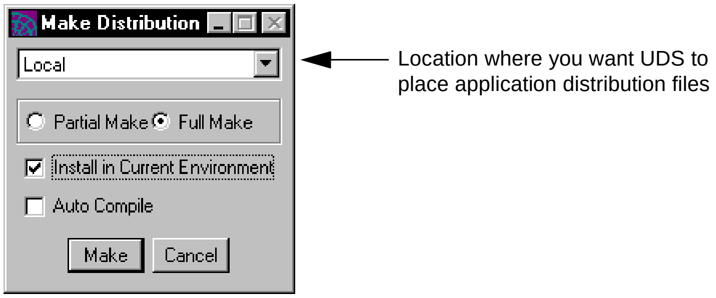
After testing, you can make an *application distribution*. During this stage, iPlanet UDS creates all the files needed to install the application into a deployment environment. During installation, which is typically done by the system manager, code for each partition is installed in a location on its assigned node.

➤ **To make an application distribution**

1. In the Partition Workshop, choose the File > Make Distribution… command.

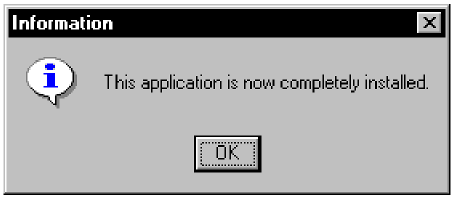
   iPlanet UDS displays a dialog that you use to specify on which node you want the files created, whether you want a full or partial distribution, and whether you want it to compile partitions that are marked as compiled.

   The first time you make a distribution, you should choose the Full Make option. Subsequently, you can choose Partial Make.


Make Distribution
Local
Partial Make
Full Make
Install in Current Environment
Auto Compile
Make
Cancel
Location where you want UDS to place application distribution files

2. Specify the name of the node where you want the application distribution files placed by selecting an item from the drop list. Unless you have good reason to do otherwise, it is best to make the distribution on your local machine.
3. Click the Full Make radio button.
4. If you are running on a machine that can be a serve, turn on the Install in Current Environment toggle. Otherwise, leave this checkbox clear.
5. Leave the Auto Compile toggle off, because you have not marked any partitions to be compiled.
6. Click the Make button.

   If you are running on a machine that is not a pure client, iPlanet UDS will create the distribution files and install them on your machine. When it is finished, iPlanet UDS displays the following message to inform you when the distribution is complete.


Information

This application is now completely installed.

   If you are making a distribution on a pure client, iPlanet UDS displays the message “Distribution made” on the status line. iPlanet UDS has created the files that need to be installed; you need to have the system manager install these files on the appropriate machines.

7. Click the OK button to close the dialog.

Running Your Application

If you have been using a UNIX or NT machine to create your application, you can run it by entering one of the following commands:

- If you are using the launch server, enter the following command:

  ```
  ftcmd run BankClient
  ```

- If you are not using the launch server, enter the following command on UNIX:

  ```
  ftexec -fi bt:FORTE_ROOT/userapp/bankclie/cl0/bankcl0
  ```

- Enter the following command on WindowsNT:

  ```
  ftexec -fi bt:FORTE_ROOT\userapp\bankclie\cl0\bankcl0
  ```

Congratulations! You have just created your first iPlanet UDS application.

Checking Concurrency

If you want, take another few minutes to see how the service object is shared by two clients.

➤ **To check concurrency**

1. Launch your application.
2. Launch your application again.

   Two Welcome windows should now be open on your screen.
3. Enter the same account number in each window and click the Display button in each window to open two Transaction windows.
4. Make a deposit or withdrawal in one of the Transaction windows.

   Note that the account and balance information is instantly updated in the other Transaction window. Your AcctUpdated event has been sent by the service object to both clients.
5. You can quit both applications now.

Index