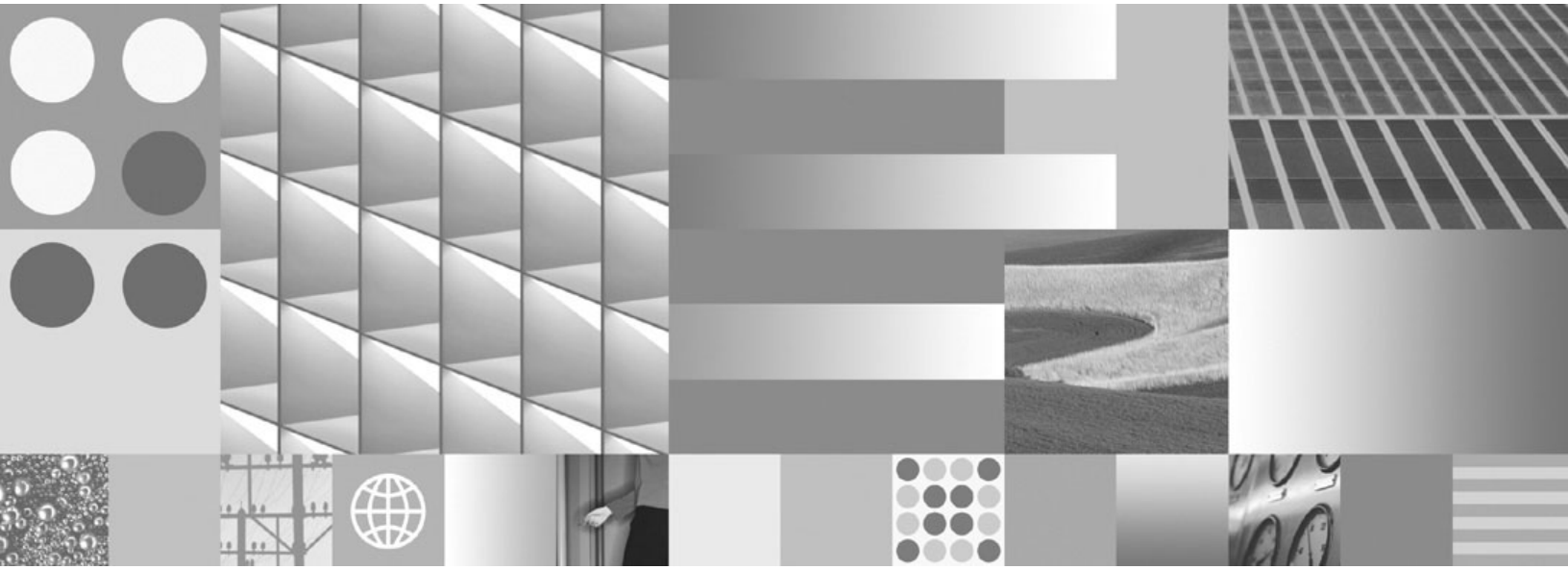
IBM Informix

Version 4.20

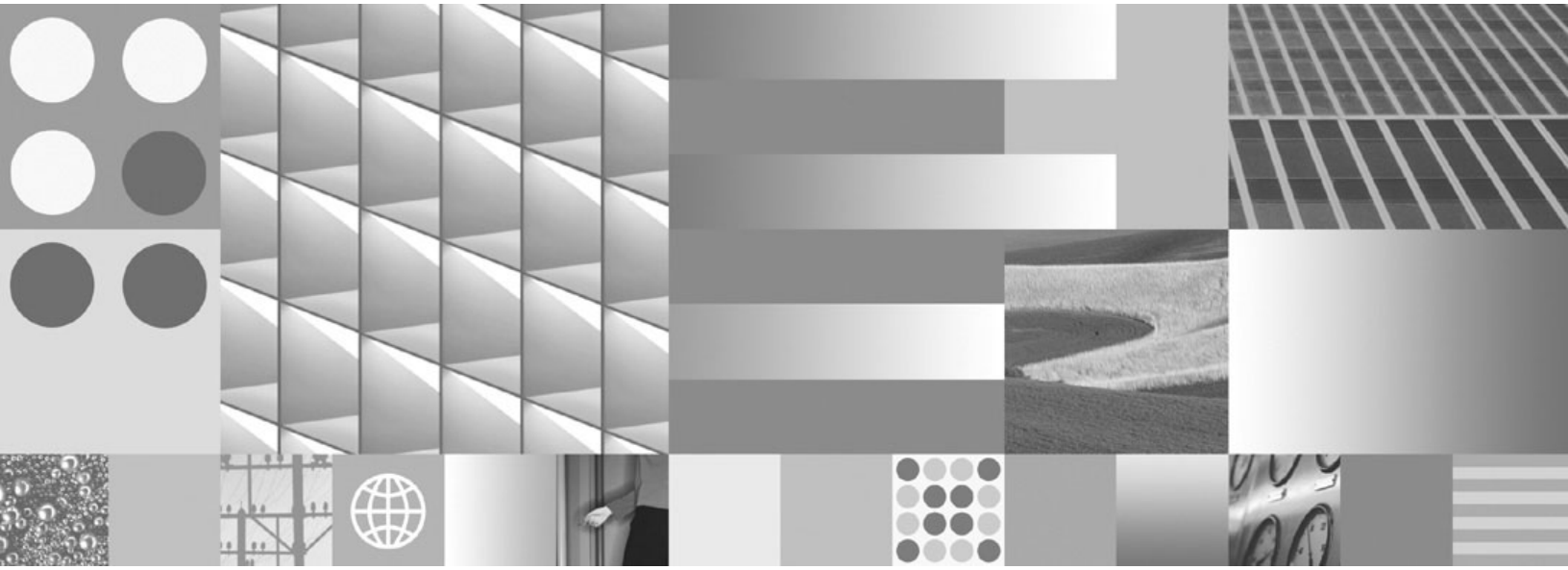


IBM Informix DataBlade Developers Kit User's Guide

G229-6366-01

IBM Informix

Version 4.20



IBM Informix DataBlade Developers Kit User's Guide

G229-6366-01

Note:
Before using this information and the product it supports, read the information in "Notices" on page E-1.

This edition replaces G229-6366-00.

This document contains proprietary information of IBM. It is provided under a license agreement and is protected by copyright law. The information contained in this publication does not include any product warranties, and any statements provided in this publication should not be interpreted as such.

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

© Copyright International Business Machines Corporation 1996, 2008. All rights reserved.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

© Copyright IBM Corp. 1996, 2008

Introduction

In This Introduction

This introduction provides an overview of the information this publication provides and the conventions it uses.

About This Publication

The *IBM Informix DataBlade Developers Kit User's Guide* describes how to use IBM® Informix DataBlade Developers Kit tools to develop and package DataBlade modules. A DataBlade module extends the functionality of Informix Dynamic Server to handle data with user-defined routines or to handle nontraditional kinds of data, such as full text, images, video, spatial, and time series data.

This section discusses the intended audience and the associated software products that you must have to develop and use the DataBlade module.

Types of Users

This guide is for experienced C, C++, or Java™ programmers who are comfortable writing libraries to support applications. You will use this guide to develop DataBlade modules that extend your Informix® database server.

If you are unfamiliar with DataBlade® modules, read *DataBlade Module Development Overview* before you read this publication.

Software Dependencies

Check the IBM Informix DataBlade Developers Kit release notes for software compatibility requirements for IBM Informix Dynamic Server and IBM Informix Client Software Development Kit (Client SDK).

To use DBDK to develop your DataBlade module in a Windows® development environment, you need to install the following software:

- Microsoft® Visual C++ 6.0
- Netscape Navigator 4.0 (or later) or Microsoft Internet Explorer 4.0 (or later)

To use BladePack to package your DataBlade module with an interactive installation for Windows, you need an InstallShield professional license.

© Copyright IBM Corp. 1996, 2008

To use the *IBM Informix DataBlade Developers Kit InfoShelf*, you need one of the following browsers:

- Netscape Navigator 4.0 or later
- Microsoft Internet Explorer 4.0 or later

For system requirements and installation instructions, see the *IBM Informix Read Me First* sheet for the Informix DataBlade Developers Kit.

Documentation Conventions

This section describes the following conventions, which are used in the product documentation for IBM Informix Dynamic Server:

- Typographical conventions
- Feature, product, and platform conventions
- Example code conventions

Typographical Conventions

This publication uses the following conventions to introduce new terms, illustrate screen displays, describe command syntax, and so forth.

Convention	Meaning
KEYWORD	Keywords of SQL, SPL, and some other programming languages appear in uppercase letters in a serif font.
italics	Within text, new terms and emphasized words appear in italics. Within syntax and code examples, variable values that you are to specify appear in italics.
boldface	Names of program entities (such as classes, events, and tables), environment variables, file names, path names, and interface elements (such as icons, menu items, and buttons) appear in boldface.
`monospace`	Information that the product displays and information that you enter appear in a monospace typeface.
`KEYSTROKE`	Keys that you are to press appear in uppercase letters in a sans serif font.
>	This symbol indicates a menu item. For example, "Choose **Tools > Options**" means choose the **Options** item from the **Tools** menu.

Feature, Product, and Platform Markup

Feature, product, and platform markup identifies paragraphs that contain feature-specific, product-specific, or platform-specific information. Some examples of this markup follow:

Dynamic Server

Identifies information that is specific to IBM Informix Dynamic Server

End of Dynamic Server

Windows Only

Identifies information that is specific to the Windows operating system

End of Windows Only

This markup can apply to one or more paragraphs within a section. When an entire section applies to a particular product or platform, this is noted as part of the heading text, for example:

Table Sorting (Windows)

Example Code Conventions

Examples of SQL code occur throughout this publication. Except as noted, the code is not specific to any single IBM Informix application development tool.

If only SQL statements are listed in the example, they are not delimited by semicolons. For instance, you might see the code in the following example:

```
CONNECT TO stores_demo
...

DELETE FROM customer
   WHERE customer_num = 121
...

COMMIT WORK
DISCONNECT CURRENT
```

To use this SQL code for a specific product, you must apply the syntax rules for that product. For example, if you are using an SQL API, you must use EXEC SQL at the start of each statement and a semicolon (or other appropriate delimiter) at the end of the statement. If you are using DB–Access, you must delimit multiple statements with semicolons.

Tip: Ellipsis points in a code example indicate that more code would be added in a full application, but it is not necessary to show it to describe the concept being discussed.

For detailed directions on using SQL statements for a particular application development tool or SQL API, see the documentation for your product.

Additional Documentation

You can view, search, and print all of the product documentation from the IBM Informix Dynamic Server information center on the Web at http://publib.boulder.ibm.com/infocenter/idshelp/v115/index.jsp.

For additional documentation about IBM Informix Dynamic Server and related products, including release notes, machine notes, and documentation notes, go to the online product library page at http://www.ibm.com/software/data/informix/pubs/library/. Alternatively, you can access or install the product documentation from the Quick Start CD that is shipped with the product.

Compliance with Industry Standards

The American National Standards Institute (ANSI) and the International Organization of Standardization (ISO) have jointly established a set of industry standards for the Structured Query Language (SQL). IBM Informix SQL-based products are fully compliant with SQL-92 Entry Level (published as ANSI X3.135-1992), which is identical to ISO 9075:1992. In addition, many features of IBM Informix database servers comply with the SQL-92 Intermediate and Full Level and X/Open SQL Common Applications Environment (CAE) standards.

How to Provide Documentation Feedback

You are encouraged to send your comments about IBM Informix user documentation by using one of the following methods:

- Send e-mail to docinf@us.ibm.com.
- Go to the Information Center at http://publib.boulder.ibm.com/infocenter/idshelp/v115/index.jsp and open the topic that you want to comment on. Click **Feedback** at the bottom of the page, fill out the form, and submit your feedback.

Feedback from both methods is monitored by those who maintain the user documentation of Dynamic Server. The feedback methods are reserved for reporting errors and omissions in our documentation. For immediate help with a technical problem, contact IBM Technical Support. For instructions, see the IBM Informix Technical Support Web site at http://www.ibm.com/planetwide/.

We appreciate your suggestions.

Chapter 1. Getting Started with DataBlade Module Development

In This Chapter

This chapter provides an overview of DataBlade module development and describes the resources and tools the Informix database server provides to facilitate development.

What is a DataBlade Module?

A *DataBlade module* is a software package that extends the functionality of your Informix database server. The package includes SQL statements and supporting code written in an external language or Informix SPL. DataBlade modules can also contain client components. A DataBlade module enables your Informix database server to provide the same level of support for new data types as it provides for built-in data types.

Users access DataBlade module services in the same way they access database server services: through SQL, SPL, and client programs linked with any of the Informix client APIs. DataBlade modules can also use the DataBlade API or SQL queries to access data types and routines in other DataBlade modules.

The Informix DataBlade Developers Kit aids you in developing DataBlade modules. It provides graphical user interfaces to complete tasks, and it generates much of the code you need for your DataBlade module.

DataBlade Developers Kit Tools

The Informix DataBlade Developers Kit provides the following graphical user interfaces for creating and working with DataBlade modules:

- **BladeSmith.** A tool for organizing a DataBlade module development project. You use BladeSmith to create a project and then define the objects (such as data types and routines) that belong to the DataBlade module. BladeSmith generates source files, header files, make files, functional test files, SQL scripts, messages,

© Copyright IBM Corp. 1996, 2008

and packaging files. Chapter 4, "Creating DataBlade Objects Using BladeSmith," on page 4-1, describes how to use this tool.

- **DBDK Visual C++ Add-In and IfxQuery.** Tools for debugging a DataBlade module using Microsoft Visual C++ on Windows. The add-in automates many of the debugging tasks and calls the IfxQuery tool to run unit tests for DataBlade module routines. Chapter 10, "Debugging and Testing DataBlade Modules on Windows," on page 10-1, describes how to use these tools.
- **BladePack.** A tool for creating a DataBlade module package. BladePack can create a simple directory tree containing files to be installed or an installation that includes an interactive user interface. Chapter 11, "Using BladePack," on page 11-1, describes how to use this tool.
- **BladeManager.** A utility for registering and un-registering DataBlade modules in Informix databases. The *IBM Informix DataBlade Module Installation and Registration Guide* describes how to use this tool.

Preparing to Develop DataBlade Modules

This section suggests how to prepare for developing DataBlade modules. This overview is intended to act as a map for information resources.

To prepare for DataBlade module development, complete these general processes:

1. If necessary, familiarize yourself with IBM Informix software and documentation (see "Becoming Familiar with IBM Informix Software and Documentation," next).
2. Design your DataBlade module (see "Designing Your DataBlade Module" on page 1-3).

After you finish your preparations, you can develop your DataBlade module and then have it certified (see "Developing Your DataBlade Module" on page 1-5).

Becoming Familiar with IBM Informix Software and Documentation

Familiarizing yourself with IBM Informix software and documentation is critical for first-time DataBlade developers. However, it is important for experienced DataBlade developers too, because IBM Informix software and documentation are enhanced in each release.

To familiarize yourself with IBM Informix products and documentation:

1. Read *DataBlade Module Development Overview*.

   This publication briefly describes the database objects you can include in your DataBlade module and other options you have when you create a DataBlade module.
2. Install the necessary IBM Informix software.

   See "Installing IBM Informix Software" on page 1-3 for more information.
3. Learn to use IBM Informix software: at the very least, your database server, the Setnet32 utility (client connectivity), and the DB-Access or SQL Editor utilities (SQL querying).
4. Complete the Informix DataBlade Developers Kit Tutorial.

   See "DataBlade Developers Kit Tutorial" on page 1-3 for more information.
5. Create your own practice DataBlade module.

   See "Creating a Practice DataBlade Module" on page 1-3 for more information.

Some of these steps are described in the following sections.

Installing IBM Informix Software

Install and become familiar with the following IBM Informix software products:

- Your database server
- IBM Informix Client Software Development Kit (Client SDK)
- Informix DataBlade Developers Kit

In addition, if you plan to develop a DataBlade module in Java, you should become familiar with IBM Informix Dynamic Server with J/Foundation and the Java Development Kit (JDK). For information on the correct version and the source of the JDK, see the release notes for your database server.

Install the latest version of the IBM Informix software for your development environment. Although the Informix DataBlade Developers Kit is only available on Windows, it can generate DataBlade modules for UNIX® as well as Windows.

For information on currently available IBM Informix software releases, see the IBM Informix Developer Zone site at http://www.ibm.com/software/data/developer/informix.

DataBlade Developers Kit Tutorial

The Informix DataBlade Developers Kit Tutorial offers several exercises, each focusing on a single aspect of DataBlade module development.

To access the exercises, start the tutorial from the Informix DataBlade Developers Kit InfoShelf home page. You can launch the InfoShelf from the BladeSmith **Help** menu or start it independently by choosing **Start > Programs > Informix > DBDK InfoShelf**.

Creating a Practice DataBlade Module

To familiarize yourself with the entire development process, create a simple practice DataBlade module containing an easily implemented object, such as a user-defined routine that takes built-in data types as arguments. Be sure to write the code, test it, and debug it. Completing a simple DataBlade module helps you create a realistic estimate of the length of your development cycle.

Designing Your DataBlade Module

DataBlade modules can contain complex operations. A good design is critical to your success.

To design your DataBlade module:

1. Read about DataBlade module SQL design concepts.

   For DataBlade module SQL design issues, see Chapter 2, "Designing DataBlade Modules," on page 2-1.

   For general information about the options you have when you extend the server, see *IBM Informix User-Defined Routines and Data Types Developer's Guide*.
2. Write a functional specification.

   See "Writing a Functional Specification" on page 1-4 for more information.
3. Read Informix coding guidelines.

   See "Programming Resources" on page 1-4 for more information.
4. Write a design specification.

   See "Writing a Design Specification" on page 1-4 for more information.

5. Create an iterative development strategy.

   See "Creating an Iterative Development Plan" on page 1-5 for more information.

Some of these steps are described in the following sections.

Writing a Functional Specification

A functional specification describes the scope and functionality of your DataBlade module, without documenting implementation details. It also documents other issues for development, such as phases of functionality, compatibility, performance, and platform. A good functional specification shows how your DataBlade module solves the problem you designed it to solve.

For a sample functional specification, see the IBM Informix Developer Zone site at http://www.ibm.com/software/data/developer/informix.

Programming Resources

For specific language options and guidelines, see Chapter 3, "Programming Guidelines," on page 3-1.

The following table lists the programming language options you have when writing DataBlade module code and refers you to sources of information about them.

Language	Information Sources
C	Chapter 5, "Programming DataBlade Module Routines in C," on page 5-1 *IBM Informix DataBlade API Programmer's Guide*
ActiveX/C++ (client-side programming and Windows server projects only)	Chapter 6, "Creating ActiveX Value Objects," on page 6-1 Chapter 7, "Using ActiveX Value Objects," on page 7-1 *IBM Informix DataBlade API Programmer's Guide*
Java	Chapter 8, "Programming DataBlade Modules in Java," on page 8-1 *J/Foundation Developer's Guide* *IBM Informix JDBC Driver Programmer's Guide*
Stored Procedure Language (SPL)	*IBM Informix Guide to SQL: Tutorial*

For further tips on coding DataBlade modules, see the IBM Informix Developer Zone at http://www.ibm.com/software/data/developer/informix.

Writing a Design Specification

A design specification describes the overall functionality of your DataBlade module and documents the specific routines available to the user, the supporting database tables used to implement the routines, error messages, and the environment used to build the DataBlade module. A design specification also documents implementation details that the DataBlade module customer does not need to know, such as internal support routines.

For a sample design specification, see the IBM Informix Developer Zone at http://www.ibm.com/software/data/developer/informix.

Creating an Iterative Development Plan

Keep the following guidelines in mind when you create an iterative development plan:

- Plan the order in which to create objects.

  Some objects can depend on others; you must create new data types before you create the routines that operate on them. Create simple data types and routines before complex ones. Create objects in the smallest independently testable groups. For example, you can test opaque data type support routines without any other objects.

- Add unit and functional test data for opaque data type support routines, user-defined routines, and cast support routines as you create them.

  Unit tests are SQL files you use to test boundary conditions while debugging your DataBlade module on Windows. After you generate unit tests for all your routines with BladeSmith, you add test data to them. If you later regenerate unit tests, the changes you made are merged into the new unit test files.

  Functional tests are scripts you execute to validate your DataBlade module on UNIX after you finish debugging it. You can also run functional tests on Windows if you use a UNIX emulator, such as MKS Toolkit. Before you generate functional tests in BladeSmith, you must enter functional test data for all your routines. You can add custom scripts, but if you alter existing scripts and then regenerate them, changes you made are overwritten. See "Adding Functional Test Data" on page 4-36 for more information.

- Include tracing when you generate code.

  If you enable tracing when you generate code in BladeSmith, BladeSmith includes enter and exit tracing for every routine. You can also add more tracing. See "Generating Files" on page 4-40 for information on how to generate code with tracing and "Tracing and Error Handling" on page 5-7 for information on the generated tracing.

- Add custom error messages.

  Anticipate how your customers will use your DataBlade module and create error messages that sensibly report problems to your users. See "Defining Errors" on page 4-13 for information on how to define error messages and "Tracing and Error Handling" on page 5-7 for information on how to add custom error handling to your DataBlade code.

Tip: Although you can use BladeSmith to define all of the objects in a DataBlade project before you edit and test the code, you might find it helpful to develop a modular plan to define and test objects one by one before you test the project as a whole.

Developing Your DataBlade Module

Developing your DataBlade module is an iterative process that involves creating objects in BladeSmith, generating code, editing and compiling code, and testing and debugging code. When you identify errors, you must repeat the process to correct errors. DataBlade development can be iterative in another way: you can create objects in BladeSmith one by one, coding and testing each one before creating the next. When you are finished developing your DataBlade module, you package it for distribution.

To create your DataBlade module:

1. Create a project in BladeSmith.

   See "Creating a New Project" on page 4-4 for more information.

2. Define the contents of your DataBlade module in BladeSmith.

   See "Creating DataBlade Module Objects" on page 4-8 for more information.
3. Generate DataBlade module code in BladeSmith.

   See "Generating Files" on page 4-40 for more information.
4. Edit and compile DataBlade module code.

   See "Editing and Compiling DataBlade Module Code" on page 1-7 for more information.
5. Debug your DataBlade module code.

   See "Debugging Your DataBlade Module" on page 1-7 for more information.
6. Repeat steps 2 through 5 until your DataBlade module is complete and the code functions properly.
7. Test your DataBlade module code.

   Run generated functional test scripts on UNIX or on Windows with a UNIX emulation program. For instructions for UNIX, see "Performing Functional Tests" on page 9-7. For instructions for Windows, see "Performing Functional Tests on DataBlade Modules" on page 10-6.
8. If necessary, repeat steps 4 through 7 until your DataBlade module is complete and the code functions properly.
9. Package your DataBlade module with BladePack.

The following diagram illustrates the basic steps in DataBlade module development and lists the tools you use for Windows, UNIX, and Java.

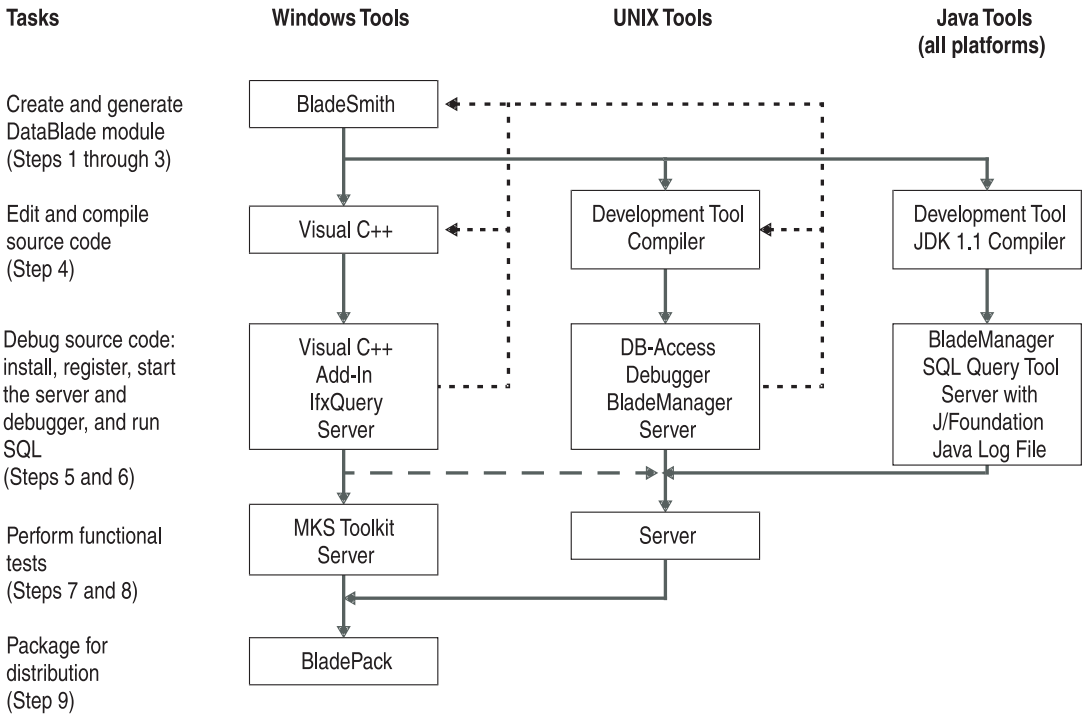


Figure 1-1. DataBlade Module Development Cycle

Editing and Compiling DataBlade Module Code

Windows Only

If you are developing a DataBlade module in C or C++, you use Microsoft Visual C++ 6.0 to edit and compile your source code on Windows.

End of Windows Only

UNIX Only

If you are developing a DataBlade module in C, you can use any standard UNIX development tool and compiler to edit and compile your source code.

End of UNIX Only

If you are developing a DataBlade module in Java, you can use any standard UNIX or Windows development tool to edit your source code. Use the JDK 1.1.*x* compiler to compile it.

For more information about programming and compiling, see the following chapters:

- Chapter 5, "Programming DataBlade Module Routines in C," on page 5-1
- Chapter 6, "Creating ActiveX Value Objects," on page 6-1
- Chapter 8, "Programming DataBlade Modules in Java," on page 8-1

Debugging Your DataBlade Module

Debugging a C or C++ DataBlade module consists of the following general steps:

1. Install the DataBlade module on the database server.
2. Set breakpoints in your source code file.
3. Start and attach the debugger to the database server process.
4. Register the DataBlade module in your test database.
5. Run SQL queries (unit tests) to test the source code marked with breakpoints.

Windows Only

If you are debugging C or C++ DataBlade module source code on Windows, you use Microsoft Visual C++, the DBDK Visual C++ Add-In, and the IfxQuery tool. The **Debug DataBlade Module** command of the add-in installs the DataBlade module on the local database server, starts the debugger and database server, and calls IfxQuery to register the DataBlade module and run the unit tests that halt at breakpoints in the source code. The **Debug DataBlade Module** command functions only if the database server is installed on the same computer on which you are debugging.

For more information on debugging C and C++ code on Windows, see Chapter 10, "Debugging and Testing DataBlade Modules on Windows," on page 10-1.

End of Windows Only

Windows NT Only

UNIX Only

If you are debugging a DataBlade module on UNIX, you must install the DataBlade module, start the database server and debugger, register the DataBlade module with BladeManager, and use DB-Access to execute SQL statements that halt at breakpoints in the source code.

For more information on debugging C code on UNIX, see Chapter 9, "Debugging and Testing DataBlade Modules on UNIX," on page 9-1.

End of UNIX Only

End of Windows NT Only

Windows NT Only

JAVA Language Support

Debugging a DataBlade module written in Java consists of the following general steps:

1. Install the DataBlade module.
2. Start the database server process.
3. Register the DataBlade module in your test database.
4. Run SQL queries (unit tests) to test the source code marked with breakpoints.
5. Examine the Java log file for errors.

For more information on debugging Java code, see "Debugging and Testing DataBlade Modules Written in Java" on page 8-9.

End of JAVA Language Support

End of Windows NT Only

Packaging Your DataBlade Module

With BladePack, you can create an interactive installation program for the following environments:

- UNIX
- Windows with InstallShield 3.1
- Windows with InstallShield 5.1

You must include the generated SQL scripts and all the shared library files you produced when you compiled your DataBlade module source code. However, consider also including:

- Custom installation extensions
- Documentation for your DataBlade module
- Examples for your DataBlade module

Chapter 2. Designing DataBlade Modules

In This Chapter

This chapter describes DataBlade module SQL design issues. For language-specific design issues, see Chapter 3, "Programming Guidelines," on page 3-1

Data Model

A *data model* is a high-level definition of a DataBlade module: what objects it represents and what operations on those objects it provides. Here are some issues to keep in mind when you design a data model:

- Consider the data model independently from the applications that use it and the user interfaces required by those applications.
- Concentrate on designing a service, rather than an application.
- Build a data model that is reusable by multiple client applications, rather than creating one tailored to a particular client application.

For example, consider the fictitious SimpleMap DataBlade module, which stores, manipulates, and displays maps. The data model for this DataBlade module might specify:

- Spatial data types, such as polygons to represent counties and cities and line segments to represent roads and rivers

© Copyright IBM Corp. 1996, 2008

- Operations performed on the spatial data types, such as search and comparison routines that determine whether a particular city lies within a particular county or whether two roads intersect

This data model allows you to make requests such as, "Find all the polygons in the map that fall inside the currently visible region, where the currently visible region is a given polygon." The query scans the database and returns only the polygons that meet the request criteria.

However, the display logic for the data types does not belong in this data model; the rendering of polygons is a user-interface issue. After the desired polygons have been retrieved from the database, the client application displays them.

The data model for a DataBlade module must be simple to understand, and the DataBlade module must provide a rich set of services using a minimal set of routines. This fundamental software design concept applies to DataBlade modules in particular, because DataBlade modules are intended for use by other developers.

For example, in the SimpleMap DataBlade module, assume that users want to find overlapping regions on a map. The DataBlade module can provide a number of different interfaces to support the query. Two examples are as follows:

```
Overlaps(Polygon, Polygon)
Contains(Polygon, Polygon)
```

The **Overlaps()** routine returns `TRUE` if any parts of the two polygons overlap, while the **Contains()** routine returns `TRUE` if the first polygon completely contains the second. These two routines are simple to understand and easy to remember.

However, if the semantics of two routines are too similar, users may have difficulty remembering which routine computes which value. For example, assume the SimpleMap DataBlade module also provides the following **Intersects()** routine:

```
Intersects(Polygon, Polygon)
```

This routine returns `TRUE` if the boundaries of the two polygons intersect. `Intersects(a, b)` is equivalent to the following statement:

```
Overlaps(a, b) and not (Contains(a, b) or Contains(b, a))
```

Intersects() and **Overlaps()** are confusing when both are supplied. Because **Overlaps()** and **Contains()** together can compute intersections, it is probably best to leave **Intersects()** undefined.

When you design a data model, separate the routines used by a single application from the more general service routines. For example, perhaps you want to provide a routine that takes a polygon as an argument and returns another polygon with a border exactly one pixel outside the original. To display the new polygon, the two polygons are used together to create the appearance of a polygon with a thick border.

Such a routine is probably useful only for a particular application that displays thick-bordered polygons; it is not useful to other applications that operate on spatial data. Thus, it is a poor candidate for inclusion in a DataBlade module.

In summary, these are the issues to consider when you design the data model for a DataBlade module:

- Separate the user interface from the abstract operations on data.

- Think about data types and routines that operate on them.
- Keep the design simple.
- If you are building a production DataBlade module, do not add server routines intended to support a single application.

Data Type Design

After you have designed a data model for your DataBlade module, you can design its specifics, such as data types to best represent your DataBlade module objects.

Your Informix database server supports a rich set of data types, known as *built-in data types*. (For information on the built-in types, see the *IBM Informix Guide to SQL: Reference*.) It is recommended that you use built-in types wherever possible; however, even with built-in types, consider the following design issues:

- How accessible must the elements of each object be?
- How large is each object?

This section provides some guidelines for making these design decisions.

Object Accessibility

Users are likely to query the data of two extended types: *row data types* and *opaque data types*. To decide which data type to use, consider how accessible the elements of each DataBlade module object should be:

- Use row data types for any object that is a container and whose elements users always want to access.
- Use opaque data types for indivisible objects or for objects whose representation you want to hide from your users.

For example, say the users of the SimpleMap DataBlade module want to see and operate directly on the street number and name, country, and postal code. You might decide to provide the **Address** data type.

If you create **Address** as an opaque type, each member of the underlying C structure can store a different element of the address. However, this means you must also define accessor routines for each element.

If you create **Address** as a row type, your Informix database server automatically provides direct access to each of the fields, as follows:

```
CREATE ROW TYPE Address
   (street_number real, street_name varchar(40),
    city varchar(100), country varchar(40),
    postal_code varchar(20));
```

This allows users to write queries like the following example:

```
SELECT * FROM employees WHERE address.city = 'Vienna';
```

In contrast, because users seldom need to examine the individual points of a polygon, you can create the Polygon type as an opaque data type. An opaque data type provides an efficient representation that you can operate on easily with C code. The query language interface remains simple.

When you design data types, ask yourself the following questions:

- Is the data type just a container for a collection of values that users can access directly? If so, use a row data type.

- Is the type naturally indivisible, or do you want to hide its representation from users? If so, use an opaque data type.
- How can you represent your data to make it easy to use in SQL and to make end-user queries simple?

Handling Large Objects

When you decide on the specific data type to represent a DataBlade module object—or its elements—keep in mind that the maximum row size for a database table is 32 KB. (*Row size* is the sum of the sizes of the columns in that row.)

Your Informix database server provides the LVARCHAR data type, which can hold up to 32,790 KB of text data. Larger objects and binary objects are called smart large objects, and your Informix database server provides facilities for high-performance access to smart large objects.

A *smart large object* is an object that is logically stored in a table column of type BLOB (binary large object, for binary data) or CLOB (character large object, for text data) but is physically stored in an sbspace.

An *sbspace* is a logical storage area that contains one or more chunks that store only BLOB and CLOB data. Sbspaces must be created before you can create any smart large objects; after sbspaces are created, they are managed by your Informix database server.

Tip: If your DataBlade module makes use of smart large objects, you can test for the existence of a particular sbspace when your DataBlade module is being registered in a database using BladeManager. For information, see Appendix C, "Testing for an Sbspace," on page C-1.

Smart large objects are "smart" because they provide the following features:

- They provide random access to their data, using an operating-system-style interface (seek, read, write, and so on).
- They are recoverable in the event of a system crash (if the sbspace was created with logging enabled), and they obey transaction isolation modes.
- They have no maximum size.
- You can create and store indexes in them.
- You can access and manipulate them using SQL, Informix ESQL/C, or the DataBlade API.

Within SQL, the only comparison operator you can use for data of types BLOB and CLOB is **Equal()** (=); however, you can perform additional operations using Informix ESQL/C or the DataBlade API from your client application.

You can also use the IBM Informix Large Object Locator DataBlade Module to handle large objects. This DataBlade module enables you to store large object data on the client computer. The IBM Informix Large Object Locator DataBlade Module is included with your Informix database server. You must register the module's routines and data types in each database in which you plan to use the module.

When you design data types, ask yourself the following questions:

- Is the object represented by 32,790 KB or less of text data? If so, use the built-in LVARCHAR data type.

- Is the object represented by more than 32,790 KB of text data, or by binary data? If so, use the smart large object facilities provided by your Informix database server.
- Does it make sense to store the large object on the client computer? If so, use the IBM Informix Large Object Locator DataBlade Module.

The following table lists large object topics and where you can find more information on them.

For information on...	See...
Using smart large objects, including examples	http://www.ibm.com/software/data/developer/informix
Overview of smart large objects	*IBM Informix Guide to SQL: Tutorial*
Creating sbspaces	*IBM Informix Dynamic Server Administrator's Guide*
Testing for the existence of a particular sbspace during DataBlade module registration	Appendix C, "Testing for an Sbspace," on page C-1
BLOB and CLOB data types	*IBM Informix Guide to SQL: Reference*
SQL smart large object functions	*IBM Informix Guide to SQL: Syntax*
Informix ESQL/C smart large object features	*IBM Informix ESQL/C Programmer's Manual*
IBM Informix DataBlade API smart large object features	*IBM Informix DataBlade API Programmer's Guide*
IBM Informix Large Object Locator DataBlade Module	*IBM Informix Database Extensions User's Guide*

Most of these publications are accessible through the InfoShelf.

Query Language Interface

The next component in DataBlade module design is the *query language interface*. Because your Informix database server is object-relational, you access it by formulating queries in SQL. DataBlade modules extend SQL by defining new types and new routines that are available to queries. Consider the syntax that users must master to use a DataBlade module.

SQL Query Structure

SQL includes Data Definition Language (DDL) statements and Data Manipulation Language (DML) statements.

DDL statements, such as CREATE, ALTER, and DROP, modify the schema of a database. DML statements, such as SELECT, INSERT, UPDATE, and DELETE, manipulate data in tables.

Most SQL queries use DML statements. When you design a DataBlade module, consider DML statements in the abstract. DML statements can be in either of the following two forms:

```
SELECT something FROM some table
   WHERE some conditions are satisfied

UPDATE some table SET something
   WHERE some conditions are satisfied
```

The italicized components serve different purposes in the DML query. The *some table* part is called "the from list" and is not important to consider when you design a DataBlade module. The *something* part is called "the target list" and identifies the columns for retrieval or update. The target list is the target on which

the query is operating. The *some conditions are satisfied* part of the query is called "a qualification" because it identifies the rows that qualify to participate in the operation.

When you develop a DataBlade module, consider where you expect a particular routine to be used. In the following two sections, the DataBlade module routines that typically appear in the target list and qualification are addressed.

The Target List

The *target list* is where simple computation occurs. Consider providing DataBlade module routines for common computations on opaque data types. You can perform the computation in the DataBlade module to eliminate the need to implement the routine in the client application. You can use DataBlade module routines in the target list to reduce the amount of data transferred from the server to the client and thereby improve performance.

Consider the following sample query from the SimpleMap DataBlade module discussed earlier in this chapter:

```
CREATE TABLE cities (name text, population integer,
   boundary Polygon);
```

Polygon is a data type supplied by the SimpleMap DataBlade module.

To retrieve a list of all cities, their populations, and population densities, you can submit the following query:

```
SELECT name, population, population / Area(boundary)
   AS density FROM cities;
```

In this example, the **Area()** routine is supplied by the SimpleMap DataBlade module. **Area()** returns a floating-point number that is the area of the supplied polygon. You can invoke the built-in division operator to compute density from population and area. This query does a simple computation in the target list, using a mixture of DataBlade module and built-in routines.

This computation can also be done on the client. However, the client must implement the **Area()** routine for polygons, and the server must ship all of the polygons to the client. This operation is more expensive than shipping the results of the division across the network because polygons can be quite large. Generally, any computation that appears in the target list can also be done by the client. Thus, place target-list routines in the DataBlade module server routines only if there is an advantage to be gained by doing so.

If there is no advantage to running the routine on the server, leave the routine out of the DataBlade module and allow the client application developers to implement it in the client. If the server routine provides any of the following advantages, include it in the DataBlade module:

- It reduces the volume of data transferred to the client.
- It simplifies application development by sharing code among clients more effectively.
- It benefits from the parallelism and scalability enabled by your Informix database server.

A simple DataBlade module that integrates well with existing data types is always better than a complicated one with many predefined routines that cannot be used in conjunction with built-in or other DataBlade module routines.

The division operator that appears in the query calls a division routine built into your Informix database server. Built-in routines and DataBlade module routines can be combined in queries, as shown in the previous example using division with **Area()**. Routines from different DataBlade modules can be mixed to provide additional services.

When you design a data model, consider using built-in types and types provided by other DataBlade modules. In the previous example, you might define a new data type, called AreaType, to represent the area of geographic objects. However, then you must implement all the math on AreaType values yourself. By using real numbers to represent areas, you can leverage existing math and computational support in the database server and allow users to mix SimpleMap DataBlade module routines with other routines.

You might define a routine that computes population density inside the SimpleMap DataBlade module. The routine takes two parameters—a polygon and an integer—and does the division itself. However, no real semantic power is derived from this design. Leave special-purpose routines out of the DataBlade module to keep the interface simple and to let developers define their own expressions or routines to compute specific values.

The Qualification

The SQL *qualification* restricts the set of rows returned to the user. The qualification filters out records that are not interesting. Only the records that pass the qualification are evaluated in the target list. Thus, a qualification is a more powerful tool than the target list.

A single expression in a qualification is called a *predicate*. A qualification can contain multiple predicates joined by the Boolean operators AND and OR.

If a DataBlade module routine is used in a qualification, it filters the records returned to the client. Your database server can filter by the contents of new data types. (This capability is not available in conventional relational databases.)

Consider whether the routines you define are more likely to be used in the target list or the qualification. Routines more commonly used in the qualification make better use of the extensibility of your database server because they support searches that cannot be done efficiently on conventional relational servers.

The following example shows a DataBlade module routine used in a qualification:

```
SELECT name, boundary FROM cities WHERE
   Overlaps(boundary, '(1,1), (5,5), (7,7), (8,9)');
```

In this example, the **Overlaps()** routine is provided by the SimpleMap DataBlade module and takes two polygon arguments: the first argument specifies the polygon you are checking; the second specifies the polygon with which the first is compared. **Overlaps()** returns `TRUE` if the two polygons overlap and `FALSE` otherwise. This query searches the **cities** table for those cities that overlap the region of interest.

The separation between routines used in the target list and those used in the qualification is not absolute. For example, the following query finds the names and populations of large cities:

```
SELECT name, population FROM cities
   WHERE Area(boundary) > 500;
```

In this example, the **Area()** routine appears in the qualification. In the section "The Target List" on page 2-6, the **Area()** routine appeared in the target list.

Some routines are better suited to the qualification than the target list. A good example of this distinction is the **Overlaps()** routine. This routine is more powerful in the qualification. While it is possible to formulate a query like the following example, it is not very common:

```
SELECT Overlaps(boundary, '(1,1), (5,5), (7,7), (8,9)')
   FROM cities;
```

This query returns a list of yes-or-no answers for each city in the table that overlaps the supplied constant polygon. It is more common to use the **Overlaps()** routine to filter rows than to compute values returned to the user. However, important and useful exceptions to this rule exist, as follows:

```
SELECT a.name, Overlaps(a.boundary, b.boundary)
   FROM cities a, cities b
   WHERE b.name = 'Los Angeles' AND
   a.name = b.name;
```

This query returns a list of all cities in the table and whether they overlap Los Angeles.

To help decide which routines to include in the DataBlade module, consider the following questions:

- What questions do users want to ask about the contents of the new data types in the DataBlade module?
- What routines allow them to ask those content-based questions? These routines are used in the qualification.

Query Processing

To develop a DataBlade module, you need a general understanding of query processing and Informix SQL. You must also understand the execution environment inside your Informix database server—the multithreading model, the collection of processes in which DataBlade module routines can execute, and concurrent access to database objects, transactions, and so on. This section describes query processing.

Predicate Evaluation

An expression in the qualification of a query is a *predicate*. The WHERE clause in the following query is a predicate:

```
SELECT name, boundary from cities
   WHERE Overlaps(boundary, '(1,1), (5,5), (7,7), (8,9)');
```

The database server evaluates the predicate for every row in the table and returns the rows that satisfy the predicate to the client. Each predicate in a qualification eliminates rows from the candidate set. After the server determines that a row does not satisfy a predicate, it moves on to the next candidate row.

Most predicate evaluation is straightforward—values of interest are extracted from a candidate row and evaluated against the predicate. However, there are some cases where predicates in the qualification behave in a unique way. These cases are described in the following sections:

- "Expensive Routines" on page 2-9, next
- "User-Defined Statistics" on page 2-9

Expensive Routines

Expensive routines are routines that either take a long time or require a great deal of disk space to run. Conventional relational database systems do not account for expensive routines; any predicate that appears in a query is assumed to be as expensive as any other. For example, comparing two floating-point numbers is not more difficult than comparing two integers. For relational databases, this is the right approach.

However, an object-relational database system must evaluate relative function costs. Some routines are very difficult to compute or require a great deal of intermediate space. For example, it can take many thousands of machine instructions to determine whether two polygons overlap.

Because an object-relational database stops evaluating predicates as soon as it determines that a row does not satisfy the criteria, the database server chooses an optimum order to evaluate the predicates in a query. If it evaluates all the expensive predicates first, the query runs slower than if it considers the inexpensive predicates first.

The strategy for choosing the best order to evaluate predicates is complex and beyond the scope of this discussion. However, the database server must evaluate the cost of invoking user-defined routines to run queries efficiently.

Most DataBlade module routines are at least as complex as a routine that compares floating-point numbers. For DataBlade module routines that are more expensive, you must describe the relative expense to the Informix server.

A good formula for estimating the expense of a routine is as follows:

```
<lines of code> + (<number of I/Os> * 100)
```

For example, if a routine has 100 lines of code and performs 5 disk I/Os or SQL queries, the cost is 100 + (5 * 100), or 600. You can enter the cost in the BladeSmith Routine wizard (see "Cost of Routine" on page 4-21).

When you estimate the cost of executing a routine, consider the following questions:

- Which DataBlade module routines take a long time to run?
- Which DataBlade module routines consume large amounts of memory or disk space?
- How expensive are the DataBlade module routines relative to one another?
- How expensive are the DataBlade module routines relative to expensive routines defined by other DataBlade modules?

User-Defined Statistics

User-defined statistics provide a way to improve performance when you compare opaque data type values. User-defined statistics compile information about the values in an opaque data type column that the optimizer can use when it creates a query plan when it needs to execute routines that compare opaque data type values.

Statistics typically consist of the following types of information about the specified column; however, you can collect more information if it is appropriate for your opaque data type:

- Minimum value
- Maximum value
- Distribution of values

When your statistics-gathering function calculates the distribution of column values, it can assign each value to a "bin." Each bin contains a range of values. For example, suppose the column values range from 1 through 10. You could have five bins: the first bin would hold values from 1 through 2, the second bin would hold values from over 2 through 4, and so on. The database server generates statistics by calling your statistics-gathering function when you run the UPDATE STATISTICS statement in medium or high mode (see the *IBM Informix Guide to SQL: Syntax*).

Important: You must understand your data and how users will query it to create meaningful statistics.

The minimum, maximum, and distribution of values can be used to compute the selectivity of a value. The optimizer can then use the selectivity of values when it determines query cost estimates. For example, suppose you want to join two tables. Normally, a join compares all values in one table to all values in the other table. However, if the optimizer knows that one of the tables has low selectivity, it can efficiently order the joins.

Selectivity is an estimate of the percentage of rows that will be returned by a filter in a query. Selectivity values range from 0.0 to 1.0, where 0.0 indicates a very selective filter that passes very few rows and 1.0 indicates a filter that passes almost all rows. The optimizer uses selectivity information to reorder expressions in the query predicate so that filters that are expensive to call given the values of their arguments are evaluated after filters that are inexpensive to call. Thus the optimizer reduces the number of comparisons and improves performance. To determine the selectivity of a routine, the database server calls the associated selectivity routine.

For example, suppose you have an opaque data type that represents a circle and you have created a distribution for the circle type based on the radius. Assume that the values of the radius range from 5 to 15. If you query for all circles with a radius of less than 4, the selectivity of the **LessThan()** function that handles the circle data type is 0 because no values qualify. Consequently, the optimizer would not execute the **LessThan()** function. Alternatively, if you query for all circles with a radius of greater than 4, the selectivity of the **GreaterThan()** function that handles the circle data type is 1.0 because all values qualify. Consequently, the optimizer would execute the **GreaterThan()** function after all other operations in the query predicate.

You can define selectivity routines for user-defined functions with the following characteristics:

- Functions that compare two opaque data types
- Functions that return a Boolean value
- Functions that act as filters (called in the WHERE clause of a SELECT statement)

For example, you can define selectivity functions for the **Equal()**, **LessThan()**, and **GreaterThan()** functions that are overloaded for an opaque data type. You can also define a selectivity function for a function like **Contains()** that compares two opaque data types.

To implement user-defined statistics, you must supply the following routines:

- Statistics support functions that collect statistics for opaque data types (see "Statistics Support" on page 4-34)
- User-defined selectivity routines that use statistics to estimate the selectivity of a routine that compares opaque data type values (see "Selectivity Functions" on page 4-22)

After you define the routines in BladeSmith, you must add code to them to provide the required functionality. See "Editing Statistics Routines in statistics.c" on page 5-27 and "Selectivity Functions" on page 5-31 for instructions.

To determine whether your opaque data type needs user-defined statistics, consider the following questions:

- Do you know enough about the data and how users will access it to write routines that compile meaningful statistics?
- Do the routines that compare your opaque data type consume large amounts of memory or disk space?

Aggregates

Most DataBlade module functions operate on single values and return one result for each time they are called. Aggregates, however, are functions that are called repeatedly, with different values, and collect their results until no more arguments are supplied.

An example of an aggregate is the built-in AVG aggregate. This aggregate computes the average value of all its arguments. For example, an SQL user could issue the following query:

```
SELECT AVG(population) FROM cities;
```

The query processing engine calls the supporting function for AVG repeatedly with population values from the **cities** table. After all the populations have been passed to AVG one at a time, it returns the average population. You can extend the aggregates that are built into the database server by overloading their operator functions for an extended data type. For more information, see the *IBM Informix DataBlade API Programmer's Guide*.

You can define new aggregates that implement user-defined functions. For example, one common spatial operation is to compute a minimum bounding rectangle that contains a collection of other rectangles. A user might write the following query using a user-defined aggregate called BOUNDING:

```
SELECT BOUNDING(boundary) FROM cities;
```

The BOUNDING aggregate takes all the polygons, one at a time, from the **cities** table and returns the smallest rectangle that contains them all. The query processing engine supplies records to the aggregate for computation; the aggregate only collects information over the arguments it is passed. For more information, see "Creating Aggregates" on page 4-9.

Like ordinary functions, aggregates may appear anywhere in the query, including in the target list and the qualification. Aggregates in the qualification are most

useful in queries that also do grouping. See "Grouping" on page 2-12 for more information on how aggregates work in grouping queries.

If you have a data type over which summary or statistical analyses are valuable, consider defining an aggregate.

When you design a DataBlade module, ask yourself the following question: Is it useful to compute a summary over values that the DataBlade module supports?

Sorting Results

SQL allows you to sort result rows when you express your queries. Sorted results are useful when you need to see records in some particular order.

The following query sorts a list of cities and their populations in descending order by population:

```
SELECT name, population FROM cities ORDER BY population desc;
```

If a DataBlade module defines a data type that can be sorted in a meaningful way, you must supply a comparison routine for the type. This routine allows the user to sort query results on that type.

In addition, you can use the results of routines that appear in the target list to sort the results of a query. For example, the following query returns a list of cities in descending order by population density:

```
SELECT name, population,
   population / Area(boundary) AS density
   FROM cities
   ORDER BY density desc;
```

The density expression, on which the query results are sorted, is a complex calculation. The expression includes a DataBlade module routine and a division operation. Because your Informix database server allows sorting by floating-point numbers, the preceding query requires no special sorting support from the DataBlade module.

To determine whether to provide sorted results, ask the following questions:

- Can my DataBlade module data types be sorted?
- Will users want to sort this data type?

Grouping

SQL allows you to write queries that group results. Grouping is a powerful facility for summarizing data, particularly in combination with aggregates such as COUNT or SUM. The following query uses grouping and aggregates:

```
SELECT COUNT(name), population FROM cities GROUP BY population;
```

This query returns the number of cities that have the same population for each distinct population value that appears in the table. The GROUP BY clause breaks the set of result rows into groups with equal populations; then the target list is evaluated for each group separately. The COUNT aggregate counts the number of city names in the group.

Consider whether any of the types you define are candidates for grouping. In the SimpleMap DataBlade module, for example, polygons are a poor candidate; users seldom want to group geographic data that contains identical polygons.

You can group results using complex expressions. For example, the following query divides cities into groups that are within 10 units of the same area and then adds the population for the group:

```
SELECT Area(boundary) / 10 AS dimensions, SUM(population)
   FROM cities GROUP BY dimensions;
```

To determine whether your DataBlade module requires support routines for grouping, ask the following questions:

- For each type in the DataBlade module, can the values sensibly be broken into groups that are equivalent?
- What is the meaning of each of these groups?
- Do users want to group values in that way?

Casts

If your DataBlade module defines types that are similar or comparable, consider defining casts between the types. You can also define casts from DataBlade module types to built-in types, and from data types in one DataBlade module to data types in another DataBlade module.

Casting values allows the query processing engine, implicitly or explicitly, to change the type of a value and use it as an argument to routines that require the destination type.

In an inheritance hierarchy, casting can provide another mechanism for type conversion. In general, subclasses can be implicitly cast to superclass types. However, downward casts (that is, from supertype to subtype) are not automatically supported because subclasses typically add instance variables not present in the supertype.

Similarly, distinct types can often be cast to their source type. For example, a distinct type called LIRA (representing the currency unit of Italy), based on the MONEY data type, might allow casting to MONEY to allow simple math operations on it. However, you probably do not want to cast MONEY to LIRA; if LIRA has only the properties of MONEY, it is not a required type.

Casts can be confusing if overused. Implicit casts hide an important fact from users—that data can be lost during type conversion. Explicit casts, which users must specify in queries, do not have this problem.

Use casts only where necessary. Be sparing in the casts you supply to users, and be sure you understand the circumstances under which you expect casts to be used.

To determine whether to provide casts, ask the following questions:

- Are any of the types in the DataBlade module comparable? Do they really need to be different types? If so, is there a need to support explicit or implicit casts between those types?
- Will users want to convert between values of one type and some other type, either an Informix built-in type or one defined by the DataBlade module?
- Which direction should the conversion go (in the example earlier in this section, from LIRA to MONEY, or from MONEY to LIRA)? In general, casts should only go one way, unless you intend them to be explicit.

Access Path Selection

During query processing, your database server takes a *nonprocedural* query and produces a *procedural* plan for satisfying it; this process is called *making an access path selection*. Queries are nonprocedural because they describe only the records of interest and what operations to perform on them. They do not prescribe an algorithm (procedure) for locating records on disk or the order in which to process them.

Your database server evaluates a collection of possible *query plans* that can execute the query correctly. The server estimates the cost of running each plan and chooses the one with the smallest cost estimate. Cost estimates are a combination of the number of expected disk I/Os, the expected number of records that must be processed, and the cost of invoking each of the routines in the query on each candidate row.

Unordered Row Processing

When you design a DataBlade module, you cannot control how queries are executed. There is no guarantee that the routines in a query are called in any particular order. DataBlade module routines are called during query processing to compute answers to queries. Do not hardcode query execution strategies. For example, an attempt to force an index scan or a sequential table scan reduces the number of choices available to the query optimizer and results in poor performance.

To ensure that your DataBlade module does not conflict with the query processing engine, ask the following questions:

- Do any routines require values to be delivered in some particular order? If so, the routines break a fundamental rule of relational database systems and must be changed.
- Is it important for routines in a query to execute in some particular sequence? Again, the routines must be changed.

Secondary Access Methods

A *secondary access method* is an index that allows queries to be evaluated more efficiently. When you create a table in SQL, you can choose to create a B-tree index on one or more columns in the table. The query processing engine can choose to use the index. For example, if there is an index on the population column of the **cities** table, the query processing engine has at least two choices for evaluating the following query:

```
SELECT name FROM cities WHERE population > 1000000;
```

The query processing engine can scan the **cities** table sequentially, examining each record in turn and comparing the population to one million, or it can use the B-tree index to quickly find only those records with populations of more than one million. When it chooses to use the B-tree index, the engine does not consider records with smaller populations and does not read them from the disk.

The B-tree index stores the key value (for example, the population) and a pointer to the record in the base table. The base table is the primary store, and the index is a secondary access method.

You can define many types of indexes. For example, most text search engines use a textual index to run searches quickly, while spatial data can be indexed in a number of ways, including grid files and R-trees.

You can allow the creation of other indexes on your data types. For example, a DataBlade module that defines a new type that can be sorted can allow users to create B-tree indexes on that type. To do so, you create an *operator class* for the type. An operator class is a collection of routines that allows the type to be used in a given access method. For example, the operator class for B-trees includes the routines **LessThan()**, **LessThanOrEqual()**, **Equal()**, **GreaterThanOrEqual()**, and **GreaterThan()**. When you define those routines on a new data type, users can create B-tree indexes on the type.

Planning for Transaction Semantics

DataBlade module code runs in SQL *transactions*. A transaction is a single, atomic, independent sequence of client/server interactions. For example, inside a transaction, a user can search a table for all the cities that overlap Los Angeles. In a separate transaction, some other user can change the boundaries of Los Angeles, as outlying areas are incorporated into the city. The two operations are independent of one another. Each user is isolated from the changes made by the other until the next transaction begins.

You must define DataBlade module code to be stateless and not based on the assumption that any particular value persists across user transactions. For example, a DataBlade module that does text matching might provide two services: one to find all documents that contain a particular set of keywords and another to highlight the matching keywords in the documents.

If a user first runs a query to find matching documents and then runs a separate query to highlight the matches, the second operation cannot rely on any cached results from the first. This is because some document contents might have changed. In addition, because your database server is a multiuser system, different users can run the same routines at the same time. There is no way to guarantee that "saved" results belong to a particular query.

To ensure that you design systems that operate correctly in this environment, ask the following questions:

- Are any of the routines based on an assumption that results from previous user actions are still valid?
- Do I try to cache results for reuse?
- What happens if two users run the same routine on the same table simultaneously?

Interoperability

The *interoperability* of a DataBlade module refers to how well that module works with your Informix database server and with other DataBlade modules.

This section discusses the following interoperability issues:

- Orthogonality
- Simple, clean interfaces

Orthogonality

In an *orthogonal* system, such as an object-relational database, the various parts work together in a natural, semantically logical way. For example, an orthogonal DataBlade module provides solutions only for the problems it is intended to solve, and it relies on the Informix server or other DataBlade modules to solve problems

outside of its domain. Similarly, an orthogonal DataBlade module allows other DataBlade modules to use its facilities in a natural, semantically logical way.

The SimpleMap DataBlade module, for example, does not implement full-text search. It is more effective if developers who are experts in text search facilities create DataBlade modules that satisfy this requirement. The SimpleMap DataBlade module can then supply just geospatial functionality; it does not need to define routines over types that it does not create.

A simple guideline for ensuring orthogonality in DataBlade module development is, "It does a small number of things well."

Simple, Clean Interfaces

Provide the users of your DataBlade module with a simple, clean interface by following these guidelines where possible:

- Give your routines meaningful, "self-documenting" names.
- Take advantage of polymorphism.
- Limit the number of arguments each routine takes.
- Avoid creating modal routines.

This section discusses each of these guidelines.

Naming Routines

Whenever possible, use generally accepted names for routines using your new data types. For example, the **Overlaps()** routine in the SimpleMap DataBlade module does precisely what its name indicates. Users know what to expect when they call it.

Because your database server supports polymorphism (see "Taking Advantage of Polymorphism" on page 2-17), it is possible that another routine of the same name already exists in the system. If you are concerned that your routine provides a different service or has the same signature as another, similarly named routine from another DataBlade module (that is, none of the arguments of your routine are of a data type defined in your DataBlade module), consider renaming the routine or qualifying its name with a three-character DataBlade module prefix such as "USR". Doing so helps avoid conflicts in the system and confusion among your users.

Assume, for example, that you are creating the OtherMap DataBlade module with a routine named **Overlaps()** that provides a different service than the **Overlaps()** routine supplied by the SimpleMap DataBlade module. In addition, your **Overlaps()** routine takes polygon data types not defined in the OtherMap DataBlade module. If the three-character prefix of your DataBlade module is *OTH*, then you might define your routine as follows:

```
OthOverlaps(Polygon, Polygon)
```

However, if your **Overlaps()** routine takes arguments of data types defined in the OtherMap DataBlade module, you might define **Overlaps()** as follows:

```
Overlaps(OthPolygon, OthPolygon)
```

Taking Advantage of Polymorphism

Your database server supports polymorphism; thus, you can have multiple routines with the same name that take different argument types. For example, a C programmer might be tempted to create distinct names for the following routines that return the larger of their arguments:

```
bigger_int(integer, integer)

bigger_real(real, real)
```

However, in SQL it is better to define the routines with the same name, as follows:

```
bigger(integer, integer)

bigger(real, real)
```

Limiting the Number of Arguments

To help your users remember how to use your DataBlade module routines, limit the number of arguments they take. Re-evaluate any routines that take more than three arguments; such routines can become unwieldy or can inadvertently become modal (defined in the next section).

Avoiding Modal Routines

When you create DataBlade module routines, avoid including arguments that make them *modal*; that is, the mode of the routine changes, depending on the third argument. For example, there are a number of different ways to call a routine that computes containment of spatial values. The SimpleMap DataBlade module might implement the following routine:

```
Containment(polygon, polygon, integer);
```

This routine determines whether the first polygon contains the second polygon, or whether the second contains the first. The caller supplies an integer argument (for example, `1` or `0`) to identify which value to compute; but the purpose of this argument is not immediately evident to a new user of the DataBlade module.

Consider a second design for calculating containment, as follows:

```
Contains(polygon, polygon)
ContainedBy(polygon, polygon)
```

This design is an improvement: not only are the routines nonmodal, but the routine names also clearly explain what computation is performed.

Chapter 3. Programming Guidelines

In This Chapter

Use this chapter to help you when you write the design specification for your DataBlade module.

Programming Language Options

This section explains the programming language options you have when you use BladeSmith to generate source code for your DataBlade module. BladeSmith supports the following external languages:

- C
- C++/ActiveX
- Java

The following table lists the objects that you can implement in an external language and the languages you can use for each.

Object	C	C++/ActiveX	Java
Cast support functions	Yes	No	Yes
Aggregates	Yes	No	Yes
Other user-defined routines	Yes	No	Yes (with restrictions)
Opaque data types routines (server implementation)	Yes	Yes (with restrictions)	No
Value object methods (client implementation)	No	Yes	No

The following subsections discuss programming language options in detail:

- "Options for Opaque Data Types" on page 3-2, next
- "Options for Routines" on page 3-5
- "Multilanguage DataBlade Module Issues" on page 3-5

© Copyright IBM Corp. 1996, 2008

Options for Opaque Data Types

Opaque data types are ultimately defined as C structures; when you create an opaque data type with BladeSmith, the built-in data types you can choose as members are C structures provided by the DataBlade API. However, you can implement opaque data types as value objects in other external languages. A *value object* is a self-contained binary object that provides standard interfaces to its users. Value objects can be used in client applications.

You create client value objects in BladeSmith by specifying an optional client implementation of your opaque data type, in addition to the mandatory server implementation. For a complete list of options you have when you create an opaque data type with BladeSmith, see "Opaque Data Type" on page 4-24.

When you decide what functionality and which language or languages to use for your opaque data types, you should consider the following options:

- Whether to create client value objects in addition to opaque data types for the database server. See "ActiveX Value Objects," next, for more information.
- Which language to use for your opaque data types. See "Limitations of Opaque Types for Each Language" on page 3-4 for more information.
- Whether to use different languages for the server and client implementations. See "Mixing Languages in Server and Client Implementations" on page 3-3 for more information.
- Whether to embed opaque data types as members of other opaque data types. See "Embedding Opaque Data Types within Opaque Data Types" on page 3-5 for more information.

ActiveX Value Objects

You can create ActiveX value objects with DBDK. An ActiveX value object is an object that is compliant with Microsoft Common Object Model (COM) and contains a client-side copy of database data.

The following table summarizes the relationship between ActiveX elements and Informix opaque data type elements.

ActiveX Element	Opaque Data Type Element
ActiveX control (or *ActiveX object*)	Opaque data type
Custom methods (dual interface)	Opaque type support routines
Properties	Accessor methods for the members of the data structure that defines the opaque type
States	N.A.
Events	N.A.
Interfaces (ActiveX value objects provide the IRawObjectAccess and ITDkValue interfaces)	N.A.

Important: Be aware that changing an object on the client will not update the object on the database server. To update the value on the database server, you must do so explicitly with an SQL UPDATE statement.

Mixing Languages in Server and Client Implementations

You can choose to combine languages for the server implementation and the optional client implementations of your opaque data type: C or C++ for the server implementation and ActiveX for the client implementation.

The following table describes some of the advantages and disadvantages you should consider when you choose server and client implementation languages.

Option	Advantage	Disadvantage
Using the same server and client implementation language	You have less code to edit. With C++, much of the server and client code overlaps (see Figure 3-1).	C++ has restrictions on the functionality of the opaque data type. You can use C++ to implement only opaque data types; you must use C or Java to implement other DataBlade module objects. You cannot port a C++ server project from Windows to UNIX platforms. See "Limitations of Opaque Types for Each Language" on page 3-4 for more information.
Using different languages for the server and client implementations	If you use C as the server language, you can implement functionality that is not available for Java or C++.	You have more code to edit because you have separate server and client source code.

Figure 3-1 illustrates the advantage of choosing C++ for both the client and server implementations: much of the same generated source code can be used to compile each project.

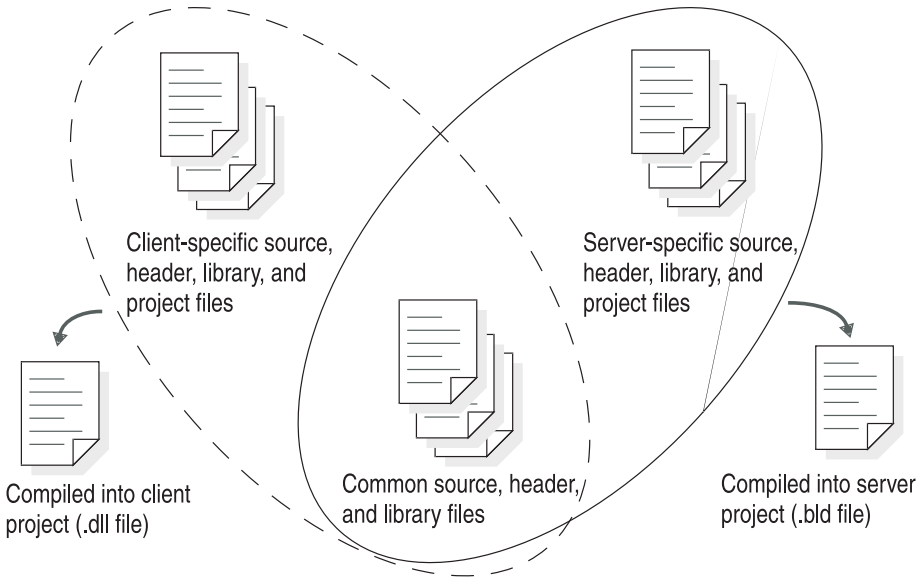


Figure 3-1. Choosing C++ for Both Client and Server Projects

See "Implementing ActiveX Value Objects" on page 6-2 for more information about the generated files.

Important: It is recommended that developers create DataBlade modules in C++ only for client projects and for server projects that use Dynamic Server on Windows only. For the latest recommendations on C++ programming options, check the IBM Informix Developer Zone at http://www.ibm.com/software/data/developer/informix.

Limitations of Opaque Types for Each Language

This section discusses the limitations you have for each language when you implement opaque data types.

Opaque Type Limitations for C: You cannot use BladeSmith to generate a client object or its accessor methods written in C.

Opaque Type Limitations for C++/ActiveX: The following limitations apply to using C++/ActiveX value object code:

- When you define the opaque type that you intend to encapsulate as an ActiveX value object, BladeSmith enforces these rules:
  - You must define the internal structure of the opaque type.
  - The opaque type must be of a fixed size.
  - The opaque type cannot contain members that are smart large objects or variable in size.
  - The opaque type can contain members that are opaque types only if they are implemented in C++.
- You cannot import opaque types from other DataBlade modules.
- The following opaque type routine categories are not supported for either the client or server implementations of an ActiveX value object:
  - **Contains large objects routines.** To support smart large objects.
  - **Type insert and delete notification routines**. To perform tasks before storing or deleting an opaque data type on disk.
  - **Statistics support functions.** To provide a way to improve performance when you compare opaque data type values.
- The following opaque type routine categories have no meaning for database clients. Thus, although you can implement these routines for the server implementation, they are not made available to the client application developer as ActiveX custom methods:
  - **Binary send and receive.** To transfer the binary representation of the opaque data type to and from the client.
  - **Text file import and export.** To transfer the text representation of the opaque data type to and from a flat file.
  - **Binary file import and export.** To transfer the binary representation of the opaque data type to and from a flat file.
  - **Hash.** To replace the built-in hashing function to cache return values.
- The following ActiveX custom methods cannot be used in a server project:
  - **IsNull**
  - **SetNullFlag**

For more information on opaque data type properties, see "Opaque Data Type" on page 4-24.

Opaque Type Limitations for Java: You cannot generate Java code for opaque data types with Version 4.0 of DBDK.

Embedding Opaque Data Types within Opaque Data Types

You can embed an existing opaque data type as a member of another opaque data type with BladeSmith only if both opaque data types have the same server implementation language. You cannot mix programming languages in opaque data types.

Options for Routines

If you choose to program your user-defined routines in C, you can choose any of the routine options available in BladeSmith. For a description of these options, see "Creating Routines" on page 4-15.

When you implement a routine in Java, you cannot specify the following options:

- That it has a selectivity function or is a selectivity function
- That it takes row data types or collection data types as arguments
- That it is internal
- Its stack size
- Its cost
- That it has a commutator function or is a commutator function

Tip: You can also choose to implement routines in IBM Informix Stored Procedure Language (SPL). See "Creating Routines" on page 4-15 for more information.

Overloading Routines in Different Languages

You can overload routines to handle different data types in either C or Java. You cannot, however, overload a routine in a different language with the same data type. For example, you can create the following two functions:

- **MyFunction(lvarchar)** written in C
- **MyFunction(int)** written in Java

However, you cannot create the following two functions:

- **MyFunction(lvarchar)** written in C
- **MyFunction(lvarchar)** written in Java

Handling Opaque Data Types Implemented in a Different Language

If you create a routine in C, it can handle an opaque data type implemented in C++ without additional code.

If you create a routine in Java that handles an opaque data type implemented in C or C++, BladeSmith generates a default implementation of the SQLData interface for the opaque data type. See "SQLData Interface Method Support Code" on page 8-3 for more information.

Multilanguage DataBlade Module Issues

If you use more than one external programming language in your DataBlade module, you might have more than one resulting shared library after you compile your source code. Figure 3-2 illustrates the shared libraries you can produce from source code generated by BladeSmith.

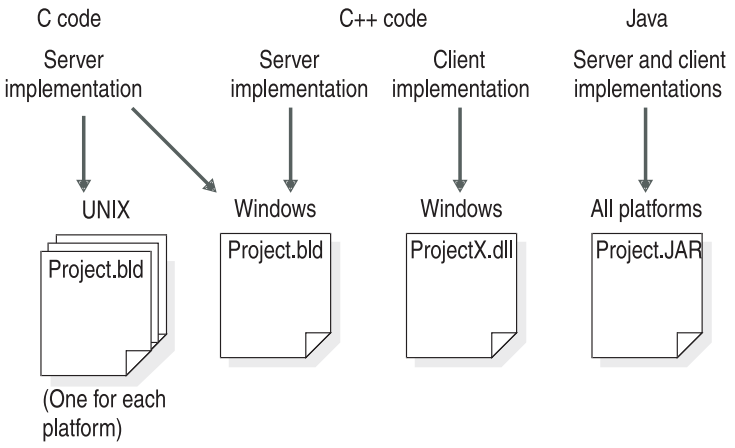


Figure 3-2. Shared Libraries for Multilanguage DataBlade Modules

For C and C++ server implementation code, BladeSmith generates a combined Visual C++ workspace file and UNIX makefile.

For Java server and client code, BladeSmith generates a single makefile that is appropriate for all platforms.

C Programming Guidelines

To take advantage of Informix database server architecture, you must use the DataBlade API and follow the guidelines in the *IBM Informix DataBlade API Programmer's Guide* when you write user-defined routines in C.

The Informix database server uses *virtual processors* (VPs) to service multiple client-application SQL requests. The database server breaks the SQL request into distinct tasks, based on the resource that the task requires. Different VP types, called *virtual processor classes* (VP classes), service different kinds of tasks.

The CPU virtual processor (CPU VP) acts as the central processor for client-application SQL requests. When a client application establishes a connection, the CPU VP creates the session thread for that client application. A CPU VP runs multiple session threads to service multiple SQL client applications. Because a session thread is the primary thread for the processing of SQL requests, any C routines in an SQL request normally execute in the CPU VP. However, your routine must be well-behaved by following certain guidelines to avoid loss of performance and data.

For example, a well-behaved user-defined routine that runs in the CPU VP must fulfill the following requirements:

- Preserve concurrency by following these rules:
  - Yield the CPU VP regularly by using the **mi_yield()** DataBlade API function.
  - Do not use blocking I/O calls.
- Be thread safe by following these rules:
  - Do not use heap-memory allocation; instead use the DataBlade API memory-management functions.
  - Do not modify global or static data; instead use the MI_FPARAM structure to preserve state information.
  - Do not modify the global state of the CPU VP.

- Do not use unsafe operating system calls that might impair concurrency or allocate local resources.

Some of these restrictions are relaxed if you assign your routine to a user-defined virtual processor. A *user-defined virtual processor* is a VP that you create. It runs only those routines that you assign to it.

The *IBM Informix DataBlade API Programmer's Guide* describes in detail each of these guidelines and their possible workarounds using user-defined virtual processors.

C++ Programming Guidelines

Follow these rules and guidelines when you edit the source code for your C++ client and server projects:

- Use the following sets of methods in your code to ensure that your code is portable between the client and server projects:
  - The object methods that are made available as ActiveX custom methods; see "Adding Project-Specific Logic to the Source Code" on page 6-3
  - The internal object methods that are not made available as ActiveX custom methods; see "Internal Object Methods" on page 6-7
  - The generated C++ support library; see "C++ Support Library" on page 6-7

  For additional functionality, use the DataBlade API. For information, see the *IBM Informix DataBlade API Programmer's Guide*.
- In the server project, use C++ *only* to implement the opaque type support routines that you intend to encapsulate as ActiveX value objects. Do not use C++ to implement any other DataBlade module objects.
- Do not use the *IBM Informix Object Interface for C++ Programmer's Guide* in server code.
- Do not change the function headers or parameter lists of any of the support routines for the opaque type.
- Do not use virtual methods or virtual base classes (either direct or inherited).
- In the server project, check for routine arguments with null values. The server will not call a routine that has an argument with a null value.
- In the client project, if your project does not need to handle null values, you can remove all calls to **IsNull()**, **SetNull()**, and **SetNullFlag()** in the generated code. Then make sure all constructor functions call **SetNotNull()**.

For a list of restrictions on the C++ code you can generate with BladeSmith, see "Opaque Type Limitations for C++/ActiveX" on page 3-4.

Important: It is recommended that developers create DataBlade modules in C++ only for client projects and for server projects that use Dynamic Server on Windows only. For the latest recommendations on C++ programming options, check the IBM Informix Developer Zone at http://www.ibm.com/software/data/developer/informix.

Java Programming Guidelines

You can use the following Java packages, interfaces, classes, and methods in a Java method:

- SQLJ packages

You can use all the basic and optional Java packages that are in JDK. That is, Java methods can use java.util.*, java.io.*, java.net.*, java.rmi.*, and so on. However, Java methods cannot use java.awt.*, java.applet.* and other GUI packages. For more information on these packages, see the SQLJ Part I draft.

- Informix SQLJ extensions

  Certain Informix extensions to SQLJ are available to applications that need to exploit the capabilities of the database server. The Informix extensions reside in the **com.informix.udr** package.

- Java Database Connectivity (JDBC) 1.0 API

  Java methods can use the JDBC 1.0 API to access the database.

- Informix JDBC extensions

  Java methods can also use Informix extensions to JDBC 1.0 to access some JDBC 2.0 functionality.

When you edit your Java source code, follow the guidelines and restrictions listed in the *IBM Informix JDBC Driver Programmer's Guide* and the *J/Foundation Developer's Guide*.

Version 4.0 of BladeSmith does not generate Java code for opaque data types. For a list of additional restrictions on the Java code you can generate with BladeSmith, see "Options for Routines" on page 3-5.

Important: You must use the IBM Informix J/Foundation upgrade to Informix Dynamic Server to enable services that use Java. For more information about J/Foundation, see the *J/Foundation Developer's Guide*.

DataBlade API Programming Tips

While you program your DataBlade modules using the DataBlade API, observe these guidelines:

- Never assume that the content of an **mi_lvarchar** data type is null-terminated.

  The Informix database server never passes a null-terminated external representation of an **mi_lvarchar** data type; however, the DataBlade API provides functions to convert **mi_lvarchar** values to and from null-terminated strings. To allocate and free memory for **mi_lvarchar** data types, use the **mi_var** accessor functions. For more information, see the documentation on the **mi_lvarchar_to_string()** function in the *IBM Informix DataBlade API Programmer's Guide*.

- Pass and return values greater than 4 bytes by reference.

  Opaque data types are wrapped in an **mi_lvarchar** data type and passed by reference.

  Write your user-defined routine code to pass arguments using a pointer. All built-in data types are passed by reference except fixed-length, noncharacter data types of fewer than 4 bytes. The **mi_real** data type (the SQL data type SMALLFLOAT) is always passed by reference. Pass opaque data types by value by creating them with the **passedbyvalue** modifier.

- Do not modify a user-defined routine argument unless it is an OUT parameter.

  Arguments to C routines cannot be modified unless you specify that the argument is an OUT parameter for a statement local variable. See *IBM Informix User-Defined Routines and Data Types Developer's Guide* for more information.

- To test if an argument for a user-defined routine is null, use the **mi_fp_argisnull()** function.

If you create a user-defined routine with the **with (handlesnulls)** modifier, your routine must check the input parameters to determine if they are null. To check whether arguments are null, pass the MI_FPARAM structure as the last argument in the C routine; then check the arguments by calling the **mi_fp_argisnull()** function.

- To set a return value to NULL, use the **mi_fp_setreturnisnull()** function.

  If you intend to return a null value from a function, you must call **mi_fp_setreturnisnull()** with MI_TRUE before the return statement. If you do not, you might receive an incorrect result or memory errors.

For code examples illustrating these and other DataBlade API coding tips, see the example DataBlade modules provided with the Informix DataBlade Developers Kit in the **%INFORMIXDIR%\dbdk\examples** directory or the IBM Informix Developer Zone at http://www.ibm.com/software/data/developer/informix.

For details on DataBlade API data types and routines, see the *IBM Informix DataBlade API Programmer's Guide*.

Chapter 4. Creating DataBlade Objects Using BladeSmith

© Copyright IBM Corp. 1996, 2008

In This Chapter

You use BladeSmith to create DataBlade modules. BladeSmith provides a visual presentation of the objects in a DataBlade module and allows you to add objects and modify properties of objects. After DataBlade module objects are defined in a BladeSmith project, use BladeSmith to generate source code files, SQL scripts, functional tests, and installation packaging files.

BladeSmith online help contains additional topics and reference information for BladeSmith. Use the online help for detailed descriptions of the BladeSmith user interface.

See Appendix A, "Source Files Generated for DataBlade Modules," on page A-1, for a complete list of generated files.

Appendix B, "Completing BladeSmith-Generated Code," on page B-1, provides reference tables that list the types of objects BladeSmith generates and indicate whether BladeSmith generates complete code or template code you must complete.

Prerequisite Tasks

Before you begin using BladeSmith, design your DataBlade module.

Write a functional specification to provide an overview of the features of your DataBlade module and a design specification to describe in detail how you plan to implement those features. Use your design specification as a reference when you supply input for BladeSmith.

See "Designing Your DataBlade Module" on page 1-3 for more information.

Task Overview

After you design your DataBlade module, complete these general tasks to implement your design with BladeSmith:

1. Create a DataBlade module project.
2. Import interfaces from other DataBlade module projects on which you want your DataBlade module to depend.
3. Define new DataBlade module objects, in this order:
   a. Data types
   b. Routines, aggregates, and casts

c. Custom SQL and client files

4. Add functional test data for each opaque data type support routine, user-defined routine, and cast support function.
5. Generate DataBlade module files.

BladeSmith uses code templates to generate much of the code for your DataBlade module objects. However, you must add code to make your routines operate the way you intend. After you edit the source code files, compile them; then test and debug them.

You can modify your project file, generate files, and recompile the source code as often as necessary until your development is complete. BladeSmith merges your previous edits into the newly generated source code files. When your DataBlade module is complete, use BladePack to create installation packages for each platform you support.

Windows

The BladeSmith project window is divided into two panes. One pane, called the *project view*, contains a tree representing the hierarchy of the objects in the project, with folders for files, imported objects, and user-defined objects. The other pane, called the *item view*, contains information about the object selected in the project view.

Figure 4-1 shows a BladeSmith project window.

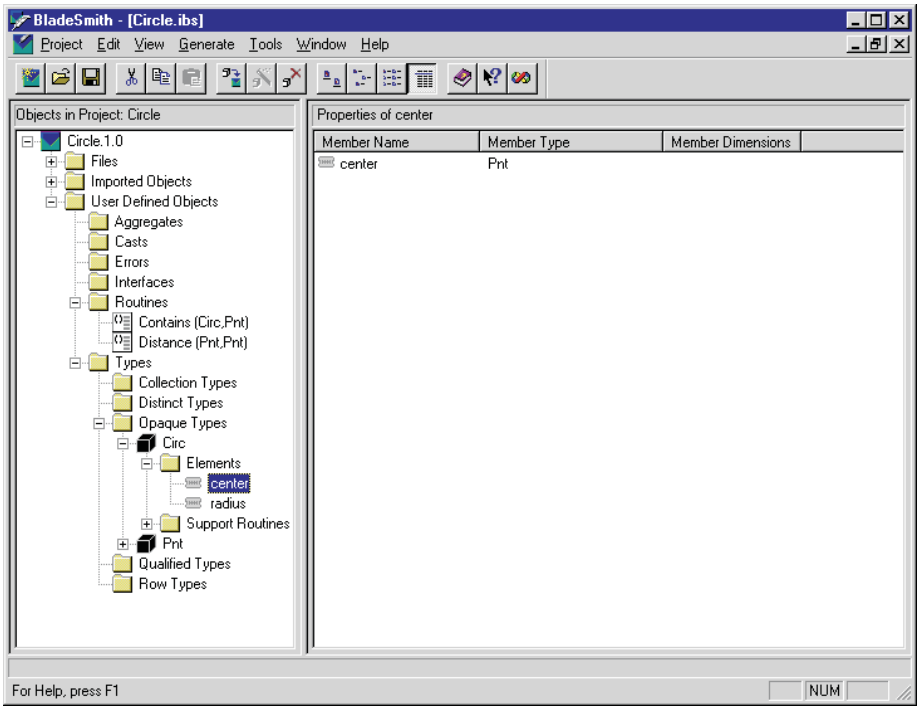


Figure 4-1. BladeSmith Project Window

In the project view, expand or collapse a folder by clicking the expander button next to the node. When you select a *node* in the project view, the item view displays the node's contents. When you select an *object* in the project view, the item view displays information about the object.

The information displayed in the item view depends on the type of object selected. The **View** menu allows you to specify what BladeSmith displays in the item view. You can choose **Small Icons**, **Large Icons**, **List**, or **Details** from the **View** menu. When you choose the **Details** view, BladeSmith lists the properties of the object.

Most objects have property sheets, which allow you to view or edit their properties. To view the property sheet for an object, right-click the object or select the object in either view and then choose **Edit > Properties**. You can also choose **Edit > Update** to start the wizard for the selected object.

Creating a New Project

The first step you complete in BladeSmith to create a DataBlade module is to create a *project* for it. BladeSmith saves DataBlade module object definitions in a project file with an **.ibs** extension. BladeSmith generates source code, SQL scripts, functional tests, and installation package files in directories that are relative to the project file. By default, BladeSmith creates subdirectories **src**, **scripts**, **functest**, and **install** in the directory where you save the project file.

Important: Create a different project directory for each DataBlade module so that BladeSmith does not overwrite any other files.

To create a new project in BladeSmith, choose **Project > New** and complete the information requested by the New Project wizard; then save your project file in the directory you created. BladeSmith creates the necessary subdirectories when you generate files.

The following table lists the properties you must specify values for when you create a project.

Property	Default Value	Description
DataBlade module name	**NewProject**	The name of the DataBlade module project. See "DataBlade Module Project Name" on page 4-5 for more information.
New object prefix	None	A three-character prefix used in naming new objects. See "New Object Prefix" on page 4-6 for more information.
Server compatibility	9.4	The version of the database server with which you want your DataBlade module to be compatible. See "Server Compatibility" on page 4-6 for more information.
Description locale	The locale of your Windows installation	The language code set for the project. See "Description Locale" on page 4-7 for more information.
Project version numbers or letters	Major: **1** Minor: **0**	Optional. One to four sets of numbers or letters separated by periods to designate version information. See "Project Version Numbers" on page 4-7 for more information.
Project description	None	Optional. A description of the DataBlade module. This information appears to the user, if requested, in BladeManager.
Vendor information	None	Optional. Information about the company developing the DataBlade module, including company name, copyright, and contact. This information appears to the user, if requested, in BladeManager. See "Vendor Information" on page 4-7 for more information.

DataBlade Module Project Name

The project name is combined with the version numbers as a unique identifier to use to register the DataBlade module in the database server and to create the installation directory. The project name must follow standard directory naming conventions.

For Dynamic Server Version 9.2 and later, the maximum length of the project name is 32 characters.

For Dynamic Server Version 9.14, the maximum length of the project name is 18 characters.

If you change the project name or version numbers, you must regenerate files in BladeSmith.

For more information on obtaining the project name, see "Developing Your DataBlade Module" on page 1-5.

New Object Prefix

Use a three-character object prefix in the name of the new objects you create in BladeSmith to ensure that your objects have unique names in the database when the DataBlade module is registered.

Use your DataBlade module new object prefix to begin the names of the following objects:

- New data types
- User-defined routines
- Aggregates
- Access methods
- Executable utilities and tools provided with the DataBlade module
- Tables, views, and other custom SQL objects included with your DataBlade module
- User-defined virtual processors
- Trace classes
- Named memory

You do not have to use the DataBlade module new object prefix for the following objects:

- Names of routines that operate on data types unique to your DataBlade module, including routines that you overload to take a new data type
- Names of routines that operate on data types provided by other DataBlade modules that you develop and maintain
- Names of routines in support libraries linked to the shared object file

Server Compatibility

When you create a new project in BladeSmith, you must specify the version of the database server with which you want your DataBlade module to be compatible. Different database server versions have different features; if you choose a feature that is not available for the database server version you specified, the feature is either disabled or BladeSmith displays a warning.

The following features are compatible only with Version 9.2 and later of Dynamic Server:

- User-defined statistics
- Long identifiers for database objects

If you specify a database server version of 9.14 and attempt to add user-defined statistics support for an opaque data type, you receive a warning stating that statistics support is not available with Version 9.14.

You can use DBDK to generate Java code for aggregates, cast support functions, and user-defined routines by specifying compatibility with Version 9.2 and later of the database server. However, you must use the IBM Informix Dynamic Server with J/Foundation upgrade to Informix Dynamic Server to enable services that use Java. For more information about IBM Informix Dynamic Server with J/Foundation, see the publication *J/Foundation Developer's Guide*.

All other features are compatible with Version 9.14 and later versions of Dynamic Server.

Description Locale

A GLS description locale is a set of files that contain information specific to a particular language and culture. A GLS locale provides the following information:

- The name of the code set that the application data uses
- The collation order to use for character data
- The format in which different types of data are displayed

The default locale is the locale that your Windows installation uses. For example, Windows installations running U.S. English use the **en_us.1252** locale. To change the locale, type a new locale specification in the **Description Locale** field. See the *IBM Informix GLS User's Guide* for more information on locales and how to access a list of available locales.

When you generate SQL scripts, BladeSmith uses the locale information to generate a locale-specific prepare script and locale-specific error scripts.

Project Version Numbers

The optional version information can be one to four sets of numbers or letters separated by periods: for example, `1.2.3.4`. The numbers correspond to the categories major, minor, revision, and release. The major and minor numbers can be up to eight characters long; the revision and release numbers can be up to six characters long. Use a consistent versioning format for all your projects.

The version numbers are combined with the project name as a unique identifier to register the DataBlade module in the database server and to create the installation directory.

If you change the version numbers or project name, you must regenerate SQL files and installation packaging files in BladeSmith.

Vendor Information

The vendor ID should be unique. All DataBlade modules with the same vendor ID display the same information when the user requests it from BladeManager. If you want to display different contact information for different DataBlade modules, you must use different vendor IDs.

Importing Interfaces from Another DataBlade Module

A DataBlade module can access data types and routines provided by another DataBlade module if you import an interface from that DataBlade module.

When you use BladeManager to register or unregister DataBlade modules in databases, BladeManager checks dependencies between modules to ensure that all required interfaces are available.

Important: The interface you import must not contain features that are not available for the database server version associated with your project.

To import an interface, you must have the project file (**project.ibs**, where **project** is the name of the DataBlade module) of the DataBlade module from which you wish to import an interface.

To import an interface:

1. Open the project files for both DataBlade modules in BladeSmith: the one to import into and the one to import from.
2. Click the object to import in the source project view.
3. Choose **Edit > Copy**.
4. Click the destination project window.
5. Choose **Edit > Import > From Clipboard**.

The object is added in the proper subfolder in the **Imported Objects** folder.

Creating DataBlade Module Objects

The following sections describe the objects that you can define in a BladeSmith project:

- "Creating Aggregates" on page 4-9
- "Creating Casts" on page 4-12
- "Defining Errors" on page 4-13
- "Defining Interfaces" on page 4-15
- "Creating Routines" on page 4-15
- "Creating Data Types" on page 4-22

BladeSmith uses wizards to create and edit objects. To start a wizard to create or add an object to your project, choose **Edit > Insert > ObjectName**. The last page of the wizard displays the SQL that BladeSmith generates for your object, if there is any.

If you are generating an object that can be secured, the last page of most wizards also allows you to specify privileges. You have these privilege options:

- Grant usage privileges

  All users can access the object, but only the owner can delete it. The owner of an object is the user ID of the user that created the object in the database.
- Grant none

  Only the owner can access or delete the object. Use this option only when there is a specific need to protect a type or routine.

See the *IBM Informix Guide to SQL: Tutorial* for more information on privileges.

Database Object Name Lengths

The limit of the lengths of SQL database objects names varies with different database server versions.

Dynamic Server Version 9.2 and later allows you to use long identifiers for database object names. Most object names can have 128 characters. The following table lists the objects whose names must be fewer than 128 characters.

Object	Maximum Characters
Project	32
Error code	5
Interface	64
Opaque data type (C)	110
Opaque data type (ActiveX)	80

Client file	64

Tip: The object name fields in BladeSmith are not 128 characters wide; therefore, you might not be able to distinguish between objects on a list if they have similar names. You can display the full name of an object with a tooltip. Select the object and place the cursor over it to display the tooltip.

Dynamic Server Version 9.14 limits database object names to 18 characters, except for error codes, which are always 5 characters, and opaque data type names, which are limited to 14 characters. The names of opaque data type support routines contain the name of the opaque data type plus a four-character routine identifier.

Warning: BladeSmith does not prevent you from specifying long identifiers if your database server version is 9.14; however, source code that contains long identifiers does not compile if the database server version is earlier than Version 9.2.

Creating Aggregates

An aggregate is a function that returns information about a set of query results. For example, the SUM aggregate adds all the query results together and returns the result. An aggregate is invoked in SQL as a single function but is implemented as one or more support functions.

You can use BladeSmith to create new, user-defined aggregates that implement user-defined routines.

Important: You cannot use the Aggregate wizard to overload built-in aggregates for extended data types; you must use the Routine wizard to overload each of the operators required by the built-in aggregate. For more information on the Routine wizard, see "Creating Routines" on page 4-15. For more information on built-in aggregates, see the *IBM Informix DataBlade API Programmer's Guide*.

You can define two aggregates that have the same name but operate on different data types. An aggregate acts as a template: the aggregate support functions must have the same names for both aggregates. If you overload an aggregate, you cannot add, remove, or change the names of its support functions. Use your new object prefix to begin the name of your aggregate to avoid accidentally overloading an aggregate in another DataBlade module.

The following table describes the properties you specify when you create an aggregate.

Property	Default Value	Description
Aggregate name	**Aggregate**	The name of the aggregate function. If you are overloading an aggregate, the name can be the name of an existing aggregate; otherwise, the name must be unique. New aggregate names must begin with the new object prefix. See "Aggregate Name" on page 4-11 for more information.

Property	Default Value	Description
Language	C	Which language to use for the aggregate functions: C or Java. You must set server compatibility to 9.2 or later to generate code for Java projects. You need the IBM Informix Dynamic Server with J/Foundation upgrade to Informix Dynamic Server to enable Java services.
Iteration type	None	The data type on which the aggregate function operates. See "Iteration Type" on page 4-11 for restrictions.
Initialization parameter	No	Optional. Used only for aggregates whose behavior can be changed dynamically. See "Initialization Parameter" on page 4-11 for more information.
Return type	None	The data type of the result of the aggregate function.
State type	None	The data type of the intermediate aggregation state. The state type is often POINTER. See "State Type" on page 4-11 for more information.
Initialization function	**AggregateInit**	The function called before the aggregation begins. Not required if the state and iteration data types are the same and there is no initialization parameter. See "Initialization Function" on page 4-11 for more information.
Iteration function	**AggregateIter**	Called once for every value that is aggregated. By default, this function accepts null values. See "Iteration Function" on page 4-11 for more information.
Combine function	**AggregateComb**	Optional. Merges results from parallel iterations. See "Combine Function" on page 4-11 for more information.
Final function	**AggregateFinl**	Performs computations on the combined state, cleans up memory, and returns the final value. See "Final Function" on page 4-12 for more information.

For information on how aggregates behave, see the *IBM Informix DataBlade API Programmer's Guide*.

The following sections describe properties of aggregates.

Aggregate Name

Use the new object prefix to begin the name of your new aggregate. The aggregate name cannot be the same as another user-defined routine or aggregate unless you are overloading an existing user-defined aggregate. For more information on the new object prefix, see "New Object Prefix" on page 4-6.

Iteration Type

You cannot use the following data types as aggregate iteration types:

- BLOB
- CLOB
- Collection data types: SET, MULTILIST, LIST
- Unnamed row data types

Initialization Parameter

The initialization parameter is an argument in the initialization function to customize the aggregation computation. For example, if you defined an aggregate to return the top *n* values of a query, your initialization parameter can be 3 to select the top three.

State Type

The state type holds the partial result information during the aggregation computation. The database server never accesses the state type, so it can be any data type or structure appropriate for the partial results. For example, if you have an aggregate that returns the three largest values from a query result set, your state type can be an array of three integers.

If you are overloading an existing aggregate, the state type must be different for each aggregate.

Select the POINTER data type from the data type list to indicate that your data type is not known to the database server.

Initialization Function

The initialization function initializes the data structures required by the rest of the aggregation computation. For example, it can set up smart large objects or temporary files for storing intermediate results.

The initialization function can take an optional initialization parameter to customize the aggregate computation.

The initialization function is not required for simple binary operators that have a state type that is the same as the iteration type.

Iteration Function

The iteration function merges a single value of the iteration type with the partial result of the state type and returns the updated partial result.

You can specify whether the iteration function handles null values. If it does not, any null values returned by the query are ignored. If it does handle null values, the iteration function includes them in its computations.

Combine Function

The database server can break up the aggregation computation into several pieces and compute them in parallel. Each piece is computed sequentially; then the

results from all pieces are combined into a single result using the combine function. The parallel computation must give the same result as the sequential computation.

The combine function merges partial results of the state type and returns the updated partial result. It can also perform cleanup work by releasing resources acquired by the initialization function.

The combine function can be the same as the iterator function if the aggregate is derived from a simple binary operator whose result type is the same as the state type.

Final Function

The final function converts a partial result of the state type into the result type. It can also release resources acquired by the initialization function to clean up the result type.

If you do not include a final function, the database server returns the final state type. The final function is not required for aggregates derived from simple binary operators whose result type is the same as the state type.

Creating Casts

A cast is a conversion from one data type to another. The cast accepts the source data type as its argument and returns the target data type.

The following table describes the properties you specify when you create a cast.

Property	Default Value	Description
Cast from type	None	The source data type. See Source and Target Data Types for more information.
Cast to type	None	The target data type. The source and target data types cannot both be built-in or qualified types. See Source and Target Data Types for more information.
Implicit cast	Yes	The kind of cast. Implicit casts are automatically called by the database server. Explicit casts are called by the users. See "Implicit and Explicit Casts" on page 4-13 for more information.
Support routine	Yes **typeCast**	If the source and target data types do not have the same binary representation, you must write a routine to support the cast. See "Cast Support Functions" on page 4-13 for more information.
Language (if you choose to create a support function)	C	Which language to use for the cast support function: C or Java. You must set server compatibility to 9.2 or later to generate code for Java projects. You need the J/Foundation upgrade to Informix Dynamic Server to enable Java services.

See the *IBM Informix Guide to SQL: Tutorial* for general information on casts.

The following sections describe properties of casts.

Source and Target Data Types

You cannot create a cast between two built-in or qualified data types.

You also cannot create a cast that includes any of the following data types as either the source type or target type for the cast:

- Collection data types: LIST, MULTISET, or SET
- Unnamed row types
- Smart-large-object data types: BLOB or CLOB

Implicit and Explicit Casts

You can specify whether a cast is called for *implicit conversions*. Implicit conversions allow the database server to use the cast when it is not called explicitly in an SQL statement.

For example, if you call the **Plus()** function with a DOLLAR argument, the database server searches for an implicit cast from DOLLAR to a data type for which the **Plus()** function is defined. If an implicit cast exists, the database server calls the conversion function and then calls the **Plus()** function without error. If no cast is specified with implicit conversion, the **Plus()** function call results in an error message from the database server.

In this example, you create an implicit cast from DOLLAR to DOUBLE PRECISION to permit the database server to execute all functions defined for DOUBLE PRECISION on DOLLAR values. However, if you define a cast from DOLLAR to INTEGER, you do not want that cast to be implicit, because the conversion function truncates dollar values, resulting in inaccurate results.

See *IBM Informix User-Defined Routines and Data Types Developer's Guide* for more information on implicit and explicit casts.

Cast Support Functions

If the source and target data types do not have the same binary representation, the database server calls a cast support function to perform the conversion. If the two types have the same binary representation, a cast support function might not be needed. You can also create a cast support function to perform other types of conversions, such as applying a mathematical formula. For example, you could create a cast support function to convert temperature between Fahrenheit and Celsius.

See *IBM Informix User-Defined Routines and Data Types Developer's Guide* for more information on creating cast support functions.

Defining Errors

DataBlade module routines can print error messages and trace messages. Error messages are printed with the **mi_db_error_raise()** function. Trace messages are written to a trace file with the **DBDK_TRACE** macros or the **gl_dprintf()** macro. See "Tracing and Error Handling" on page 5-7 for more information on tracing and error handling.

Although it is possible to hard-code messages in your routines, defining them in BladeSmith makes them easier to edit. Also, BladeSmith generates code that uses the IBM Informix Global Language Support (GLS) API, so messages that you create in BladeSmith can be easily localized.

The following table lists the properties you specify when you create an error.

Property	Default Value	Description
SQL error code	None	A five-character error code. This character string uniquely identifies the error or trace message. See "SQL Error Code" on page 4-14 for more information.
Error locale	The locale of your Windows installation	An Informix locale specification for the message. See "Error Locale" on page 4-14 for more information.
Register message as error, trace, or both?	**Error**	Error messages are added to the **syserrors** system table. Trace messages are added to the **systracemsgs** system table. If you choose **Both**, the message is added to both system tables.
SQL error text	None	A character string that can contain embedded parameters to be replaced with current values at runtime. See "SQL Error Text" on page 4-15 for more information.

The following sections describe properties of errors.

SQL Error Code

To ensure that your error codes do not conflict with built-in error codes and those of other DataBlade modules, consider qualifying the code with a three-character DataBlade module prefix such as *USR*.

See "Developing Your DataBlade Module" on page 1-5 for information on how to design your error codes.

Error Locale

The error locale enables the database server to select a translated error or trace message for a localized database. The locale is specified using the format *language_country.codeset*. Be sure to create messages for all of the locales in which your DataBlade module executes.

UNIX Only

The default BladeSmith locale, **en_us.8859-1**, is for U.S. English using code set **8859**. This is the default locale for the Informix database server on UNIX platforms.

Windows Only

The default code set for the Informix database server on Windows is **1252**. Create U.S. English messages using locale **en_us.1252** for Windows database servers.

End of Windows Only

For more information on locales, see the *IBM Informix GLS User's Guide*.

SQL Error Text

The SQL error text is displayed with the error code in the language specified by the message locale. To specify parameters in messages, assign each parameter a unique name enclosed in percent characters (%). For example, an input function could send the following message when it is unable to translate an input value:

```
%FUNCNAME%: Unable to decipher input '%INPUT%'.
```

For information about tracing and calling error messages, see the *IBM Informix DataBlade API Programmer's Guide*.

Defining Interfaces

If you expect other DataBlade modules to use the functionality provided by your DataBlade module, create an interface. DataBlade developers can include the interface in a DataBlade module to ensure that BladeManager registers the DataBlade module with the interface before registering the DataBlade module dependent on the interface.

The interface you define encompasses all of your DataBlade module.

The following table lists the properties you specify when you create an interface.

Property	Default	Description
Interface name	**INewInterface**	The name of the interface. Must be unique. Change the name to **Iproject**, where **project** is the name of your DataBlade module.
Interface description	None	Optional. A description of the interface and its intended purpose.

For Dynamic Server Version 9.2 and later, the maximum length of an interface name is 64 characters.

For Dynamic Server Version 9.14, the maximum length of an interface name is 18 characters.

Creating Routines

You can define public or private user-defined routines that support your DataBlade module. You can specify if the routine is called by SQL or is an internal routine.

Routines can be *functions*, which return values, or *procedures*, which do not return values. Routines can be written in the C or Java programming languages or the Informix Stored Procedure Language (SPL).

Use the New Routine wizard to:

- Overload existing routines for extended data types.

  Existing routines can be built-in or user-defined. Built-in routines include operator and other arithmetic functions, and support routines. For a list of built-in routines you can overload, see *IBM Informix User-Defined Routines and Data Types Developer's Guide*.
- Overload operators for built-in aggregates for extended data types.

  Built-in aggregates include AVG, DISTINCT, MAX, MIN, RANGE, SUM, STDEV, and VARIANCE. For a list of the operators you must overload for each built-in aggregate, see *IBM Informix User-Defined Routines and Data Types Developer's Guide*.
- Create new user-defined routines for built-in or extended data types.

The following table lists the properties you specify when you create a routine.

Property	Default Value	Description
Routine name	***prefix*Routine**	The name of the routine. If you are overloading a routine, the name can be an existing routine name; otherwise, it must be a unique name. See "Routine Name" on page 4-18 for more information.
Return type	No return type	The data type that is returned by the routine. Functions return a value, but procedures do not.
Statement local variable? (Available for routines that return values only)	No	Whether the last argument passed to the function is an OUT parameter for a statement local variable, allowing the function to return two values. See "Statement Local Variables" on page 4-19 for more information.
Arguments	None	The name, data type, and default value of each argument passed to the routine. See "Routine Arguments" on page 4-19 for more information.
Language	**C**	The programming language in which to write the routine: C, Java, or Stored Procedure Language (SPL). You must set server compatibility to 9.2 or later to generate code for Java projects. You need the J/Foundation upgrade to Informix Dynamic Server to enable Java services.
SQL routine body (SPL routines)	None	The SPL statements that define the routine. See the *IBM Informix Guide to SQL: Syntax* for more information on SPL.

Property	Default Value	Description
Does not accept null values? (C and Java routines)	Yes	Whether the routine accepts null values. If a routine that does not accept nulls is passed a null value, the database server returns a null value without calling the routine.
Is variant? (C and Java routines)	Yes	Variant routines can return different values with the same input arguments. The database server never caches results from variant routines. See "Variant Functions" on page 4-19 for more information.
Is parallelizable? (C and Java routines)	No	Parallelizable routines can be split into subqueries and processed in parallel. See "Parallelizable Routines" on page 4-20 for more information.
Is a DBA routine? (C and Java routines)	No	The routine can be created or executed only by a user with DBA permissions.
Never called from SQL? (C routines)	No	If a routine cannot be called from SQL, it is an internal routine that can only be called directly by the database server: for example, primary access method routines.
An iterator? (C and Java routines)	No	Iterator routines return a set of values, one value at a time. See the *IBM Informix DataBlade API Programmer's Guide* for more information.
C routine name (C routines)	***prefix*Routine**	The name of the routine in the shared object file. Must be unique. See "C Routine Name" on page 4-20 for more information.
Shared object path (C and Java routines)	**$INFORMIXDIR/extend/%SYSBLDDIR%/project.bld** (C routines) **%JAVAPATH%** (Java routines)	The relative or absolute path and filename of the shared object. The default path and filename is recommended.
Well behaved or poorly behaved? (C and Java routines)	Well behaved	Well-behaved routines can run in the CPU virtual processors; poorly behaved routines should run in a user-defined virtual processor. See "Routine Behavior" on page 4-20 for more information.
User-defined virtual processor class (C and Java routines)	**default_class**	The name of the user-defined virtual processor class in which a poorly behaved routine runs. See "User-Defined Virtual Processor Class Name" on page 4-20 for more information.

Property	Default Value	Description
Special stack size requirements? (C routines)	No	Whether the routine needs an unusually large amount of virtual shared memory to execute. See "Stack Size" on page 4-21 for more information.
Cost of routine (C routines)	**0**	The relative cost of the routine, for query optimization. See "Cost of Routine" on page 4-21 for more information.
Negator routine?	No	A routine that returns the opposite Boolean result with the same arguments. Used for query optimization. See "Related Routines" on page 4-21 for more information.
Commutator routine? (C routines)	No	A routine that returns the same Boolean result with the arguments in reverse order. Used for query optimization. See "Related Routines" on page 4-21 for more information.
Selectivity routine? (C routines)	No	A routine that estimates the percentage of rows returned by the routine. Used for query optimization. See "Related Routines" on page 4-21 for more information.

The SQL that BladeSmith generates for routines uses the ALTER FUNCTION statement to specify all but the following properties:

- Routine name
- Return type
- Statement local variable
- Arguments
- Language
- DBA routine

Using the ALTER FUNCTION statement allows BladeManager to re-register the routine without dropping and recreating it.

The following sections describe properties of routines.

Routine Name

Specify the name of an existing routine to overload it for a new data type, or specify a unique routine name to create a new routine.

If you are creating a selectivity routine for a user-defined routine, name the selectivity routine **RoutineSelectivity**.

You can overload built-in operator and other arithmetic routine for collection, row, and distinct data types. (You can overload most arithmetic routines for opaque data types with the New Opaque Type wizard.) How arithmetic routines operate on collection and row data types is determined by the code you write for them. For example, if you overload the **Plus()** function for a row data type, it might either:

- Add the values of the fields and return a row data type with the same number of fields as the originals.
- Return a row data type with twice as many fields as the originals.

See *IBM Informix User-Defined Routines and Data Types Developer's Guide* for a list of built-in routines you can overload.

Although it is not necessary, you can create new support routines for collection, row, and distinct data types.

Statement Local Variables

If you want your function to return two values, check the statement local variable check box. The last argument for your function is then defined as an OUT parameter. The OUT parameter corresponds to a value the function returns indirectly, through a pointer, to a statement local variable (SLV). The value the function returns through the pointer is an extra value, in addition to the value it returns explicitly.

The SLV provides a temporary name that a single statement can manipulate. An SQL statement uses each SLV to transmit the output from a single function to other parts of the SQL statement.

See the *IBM Informix DataBlade API Programmer's Guide* for more information.

Routine Arguments

A routine can accept 0 to 20 arguments.

Arguments passed to a routine have the following properties:

- **Name.** The name must follow SPL, C, or Java language naming conventions.
- **Data type.** For SPL and C, any existing data type that appears on the list; for Java, any existing data type except row or collection data types. If you want to use a data type that is not on the list, you must first create it in BladeSmith. For more information on SQL data types, see *IBM Informix Guide to SQL: Reference.*
- **Default value.** Optional. The value of the argument if a value for the argument is not specified when the routine is called.

Variant Functions

By default, user-defined functions are *variant*. Variant functions can return different values or have varying side effects, given the same arguments. For example, a function that returns the current date or time is a variant function. However, a function that appears nonvariant can also have varying side effects, such as updating a table or external file.

The cost of defining a nonvariant function as variant is low: you might experience slightly diminished performance. However, the cost of defining a function that exhibits variant behavior as nonvariant can be high, because a query might return incorrect results.

Most functions are not variant; marking them as nonvariant improves performance. If the function is nonvariant, the database server might cache the return values of expensive functions or run parallel queries. Functional indexes are only allowed on nonvariant functions.

See the *IBM Informix User-Defined Routines and Data Types Developer's Guide* publication for more information.

Parallelizable Routines

Mark a routine as parallelizable if it can be executed within a parallel database query (PDQ) statement. PDQ statements allow the Informix database server to distribute the executions of one query among several processors by dividing the query into subqueries. The database server then allocates subqueries to separate threads for parallel processing and thus improves performance. See *IBM Informix User-Defined Routines and Data Types Developer's Guide* for more information about using the parallelizable option.

Use routine parallelization if your routine is used as an expression in qualification clauses, in GROUP BY lists, or as an overloaded comparison operator.

A routine cannot be parallelizable if it accepts row or collection data types as arguments.

C and Java routines are parallelizable if they call only the DataBlade API routines listed in the following categories from the *IBM Informix DataBlade API Programmer's Guide*:

- Data handling, except for collection manipulation functions
- Session, thread, and transaction management
- Function execution
- Memory management
- Exception handling
- Callbacks
- Miscellaneous

See the *IBM Informix DataBlade API Programmer's Guide* for more information about the routines under each category.

C Routine Name

SQL allows overloading of routine names; however, the C language does not. Therefore, if you overload a routine, you must give it a unique C name.

Routine Behavior

A routine is well-behaved within the context of Informix database server architecture if it:

- Yields the virtual processor on a regular basis to other threads.
- Does not use blocking operating-system calls.

If your routine violates one of these conditions, mark it as poorly behaved and type the name of a user-defined virtual processor in the user-defined virtual processor class field.

User-Defined Virtual Processor Class Name

The name of the grouping class for the user-defined virtual processor must be 128 alphanumeric characters or fewer, and it must be unique. The class name is case

insensitive. It is recommended that you begin the name of your virtual processor class with your DataBlade module new object prefix.

Tip: You can create a routine that references a virtual processor class before that class exists. However, you must create the virtual processor class and create virtual processors in it before you register your DataBlade module in the database.

Stack Size

You can specify stack size only for a user-defined routine written in C.

Stack space is allocated from a common region in shared memory that can be overrun if a routine consumes more stack space than is allocated for it. To avoid stack overrun:

- allocate sufficient stack space for all the local variables in the routine. Monitor stack usage with the **onstat** utility. See the *IBM Informix Administrator's Guide* for more information on the **onstat** utility.
- execute recursively called C routines with the **mi_call()** function. Pass any arguments other than **mi_integer** data types by reference. See the *IBM Informix DataBlade API Programmer's Guide* for more information on **mi_call()**.

When you specify a stack size for a user-defined routine, the database server allocates the specified amount of memory for every execution iteration of the routine.

See the *IBM Informix Dynamic Server Administrator's Guide* for more information on stacks.

Cost of Routine

You can specify cost only for user-defined routines written in C.

The relative cost of the routine is used by the query optimizer to determine the order in which to process WHERE clauses in a SELECT statement. Expensive routines are called after inexpensive routines. A cost of `0` indicates that the routine costs about the same as the routines in the reference list that have a cost of `0`. The reference list shows all user-defined routines created in the project. The standard formula for computing routine cost is:

lines_of_code + (*I/O_operations* x 100)

Because the optimizer compares routine costs, the actual cost is irrelevant; only the relative cost matters. However, follow the general formula to ensure that your routines interact with other DataBlade module routines in a predictable way.

Related Routines

If your user-defined function compares or acts as a filter for two instances of the same data type and returns a Boolean result, you can specify related functions to optimize the execution of the function when it is called in the WHERE clause of a SELECT statement.

Important: Related functions must exist before you can choose them. You can create them before you create the function they are related to, or you can update the original routine to add related routines after you create them.

Commutator and Negator Functions: You can specify a commutator function only for user-defined routines written in C.

The database server calls a commutator or a negator function instead of the original function if the query optimizer determines that it is faster. A commutator function returns the same Boolean result as the original function with the same arguments but with the arguments in reverse order. A negator returns the opposite Boolean result as the original function with the same arguments in the same order.

Selectivity Functions: You can specify a selectivity function only for user-defined routines written in C.

A selectivity function estimates the percentage of rows that might be returned by your function, given a set of arguments. Define a selectivity function if you want to determine the cost of your function so that the query optimizer can determine when it is most efficient to call your function. Selectivity functions determine the cost of a function with statistics gathered about the values of the data type on which the function operates. See "User-Defined Statistics" on page 2-9 for more information on selectivity functions and how they process user-defined statistics.

You can create a selectivity function for your user-defined function if your function compares or acts as a filter for two values of the following kind of data types:

- An opaque data type for which you have created user-defined statistics support routines (see "Statistics Support" on page 4-34)
- A built-in data type

The B-tree functions **Equal()** and **NotEqual()** that are overloaded for an opaque data type are good candidates for selectivity functions. Because the **Equal()** and **NotEqual()** functions are created with the Opaque Type wizard, you must add selectivity support by assigning selectivity routines on their properties pages after you create them.

Built-in data types have built-in statistics support routines, and all qualifying built-in functions (such as B-tree functions) have built-in selectivity functions. You can only create selectivity functions for functions that take built-in data types if those functions are user-defined.

A selectivity function must have the following properties:

- A name in the form of **FunctionSelectivity()**, where **Function** is the name of the function to which it is assigned
- A double-precision return type
- Two arguments of type POINTER
- All other properties as their default values

For a description of user-defined statistics and selectivity, see "User-Defined Statistics" on page 2-9.

Creating Data Types

You can create the extended data types described in the following sections in BladeSmith:

- "Collection Data Type" on page 4-23, next
- "Distinct Data Type" on page 4-24
- "Opaque Data Type" on page 4-24
- "Row Data Type" on page 4-35

In addition, you must define qualified built-in data types (see "Qualified Data Type" on page 4-34) before you can use them in a BladeSmith project.

Collection Data Type

A collection data type is a set of elements of another, single data type. Collection elements can never be null.

You can overload existing user-defined routines and built-in routines to work on your collection data type. You can also define custom support routines for your collection data type. See "Creating Routines" on page 4-15 for instructions.

The following table lists the properties you specify when you create a collection data type.

Property	Default Value	Description
Type	None	The data type that makes up the collection. See Valid Element Data Types for more information.
Constructor	None	The type constructor: LIST, MULTISET, or SET. See "Type Constructors" on page 4-23 for more information.

See the *IBM Informix Guide to SQL: Tutorial* for general information on collection data types.

The following sections describe properties of collection data types.

Valid Element Data Types: You can create a collection with elements of any data type listed in your project except SERIAL or SERIAL8. You can define a collection type of an existing collection or row data type. For example, you can define a list of a set of integers in SQL:

```
LIST(SET(integer not null))
```

You can also create collections of opaque or distinct data types.

Tip: If you create a collection with an element of type BLOB or CLOB, you can test for the existence of a particular sbspace when your DataBlade module is being registered in a database using BladeManager. For information, see Appendix C, "Testing for an Sbspace," on page C-1.

Type Constructors: The type constructor determines the structure of the collection. The following table shows the options between the type constructors.

Constructor	Elements Ordered?	Duplicates Allowed?
LIST	Yes	Yes
MULTISET	No	Yes
SET	No	No

See the *IBM Informix Guide to SQL: Reference* for more information about collection type constructors.

Distinct Data Type

A distinct data type has an internal and external representation identical to another data type, but the database server treats it as a different data type. Any existing routines on the source data type are automatically registered on the distinct data type. However, you can define new routines that operate only on the distinct data type.

You can define custom support routines and user-defined routines for your distinct data type. You can also overload existing user-defined routines and built-in routines to work on your distinct type. See "Creating Routines" on page 4-15 for instructions.

The following table lists the properties you specify when you create a distinct data type.

Property	Default Value	Description
Name	**prefixDistinctType**	The name of the distinct type. This name must be unique.
Source type	None	The data type the distinct type is based on. Can be any existing data type. The distinct type inherits all properties of the source type.

When you create a distinct data type, the database server creates explicit casts between the source data type and the distinct data type; however, you can also create implicit casts between a distinct data type and its source data type.

Tip: If you create a distinct type with a source type of BLOB, CLOB, or an opaque data type containing BLOB or CLOB arguments, you can test for the existence of a particular sbspace when your DataBlade module is being registered in a database using BladeManager. For information, see Appendix C, "Testing for an Sbspace," on page C-1.

See the *IBM Informix Guide to SQL: Reference* for more information on distinct data types.

Opaque Data Type

An opaque data type is a C structure or C++/ActiveX class. The database server does not interpret the contents of the structure. Instead, it calls support routines that you provide to manipulate the structure.

BladeSmith generates much of the code for the support routines. You must complete the code and compile the source code.

The following table lists the properties you specify when you create an opaque data type.

Property	Default Value	Description
Name	**prefixOpaqueType**	The name of the opaque type. See "Opaque Data Type Name Lengths" on page 4-26 for more information.

Property	Default Value	Description
Server implementation	**C**	Which language to use for database server source code for your opaque data type: C, C++, or Java. You must set server compatibility to 9.2 or later to generate code for Java projects. You need the J/Foundation upgrade to Informix Dynamic Server to enable Java services. See "Server Implementation" on page 4-26 for more information.
Client implementation	None	Whether to generate value objects as a client interface for your opaque data type in ActiveX. See "Client Implementation" on page 4-26 for more information.
Generate accessor methods? (ActiveX client implementation)	None	Whether to create accessor methods for value objects; that is, whether to expose the members of the data structure that defines your opaque data type as properties. See "Accessor Methods" on page 4-27 for more information.
Define internal structure? (C)	Yes	Whether you enter information on the internal members of the opaque type in BladeSmith. See "Definition of Internal Structure" on page 4-27 for more information.
Fixed or variable size? (C)	Fixed size	Whether the opaque data type varies in size. See "Fixed or Variable Size" on page 4-27 for more information.
Total size (if you choose not to specify the internal structure to BladeSmith and choose fixed size)	None	The total size of the opaque data type. The maximum size is 32 KB. If you do not specify a size, BladeSmith calculates it.
Member information (if you choose to specify the internal structure to BladeSmith)	None	The name, data structure, and array size of the members making up the opaque type. See "Member Information" on page 4-27 for more information.
Limit allocation size? (variable-length opaque data types)	No	The maximum size allowed the opaque data type, not to exceed 32 KB. See "Maximum Size" on page 4-28 for more information.

Property	Default Value	Description
Memory alignment (if you choose not to specify the internal structure to BladeSmith)	**4**	The alignment value for the first member of the opaque data type. See "Memory Alignment" on page 4-28 for more information.
Support routines	Basic text input/output Binary send/receive with client (C and C++) Text file import/export Binary file import/export (C and C++) Type compare support	The routines necessary to operate on the internal structure of the opaque data type and optional built-in routines. See "Support Routines" on page 4-29 for more information.

See *IBM Informix User-Defined Routines and Data Types Developer's Guide* for more information on opaque data types.

The following sections describe the properties of opaque data types you need to define when you create an opaque data type with BladeSmith.

Opaque Data Type Name Lengths: The limit of the opaque data type name is determined by the version of the database server and the client implementation language, as shown in the following table.

Language	Version 9.14	Version 9.2 or later
C	14	110
C++/ActiveX	14	80

Server Implementation: The server implementation is the programming language in which you implement your opaque data type within the database server. Each language has restrictions on the functionality you can specify for your opaque data type:

- **C.** You cannot choose a client implementation or accessor methods in C.
- **C++.** See "Opaque Type Limitations for C++/ActiveX" on page 3-4 for a list of C++ restrictions.
- **Java.** You cannot generate Java code for opaque types with DBDK 4.0.

Client Implementation: A client implementation of your opaque data type is a value object, which BladeSmith generates in the programming language you specify. A value object is a client-side interface to an opaque data type and its support routines. Client and server implementations need not be in the same programming language.

For information on the implications of using different languages for server and client implementations, see "Mixing Languages in Server and Client Implementations" on page 3-3.

For more information on value objects, see "ActiveX Value Objects" on page 3-2.

Accessor Methods: If you choose to create a client implementation of your opaque data type, you can specify whether to generate accessor methods. Choosing this option makes the members of the opaque data type available as properties, allowing client-side access to those values. BladeSmith generates set and get methods for each property.

For more information on ActiveX accessor methods, see "ActiveX Properties" on page 6-4.

Definition of Internal Structure: You can specify an undefined internal structure for an opaque data type with a server implementation in C or Java.

The internal structure of the opaque data type is not known to the database server. The support routines you define for the opaque data type operate on the internal structure.

If you define the internal structure of your opaque data type to BladeSmith, BladeSmith generates useful code for it. If you do not specify the internal structure, BladeSmith generates code that operates as if your opaque data type is a stream of bytes.

Fixed or Variable Size: You can specify a variable-sized structure for an opaque data type with a server implementation in C or Java.

An opaque data type can have a fixed size that is determined by the sum of the sizes of the data structures within the opaque data type. The maximum size is 32 KB.

Alternatively, an opaque data type can have a variable size if one of its internal data structures does not have a fixed size. Typically, variable data structures are smart large objects or other opaque data types. Variable-length data structures can have a maximum size. Variable-length opaque data types are treated as bit-varying types.

Member Information: Specify the following information about the internal members of your opaque data type:

- **Name.** Must be unique within the opaque data type.
- **Data structure.** Select from the list, which includes any extended data types you have defined or imported in the project and the data structures that correspond to the programming language you choose.
- **Array size.** Specifies the number of components and subcomponents.

The following table maps the DataBlade API data structures listed in the Opaque Type wizard to their external programming language equivalents.

DataBlade API Data Types	C and ESQL/C Data Types	ActiveX Data Types
gl_wchar_t	char	BSTR
mi_boolean	boolean	BOOL
mi_char	char	BSTR
mi_char1	char	BSTR
mi_date	int	BSTR
mi_datetime	dtime_t	BSTR
mi_decimal	dec_t	BSTR
mi_double_precision	double	double
mi_int1	char	short
mi_int8	ifx_int8_t	BSTR
mi_integer	int, long	long
mi_interval	intrvl_t	BSTR
MI_LO_HANDLE	ifx_lo_t	
mi_money	dec_t	BSTR
mi_numeric	dec_t	BSTR
mi_real	float	double
mi_smallint	short	short
mi_string	char	BSTR
mi_unsigned_char1	unsigned char	short
mi_unsigned_int8	ifx_int8_t	BSTR
mi_unsigned_integer		long
mi_unsigned_smallint	uint2	short
mi_wchar	uint2	BSTR

If you choose to create a variable-length opaque data type, a member is automatically added as an **mi_int1** of variable size. Change the **mi_int1** data structure to be the one you need. Be sure to list the variable-length member last.

Tip: If you create an opaque type with a member of type MI_LO_HANDLE, you can test for the existence of a particular sbspace when your DataBlade module is being registered in a database using BladeManager. For information, see Appendix C, "Testing for an Sbspace," on page C-1.

Maximum Size: If you create a variable-length opaque data type, specify the maximum allocated length of that data type. The database server does not allow an opaque data type to grow beyond its maximum length. If you choose to specify a maximum length, the maximum value is 32,767 bytes. This value, however, is the maximum size of a row in a database table. Therefore, if your opaque data type is 32,767 bytes, you cannot have any other columns in your table.

Memory Alignment: If you do not specify the internal structure of your opaque data type in BladeSmith, you must choose the memory alignment of the first member; your compiler aligns the other members with this value. Choose an alignment value that corresponds to the greatest alignment requirement in the data

structure. The default alignment is **4**. If you do not know the alignment of the member with the greatest alignment, choose **8**.

See *IBM Informix User-Defined Routines and Data Types Developer's Guide* for more information on memory alignment.

Support Routines: You can select from the following categories of support routines to support your opaque data type:

- "Basic Text Input and Output" on page 4-29
- "Binary Send and Receive" on page 4-29
- "Text Import and Export" on page 4-30
- "Binary File Import and Export" on page 4-31
- "Contains Large Objects" on page 4-31 (not available for C++)
- "Type Insert and Delete Notification" on page 4-31 (not available for C++)
- "Type Compare Support" on page 4-32
- "B-Tree Indexing Support" on page 4-32
- "Type Mathematic Operators" on page 4-33
- "More Mathematic Operators" on page 4-33
- "Type Concatenation Operator" on page 4-33
- "Type Hash Support" on page 4-33
- "Statistics Support" on page 4-34 (not available for C++ or Java)

These support routines are described in the following sections.

Basic Text Input and Output: This category is valid for the C and C++ languages. BladeSmith generates this category by default.

Basic text input and output functions convert between the text representation of the opaque data type and the internal database server format.

The text representation of an opaque data type is an **mi_lvarchar** value that contains a printable representation of an instance of the data type. The text representation enters values for the data type in SQL statements such as INSERT and displays values in output from SQL statements such as SELECT.

The names of these functions differ for different programming languages, as listed in the following table.

Language	Function Names
SQL	**OpaqueIn()**
	OpaqueOut()
C	**OpaqueInput()**
	OpaqueOutput()
C++	**FromString()**
	ToString()

Binary Send and Receive: This category is valid for the C and C++ languages. BladeSmith generates this category by default.

Binary send and receive functions transfer the binary representation of the opaque data type to and from the client.

Binary send and receive functions allow the client and server to execute on different platforms, with different data type representations. When a client connects with a server, it sends a description of its data representation. The server calls the binary send function to convert opaque data type values to the client format before sending them to the client. The binary receive function converts a value arriving from the client binary format to the server binary format.

You do not have to know the specifics about data representation on different platforms to convert an instance of a data type. Binary send and receive functions call DataBlade API routines for each member of the structure to convert values to the appropriate C data type representation for the destination platform.

The names of these functions differ for different programming languages, as listed in the following table.

Language	Function Names
SQL	**OpaqueSend()**
	OpaqueRecv()
C	**OpaqueSend()**
	OpaqueReceive()
C++	**Send()**
	Receive()

Text Import and Export: This category is valid for the C and C++. BladeSmith generates this category by default.

Text file import and export functions transfer the text representation of the opaque data type to and from a flat file.

Text file import and export functions enable bulk copy for opaque data types. When you copy data from a file into a database with PLOAD or the DB–Access LOAD command, the server calls a text file import function to convert the incoming value to the server binary format. When data is copied out of the database into an external file, the server calls a text file export function to convert the value from server binary format to text file format.

You need text file import and export functions for opaque data types that include large objects or that are exported to a disk file. On copy-out, the text file export function creates a file on the client, writes the large object data to it, and then sends the name of the file as the data value for storage in the copy file. The text file import function takes the filename, opens it, and loads the large object data. This method stores large object data independent from the copy file so that the copy file is smaller and easier to read.

If you do not define text file import and export routines, the server calls the text input and output routines.

The names of these functions differ for different programming languages, as listed in the following table.

Language	Function Names

SQL	**OpaqueImpT()**
	OpaqueExpT()
C	**OpaqueImportText()**
	*Opaque***ExportText()**
C++	**ImportText()**
	ExportText()
Java	**textImport()**
	textExport()

Binary File Import and Export: This category is valid for the C and C++ languages. BladeSmith generates this category by default.

Binary file import and export functions transfer the binary representation of the opaque data type to and from a flat file.

Use the binary file import and export functions for bulk copy of binary data. These functions are the same as the text file import and export functions, except that they operate on binary representations of the data type. The functions are called when PLOAD executes.

The names of these functions differ for different programming languages, as listed in the following table.

Language	**Function Names**
SQL	**OpaqueImpB()**
	OpaqueExpB()
C	**OpaqueImportBinary()**
	OpaqueExportBinary()
C++	**ImportBinary()**
	ExportBinary()

Contains Large Objects: This category is valid for the C language.

The **LOhandles()** function retrieves a list of the pointer structures for the smart large objects embedded in the opaque data type. The database server calls the **LOhandles()** function to obtain a list of large objects used by an opaque data type. The **LOhandles()** function takes a pointer to an instance of the data type and returns an array of the large object handles used by the object.

This category also includes the **Assign()** function and the **Destroy()** procedure described in the section "Type Insert and Delete Notification," next.

For the C language, BladeSmith prefixes the names of these functions with the name of the data type for which they are specified: **OpaqueLOhandles()**, **OpaqueAssign()**, and **OpaqueDestroy()**.

Type Insert and Delete Notification: This category is valid for the C language.

The **Assign()** function and the **Destroy()** procedure perform tasks before storing or deleting an opaque data type on disk: for example, to ensure proper reference counting on smart large objects.

You can provide **Assign()** and **Destroy()** routines for an opaque data type that requires special processing when an instance is stored in a database or removed from the database. The database server calls the **Assign()** function before it writes a value to a table. The database server calls the **Destroy()** procedure before it deletes a value from a table.

For example, opaque data types that include large objects require special handling before they are stored on disk or removed from a table. The database server maintains a reference count for large objects to ensure that an object is not dropped while a row in the database references it. When a reference to a large object is inserted into a table, the **Assign()** function increments the reference count. When a reference to a large object is deleted from a table, the **Destroy()** procedure decrements the reference count.

For the C language, BladeSmith prefixes the names of these functions with the name of the data type for which they are specified: **OpaqueAssign()** and **OpaqueDestroy()**.

Type Compare Support: This category is valid for the C and C++ languages. BladeSmith generates this category by default.

Type comparison functions **Compare()**, **Equal()** (bound to the = operator), and **NotEqual()** (bound to the <> and != operators) compare two opaque data types: for example, to support an ORDER BY clause in a query.

For the C language, BladeSmith prefixes the names of these functions with the name of the data type for which they are specified: **OpaqueCompare()**, **OpaqueEqual()**, and **OpaqueNotEqual()**.

B-Tree Indexing Support: This category is valid for the C and C++.

The following B-tree strategy and support functions support using the B-tree secondary access method to create an index on your opaque data type column:

- **Compare()**
- **Equal()** (bound to the = operator)
- **LessThan()** (bound to the < operator)
- **GreaterThan()** (bound to the > operator)
- **LessThanOrEqual()** (bound to the <= operator)
- **GreaterThanOrEqual()** (bound to the >= operator)
- **NotEqual()** (bound to the != and the <> operators)

Defining these routines for opaque data types allows fast B-tree index searches on the new data types. If a query uses the operator bound to one of the functions, the optimizer can evaluate strategies that use the index.

For the C language, BladeSmith prefixes the names of these functions with the name of the data type for which they are specified: ***Opaque*Compare()**, ***Opaque*Equal()**, ***Opaque*LessThan()**, ***Opaque*GreaterThan()**, ***Opaque*LessThanOrEqual()**, and ***Opaque*GreaterThanOrEqual()**.

R-Tree Indexing Support: Version 4.0 of BladeSmith does not generate code for R-tree support routines.

Refer to the *IBM Informix R-Tree Index User's Guide* or the IBM Informix Developer Zone at http://www.ibm.com/software/data/developer/informix for information about creating DataBlade modules that use the R-tree secondary access method.

Type Mathematic Operators: This category is valid for the C and C++ languages.

Binary arithmetic operators **Plus()** (bound to the + operator), **Minus()** (bound to the - operator), **Times()** (bound to the * operator), and **Divide()** (bound to the / operator) perform operations on your opaque data type.

If you define these routines for an opaque data type, the database server can resolve mathematical expressions in the select list or WHERE clause of a query.

For the C language, BladeSmith prefixes the names of these functions with the name of the data type for which they are specified: ***Opaque*Plus()**, ***Opaque*Minus()**, ***Opaque*Times()**, and ***Opaque*Divide()**.

More Mathematic Operators: This category is valid for the C and C++ languages.

Unary arithmetic functions **Positive()** (bound to the + operator) and **Negate()** (bound to the - operator) perform operations on your opaque data type.

If you define these routines for an opaque data type, your database server can resolve mathematical expressions in the select list or WHERE clause of a query.

For the C language, BladeSmith prefixes the names of these functions with the name of the data type for which they are specified: ***Opaque*Positive()** and ***Opaque*Negate()**.

Type Concatenation Operator: This category is valid for the C and C++ languages.

The **Concat()** function (bound to the `||` operator) concatenates the values of two opaque data types.

For the C language, BladeSmith prefixes the names of these functions with the name of the data type for which it is specified: ***Opaque*Concat().**

Type Hash Support: This category is valid for the C and C++ languages.

You should define a **Hash()** function for your opaque data type if the database server cannot use the built-in hashing function to cache its return values.

Most data types are *bit-hashable* and can use the built-in hash routine.

Bit-hashable data types have the property that for any hash routine:

if A = B then *hash*(A) = *hash*(B)

In practice, this means that A and B have identical bit representations.

There are some data types for which two equal values have different bit representations. For example, in one's-complement notation, there are two distinct representations for `0` (`+0` and `-0`). The SQL rules for the data type VARCHAR

require that trailing blanks be ignored in equality comparisons. Thus, two VARCHAR values with different numbers of trailing blanks will have different bit representations, but they should still be considered equal.

For data types that are not bit-hashable, you must provide a **Hash()** function.

For the C language, BladeSmith prefixes the names of these functions with the name of the data type for which it is specified: ***Opaque*Hash().**

Statistics Support: Statistics support is available with Dynamic Server Version 9.2 and later.

This category is valid for the C language.

User-defined statistics provide a way to improve performance when you compare opaque data type values. User-defined statistics compile information about the values in an opaque data type column that the query optimizer can use when it creates a query plan.

You can define statistics support functions for an opaque data type and then a selectivity function for a routine that takes opaque data types as its arguments. See "User-Defined Statistics" on page 2-9 for more information on user-defined statistics.

Statistics support functions are **OpaqueStatCollect()**, **OpaqueStatPrint()**, **Opaque_SetMinValue()**, **Opaque_SetMaxValue()**, and **Opaque_SetHistogram()**.

Tip: Statistics support functions reside in the **statistics.c** source code file, instead of in the *opaque*.**c** source code file with all other opaque data type support routines.

Qualified Data Type

A qualified data type is a built-in data type with additional specifications that provide information about the storage size, range of values, or precision of the data type. For example, CHAR is a built-in data type, but CHAR(16) is a qualified data type because you are fixing its length. You must add a qualified data type to a BladeSmith project before you can use it as a component of an extended data type.

When you create a qualified data type in a BladeSmith project, BladeSmith adds to the list of data types from which you choose when creating extended data types. Qualified data types do not need SQL or source code.

For example, to create a collection data type that stores sets of 16-byte character strings, you must first create a CHAR(16) qualified data type. Then create the collection data type, choosing CHAR(16) as the base data type and SET as the constructor function. The new data type has the following SQL definition:

```
SET(CHAR(16) not null)
```

The following table lists the data types that take qualifications.

Data Type	Qualification
CHARACTER, CHAR	(*size*)
CHARACTER VARYING	(*size*, *minimum*)
DATETIME	*largest_qualifier* TO *smallest_qualifier*

DECIMAL, DEC	(*precision*, *scale*)
INTERVAL	*largest_qualifier*(*n*) TO *smallest_qualifier*(*n*)
MONEY	(*precision*, *scale*)
NCHAR	(*size*)
NVARCHAR	(*size*, *minimum*)
SERIAL, SERIAL8	(*start value*)
VARCHAR	(*size*, *minimum*)

BladeSmith restricts your input for qualification values to valid choices.

See *IBM Informix Guide to SQL: Reference* for more information about qualified data types.

Row Data Type

A row data type is a group of fields of existing data types arranged like a row in a table. The fields of a row data type can be almost any data type that exists in your project, including other row data types.

You can overload existing user-defined routines and built-in routines to work on your row type. See "Creating Routines" on page 4-15 for instructions.

The following table lists the properties you specify when you create a row data type.

Property	Default Value	Description
Name	**prefixRowType**	The name of the row type. Must be unique. To create an unnamed row type, leave this field blank. See Named and Unnamed Row Data Types for more information.
Inherits from parent?	No parent	A row type can inherit the fields and routines of another (parent) row type. See "Row Data Type Inheritance" on page 4-36 for more information.
Field information	None	The name, data type, and nullability of the fields within the row type. See "Row Data Type Fields" on page 4-36 for more information.

The following sections describe properties of row data types.

Named and Unnamed Row Data Types: You can create a named or an unnamed row data type.

A named row data type has these general characteristics:

- Its name is unique.
- It supports inheritance from a parent row data type or to a child row data type.
- It can be used as the basis of a typed table.

An unnamed row type has these general characteristics:

- It is equivalent to any other unnamed row type with the same structure. The structure is defined by the number of fields, the data types of the fields, and the order of the fields.
- It does not support inheritance.
- It cannot be used as the basis of a typed table.

See the *IBM Informix Guide to SQL: Tutorial* for more information on named and unnamed row data types.

Row Data Type Inheritance: Named row data types can inherit from other named row data types. A child row data type inherits its parent's fields and can be passed to all routines defined for the parent data type.

You can add additional fields and routines that are only valid for the child data type.

See the *IBM Informix Guide to SQL: Tutorial* for more information on inheritance.

Row Data Type Fields: Fields in row data types can be any existing data type except SERIAL and SERIAL8.

Tip: If you create a row data type with a field of type BLOB or CLOB, you can test for the existence of a particular sbspace when your DataBlade module is being registered in a database using BladeManager. For information, see Appendix C, "Testing for an Sbspace," on page C-1.

Adding Functional Test Data

You can perform functional tests on your DataBlade module routines using the functional tests generated by BladeSmith. You must enter test data for functional tests in BladeSmith. You run functional tests on UNIX, or you run them on Windows using a UNIX-compatible toolkit, such as MKS Toolkit.

Tip: You can also generate unit tests, which run on Windows with the DBDK Visual C++ Add-In. See "Debugging Your DataBlade Module" on page 1-7 for more information.

You can add functional test data for opaque type support routines, user-defined routines, and cast support functions. Using the test data you enter, BladeSmith generates a functional test for each routine. BladeSmith creates UNIX shell scripts and SQL scripts to create test tables in a database, populate them with your test data, and run SQL scripts that execute the DataBlade module routines.

BladeSmith generates functional tests for an object only if you enter test data for it. You must regenerate functional tests whenever you add test data to update the functional test scripts.

Chapter 9, "Debugging and Testing DataBlade Modules on UNIX," on page 9-1 describes how to use the functional tests that BladeSmith generates.

To enter test data for an object, select the object and choose **Edit > Gather Test Data**.

The following sections describe the test data you enter in BladeSmith.

Test Data for Opaque Type Support Routines

To enter test data for an opaque type support routine, select the routine and choose **Edit > Gather Test Data**.

For opaque type support routines, each test data item includes the following elements:

- **An input value for the data type.** This value must be in the format specified for the text input routine.
- **The expected output value for the input value.** This value must be in the format specified for the type's text output routine.
- **An error code, if the input value is not valid.** Leave this entry blank when the input value is expected to be correct.

Enter values to test the opaque type boundaries. For example, if a type does not accept negative input values, enter test data with negative values and specify the error code you expect to receive from the text input routine.

The data you enter for an opaque type is used to test all of the supporting routines defined for the type. BladeSmith generates SQL scripts to test each supporting routine, including the text input and output routines and other routines, such as binary input and output routines and comparison routines. Add test data values that thoroughly test each of these routines.

Test Data for User-Defined Routines

To enter test data for a routine, select the routine and choose **Edit > Gather Test Data**.

The test data for user-defined routines includes the following items:

- **The input parameters for the test case.** Enter the input parameters in the same format you type them in an SQL statement. Enclose text parameters in single quotes.
- **The result expected from the function, if the input parameters are valid.** If you enter invalid input parameters, leave this field blank. If the user-defined routine is a procedure, the result field is not available because procedures do not return values.
- **An error code, if the input parameters are invalid.** Leave this field blank if the input parameters are expected to be valid.

For example, the Circle DataBlade module defines a **Contains()** function that takes a **Circle** value and a **Pnt** value and returns a Boolean result. To test the **Contains()** function with a **Circle** value of `'(12,12,2)'` and a **Pnt** value of `'(12,12)'`, enter the following input parameters:

```
'(12,12,2)','(12,12)'
```

Calling **Contains()** with these values should return a true result, which you can enter as `t`. Because the input parameters are valid, you leave the **Error code** field blank.

Test Data for Cast Support Routines

To enter test data for a cast support function, select the cast and choose **Edit > Gather Test Data**.

The test data for a cast support functions includes the following items:

- Input data, in the text input format specified for the source data type.
- Expected output data, in the text output format specified for the destination data type.
- The error code expected if the input data is not valid. If the input data is valid, leave this field blank.

Enter invalid input values and values that test boundary conditions for the data type.

Adding SQL Files

BladeSmith allows you to add custom SQL commands to the scripts that describe a DataBlade module and its objects. You can include SQL commands to create tables, indexes, or SPL procedures your DataBlade module requires. For example, if your DataBlade module uses smart large objects, you can include a statement to test for the required sbspace when you register the DataBlade module. For information, see Appendix C, "Testing for an Sbspace," on page C-1.

Use the three-character new object prefix assigned to your project in the name of every custom SQL object you create. The maximum size of an SQL file is 20 KB.

If you create any objects for your DataBlade module, add corresponding SQL DROP statements so that the objects are dropped when your DataBlade module is unregistered.

To add custom SQL, start the wizard by choosing **Edit > Insert > SQL Files**. Enter SQL commands directly into the BladeSmith edit window, or import a disk file into the BladeSmith project. When you import a file, copy the contents of the file at the time you import, or import the file by reference so that its contents are copied whenever you generate SQL scripts.

The following table lists the properties you specify when you include custom SQL statements.

Property	Default Value	Description
Descriptive name	**SQLfile**	A descriptive name for the custom SQL statements.
Read SQL text from file	None	Use to import SQL statements into the CREATE and DROP fields or to import by reference. See "Importing SQL Text from a File" on page 4-39 for more information.
Custom SQL CREATE text	None	A text field in which to type SQL CREATE statements.
Custom SQL DROP text	None	A text field in which to type SQL DROP statements.
Depends on objects	None	A list of objects on which the custom SQL depends. See "Object Dependencies" on page 4-39 for more information.
Which objects require SQL	None	A list of objects that depend on the custom SQL. See "Object Dependencies" on page 4-39 for more information.

The following sections describe properties of custom SQL files.

Importing SQL Text from a File

If you import SQL statements from a text file, separate the CREATE and DROP statements with a line of 40 hyphens (-). To import a file by reference, click **Browse** and select a file from the Open dialog box. If you import by reference, BladeSmith stores the full path and filename.

Object Dependencies

You can specify whether an SQL command depends upon a DataBlade module object or upon your SQL. These dependencies determine the sequence of SQL commands in the generated **objects.sql** script. BladeSmith generates SQL in the following sequence:

1. SQL commands to create objects upon which custom SQL commands depend
2. Custom SQL commands
3. SQL commands to create objects that depend upon custom SQL commands

The generated SQL scripts register dependencies when the DataBlade module is registered in a database.

Adding Client Files

Client files you include in your BladeSmith project are downloaded to client workstations after a DataBlade module is installed on a database server. Client files include:

- Graphical user interfaces
- Documentation and help files

- Shared object files, dynamic link libraries, or header files containing DataBlade module routines executed in the client address space

Client files are installed on the database server in platform-specific directories in the DataBlade module installation directory. Clients use BladeManager to download the files to their workstations.

To add a client file to your DataBlade module, choose **Edit > Insert > Client Files**. Type the full path and filename in the local path field or click the browse button (...) and select a file from the Open dialog box. This file must be accessible to both BladeSmith and BladePack. Select the appropriate client operating system, version, and architecture for your file. This information is used by BladeManager to install the specific client files on the correct client computer.

For Dynamic Server Version 9.2 and later, the maximum length of a client filename is 64 characters.

For Dynamic Server Version 9.14, the maximum length of a client filename is 18 characters.

Generating Files

When you generate files, BladeSmith creates files that describe the objects you defined in your project. The files include:

- SQL scripts that BladeManager executes to register the DataBlade module in databases and SQL scripts that create the DataBlade module objects in user databases
- Source files that contain basic code for the routines defined in your project
- Unit and functional test files
- Setup files that you use with BladePack to create an installation package

Generate SQL files, source files, functional test files, and installation files at any time. See "Regenerating Files" on page 4-46 for more information on when to regenerate. If you change any of the output directories, regenerate all files.

To generate files or change the properties of generated files, choose **Generate > DataBlade** to display the Generate DataBlade dialog box, as shown in Figure 4-2.

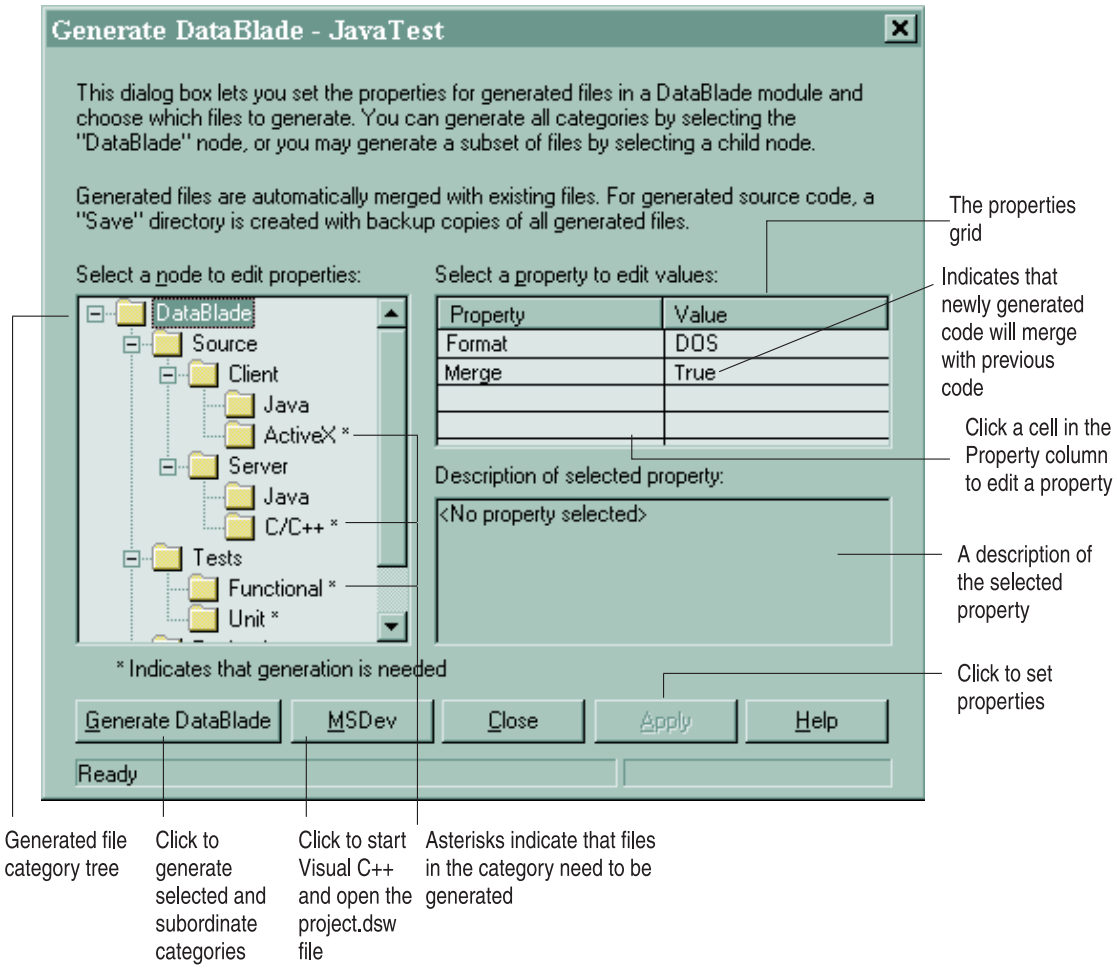


Figure 4-2. Generate DataBlade Dialog Box

The Generate DataBlade dialog box contains a file tree that shows categories of generated files. Each category is represented by a node in the tree. When you click a node, the **Generate** button changes to reflect the name of the category.

The directory structure of the generated files is illustrated by Figure 4-3.

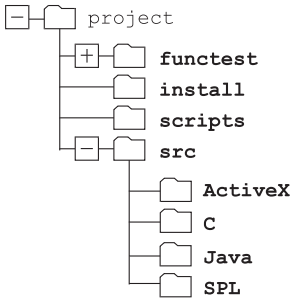


Figure 4-3. Generated File Directory Structure

Programming language subdirectories are created during generation only if you have defined objects in those languages.

This section includes the following subsections that describe the tasks you can perform using the Generate DataBlade dialog box:

Setting Generated File Properties

Most categories of generated files have properties that you can change. The properties appear in the Generate DataBlade dialog box **Properties** grid when you select a category in the file tree.

The following table lists the properties of the generated file categories, their default values, categories to which they belong, and a brief description of each.

Property	Default Value	Category	Description
Format	DOS	Generate DataBlade	The format of the generated files: • DOS: text lines end with a carriage return/linefeed pair. • UNIX: text lines end with a linefeed character.
Merge	True	Generate DataBlade	Whether to merge custom changes from previous source code files into the new files or to overwrite existing files. See "Merging Changes in Source Code and Unit Test Files" on page 4-46 for more information.
Directory	install	Packaging	The name of the directory that receives the generated files from each category. The path is relative to the directory that contains the project file.
	src	Source	
	scripts	SQL	
	functest	Tests	
Logging	False	Source	Whether to generate logging information.
Tracing	False	Source	Whether to add tracing to your generated source code. See "Tracing and Error Handling" on page 5-7 for more information.

Property	Default Value	Category	Description
MMX	False	C	Whether to allow Intel® MMX media enhancement technology in your DataBlade module. If you change the value to **True**, BladeSmith generates the **Gen_IsMMXMachine** function to check for an Intel MMX processor. See "The Gen_IsMMXMachine() Utility Function" on page 5-14 for more information.

To change a property of a generated file category:

1. Click the node of the category whose properties you want to change.
2. Click the name of the property whose value you want to edit in the **Property** column of the **Properties** grid.
3. Edit the value by typing a new value or selecting a new value from the popup list in the **Value** column.
4. Click **Apply**.

Generating All Files

The top-level node in the Generate DataBlade dialog box file tree is **Generate DataBlade** (see Figure 4-2 on page 4-41).

To generate all the files for your DataBlade module:

1. In the Generate DataBlade dialog box, click the **Generate DataBlade** node.
2. Edit the properties of the generated file categories, if necessary. See "Setting Generated File Properties" on page 4-42 for instructions.
3. Click **Generate DataBlade**.

The **Generate DataBlade** node has the following properties:

- **Format**. Use the **Format** property to specify how text lines end in the generated files:
  - **DOS.** Text lines end with a carriage return/linefeed pair.
  - **UNIX.** Text lines end with a linefeed character.
- **Merge.** The default value is **True**. To merge custom changes from previous source code and unit test files into the new files, select **True**. To overwrite existing files, select **False**. See "Merging Changes in Source Code and Unit Test Files" on page 4-46 for more information.

The default value for the **Format** property is **DOS**.

Generating SQL Scripts

To generate only the SQL scripts, click the **SQL** node in the Generate DataBlade dialog box; then click **Generate Scripts**.

The property associated with the **SQL** node is **Directory**. The default directory is **scripts**. You can change the name of the directory the SQL scripts are saved in, but the path must be relative to the project directory.

The following table describes the generated SQL scripts.

SQL Script	Purpose
prepare.sql	Contains SQL statements that describe the DataBlade module to BladeManager.
objects.sql	Contains SQL statements that update the **sysbldobjects** system table with information about the DataBlade module objects that are created in a database. BladeManager uses the information in the table to register, unregister, and upgrade DataBlade modules.
test.sql	Contains the SQL statements to create all objects in the DataBlade module projects and a GRANT EXECUTE statement.

BladeSmith generates separate files for locale-specific objects such as error messages. For example, the files for the default U.S. English locale are **prepare.en_us.sql** and **errors.en_us.1252**. Only one error message file is necessary per language. The database server automatically translates between languages. For example, the **errors.en_us.1252** file is sufficient for all **en_us** encodings; you do not need additional encodings like **error.en_us.8859-1**.

You can add SQL statements to the generated SQL scripts by adding an SQL file object to your project. See "Adding SQL Files" on page 4-38 for more information about adding SQL statements to a DataBlade module.

Warning: Do not edit generated SQL scripts. Use BladeSmith to make changes; then regenerate the scripts.

Generating Source Files

To generate only the source files for objects defined in your project, click the **Source** node or one of its subordinate nodes in the Generate DataBlade dialog box (see Figure 4-2 on page 4-41); then click **Generate Source**. The following table lists the source code file generation options.

Node	Code Generated
Source	All source code in the coding languages you use for your DataBlade module objects
Client	Client code (ActiveX or Java)
Server	Server code in the coding languages you specified when you create objects in BladeSmith
Individual language: **ActiveX**, **C++**, **C**, **Java**, or **SPL**	Source code only for the selected language

The **Source** node has these properties:

- **Directory.** The default directory is **src**. To change the name of the top-level directory the source code files are saved in, specify a new directory or path. The path must be relative to the project directory.
- **Logging.** The default value is **False**. To add logging information to your source code files, select **True**.
- **Tracing.** The default value is **False**. To add tracing to your generated source code, select **True**. See "Tracing and Error Handling" on page 5-7 for more information.

The property associated with the **C** node under the **Server** node is **MMX**. The **MMX** default value is **False**. You can choose whether to allow Intel MMX media enhancement technology in your DataBlade module. To generate the **Gen_IsMMXMachine** function to check for an Intel MMX processor, specify **True**. See "The Gen_IsMMXMachine() Utility Function" on page 5-14 for more information.

BladeSmith writes a header file, source files, and makefiles for Windows and UNIX platforms. It also generates other necessary files, depending on the coding language. For information on the source files BladeSmith generates, see "Source Files Generated by BladeSmith" on page 5-3 and Appendix A, "Source Files Generated for DataBlade Modules," on page A-1.

After you generate source files, edit the source files to add your code to the routine declarations BladeSmith generated. For a description of the contents of the generated files and how to modify and compile the generated code, see Chapter 5, "Programming DataBlade Module Routines in C," on page 5-1, Chapter 6, "Creating ActiveX Value Objects," on page 6-1, or Chapter 8, "Programming DataBlade Modules in Java," on page 8-1.

Generating Test Files

To generate only the test files, click the **Tests** node in the Generate DataBlade dialog box (see Figure 4-2 on page 4-41); then click **Generate Tests**. You can also choose to generate only functional tests or only unit tests.

The property associated with the **Tests** node is **Directory**. The default directory is **functest**. You can change the name of the directory the test files are saved in, but the path must be relative to the project directory. The **functest** directory only applies to functional tests; unit tests are generated in the **src** directory.

See Chapter 9, "Debugging and Testing DataBlade Modules on UNIX," on page 9-1, for information about executing functional tests.

Generating Installation Package Files

To generate only the installation packaging files that BladePack uses to build installation packages, click the **Packaging** node in the Generate DataBlade dialog box (see Figure 4-2 on page 4-41); then click **Generate Packaging**.

The property associated with the **Packaging** node is **Directory**. The default directory is **install**. You can change the name of the directory where the installation files are saved, but the path must be relative to the project directory.

When you generate installation package files, BladeSmith creates a set of files that you use with BladePack to generate installation scripts for your DataBlade module:

- **project.bom.** A bill of materials file.
- **project.cmp.** A components file.
- **project.prd.** A product file.
- **project.str.** A string file.

Important: Do not edit the generated installation package files. Instead, use BladeSmith to regenerate the installation package files after you have added or removed DataBlade module objects in the project file. For details about adding files to your installation package and generating installation packages, see Chapter 11, "Using BladePack," on page 11-1.

Regenerating Files

After you make changes in a BladeSmith project, regenerate the appropriate files. For example, if you add a cast to a DataBlade module, you must regenerate SQL scripts. If the cast has a support routine, you must also regenerate source files. When a node in the generated file category tree on the Generate DataBlade dialog box needs to be generated, it is followed by an asterisk.

BladeSmith uses two processes to regenerate files, depending on the file:

- **Source code and unit test files.** If merging is enabled, BladeSmith merges user changes into the newly generated code.
- **Visual C++ project, SQL, functional test, and installation files.** BladeSmith replaces existing files.

Merging Changes in Source Code and Unit Test Files

When BladeSmith finds existing source code and unit test files, it copies them to backup files before regenerating them.

If you add SQL statements to unit test files and then regenerate them, BladeSmith merges your code automatically.

If the **Merge** property of the **DataBlade** node in the Generate DataBlade dialog box is **True**, BladeSmith copies the changes you made to your source and unit test files into the newly generated source files. In this way, you can update your objects in BladeSmith without losing the code you added.

In source code files, BladeSmith does not remove code for routines that no longer exist in the project; you must manually remove such code. For example, suppose you create an opaque data type with an **Assign()** function specified and generate code; then you alter the opaque data type to no longer have an **Assign()** function. When you regenerate the source code, the **Assign()** function code remains.

If the **Merge** property of the **DataBlade** node in the Generate DataBlade dialog box is **False**, BladeSmith does not merge your previous changes into the new source files, but existing files are backed up. You can copy changes from the backup files into the newly generated files using a text editor.

If BladeSmith encounters problems while generating files, it displays the Generate Code Problem dialog box. For help in resolving merging problems, click **Help** on this dialog box and read the online help.

For information on how to use merging to upgrade projects created with a previous version of the Informix DataBlade Developers Kit, see the release notes.

Replacing Visual C++ Project, SQL, Functional Test, and Installation Files

BladeSmith does not merge changes to Visual C++ project, SQL files, functional test files, or installation files. When you regenerate SQL files and functional test files, BladeSmith regenerates the previous files as well as the Visual C++ project file. When you regenerate installation files, BladeSmith deletes relevant entries in the bill of materials file and adds them again. If you use BladePack to add files to the package, regenerating installation files does not affect your additions.

Opening the Project File in Visual C++

After you generate your C or C++ source code, you can launch Microsoft Visual C++ and open the project workspace file for your DataBlade module by clicking **MSDev** on the Generate DataBlade dialog box.

You can also open the DataBlade module project in Visual C++ from BladeSmith at any time by choosing **Tools > MSDev** or clicking the **MSDev** button in the toolbar.

To launch Visual C++ and open the Visual C++ workspace file:

1. With the project open in BladeSmith, choose **Tools > MSDev**.

   If the DBDK Visual C++ Add-In toolbar does not appear, you must add it manually. For instructions, see "Manually Loading the Add-In" on page 10-5.

   A dialog box asks if you want to select a local database server.
2. Click **Yes**.

   The add-in Properties dialog box appears. See "Specifying Properties for a Project" on page 10-5 for more instructions on assigning a database server and database to your project.

Chapter 5. Programming DataBlade Module Routines in C

© Copyright IBM Corp. 1996, 2008

In This Chapter

This chapter contains information to help you edit and compile C language source code generated by DataBlade. It includes the following sections:

- "Prerequisite Tasks," next
- "C Programming Task Overview" on page 5-3
- "Source Files Generated by BladeSmith" on page 5-3
- "Using Generated Code" on page 5-5
- "Editing Opaque Type Support Routines in opaque.c" on page 5-15
- "Editing Statistics Routines in statistics.c" on page 5-27
- "Editing Routines in udr.c" on page 5-28
- "Compiling DataBlade Module Code" on page 5-33

Prerequisite Tasks

Before editing and compiling your DataBlade module code, complete these tasks:

1. Write functional and design specifications that comply with Informix coding standards.

   See Chapter 3, "Programming Guidelines," on page 3-1, for more information.

2. Create your DataBlade module in BladeSmith.
   See "Creating DataBlade Module Objects" on page 4-8 for instructions.
3. Generate source code and SQL files in BladeSmith.
   See "Generating Files" on page 4-40 for instructions.

C Programming Task Overview

After you generate code with BladeSmith, complete these general tasks to finish your DataBlade module code:

Windows Only

1. Open the **project.dsw** file in Visual C++. You can do this from within BladeSmith. See "Opening the Project File in Visual C++" on page 4-47 for instructions.

End of Windows Only

2. Add code to these source code files to enable your routines to function as you intend:
   - **Opaque.c.** Add functionality for opaque data type support routines, as necessary. See "Editing Opaque Type Support Routines in opaque.c" on page 5-15 for instructions.
   - **statistics.c.** Add functionality for statistics support routines, as necessary. See "Editing Statistics Routines in statistics.c" on page 5-27 for instructions.
   - **udr.c.** Add functionality for user-defined routines, cast support functions, and aggregates. See "Editing Routines in udr.c" on page 5-28 for instructions.
3. Compile and link your source code files using a makefile or Visual C++ workspace file generated by BladeSmith. See "Compiling DataBlade Module Code" on page 5-33 for instructions.

To avoid merging conflicts when you regenerate your code, add code only in areas marked by `TO DO:` comments or after the generated code. If you do modify code outside the designated areas, after you regenerate you might have two copies of the routine: the one you modified and the one BladeSmith generated. Although your changes remain, you must resolve conflicts in the two pieces of code.

Source Files Generated by BladeSmith

When you create new objects, BladeSmith generates the source files; some filenames are prefixed with the name of the DataBlade module (indicated by **project**). By default, BladeSmith creates the source files in the **src** and **src\c** subdirectories of the directory that contains the BladeSmith project file. Generated source files are listed in the following table.

Filename	Directory	Type of File	More Information
project.h	src\c	C header file	See C Header File for more information.
support.c **udr.c** **Opaque.c** **statistics.c**	src\c	C source code file	You should edit only the **udr.c**, **Opaque.c**, and **statistics.c** files. See "C Source Code Files" on page 5-4 for more information.
project.def	src\c	C definition file	This file lists the exported routines declared in the source code file.
project.dsp **project.dsw**	src	Visual C++ files	See "Microsoft Visual C++ Files" on page 5-5 for more information.
readme.txt	src\c	Text file	This file describes the files in the **src\c** directory.
warning.txt	src\c	Text file	This file describes potential problems with your source code. See "Warning File" on page 5-5 for more information.
ProjectU.mak	src	Makefile	Use this file for compiling on UNIX. See "Compiling on UNIX" on page 5-33 for more information.

Some of these files are described in the following subsections.

C Header File

The **project.h** header file for the DataBlade module contains:

- definitions for error messages used in generated code.
- function prototypes for utility and tracing functions BladeSmith provides.
- DBDK_TRACE macros for the BladeSmith tracing facility.
- a type definition for the MI_LO_HANDLES structure returned by smart large object functions.
- type definitions for opaque type structures defined in the BladeSmith project.

C Source Code Files

The **support.c** source code file for the DataBlade module project contains:

- **#include** directives for standard C and DataBlade API header files.
- utility functions called from BladeSmith-generated routines.

The **udr.c** source code file for the DataBlade module project contains function declarations for user-defined C routines, cast support routines, and aggregates in the BladeSmith project.

Each opaque data type has a source code file, **Opaque.c**, where **Opaque** is the name of the opaque data type. The **Opaque.c** source code files contain function definitions for opaque type support routines specified in the DataBlade module project.

The **statistics.c** source code file for the DataBlade module project contains user-defined statistics support routines for each opaque data type with statistics support.

Microsoft Visual C++ Files

The **project.dsp** file is the Visual C++ project file that contains the project information for both C and C++/ActiveX code: for example, a list of the source code files. For more information about C++/ActiveX code, see Chapter 6, "Creating ActiveX Value Objects," on page 6-1.

The **project.dsw** file is the Visual C++ workspace file that contains workspace information and refers to the project file for project information. You open the **project.dsw** file to edit and compile your source code.

For more information on compiling using the **project.dsw** file, see "Compiling on Windows" on page 5-34.

Warning File

The **warning.txt** file includes the following types of warnings about your source code:

- **Unfinished code.** The file lists the routines to which you need to add code.
- **User-defined statistics.** If you included statistics-support functions for an opaque data type, a warning states that user-defined statistics are only available for Dynamic Server Version 9.2 and later.
- **Long identifiers.** If you created any objects with long identifiers, a warning states that long identifiers are only available for Dynamic Server Version 9.2 and later.
- **The mi_selfuncarg data type.** If you included the **mi_selfuncarg** data type in an extended data type, a warning states that the data type is deprecated for Dynamic Server Version 9.2 and later.

The **warning.txt** file might contain other warnings, as appropriate for your source code.

Using Generated Code

This section contains the following subsections:

Identifying the Source of Generated Code

When BladeSmith generates source code for your DataBlade module, it uses routines and data structures from various libraries.

The following table lists common prefixes for data types and routines appearing in generated DataBlade module code and lists their sources and where they are documented.

Prefix	Library	More Information
mi_	DataBlade API	Almost all DataBlade API routines and data types have the **mi_** prefix. See the *IBM Informix DataBlade API Programmer's Guide* for more information.
Gen_	BladeSmith	All variable names and routine names not explicitly named in the project have the **Gen_** prefix. See "Utility Functions Generated by BladeSmith" on page 5-13 for more information on utility functions.
DBDK_TRACE_	BladeSmith	BladeSmith uses four macros for error handling and tracing in generated code. See "Tracing and Error Handling" on page 5-7 for more information.
gl_	DataBlade API	The **gl_dprintf()** and **gl_tprintf()** functions are used for internationalized tracing. See the *IBM Informix DataBlade API Programmer's Guide* for more information.
ifx_gl_	GLS API	All GLS API routines have the **ifx_gl_** prefix. See the *IBM Informix GLS User's Guide* for more information.
ifx_	ESQL/C	In code generated by BladeSmith, this prefix indicates routines and data types from ESQL/C. See the *IBM Informix ESQL/C Programmer's Manual* for more information.

Comments in Generated Code

BladeSmith adds comments to the code it generates. Each routine begins with a prologue that describes the purpose of the routine, its arguments, and its return value. Comments throughout the code describe variable declarations and the results of generated C statements and routine calls.

In comments at the beginning and end of each generated routine, BladeSmith stores information it uses when regenerating source code. The prologue includes a routine ID. A comment at the end of the routine contains a calculated checksum.

Warning: Do not modify either of these comments; BladeSmith uses them to merge your edits into the regenerated code.

BladeSmith adds a **Gen_** prefix to all variable names and routine names you did not create or explicitly define in BladeSmith: for example, utility functions and the *Gen_Con* connection argument.

MI_FPARAM Function Argument

BladeSmith adds an extra argument to all routines it generates: a pointer to an MI_FPARAM structure. However, with the exception of iterator functions and user-defined functions that allow null arguments, the generated code does not manipulate the values stored in MI_FPARAM structures. The MI_FPARAM argument is included for your convenience. If you want to use the MI_FPARAM structure, you must add code to all noniterator routines.

Typically, you only need to use the MI_FPARAM structure for the following tasks:

- Check for `NULL` arguments or return values

- Set arguments or return values to NULL
- Get data type information about arguments or return values
- Manage iterative calls to a function

To manipulate the values in an MI_FPARAM structure, you must use its DataBlade API accessor functions. Do not access MI_FPARAM structure members directly, because the structure might change between versions of the DataBlade API.

In addition to references for each of the MI_FPARAM accessor functions, the *IBM Informix DataBlade API Programmer's Guide* includes a chapter that describes the information stored in the MI_FPARAM structure and tells you how to get values from or store values in the structure and how to use the structure for creating iterative functions.

For an explanation of how generated code uses the MI_FPARAM structure in an iterator function, see "Iterator Functions" on page 5-32. The **ExmAmortize()** function in the example Business DataBlade module uses the MI_FPARAM structure in an iterative function.

Server Connection Handle

BladeSmith calls **mi_open()** at the beginning of many of the routines it generates. The **mi_open()** call obtains a database server connection handle, which is a required argument in many DataBlade API calls. If your routine does not need a connection handle, remove the **mi_open()** and **mi_close()** functions that BladeSmith adds to your code.

Tip: If the only DataBlade API call your routine makes is to **mi_db_error_raise()**, you do not need a connection handle. You can pass a null value to **mi_db_error_raise()**.

For routines running in the database server address space, except the large object DataBlade API routines, the connection handle enables client and database server routines to use the same DataBlade API routines.

When **mi_open()** is called at the beginning of a generated routine, **mi_close()** is called to release the handle immediately before the routine returns.

Tracing and Error Handling

BladeSmith adds tracing and error handling code throughout the generated source code if the tracing option is set to **True** when you generate source code in BladeSmith.

A generated utility function, **Gen_Trace()**, processes all tracing and error handling. Your routines must not call **Gen_Trace()** directly. To perform tracing and error handling tasks, use the **DBDK_TRACE** macros defined in the generated header file.

This section describes:

- How to use the supplied **DBDK_TRACE** macros to add tracing and error handling to your generated code (see "Adding Tracing and Error Handling" on page 5-9)
- How to enable tracing and use the generated **TraceSet**_*project* procedure (see "Enabling Tracing in a DataBlade Module" on page 5-10 and "Enabling Tracing in a Database Session" on page 5-11)

- The standard Informix-supplied error messages (see "Standard Error Messages" on page 5-13)

How Tracing Works

Tracing is the process of writing status messages for routines to a file. Use tracing for debugging; tracing can generate a high volume of output, which is not appropriate for production databases.

By default, tracing is disabled whenever you start a new database session.

Each tracing message has a tracing level associated with it. When you enable tracing, you set a threshold for tracing levels. Messages with a trace level less than or equal to the threshold are written to the trace file.

Tracing messages are written to a trace file, which is created with a default name and location, or with a name and location you specify. If you remove the trace file while tracing is enabled, it is automatically re-created. The default name and location of the trace file is **tmp/session_id.trc**, where **session_id** is the four-digit identifier of the current database server session. To obtain your current session ID, use the **onstat -g ses** command.

Messages are written to the trace file only if you:

- Generate source code with tracing. See "Generating Source Files" on page 4-44 for instructions.
- Compile the DataBlade module. See "Compiling DataBlade Module Code" on page 5-33 for instructions.
- Enable tracing in your DataBlade module by adding a trace class to the **systraceclasses** system catalog and creating the **TraceSet_project()** procedure. See "Enabling Tracing in a DataBlade Module" on page 5-10 for more information.
- Enable tracing for a database session by setting a threshold and optionally specifying a trace file with the **TraceSet_project()** procedure. See "Enabling Tracing in a Database Session" on page 5-11 for instructions.

Trace messages include the name of the executing routine, the source filename, and the line number of the call to **Gen_Trace()** with the embedded parameters *%FUNCTION%*, *%FILENAME%*, and *%LINENO%*. For example, the following example is a portion of a log file resulting from calling the "enter routine" and "exit routine" macros, **DBDK_TRACE_ENTER()** and **DBDK_TRACE_EXIT()**, for the **Distance()** function:

```
========================================
Tracing session: 12 on 3/4/1998

10:55:32 Entering function Distance (Circle.c)

10:55:32 Successfully exiting Distance (Circle.c)
```

Important: If you want to include parameters other than %FUNCTION%, %FILENAME%, and %LINENO% in a message, you must call the **gl_dprintf()** function for trace messages or the **mi_db_error_raise()** function for error messages. For an example of calling these functions, see the **Gen_Trace()** function in the generated source code. See the *IBM Informix DataBlade API Programmer's Guide* for more information on using these functions.

Adding Tracing and Error Handling

To add tracing and error handling to the generated source code, edit the generated source code file and insert **DBDK_TRACE** macro calls.

The following table describes the tracing and error handling macros.

Macro	Action (if tracing is enabled)
DBDK_TRACE_MSG()	Prints a message in the trace file.
DBDK_TRACE_ERROR()	Prints a message in the trace file and raises an error by calling **mi_db_error_raise()**.
DBDK_TRACE_ENTER()	Prints a message in the trace file upon entering a routine.
DBDK_TRACE_EXIT()	Prints a message in the trace file upon exiting a routine.

The macros are described in the following sections. See "Setting a Trace Output File and a Trace Threshold" on page 5-12 for information on the name and location of the trace message file.

The DBDK_TRACE_MSG() and DBDK_TRACE_ERROR() Macros: When tracing is enabled, use the **DBDK_TRACE_MSG()** and **DBDK_TRACE_ERROR()** macros to write messages to the trace message file. The **DBDK_TRACE_ERROR()** macro also raises an error.

If tracing is not enabled, no messages are written to the trace message file; **DBDK_TRACE_ERROR()** still raises an error.

The syntax for the **DBDK_TRACE_MSG()** and **DBDK_TRACE_ERROR()** macros is as follows:

```
DBDK_TRACE_MSG(caller, mesgNo, level);
DBDK_TRACE_ERROR(caller, mesgNo, level);
```

caller The name of the C routine to which you are adding the macro.

mesgNo The number for an error or trace message in the **syserrors** or **systraceclasses** system catalog. Use numbers for messages defined in the BladeSmith project or messages BladeManager installs with all DataBlade modules.

See "Defining Errors" on page 4-13 for instructions on creating error and trace messages with BladeSmith.

The standard error messages are listed in "Standard Error Messages" on page 5-13.

level An integer that determines the trace level for the message. If *level* is less than or equal to the tracing threshold, then the message is printed in the trace file. For example, the trace level for the **DBDK_TRACE_ENTER()** and **DBDK_TRACE_EXIT()** macros is 20.

See "Setting a Trace Output File and a Trace Threshold" on page 5-12 for more information.

For example, if you have a function called **ExmAmortize()** and a trace message number **UE001** with a trace level of 20, use the following code fragment to add tracing to the **ExmAmortize()** function:

```
DBDK_TRACE_MSG("ExmAmortize", "UE001", 20);
```

The DBDK_TRACE_ENTER() and DBDK_TRACE_EXIT() Macros: If you have tracing enabled and the trace threshold set to 20 or above, the **DBDK_TRACE_ENTER()** and **DBDK_TRACE_EXIT()** macros write messages to the trace file when the called routine is entered and exited, respectively. BladeSmith automatically adds these macros to generated code for every routine.

The syntax for the **DBDK_TRACE_ENTER()** and **DBDK_TRACE_EXIT()** macros is as follows:

```
DBDK_TRACE_ENTER(caller);
DBDK_TRACE_EXIT(caller);
```

The *caller* parameter specifies the name of the C routine to which you are adding the macro.

For example, if you have a routine called **ExmAmortize(),** the following code fragment sends a message to the message file when the **ExmAmortize()** routine is entered:

```
DBDK_TRACE_ENTER("ExmAmortize");
```

Enabling Tracing in a DataBlade Module

After you generate code with tracing and compile your code, enable tracing in your DataBlade module.

To enable tracing in your DataBlade module:

1. Create a trace class.
2. Create the **TraceSet_project()** procedure.

These steps are described in the following sections.

Creating a Trace Class: Tracing classes are categories of tracing that can be activated independently, allowing you to tune your tracing output to suspected problem areas.

To enable tracing in a database, you must insert the DataBlade trace class as a record in the **systraceclasses** system catalog.

The tracing generated by BladeSmith provides a single trace class, with the same name as your DataBlade project.

This example creates a trace class for the Business DataBlade module:

```
insert into informix.systraceclasses(name)
   values('Business');
```

You can create your own tracing classes for customized tracing. See the *IBM Informix DataBlade API Programmer's Guide* for more information.

Creating the TraceSet_project Procedure: The **TraceSet_project()** procedure (where **project** is the name of your DataBlade module project) sets the tracing output file and the trace threshold for a database server session by calling the DataBlade API functions **mi_tracefile_set()** and **mi_tracelevel_set()**.

The **TraceSet_project()** procedure is included in generated source code by BladeSmith when you choose to generate code with tracing in the Generate DataBlade dialog box. Although the **TraceSet_project()** procedure is included in the

generated C code, the SQL statements to create it in the database are not included in the generated SQL scripts. This omission prevents end-users from accessing the **TraceSet_project()** procedure from SQL statements.

After you install and register your DataBlade module in the database, create the **TraceSet_project()** procedure using the following SQL statement:

```
CREATE PROCEDURE TraceSet_project(LVARCHAR, INT)
WITH(NOT VARIANT)
EXTERNAL NAME "/$INFORMIXDIR/extend/project/project.bld(TraceSet_project)"
LANGUAGE C
END PROCEDURE;
```

project is the name of your DataBlade module.

Tip: The comments for the **TraceSet_project()** procedure in your source code show the exact syntax to create the procedure for your DataBlade module.

The syntax for using the **TraceSet_project()** procedure is as follows:

```
EXECUTE PROCEDURE TraceSet_project(
          "traceFile",
          traceThreshold
);
```

traceFile The path and filename of the trace output file for the current database server session, surrounded by quotation marks. If you do not specify a filename, the default file, **/tmp/session_ID.trc**, is created. If you specify a filename and then execute **TraceSet_project()** again during the same session without specifying a filename, the filename is not changed.

See "Setting a Trace Output File and a Trace Threshold" on page 5-12 for and example.

traceThreshold The trace threshold for the current database server session. There are three possible values:

`0` Tracing is disabled.

`> 0` Tracing is enabled and the threshold is set to that number.

`< 0` The tracing threshold is not changed.

See "Setting a Trace Output File and a Trace Threshold" on page 5-12 for an example.

Enabling Tracing in a Database Session

After you enable tracing in your DataBlade module, enable tracing for the database session. By default, when you start a database server session, tracing is disabled.

To enable tracing in your Database session:

1. Set the appropriate locale environment variables.
2. Register tracing routines in the database with the **EnableTracing.sh** shell script.
3. Set the trace output file and the trace threshold for the current session.

These steps are described in the following sections.

Setting Your Locale: The system only displays and writes messages to your session that match the locale specified in the session environment variables.

Therefore, to see your trace messages, you must set the **SERVER_LOCALE** environment variable to the same locale you used when you created your messages in BladeSmith.

Tip: To determine the locale for your trace message, look at its properties in BladeSmith; click the message in the project view, under the Errors folder, and choose **Edit > Properties**. See "Error Locale" on page 4-14 for more information.

Registering Tracing Routines: To register the tracing routines for a particular DataBlade module in a database, specify the following syntax from the UNIX command line or MKS Korn Shell:

```
EnableTracing.sh database DataBlade [Project]
```

In this command, *database* is the name of the database in which you want to register the routines and *DataBlade* is the name of the DataBlade module that contains the tracing routines. The square brackets [] indicate that *Project* name is an optional argument. Specify a path for *Project* if you have moved your DataBlade module from its default directory. When you specify *Project*, specify only the part of the path that follows **%INFORMIXDIR%/extend/**.

After you have registered the tracing routines in the database, set the output trace file and tracing level as shown in "Setting a Trace Output File and a Trace Threshold" on page 5-12. The filename and trace level settings must be reset if you change DB–Access sessions or restart the server.

To unregister the tracing routines for a particular DataBlade module in a database, specify the following syntax from the UNIX command line or MKS Korn Shell:

```
DisableTracing.sh database DataBlade [Project]
```

In this command, *database* is the name of the database in which you want to register the routines and *DataBlade* is the name of the DataBlade module that contains the tracing routines. The square brackets [] indicate that *Project* name is an optional argument. Specify a path for *Project* if you have moved your DataBlade module from its default directory. When you specify *Project*, specify only the part of the path that follows **%INFORMIXDIR%/extend/**.

Important: Disable tracing in production DataBlade modules because tracing can substantially decrease performance, and output trace files can use considerable space.

Setting a Trace Output File and a Trace Threshold: To set the trace output file and the trace threshold, use the **TraceSet_project()** procedure. The **DBDK_TRACE_ENTER()** and **DBDK_TRACE_EXIT()** macros provided by BladeSmith use the trace level 20.

The following example sets the filename to **Business.trc** in the **/tmp** directory and sets the threshold to 20:

```
EXECUTE PROCEDURE TraceSet_Business("/tmp/Business.trc", 20);
```

To change the trace output file without altering the trace threshold, specify the trace threshold as **-1**. To change the trace threshold without altering the trace output file, do not put a filename between the quotation marks.

Standard Error Messages

BladeSmith uses a standard set of messages in the code that it generates. These messages are shared by all DataBlade modules created with BladeSmith and cannot be changed. BladeManager adds the error messages to the **syserrors** system catalog and the **systracemsgs** system catalog when a DataBlade module is registered in a database.

The messages have the same text and error numbers in the two system tables, except that messages in the **systracemsgs** system table include the text "(*%FILENAME%, %LINENO%*)" after the *%FUNCTION%* parameter to ensure that the source filename and line number appear in the trace file.

The following table lists the standard U.S. English DataBlade module error messages.

Error Number	Message Text
UGEN1	Connection has failed in %FUNCTION%.
UGEN2	Memory allocation has failed in %FUNCTION%.
UGEN3	Error creating large object from client file in %FUNCTION%.
UGEN4	Large object handle is invalid in %FUNCTION%.
UGEN5	Error creating large object from client file in %FUNCTION%.
UGEN6	Error saving large object to client file in %FUNCTION%.
UGEN7	Double-quoted string expected in %FUNCTION%.
UGEN8	Interval format conversion has failed in %FUNCTION%.
UGEN9	Input string is not terminated with double-quote in %FUNCTION%.
UGENA	Input string is too long in %FUNCTION%.
UGENB	Input data format error in %FUNCTION%.
UGENC	Output LO file creation has failed in %FUNCTION%.
UGEND	Entering function %FUNCTION%.
UGENE	Successfully exiting function %FUNCTION%.
UGENF	The collection could not be created in %FUNCTION%.
UGENG	Insertion into the collection has failed in %FUNCTION%.
UGENH	Invalid iterator state used in %FUNCTION%.

The generated header file defines constants **ERRORMSG1** through **ERRORMSG17** for these error numbers.

You can define additional messages used in your DataBlade module. Define them in the BladeSmith project to ensure that they are loaded into the database when your DataBlade module is registered. See "Developing Your DataBlade Module" on page 1-5 for information about reserving error codes for your DataBlade module.

Utility Functions Generated by BladeSmith

BladeSmith generates support functions that it calls from other generated code. These functions include:

- **Gen_IsMMXMachine()**. This function determines whether the database server is running on a computer with an Intel MMX processor. This function is only generated if you specify that you want MMX-enabled functions when you generate source code. See "The Gen_IsMMXMachine() Utility Function" on page 5-14 for more information.
- **Gen_LoadLOFromFile()**. When an opaque type includes a large object handle, BladeSmith includes this function to retrieve the large object data from a disk file.
- **Gen_nstrwords()**. This function counts the number of values (each separated by a blank space) in a formatted string. This function is called from input and import functions to retrieve values from variable-length opaque types.
- **Gen_sscanf()**. This function is called from input and import functions to convert text data to the C structure for an opaque type. See "The Gen_sscanf() Utility Function" on page 5-14 for more information.
- **Gen_StoreLOToFile()**. When an opaque type includes a large object handle, BladeSmith includes this function to write large object data to a disk file.
- **Gen_Trace()**. This function processes trace messages and errors. This function is generally invoked by the macros **DBDK_TRACE_ENTER()**, **DBDK_TRACE_EXIT()**, **DBDK_TRACE_MSG()**, and **DBDK_TRACE_ERROR()**. See "Tracing and Error Handling" on page 5-7 for more information.

Most of the generated utility functions are called by code that BladeSmith generates, and you typically do not use them in your code. The **Gen_sscanf()** utility function, however, can be useful in your input/output functions. You can use the **Gen_IsMMXMachine()** function if you use Intel MMX instructions in your code.

The Gen_sscanf() Utility Function

The **Gen_sscanf()** utility function scans one value from an input string and stores it at a given address. **Gen_sscanf()** returns a pointer that points just past the value it scanned from the input string.

Gen_sscanf() takes the following arguments:

Gen_Con	The database connection handle
Gen_Caller	The name of the calling function
Gen_InData	A pointer to the text to be scanned
Gen_InDataLen	An integer containing the length of the text (**mi_lvarchar** strings are not null-terminated)
Gen_Width	An integer containing the maximum data length for text data
Gen_Format	A string containing a **sscanf()** format string for the structure member to be scanned
Gen_Result	A pointer to the member in the structure where **Gen_sscanf()** stores the scanned value

The generated input and import functions call **Gen_sscanf()** once for each structure member. **Gen_sscanf()** requires an input string in the current locale and uses the IBM Informix GLS routines to scan the string.

The Gen_IsMMXMachine() Utility Function

The **Gen_IsMMXMachine()** utility function can be used when you include Intel MMX media enhancement technology in your DataBlade module. The function

tests the processor in the database server computer to determine if it has MMX technology support. If MMX technology support is found, **Gen_IsMMXMachine()** returns 1.

If the database server machine does not have MMX technology support, or if the **FORCE_NO_MMX** environment variable is set in the database server environment, **Gen_IsMMXMachine()** returns `0`. On UNIX machines, **Gen_IsMMXMachine()** always returns `0`.

To execute MMX instructions when possible and to execute portable C code on computers that do not have MMX technology support, call **Gen_IsMMXMachine()** in an IF statement.

Gen_IsMMXMachine() declares a static INT flag, **MMXType**. It first looks for the **FORCE_NO_MMX** environment variable, which must be set in the environment before the database server is started.

If **FORCE_NO_MMX** is found, the function sets **MMXType** to `0` without testing the CPU. If **FORCE_NO_MMX** is not found, the function tests the processor and sets the **MMXType** variable to `1` if MMX technology support is found or `0` if not. After the value of **MMXType** is set, **Gen_IsMMXMachine()** returns its value immediately, so that tests are performed once after the DataBlade module object file is loaded.

Editing Opaque Type Support Routines in opaque.c

BladeSmith DataBlade generates code for opaque type support routines, as described in the following subsections:

The following subsections describe the code BladeSmith generates for each routine and modifications you might need to make to the generated code.

Each of the generated support routines contains an MI_FPARAM argument. BladeSmith includes the MI_FPARAM argument in generated code for your convenience; you can add code to use it. The generated code, however, does not use the MI_FPARAM argument. See "MI_FPARAM Function Argument" on page 5-6 for more information.

Important: To avoid merging conflicts when you regenerate your code, add code only in areas marked by `TO DO:` comments or after the generated code. If you do modify code outside the designated areas, after you regenerate you might have two copies of the routine: the one you modified, and the one BladeSmith generated. Although your changes remain, you must resolve the conflicts in the two pieces of code.

Text Input and Output Functions

The text input function converts from the textual representation of an opaque data type to the internal format. The C name of the text input function for each opaque data type is **OpaqueInput()**.

The text output function converts from the internal format of an opaque data type and the textual representation. The C name of the text output function for each opaque data type is **OpaqueOutput()**.

BladeSmith generates complete C code for these functions.

The Generated Code

The generated code for the text input and output functions uses a default text representation: a string containing all members of the structure, delimited with spaces. Strings are enclosed in single quotation marks ('). Large objects are represented as external filenames enclosed in quotation marks. The input function calls the **sscanf()** C library function. Use character representations for these types that **sscanf()** recognizes.

For example, the Circle DataBlade module defines a **Pnt** data type with two **mi_double_precision** members, **x** and **y**. The **Circ.h** header file contains the following structure definition:

```
typedef struct
{
   mi_double_precision           x;
   mi_double_precision           y;
}
Pnt;
```

The default text representation for the **Pnt** data type is as follows:

```
'x y'
```

x and *y* are character strings that **sscanf()** can convert to double-precision values. For example, the following statement inserts a **Pnt** value into a table, using the default text representation:

```
insert into mytable (col1) values ('12.3 66.9');
```

Text Input Function: The text input function has two arguments: an **mi_lvarchar** pointer, which points to the text that is to be converted, and an unused MI_FPARAM pointer.

The text input function returns a pointer to a filled-in C structure for the **Pnt** data type. The contents of this structure are written to the database table. The function allocates memory for the opaque type it returns, as follows:

```
/* Allocate memory room to build the UDT in. */
Gen_RetVal = (Pnt *)mi_alloc( sizeof(Pnt));
if(Gen_RetVal == 0)
{
   /*
   ** Memory allocation has failed so issue
   ** the following message and quit.
   **
   **    "Memory allocation has failed in PntInput."
   */
   mi_db_error_raise(Gen_Con, MI_SQL, ERRORMESG2,
      "FUNCTION%s", "PntInput", (char *)NULL);
```

The database server frees the allocated memory.

Next, the function scans the text string, obtaining a value for one structure member at a time, as follows:

```
/* Get the data value for Gen_OutData->x. */
Gen_InData = Gen_sscanf(Gen_Con, "PntInput", Gen_InData,
         mi_get_varlen(Gen_param1), 0,
         "%lf %n",
         (char *)&Gen_OutData->x);

/* Get the data value for Gen_OutData->y. */
Gen_InData = Gen_sscanf(Gen_Con, "PntInput", Gen_InData,
         mi_get_varlen(Gen_param1), 0,
         "%lf %n",
         (char *)&Gen_OutData->y);
```

This code calls the **Gen_sscanf()** utility function, which BladeSmith adds to each generated C source file.

Finally, the text input function returns a pointer to the completed C structure, as follows:

```
/* Return the UDT value. */
return Gen_RetVal;
```

Text Output Function: The text output function takes a pointer to the opaque type structure and an MI_FPARAM pointer and returns a text representation of the data type in an **mi_lvarchar** value. BladeSmith operates under the assumption that the text representation is a string containing character representations of all of the structure members delimited with spaces. The function encloses strings and filenames for large objects in double quotation marks.

The generated text output function computes the maximum length of the string it returns and calls **mi_new_var()** to allocate an **mi_lvarchar** argument, **Gen_RetVal**. The database server frees the allocated memory later.

The function calls the C standard library function **sprintf()** once for each structure member to write the value and a space in the data area pointed to by the **Gen_OutData** argument. The following code shows the **sprintf()** calls from the generated **PntOutput()** function in the Circle DataBlade module:

```
/* Format the value for Gen_InData->x. */
sprintf(Gen_OutData, "%lf ", Gen_InData->x);
Gen_OutData += strlen(Gen_OutData);

/* Format the value for Gen_InData->y. */
sprintf(Gen_OutData, "%lf", Gen_InData->y);
Gen_OutData += strlen(Gen_OutData);
```

Between calls to **sprintf()**, the function resets **Gen_OutData** to point to the end of the string.

Customizing the Code

To use an input text format different from the default format, or if the C structure contains data types that BladeSmith cannot scan with **Gen_sscanf()**, you must modify the generated text input function code.

If you want an input text format that is different than the default, replace the generated code with your own. For example, you can choose to delimit values with commas instead of spaces. Your code might be able to call the **Gen_sscanf()** function, or you might need to write your own scanning function.

In the Circle DataBlade module, the text representation of the **Pnt** data type is changed from the default format, '`x y`', to a new format: '`(x, y)`'.

To support the new format, **Gen_sscanf()** calls in the **PntInput()** function are modified to include the parentheses and comma in the format string, as shown in the following code:

```
/* Get the data value for Gen_OutData->x. */
Gen_InData = Gen_sscanf(Gen_Con, "PntInput", Gen_InData,
         mi_get_varlen(Gen_param1), 0,
         "(%lf %n,",
         (char *)&Gen_OutData->x);

/* Get the data value for Gen_OutData->y. */
Gen_InData = Gen_sscanf(Gen_Con, "PntInput", Gen_InData,
         mi_get_varlen(Gen_param1), 0,
         "%lf %n)",
         (char *)&Gen_OutData->y);
```

If you change the text input function to support a text format different from the default, also change the text output function. The string returned by the text input function should be a valid string for the text output function.

For example, to support the new text representation for the **Pnt** data type, the parentheses and comma are added to the **sprintf()** calls in the **PntOutput()** function, as follows:

```
/* Format the value for Gen_InData->x. */
sprintf(Gen_OutData, "(%lf,", Gen_InData->x);
Gen_OutData += strlen(Gen_OutData);

/* Format the value for Gen_InData->y. */
sprintf(Gen_OutData, "%lf)", Gen_InData->y);
Gen_OutData += strlen(Gen_OutData);
```

Smart Large Object Considerations

Large objects are represented as external filenames enclosed in quotation marks.

Examples

The following example DataBlade modules contain text input and output functions:

- **Strings DataBlade module.** The text input and output functions for the **CompressStr** opaque data type call user-defined routines to compress and uncompress a string, use the MI_FPARAM argument, and pass a smart large object handle.
- **Circle DataBlade module.** The text input and output functions for the **Pnt** opaque data type show a modified text representation.
- **Shapes DataBlade module.** The text input and output functions for the **MyShape** opaque data type contain code for each of the three specific cases of **MyShape**: **MyBox**, **MyCircle**, and **MyPoint**. The text input and output functions for the **MyBox**, **MyCircle**, and **MyPoint** data types call the text input and output functions for **MyShape**.
- **FuzzyMatch DataBlade module.** The text input function for the **ColorType** opaque data type converts the textual name of a color to a three-integer value by looking up the value in a map file.
- **UDTExporter DataBlade module.** The text input and output functions for the **ComplexNumber** opaque data type process an integer array.
- **MMXImage DataBlade module.** The text input and output functions process the variable-length **Image** opaque data type using MMX technology.

Binary Send and Receive Functions

The binary send function converts opaque data type values to the client format before sending them to the client. The C name of the binary send function for each opaque data type is **OpaqueSend()**.

The binary receive function converts opaque data type values from the client format before sending them to the database server. The C name of the binary receive function for each opaque data type is **OpaqueReceive()**.

BladeSmith generates complete C code for these functions.

The Generated Code

The binary send and receive functions adjust data types for differing byte order and alignment requirements on the client and server computers. The functions call the appropriate **mi_put** and **mi_get** DataBlade API accessor functions to transform individual members of the opaque type structure.

The binary send function takes the opaque data type as its argument and returns a pointer to an **mi_sendrecv** type, which contains the opaque type structure, and an unused MI_FPARAM pointer. The **mi_sendrecv** type is the client form of the opaque data type. The binary receive function takes an **mi_sendrecv** type as its argument and returns the opaque data type. If the opaque data type is variable length, then the binary send function takes an **mi_bitvarying** type as its argument, and the binary receive function returns an **mi_bitvarying** type.

The binary send function calls **mi_put** DataBlade API accessor functions to store the client representation of the data type in the structure to be returned to the database server. The binary receive function calls **mi_get** DataBlade API accessor functions to retrieve values for each structure member from the input received from the client.

The following code, taken from the Circle DataBlade module **CircSend()** function, calls **mi_put_double_precision()** to store values from an input **Circ** data type (addressed by the **Gen_InData** argument) to an output **Circ** data type (addressed by the **Gen_OutData** argument):

```
/* Prepare the value for Gen_OutData->center.x. */
mi_put_double_precision((mi_unsigned_char1 *)&Gen_OutData->center.x, &Gen_InData->center.x);

/* Prepare the value for Gen_OutData->center.y. */
mi_put_double_precision((mi_unsigned_char1 *)&Gen_OutData->center.y, &Gen_InData->center.y);

/* Prepare the value for Gen_OutData->radius. */
mi_put_double_precision((mi_unsigned_char1 *)&Gen_OutData->radius, &Gen_InData->radius);
```

The corresponding code for the **CircReceive()** function is identical, except it uses **mi_get_double_precision()** instead of **mi_put_double_precision()**.

Customizing the Code

In general, you do not need to modify the generated binary send and receive functions. If you do add code to one of the functions, you must make corresponding changes to the other function. You can alter the binary send and receive functions to encrypt data in the **mi_sendrecv** type before sending data to the client and decrypt data from the **mi_sendrecv** type before receiving data into the database.

Examples

The following example DataBlade module use the binary send and receive functions without modification:

- UDTExporter DataBlade module
- Matrix DataBlade module
- Circle DataBlade module
- MMXImage DataBlade module

Text File Import and Export Functions

The text file import function transfers a flat file to the text representation of the opaque data type. The C name of the text file import function for each opaque data type is **OpaqueImportText()**.

The text file export function transfers the text representation of the opaque data type to a flat file. The C name of the text file export function for each opaque data type is **OpaqueExportText()**.

BladeSmith generates complete C code for these functions.

The Generated Code

The database server calls the text file import function with a pointer to an **mi_impexp** type containing the input text, which it retrieves from an external file, and an unused MI_FPARAM pointer. The text file import function converts the text to an instance of the opaque type and returns a pointer to it.

The text file import function allocates memory for the opaque structure that it returns and then calls **Gen_sscanf()** for each member of the structure, storing the scanned values in the allocated memory.

The database server calls the text file export function with a pointer to an instance of the opaque type and an unused MI_FPARAM pointer. The text file export function converts the opaque type value to a text value stored in an **mi_lvarchar** variable that it allocates. The generated code works the same way as the text output function.

The text file export function computes the maximum length of the text value it can return by adding the maximum lengths for each structure member. Then it calls **mi_new_var()** to allocate an **mi_impexp** argument large enough to hold the largest possible text value.

To create the output text, the function calls **sprintf()** once for each member of the opaque type structure, concatenating a text representation of the value to the string in the **mi_impexp** variable. Each value is followed by a space.

The default text file export function uses the same text representation as the input and import functions. This format allows database users to enter values for the opaque type and enables opaque types to be displayed. For bulk copy operations, however, a user-readable format is not necessary.

Customizing the Code

The default format for imported text is the same as for the text input function: a string containing each structure member, delimited with spaces. If you use a different text representation for your data type, you can modify the format strings in the **Gen_sscanf()** calls.

To conserve space in the external file or to match the representation required by some another application that uses the export file, you can use a different text representation for bulk copy. When you modify the text representation that the text file export function uses for copy-out operations, make corresponding modifications in the text file import function.

Smart Large Object Considerations

If the data type contains the smart large object handle data type **MI_LO_HANDLE**, the input contains the large object filename in double quotation marks. The text file import function calls **Gen_LoadLOFromFile()** to retrieve the smart large object data from the external file. The text file export function calls **Gen_StoreLOToFile()** to save the smart large object in an external file.

Binary File Import and Export Functions

The binary file import function transfers the binary representation of the opaque data type to a flat file. The C name of the binary file import function for each opaque data type is **OpaqueImportBinary()**.

The binary file export function transfers the binary representation of the opaque data type from a flat file. The C name of the binary file export function for each opaque data type is **OpaqueExportBinary()**.

BladeSmith generates complete C code for these functions.

The Generated Code

The Informix database server calls the binary file import function with an **mi_impexpbin** pointer containing the binary representation of an opaque type value, read from an external file. The function also receives an unused MI_FPARAM pointer.

The binary file import function translates the binary data in the **mi_impexpbin** structure into an instance of the opaque type and returns a pointer to the C structure containing the opaque data type. The BladeSmith-generated code allocates memory for the return structure and then calls DataBlade API **mi_get** functions to retrieve a value for each member of the structure.

The Informix database server calls the binary file export function with a pointer to a C structure, which contains an instance of the opaque type, and an unused MI_FPARAM pointer. The function translates the opaque type value into a binary image and returns it in an **mi_impexpbin** structure. The Informix database server writes the returned binary value into external files.

The binary file export function calls **mi_new_var()** to allocate an **mi_bitvarying** variable and then calls an **mi_put** function for each element of the opaque structure to store the value.

Customizing the Code

You should not modify the generated code for the binary file import and export functions.

Smart Large Object Considerations

If the opaque type includes large objects, the binary file import function calls **Gen_LoadLOFromFile()** to read the large object data from a file. The binary file export function calls **Gen_StoreLOToFile()** to save the smart large object in an external file.

The Assign and Destroy Routines

The **Assign()** function performs tasks before saving an opaque data type to disk. The C name for each opaque data type is **OpaqueAssign()**.

The **Destroy()** procedure performs tasks before saving an opaque data type to disk. The C name for each opaque data type is **OpaqueDestroy()**.

BladeSmith generates complete C code for **Assign()** and **Destroy()** routines that manage large object reference counts. For other types of special processing, you must add code to the generated code.

Important: If you use text input and output functions for smart large objects, you must define **Assign()** and **Destroy()** routines to prevent runtime errors from the database server.

The Generated Code

The Informix database server calls the **Assign()** function with a pointer to an instance of an opaque type and a pointer to an MI_FPARAM structure that is not used by the generated code. The function returns a pointer to the opaque type to the database server. Usually, the pointer returned by the **Assign()** function is the same one the Informix database server passed to it. If the **Assign()** function alters the input opaque type in any way and returns a pointer to it, the database server stores the modified value in the database.

The Informix database server calls the **Destroy()** procedure before removing an opaque type from the database. It passes the **Destroy()** procedure a pointer to the opaque type value that is about to be removed from the database and an unused MI_FPARAM pointer. The procedure returns no value.

Customizing the Code

If your opaque data type does not contain a smart large object, you must add code to perform the required task.

Smart Large Object Considerations

If your opaque data type contains a smart large object, then the generated **Assign()** function manages smart large object reference counts. It calls **mi_lo_validate()** to determine if a valid large object exists and **mi_lo_increfcount()** to increment the reference count for the large object.

The **Destroy()** procedure calls **mi_lo_validate()** for each large object. For valid large objects, it calls **mi_lo_decrefcount()** to decrement the reference count for the large object.

Examples

The following example DataBlade modules use **Assign()** and **Destroy()** routines for smart large object processing:

- Strings DataBlade module
- MMXImage DataBlade module

The MultiRep DataBlade module uses the **Assign()** function to determine whether to put the multi-representational opaque type in the database table or in a smart large object, depending on its size. The **Destroy()** procedure removes the reference counts to smart large objects, but has not effect on in-row data.

The Creating Distinct Types and Casts exercise in the tutorial uses the **Assign()** function for the **FTemp** (representing Fahrenheit degrees) and **CTemp** (representing Celsius degrees) distinct data types to prevent a user from entering a temperature value below absolute zero.

LOhandles() Function

The Informix database server calls the **LOhandles()** function to retrieve the smart large object handles or list of smart large object handles used by an opaque type. The **LOhandles()** function receives a pointer to the opaque type and a pointer to an MI_FPARAM structure.

The **LOhandles()** function returns a pointer to an **mi_bitvarying** variable containing an **MI_LO_HANDLES** structure. BladeSmith defines **MI_LO_HANDLES** in the generated header file; it is not part of the DataBlade API. The structure holds a list of **MI_LO_HANDLE** structures. It has the following definition:

```
/* This data structure returned by LOhandles. */
typedef struct
{
   mi_integer    nlos;   /* Number of large object handles. */
   MI_LO_HANDLE  los[1]; /* Valid large object handles.     */
} MI_LO_HANDLES;
```

The **LOhandles()** function calls the **mi_lo_validate()** function for each large object in the opaque type structure, accumulating a count of valid large objects. If there are no valid large objects in the opaque type, the **LOhandles()** function returns `0` to its caller.

If the opaque type contains valid large objects, the **LOhandles()** function performs the following tasks:

1. Allocates an **mi_bitvarying** variable to hold the **MI_LO_HANDLES** structure
2. Copies valid large object handles into the **los** array in the **MI_LO_HANDLES** structure
3. Sets the **MI_LO_HANDLES nlos** member to the number of large objects in the array
4. Returns a pointer to the **MI_LO_HANDLES** structure

Comparison Functions

The Informix database server calls DataBlade module comparison functions to compare two opaque type values. For example, database users can compare opaque values in SQL statements. If you want to support B-tree or R-tree indexes on an opaque type, you must provide an additional set of comparison functions.

Compare Function

The **Compare()** function compares two opaque types and returns an integer indicating the result of the comparison. The C name for each opaque data type is **OpaqueCompare()**.

BladeSmith generates the complete C code for this function.

The Generated Code: The generated code compares the members of an opaque type C structure to the members in the other opaque type, in the order defined in the structure. This algorithm might not be the appropriate way to compare your data; if it is not, you must customize the code as described in the next section.

The **Compare()** function returns:

- `-1` if the values of two corresponding members are not equal and the value of the first is less than the value of the second.
- `0` if the value of all members of the two opaque data types are equivalent.
- `+1` if the values of two corresponding members are not equal and the value of the first is greater than the value of the second.

For example, the Circle DataBlade module defines a **Pnt** data type as follows:

```
typedef struct
{
   mi_double_precision           x;
   mi_double_precision           y;
}
Pnt;
```

The **Compare()** function generated for the **Pnt** type first compares the two values of **x**. If the two values of **x** are not equal, **Compare()** stops and returns the result of the comparison. If the two values of **x** are equal, **Compare()** proceeds to compare the two values of **y**.

For data types such as **mi_boolean**, which cannot be compared for relative magnitude, the **Compare()** function returns +1 if the values differ. If all structure members are equal, it returns `0`.

Customizing the Code: The algorithm used to generate the **Compare()** function cannot evaluate the semantic content of an opaque type. Therefore, for many opaque types, replace the generated code with more appropriate code.

For example, the **Circ.h** file of the Circle DataBlade module defines the **Circ** data type as follows:

```
typedef struct
{
       Pnt                 center;
       mi_double_precision radius;
}
Circ;
```

The **Pnt** member has two **mi_double_precision** members: **x** and **y**. The generated code for **Compare()** compares the three **mi_double_precision** values individually: first **x**, then **y**, and then **radius**. However, if the size of your circles is more important than their origins, you could remove the code that compares the **x** and **y** members to base the comparison on the length of the radius only.

If you want to use B-tree indexing, the **Compare()** function is the B-tree support function. Therefore, you should analyze how you want to index your opaque data types when modifying **Compare()**.

Smart Large Object Considerations: The generated **Compare()** function does not compare the values of the smart large objects; it compares the smart large object handles. If the smart large object handles are the same, then both handles refer to the same object. You can customize the code to compare the actual values of the smart large objects.

Examples: The following example DataBlade modules implement the **Compare()** function to compare opaque data types member by member:

- Matrix DataBlade module

- Circle DataBlade module

The Shapes DataBlade module uses **Compare()** to perform a bitwise comparison on its **Circle** and **Box** data types.

B-Tree Comparison Functions

The Informix database server calls the following comparison operators when constructing B-tree indexes for opaque data types:

- **Equal()** (=)
- **LessThan()** (<)
- **LessThanOrEqual()** (<=)
- **GreaterThan()** (>)
- **GreaterThanOrEqual()** (>=)
- **NotEqual()** (!= and <>)

The C names of each of these functions are prefixed by the name of the opaque data type for which they are defined.

Important: The **Equal()** function is required if you use Visual Basic to develop a DataBlade module.

BladeSmith generates the complete C code for these functions.

The Generated Code: For these functions, BladeSmith generates code that calls the **Compare()** function described in "Compare Function" on page 5-23. For example, the **Matrix2dEqual()** function generated for the **Matrix2d** type calls the **Matrix2dCompare()** function, as follows:

```
/* Call Compare to perform the comparison. */

return (mi_boolean)(0 != Matrix2dCompare(Gen_param1, Gen_param2, Gen_fparam));
```

Customizing the Code: You should not modify these functions. You can, however, modify the **Compare()** function that these functions call.

Smart Large Object Considerations: Because these functions call the **Compare()** function, they only evaluate the smart large object handles. You must customize the **Compare()** function to evaluate the actual contents of smart large objects.

Examples: The following example DataBlade modules contain some of the B-tree comparison functions:

- Matrix DataBlade module
- Circle DataBlade module
- Shapes DataBlade module

R-Tree Comparison Functions

Version 4.0 of BladeSmith does not generate code for R-tree comparison functions.

Refer to the *IBM Informix R-Tree Index User's Guide* or the IBM Informix Online Documentation site for information about creating DataBlade modules that use the R-tree secondary access method.

Mathematical Functions

If you choose to generate mathematical functions, BladeSmith generates the following mathematical functions that take two opaque data type arguments, return an opaque data type, and are bound to operators:

- **Plus()** (+)
- **Minus()** (-)
- **Times()** (*)
- **Divide()** (/)

BladeSmith generates the following mathematical functions that take one opaque data type argument, return an opaque data type, and are bound to operators:

- **Positive()** (+)
- **Negate()** (-)

The C name of each of these functions is prefixed by the name of the opaque data type for which they are defined.

BladeSmith generates only template code for these mathematical functions.

The Generated Code

These functions have an unused MI_FPARAM argument and one or two opaque data type arguments, and they return a pointer to the resulting opaque type structure.

In the generated code, the return value is set to `0`. You must add code to perform the required operation.

Completing the Code

To complete the code for these mathematical functions, you must:

- Add your declarations, if necessary.
- Remove the call to **mi_db_error_raise()**, which raises an error stating that the routine is not implemented.
- Compute the return value and store it in the **Gen_RetVal** argument.

The database server frees the allocated memory when it has finished processing the result.

See *IBM Informix User-Defined Routines and Data Types Developer's Guide* for more information on these functions.

Example

The Matrix DataBlade module contains some of the mathematical functions.

Concat() Function

The **Concat()** function concatenates the values of its two opaque data type arguments and returns the result. It is bound to the `||` operator.

The C name of this function is prefaced by the name of the opaque data type: **OpaqueConcat()**.

BladeSmith generates only a template for this function. You must add code to perform the required operation.

Hash() Function

The database server uses the **Hash()** function when evaluating the **Equal()** function for two opaque data types that do not have identical bit representations.

The C name of this function is prefaced by the name of the opaque data type: **OpaqueHash()**.

BladeSmith generates only a template for this function. You must add code to perform the necessary operation.

Editing Statistics Routines in statistics.c

If you selected statistics support routines when creating an opaque data type (see "Statistics Support" on page 4-34), BladeSmith generates code for the following statistics support routines for your opaque data type in the **statistics.c** source code file:

- "The Statistics Collection Function," next
- "The Statistics Print Function" on page 5-28
- "The Statistics Minimum, Maximum, and Distribution Functions" on page 5-28

To implement user-defined statistics, you must also create selectivity routines to calculate the selectivity of routines that compare your opaque data type. Selectivity routines call the statistics support functions. For a description of user-defined statistics, see "User-Defined Statistics" on page 2-9.

Important: To avoid merging conflicts when you regenerate your code, add code only in areas marked by `TO DO:` comments or after the generated code. If you do modify code outside the designated areas, after you regenerate you might have two copies of the routine: the one you modified and the one BladeSmith generated. Although your changes remain, you must resolve the conflicts in the two pieces of code.

The Statistics Collection Function

The **OpaqueStatCollect()** function collects statistics (minimum value, maximum value, and distribution information) for the **Opaque** opaque data type when a user executes the UPDATE STATISTICS statement in medium or high mode. For more information on the UPDATE STATISTICS statement, see the *IBM Informix Guide to SQL: Syntax*.

The Generated Code

The **OpaqueStatCollect()** function calls the **Opaque_SetMinValue()**, **Opaque_SetMaxValue()**, and **Opaque_SetHistogram()** functions.

BladeSmith generates complete code for the **OpaqueStatCollect()** function. The **OpaqueStatCollect()** function is an iterator that processes each row in a table. It compiles statistical information by calling the **Opaque_SetMinValue()**, **Opaque_SetMaxValue()**, and **Opaque_SetHistogram()** functions. The **OpaqueStatCollect()** function stores statistical information in a multirepresentational type.

Customizing the Code

You must understand your data and how users will query it to create meaningful statistics. BladeSmith generates statistics code under the assumption that the minimum, maximum, and distribution of values are appropriate for your opaque

data type; however, they might not be. In that case, you must rewrite the **OpaqueStatCollect()** function to call other functions that you provide.

The Statistics Print Function

The **OpaqueStatPrint()** function prints formatted statistics for the **Opaque** opaque data type. You can view statistics information with the **dbschema** utility.

BladeSmith generates complete code for the **OpaqueStatPrint()** function; however, you can alter it to customize the information it prints.

The Statistics Minimum, Maximum, and Distribution Functions

The **Opaque_SetMinValue()** function computes the minimum value of the **Opaque** opaque data type.

The **Opaque_SetMaxValue()** function computes the maximum value of the **Opaque** opaque data type.

The **Opaque_SetHistogram()** function computes the distribution of the values of the **Opaque** opaque data type.

The Generated Code

BladeSmith generates only function stubs for the **Opaque_SetMinValue()**, **Opaque_SetMaxValue()**, and **Opaque_SetHistogram()** functions.

Completing the Code

To complete the code for the statistics support functions, you must add code to compute the minimum, maximum, and distribution of the values of your opaque data type.

When the database server computes the minimum, maximum, and distribution of values for built-in data types, it uses the standard ASCII sequence to order the values. However, opaque data types can be much more complicated. For example, suppose you have a box opaque data type that contains values for its height and its width. You might decide that area of the box is an appropriate measure for determining the minimum and maximum values. The distribution, however, can be more complicated. If the area of the boxes varies, then area might be the appropriate way to assign boxes to bins. If the areas of your boxes are all very similar, but the heights and widths vary, then height or width might be the best distribution criteria.

Example

The Box DataBlade module has an example of statistics support routines.

Editing Routines in udr.c

BladeSmith DataBlade generates code for the following types of routines in the **udr.c** source code file:

- "Most User-Defined Routines" on page 5-29, next
- "Cast Support Functions" on page 5-29
- "Aggregate Functions" on page 5-30
- "Selectivity Functions" on page 5-31
- "Iterator Functions" on page 5-32

Important: Avoid code merge problems by modifying code only in the sections marked with a `TO DO:` note. If you do modify code outside the designated areas, after you regenerate you may have two copies of the routine: the one you modified and the one BladeSmith generated. Although your changes remain, you must resolve the conflicts in the two pieces of code.

Most User-Defined Routines

BladeSmith generates only minimal code for most routines you create with the Routine wizard.

The Generated Code

BladeSmith generates only templates for most user-defined routines.

The generated routine declares the routine, its return type, and arguments. In addition to the arguments you specified when creating the routine, these functions also have an MI_FPARAM argument. Only generated routines that allow null values have code that uses the MI_FPARAM argument. The generated code in these routines uses MI_FPARAM to set the return value of the routine to `NULL`.

Completing the Code

To complete the code for most user-defined routines, you must:

- Add your declarations, if necessary.
- Remove the call to **mi_db_error_raise()**, which raises an error stating that the routine is not implemented.
- Compute the return value and store it in the **Gen_RetVal** argument.
- Remove the call to **mi_fp_setreturnisnull()** that sets the return value of your routine to `NULL`, if necessary.

For more information on programming routines, see the *IBM Informix DataBlade API Programmer's Guide*.

Examples

The following example DataBlade modules have user-defined routines:

- **Business DataBlade module.** Provides mathematical functions for calculating loans.
- **Circle DataBlade module.** Provides distance and containment functions that operate on opaque data types.
- **FuzzyMatch DataBlade module.** Provides functions for handling row data types and comparing opaque data types.
- **Parts Explosion DataBlade module and DataBladeAPI DataBlade module.** Provide functions for handling and returning collection data types.
- **Strings DataBlade module.** Provides character-string manipulation functions.

The Mercury DataBlade module exercise in the tutorial provides examples of cast support functions.

Cast Support Functions

If you specified a cast support function when you created a cast, BladeSmith generates the cast support function in the **udr.c** file.

The Generated Code

BladeSmith generates only templates for cast support functions.

The generated function declares the routine, its return type, and arguments. In addition to the arguments you specified when creating the function, these functions also have an MI_FPARAM argument, which is not used by the generated code.

Completing the Code

To complete the code for cast support functions, you must:

- Add your declarations, if necessary.
- Remove the call to **mi_db_error_raise()**, which raises an error stating that the routine is not implemented.
- Convert one data type to the other.
- Store the return value in the **Gen_RetVal** argument.

In a cast support function, you might convert from one binary representation to another, if the data types involved in the cast have differing binary representations. Alternatively, you might perform a calculation to convert one data type to another.

Example

The Creating Distinct Types and Casts exercise in the tutorial uses cast support functions.

Aggregate Functions

If you created a user-defined aggregate with the Aggregate wizard, BladeSmith generates aggregate functions in the **udr.c** source code file.

The Generated Code

BladeSmith generates only templates for aggregate functions.

The generated function declares the function, its return type, and arguments. In addition to the arguments you specified when creating the function, these functions also have an MI_FPARAM argument. Only generated functions that allow null values have code that uses the MI_FPARAM argument. The generated code in these functions use the MI_FPARAM to set the return value of the function to `NULL`.

Completing the Code

To complete the code for aggregate functions, you must:

- Add your declarations, if necessary.
- Remove the call to **mi_db_error_raise()**, which raises an error stating that the routine is not implemented.
- Compute the return value and store it in the **Gen_RetVal** argument.
- Remove the call to **mi_fp_setreturnisnull()** that sets the return value of your routine to `NULL`, if necessary.

For more information on programming aggregate functions, see the *IBM Informix DataBlade API Programmer's Guide*.

The Initialization Function: If you selected an initialization function, **AggregateInit()**, you must add code to it to initialize the state type required by the aggregate computation. You can set up smart large objects or temporary files for storing intermediate results as the state type. The **AggregateInit()** function returns the state type.

The first argument of the **AggregateInit()** function is a dummy argument whose value is always NULL. The second argument is an optional initialization parameter to customize aggregate computation. The initialization parameter cannot be a lone host variable reference.

The Iteration Function: You must add code to the iteration function, **AggregateIter()**, to perform the aggregate computations.

The **AggregateIter()** function should not maintain additional states in its MI_FPARAM argument because the MI_FPARAM argument is not shared among the aggregate functions. However, you can use the MI_FPARAM argument to hold information that does not affect the aggregate result.

Tip: Although the iteration function is called by the database server multiple times to calculate the aggregation, it is not implemented as an iterator function that returns a set of results.

The Combine Function: If you selected a combine function, **AggregateComb()**, you must add code to it to merge one partial result with another and return the updated state type.

The Final Function: If you selected a final function, **AggregateFinl()**, you must add code to convert the state type to the result type.

You can also add code to the **AggregateFinl()** function to release resources acquired by the initialization function. However, the **AggregateFinl()** function must not free the state type.

Selectivity Functions

If you create a user-defined function and mark it as a selectivity function for another function (see "Selectivity Functions" on page 4-22), BladeSmith generates the selectivity function in the **udr.c** source code file.

For a description of selectivity and user-defined statistics, see "User-Defined Statistics" on page 2-9.

The Generated Code

BladeSmith generates only templates for selectivity functions.

The generated code declares the function, its return type, and arguments. In addition to the arguments you specified when you created the function, BladeSmith also generates an MI_FPARAM argument, which is not used by the generated code.

Completing the Code

You must add code to the selectivity function to call the statistics support functions and calculate the selectivity of the associated function for a given set of arguments. For built-in data types, call the built-in statistics functions, such as **StatCollect()**. For opaque data types, call the statistics support functions in the **statistics.c** file, such as **OpaqueStatCollect()**. For more information on statistics support functions for opaque data types, see "Editing Statistics Routines in statistics.c" on page 5-27.

For example, if you have a selectivity function on an **OpaqueEqual()** function that is overloaded for an opaque data types, the code for the selectivity function, **OpaqueEqualSelectivity()**, might perform the following tasks:

- Determine if either of the arguments has a null value. If so, the selectivity of the **OpaqueEqual()** function is 0.
- Determine if either of the arguments is greater than the maximum or minimum value of your opaque data type. If so, the selectivity of the **OpaqueEqual()** function is 0.
- Determine where in the distribution one of the arguments falls. Because you know how many values are in each bin, from the location in the distribution you can estimate how many values are less than the argument. The selectivity of the **OpaqueEqual()** function is then the number of values less than the argument divided by the total number of values.

For more information on coding selectivity functions, see the *IBM Informix DataBlade API Programmer's Guide*.

Example

The Box DataBlade module has selectivity functions.

Iterator Functions

If you create an iterator function that returns a set one row at a time, BladeSmith adds code to process the set. The Informix database server calls iterator functions repeatedly to process all of the return values.

The Generated Code

In addition to the arguments you specified when creating it, an iterator function contains an MI_FPARAM argument. The Informix database server uses an MI_FPARAM structure to control iteration over the set. The generated code includes a C **switch** statement with different cases to process the set. The **switch** statement uses the **mi_fp_request()** function to obtain the request flag from the MI_FPARAM structure. The Informix database server sets this flag to one of the following values before calling the function:

- `SET_INIT`. The initial call to the iterator function. The iterator function allocates and initializes memory for state information. The memory allocated must use the **mi_alloc(***size***,** PER_COMMAND**)** function to be available in subsequent calls.
- `SET_RETONE`. The iterator function is called with this request flag once for each value in the set.

  For each value in the set, the function places the address of the next value in the set in the **Gen_RetVal** argument and returns **Gen_RetVal**.

  When there are no more values to return, the iterator function must call the **mi_fp_setisdone()** function to signal the Informix database server that all of the set values have been returned, as follows:

  ```
  mi_fp_setisdone(Gen_fparam, MI_TRUE);
  ```

  On this call, the iterator function returns a null pointer.
- `SET_END`. The request flag the Informix database server sets after all values in the set have been returned. The iterator function frees allocated memory and releases any other resources it has obtained.

In the generated code, each of these sections has a `TO DO:` note. To avoid code merging problems, make changes only where indicated.

Completing the Code

To complete the iterator code, you must:

- Add information declarations.
- Initialize the iterator function.

- Allocate private state information.
- Compute the value of the iteration.
- Call **mi_fp_setisdone()** when the iteration is complete.
- Free private resources.

For more information on programming iterator functions, see the *IBM Informix DataBlade API Programmer's Guide*.

Example

The **LoanAmortization()** function in the Business DataBlade module is an iterator function.

Compiling DataBlade Module Code

This section describes how to compile DataBlade module code.

BladeSmith generates makefiles for UNIX and Visual C++ project files for Windows. When you compile the generated C source code, you produce a shared object file or dynamic link library, called **project.bld**, in the source code directory **src\OS-platform**, where **OS-platform** is the name of the operating system and platform on which you are compiling. For example, **src\WinNT-i386** holds the shared object file compiled on a Windows NT computer.

Important: When you generate code in BladeSmith, set the **Format** property of the **DataBlade** folder to the correct file format for your operating system (**UNIX** or **DOS**). The default is **DOS**. See "Generating Source Files" on page 4-44 for more information.

Compiling with Tracing Support

By default, DataBlade modules are compiled without tracing support.

To compile with tracing, you must have generated source code in BladeSmith with tracing. BladeSmith adds the **DBDK_TRACE** macros and the **TraceSet_project** procedure to your code if the **Tracing** property of the **Source** folder is set to **True** in BladeSmith's Generate DataBlade dialog box. See "Generating Source Files" on page 4-44 for more information.

After you compile with tracing, you must enable tracing for the DataBlade module after you register it in a database. See "Enabling Tracing in a DataBlade Module" on page 5-10 for instructions. You must also enable tracing in the database session. See "Enabling Tracing in a Database Session" on page 5-11 for instructions.

Compiling on UNIX

On UNIX platforms, you use the generic **ProjectU.mak** makefile. This makefile includes platform-specific makefiles as files named **makeinc.platform**. To specify the UNIX platform, set the **TARGET** environment variable to the path and filename of the include file for your platform. Platform-specific files are located in the directory $**INFORMIXDIR/incl/dbdk**.

The makefile requires the **INFORMIXDIR** environment variable to be set to the Informix database server installation directory. The **BINDIR** variable in the makefile determines where the shared object file or dynamic link library is written.

BladeSmith creates **server**, **all**, and **clean** targets in the makefile. The **server** target builds the shared object file. The **clean** target deletes the shared object file or dynamic link library. The default **all** target is equivalent to the **server** target.

Important: Generate code in BladeSmith with the **Format** property set to **UNIX**. If you generate code for a UNIX DataBlade module with the DOS file format, you must convert the files to UNIX format before compiling.

To compile and link your DataBlade module shared object file:

1. Copy the generated **src/c** directory and all of its contents to your UNIX machine.
2. To compile and link shared objects on a Sun Solaris 2.5 computer using the SPARC compiler, execute the following command at the C shell:

```
setenv TARGET $INFORMIXDIR/incl/dbdk/makeinc.solaris
make -f ProjectU.mak
```

The **project.bld** file is created in the **src/solaris-sparc** directory.

Important: For compiling information specific to your operating system, see your machine notes.

Unresolved Symbols (IDS 9.14)

When you link on UNIX, the system displays a list of unresolved symbols. This list can contain these types of unresolved symbols:

- Symbols that are later resolved by the database server when it loads the DataBlade module shared object. This is expected behavior.
- Symbols that are misspelled; you must fix these. Check the list carefully for misspellings, including incorrect case.
- Symbols that are not yet coded.
- Symbols that are not found in a private library.

Compiling with Debug Support

To debug your DataBlade module while it executes in a database server process, you must build the shared object file with debugging symbols. You can either modify the makefile and add the required compiler flags to the **CFLAGS** variable, or set the **COPTS** variable on the **make** command line.

On Solaris, the following commands build shared object files with debugging symbols from the C shell:

```
setenv TARGET $INFORMIXDIR/incl/dbdk/makeinc.solaris
make -f ProjectU.mak COPTS="-g -xs"
```

Compiling on Windows

On Windows, you use the **project.dsw** file generated by BladeSmith to build your DataBlade module with Visual C++ 6.0.

The compiled DataBlade module links to **sapi.lib**. This library resolves the **mi_** and **ifx_** symbols that the database server uses internally.

To compile and link a dynamic link library using Visual C++:

1. Open the **project.dsw** in Visual C++.
2. Choose **Build > Set Active Configuration**.

3. Select a version of the project in the Set Active Project Configuration dialog box:
   - **Release.** This version is suitable for release and does not contain debugging support.
   - **Debug.** This version contains support for debugging.
4. Click **OK**.
5. Choose **Build > Rebuild All** to compile.

Visual C++ creates both a **WinNT-i386** and a **Debug** directory under the **src\c** directory to hold the release version and the debug version, respectively, of the dynamic link library.

Important: Do not link the client DataBlade API library in **%INFORMIXDIR%\lib\dmi** into the DataBlade module; that library resolves client services instead of database server services.

Visual C++ also performs the following tasks on the computer on which the Informix database server resides:

1. Creates a **project.0** directory under the directory where your database server is installed (**%INFORMIXDIR%\extend**)
2. Copies the **project.bld** file and the SQL scripts to that directory
3. Marks the **project.bld** file as read-only

See Chapter 10, "Debugging and Testing DataBlade Modules on Windows," on page 10-1, for instructions on using the DBDK Visual C++ Add-In to edit, compile, and debug a DataBlade module on Windows.

Chapter 6. Creating ActiveX Value Objects

In This Chapter

This chapter describes how to use the Informix DataBlade Developers Kit to create ActiveX value objects.

This chapter discusses using C++ to implement opaque type support routines. These routines provide the underlying logic for the custom methods of the ActiveX value objects you create with BladeSmith. You cannot use C++ to implement any other DataBlade module objects.

Prerequisite Tasks

Before you edit and compile your DataBlade module code, complete these tasks:

1. Write functional and design specifications that comply with Informix coding standards.

   See Chapter 3, “Programming Guidelines,” on page 3-1, for more information.
2. Create your DataBlade module in BladeSmith.

   See “Creating DataBlade Module Objects” on page 4-8 for instructions.
3. Generate source code and SQL files in BladeSmith.

   See “Generating Files” on page 4-40 for instructions.

ActiveX Programming Task Overview

After you generate code with BladeSmith, complete these general tasks to finish your DataBlade module code:

1. Add code to these source code files to enable your routines to function as you intend:
   - **OpaqueCommon.cpp.** Contains the logic for the opaque type routines that are implemented both as ActiveX custom methods and server project routines.

© Copyright IBM Corp. 1996, 2008

- **OpaqueCommon.h.** Contains the logic for the **IsNull()** and **SetNullFlag()** custom methods.
- **OpaqueServer.cpp.** Contains the logic for the opaque type routines that are implemented only for the server project.

See "Implementing ActiveX Value Objects" on page 6-2 for instructions.

2. Compile your source code files using the generated makefiles. See "Compiling Client and Server Projects" on page 6-5 for instructions.

To avoid merging conflicts when you regenerate your code, add code only in areas marked by `Developer:` comments or after the generated code. If you do modify code outside the designated areas, you might have two copies of the routine after you regenerate: the one you modified and the one BladeSmith generated. Although your changes remain, you must resolve conflicts in the two pieces of code.

Important: In addition to adding logic to the opaque support routines, you can add your own functions to the C++ classes in the **OpaqueCommon**, **OpaqueClient**, and **OpaqueServer .cpp** and **.h** files. Do not modify any of the other generated source files.

Source Files Generated by BladeSmith

This section provides an overview of the code that BladeSmith generates for client and server projects. For a complete list of generated files, see Appendix A, "Source Files Generated for DataBlade Modules," on page A-1.

BladeSmith generates the following source code for each client and server project:

- Makefiles, project files, header files, definitions files, and so on, for both the client project and the server project
- A support library for use by the client and server projects

BladeSmith generates the following source code for each ActiveX value object:

- Interfaces to the object for use by the client application developer
- C++ common code that provides the internal logic for both the client and server projects
- Server project opaque type support routines not available as ActiveX custom methods and other code for use by the Informix database server

Implementing ActiveX Value Objects

To implement the client and server projects of an ActiveX value object, you add project-specific logic to particular C++ source files generated by BladeSmith.

This section contains the following subsections:

The Generated Code

The contents of the generated C++ source code differ from generated C source code (described in Chapter 5, "Programming DataBlade Module Routines in C," on page 5-1) in the following ways:

- **Comments.** BladeSmith includes comments to the developer regarding which sections must or may be modified; for more information, see "Adding Project-Specific Logic to the Source Code" on page 6-3.
- **MI_FPARAM argument.** This is not included in C++ code; it is a C language argument.
- **Server connection handle.** This handle is not needed for C++ code.
- **Tracing.** BladeSmith does not insert tracing logic into the generated C++ code. However, you can use the DataBlade API tracing macros in your server code; see "Tracing and Error Handling" on page 5-7 for instructions.
- **Error handling.** BladeSmith inserts the **DkErrorRaise()** method into the generated routines to which you must add project-specific logic, naming the routine that has not been implemented and the file in which it resides (see Adding Project-Specific Logic to the Source Code). You can add **DkErrorRaise()** to other areas of the generated code and to your project-specific logic as appropriate. For information on the **DkErrorRaise()** object method, see "Internal Object Methods" on page 6-7.
- **Utility functions.** BladeSmith generates a C++ support library for each client and server project and uses the routines and methods of the library in its generated code. For information on this support library, see "C++ Support Library" on page 6-7.

Adding Project-Specific Logic to the Source Code

For each routine in each support routine category that you specify for a particular opaque type, BladeSmith generates one of the following functions:

- The function definition and a function body that contains only a call to **DkErrorRaise()**. These routines are indicated by the comment `Developer: TO DO`.

  You must supply project-specific internal logic to these routines.

  BladeSmith inserts the **DkErrorRaise()** method into these routines, naming the routine that has not been implemented and the file in which it resides. When you supply the logic to these routines, you can remove the call to **DkErrorRaise()** or modify it to return an error more appropriate to the added logic. (For information on the **DkErrorRaise()** object method, see "Internal Object Methods" on page 6-7.)
- **The function definition and a default function body.** These routines are indicated by the comment `Developer: Make changes in this section if necessary`.

  You can keep the default logic, or you can replace or modify it as appropriate for your project.

Files to Edit

The following table lists each opaque type routine (by category and name), the source file where it is defined, and whether adding project-specific logic to that routine is required or optional.

For information on the usual behavior of the ActiveX custom methods (those defined in the **OpaqueCommon.cpp** and **OpaqueCommon.h** files), see Chapter 7,

"Using ActiveX Value Objects," on page 7-1. For information on the default behavior of the server project routines, see "Editing Opaque Type Support Routines in opaque.c" on page 5-15.

Routine Category	Opaque Type Routine/ ActiveX Custom Method	Source File	Add Logic? (Optional/ Required)
Basic Text Input/Output	**FromString()**	**OpaqueCommon.cpp**	Optional
	ToString()		Optional
Binary Send/Receive With Client	**Send()**	**OpaqueServer.cpp**	Optional
	Receive()		Optional
Text File Import/Export	**ImportText()**	**OpaqueServer.cpp**	Optional
	ExportText()		Optional
Binary File Import/Export	**ImportBinary()**	**OpaqueServer.cpp**	Optional
	ExportBinary()		Optional
Type Compare Support	**Compare()**	**OpaqueCommon.cpp**	Optional
	Equal()*		Optional
	NotEqual()		Optional
B-Tree Indexing Support	**Equal()***	**OpaqueCommon.cpp**	Optional
	GreaterThan()		Optional
	GreaterThanOrEqual()		Optional
	LessThan()		Optional
	LessThanOrEqual()		Optional
Type Mathematic Operators	**Plus()**	**OpaqueCommon.cpp**	Required
	Minus()		Required
	Times()		Required
	Divide()		Required
More Mathematic Operators	**Positive()**	**OpaqueCommon.cpp**	Required
	Negate()		Required
Type Concatenation Operator	**Concat()**	**OpaqueCommon.cpp**	Required
Type Hash Support	**Hash()**	**OpaqueServer.cpp**	Required
N.A.	**IsNull()**	**OpaqueCommon.h**	Optional
	SetNullFlag()		

* Only one **Equal()** routine generated, even if you specify all three categories that include it.

ActiveX Properties

If you choose to generate access methods in the BladeSmith New Opaque Type wizard, BladeSmith generates code to make the members of the data structure available as ActiveX properties so the client application developer can access those values.

If a member of an opaque type is an array, the following additional properties are made available:

- **One-dimensional noncharacter array**. A read-only property named **NameDim** is created to indicate the array dimension, and the property *Name* takes a one-based index as a parameter and returns the requested element.
- **Two-dimensional noncharacter array.** Properties named **NameDim1** and **NameDim2** are created, and the property *Name* takes a one-dimensional character array. Returns a string of type BSTR.
- **Two-dimensional character array.** Treated the same as a one-dimensional noncharacter array: **NameDim** indicates the dimension; the property **Name** takes a one-based index and returns the element.

Accessing Properties Using Visual Basic

This section describes how you can get and set ActiveX properties if you are using Visual Basic as your development environment.

For an ActiveX value object based on the opaque type named *Opaque*, with a non-array data structure member named **x**, you can get the corresponding property as follows:

```
member_value = Opaque.x
```

You can set the property as follows:

```
Opaque.x = member_value
```

If a data structure member is an array, you can get the property as follows:

```
count = Opaque.xDim
```

You can set or put the property as follows:

```
member_value = Opaque.x(i)
```

Compiling Client and Server Projects

Among the code files that BladeSmith generates are the following makefiles and project files:

- **ProjectU.mak.** A UNIX makefile for UNIX servers generated in the **src** directory
- **Project.dsw.** A Visual C++ workspace file for both C and C++ server code generated in the **src** directory
- **ProjectX.dsp.** A Visual C++ project file for Windows clients generated in the **src\ActiveX** directory

When you compile a server project, a **Project.bld** file is created. When you compile a client project, a **ProjectX.dll** file is created.

This section describes how to compile both a server project and a client project.

Compiling a Windows Server Project

For Windows server projects, BladeSmith generates a **Project.dsw** file in the **src** directory to use with Microsoft Visual C++ 6.0.

To compile a server project, the **INFORMIXDIR** environment variable must be set to the Informix server installation directory.

Because BladeSmith does not generate tracing routines in the source code, projects are not built with tracing support. If you have added DataBlade API tracing routines to your code, you must add the instruction to compile with tracing support to your makefile or project file. For more information, see "Tracing and Error Handling" on page 5-7.

To compile a Windows server project:

1. If necessary, open your **project.dsw** file in the **src** directory in Visual C++.
2. Choose **Build > Set Active Configuration**.
3. Select a version of the project in the Set Active Project Configuration dialog box:
   - **Release.** This version is suitable for release and does not contain debugging support.
   - **Debug.** This version contains support for debugging.
4. Click **OK**.
5. Choose **Build > Rebuild All** to compile.

Visual C++ creates both a **WinNT-i386** and a **Debug** directory under the **src\ActiveX** directory to hold the release version and the debug version, respectively, of the dynamic link library.

Compiling a Client Project

For Windows client projects, BladeSmith generates a **ProjectX.dsp** file in the **src\ActiveX** directory to use with Visual C++ 6.0 or later.

The general process for compiling a client project is:

1. Set the include and library file directories in Microsoft Developer Studio.
2. Compile the **ProjectX.dsp** file.

To set the include files and library file directories:

1. In Microsoft Developer Studio Visual C++, choose **Tools > Options**.
2. Click the **Directories** tab in the Options dialog box.
3. Select **Include files** from the **Show directories for** list box.
4. If the following directories are not on the list, add them:
   - **\informix\incl\c++**
   - **\informix\incl\dmi**
   - **\informix\incl\esql**
5. Select **Library files** in the **Show directories for** list box.
6. If the following directories are not on the list, add them:
   - **\informix\lib**
   - **\informix\lib\c++**
   - **\informix\lib\dmi**
7. Click **OK** to exit the Options dialog box.

To compile a Windows client project:

1. In Microsoft Developer Studio Visual C++, choose **File > Open** and open **ProjectX.dsp**.
2. Choose **Build > Set Active Configuration**.

3. Select a version of the project in the Set Active Project Configuration dialog box:
   - **Release.** This version is suitable for release and does not contain debugging support.
   - **Debug.** This version contains support for debugging.
4. Click **OK**.
5. Choose **Build > Rebuild All** to compile.

Visual C++ creates both a **WinNT-i386** and a **Debug** directory under the **src\ActiveX** directory to hold the release version and the debug version, respectively, of the dynamic link library.

Support Methods Reference

This section describes the internal object and support library methods that you can use when you add project-specific logic to your client and server projects. Use these methods (and the methods made available as ActiveX custom methods; see "Files to Edit" on page 6-3) to ensure that your code is portable between client and server projects.

Internal Object Methods

For each ActiveX value object that you implement, a set of internal methods is created. Although these methods are not made available to the client application developer as ActiveX custom methods, you can use them when you add project-specific logic to your client and server projects.

These are the internal object methods, where **Opaque** is the name of the opaque type that defines the current object.

Method	Description
static OpaqueCommon * CreateNew()	Creates an instance of the current object. The object name is **OpaqueServer** if called by server code or **OpaqueClient** if called by client code.
void DkErrorRaise(MI_CONNECTION *conn, mi_integer msg_type, char *msg, ...)	Maps to **mi_db_error_raise** on the server and raises an error on the client; for details, see the *IBM Informix DataBlade API Programmer's Guide*.
OpaqueStruct * GetData()	Returns a pointer to the data structure representing the current object.
mi_boolean IsDirty()	Returns mi_true if the current object has been modified or mi_false if it has not.
OpaqueStruct * RawCopy()	Allocates a C data structure and fills it with a copy of the raw data of the current object.
void SetClean()	Flags the current object as having not been modified.
void SetData(const OpaqueStruct *value)	Fills the current object with the data supplied by the input data structure.
void SetDirty()	Flags the current object as having been modified.
void SetNotNull()	A protected method that sets the current object to not null.

C++ Support Library

When you use BladeSmith to generate source code for your ActiveX value objects, the following C++ support library files are generated:

- **DkClient.cpp**
- **DkIntf.h**
- **DkIntf_i.c**
- **DkIntfImpl.h**
- **StdDbdk.cpp**
- **StdDbdk.h**

These files are used to compile both the client project and the server project; they are automatically included in the appropriate source files.

The C++ support library contains these classes and routines:

- **DkInStream** class (text input parser)
- **DkOutStream** class (text output parser)
- Memory management routines

There are two types of delimiters you must be aware of when using the **DkInStream** and **DkOutStream** classes: string delimiters and field delimiters.

String delimiters are a pair of single-byte characters that indicate the beginning and ending of a string. By default, the string delimiters are the open and close quote characters (`" "`), but you can specify other characters by using the **SetStringDelimiters()** method.

Important: If you set the **DkInStream** class string delimiters to a different pair than the **DkOutStream** string delimiters, then these text parser classes cannot exchange strings.

Field delimiters indicate the beginning and ending of a field. By default, space characters are always field delimiters. In addition, you can specify a multibyte string to also be a field delimiter, using the **SetFieldDelimiters()** method.

A string can contain multiple fields and their delimiters. However, a field cannot contain a string.

For example, if the default string and field delimiters are in use and given the characters `"Date: 4 28 97"`, then there is one string, `Date: 4 28 97`, and four fields: `Date:`, `4`, `28`, and `97`.

To include a string-delimiter character in a string, precede it with a backslash character (`\`). For example, to read the string `Date: "4 28 97"`, specify the string as follows: `"Date: \"4 28 97\""`.

The rest of this section provides reference information for the text parsing classes and memory management routines.

DkInStream

The **DkInStream** class provides methods that read an input text stream and populate an instance of the object (an opaque type if invoked by server code or an ActiveX value object if invoked by client code). This class has a built-in cursor that tracks how much of the input stream has been read.

All of the read methods return an **mi_boolean** value: `mi_true` if the read is successful or `mi_false` if it is not. In addition, all read methods except **ReadChar**, **ReadGLWChar**, and **ReadWChar** skip field and string delimiters before reading.

The **gl_wchar** data type is configurable, but it is a 4-byte character by default. For more information on this data type, see the discussion of the IBM Informix GLS API in the *IBM Informix GLS User's Guide*.

The **DkInStream** class provides the following methods.

Method	Description
DkInStream(mi_lvarchar* inputString)DkInStream(const char* inputString)	Reads **inputString**, which can be a multibyte string.
char* CurString()	Returns a pointer to the string at the current cursor position.
mi_boolean Match(char* str)	Returns mi_true if the exact sequence of characters specified in **str** is found in the input string. This method is the opposite of **DkOutStream.WriteLiteral**.
mi_boolean operator+=(size_t skip)	Returns mi_true if the number of characters specified in **skip** is successfully skipped.
mi_boolean operator-=(size_t rew)	Returns mi_true if the number of characters specified in **rew** is successfully "rewound" (skipped backwards).
mi_boolean ReadBoolean(mi_boolean* value)	Returns mi_true if one of the following values is successfully read: TRUE, True, true, FALSE, False, false.
mi_boolean ReadChar(mi_char* value)	Returns mi_true if a value of type **mi_char** is successfully read. Field and string delimiters are not skipped before reading.
mi_boolean ReadDate(mi_date* value)	Returns mi_true if a date value is successfully read. If the date value contains spaces or field delimiters, enclose it in string delimiters.
mi_boolean ReadDateTime(mi_datetime* value)	Returns mi_true if a date-time value is successfully read. If the date-time value contains spaces or field delimiters, enclose it in string delimiters.
mi_boolean ReadDecimal(mi_decimal* value)	Returns mi_true if a decimal or numeric value is successfully read.
mi_boolean ReadDoublePrecision(mi_double_precision* value)	Returns mi_true if a double-precision value is successfully read.
mi_boolean ReadGLWChar(gl_wchar_t* value)	Returns mi_true if a value of type **gl_wchar** (a 4-byte character, by default) is successfully read. Field and string delimiters are not skipped before reading.
mi_boolean ReadGLWString(gl_wchar_t* value, size_t length)	Returns mi_true if a string of **gl_wchar** values, size **length**, is successfully read. If the string is longer than **length**, it is truncated and not null-terminated. To include a string-delimiter character in the string, precede it with the backslash character (\).
mi_boolean ReadInt1(mi_int1* value)	Returns mi_true if a 1-byte integer value is successfully read.
mi_boolean ReadInt8(mi_int8* value)	Returns mi_true if an 8-byte integer value is successfully read.
mi_boolean ReadInteger(mi_integer* value)	Returns mi_true if a 4-byte integer value is successfully read.
mi_boolean ReadInterval(mi_interval* value)	Returns mi_true if an interval value is successfully read.
mi_boolean ReadMoney(mi_money* value)	Returns mi_true if a money value is successfully read.
mi_boolean ReadReal(mi_real* value)	Returns mi_true if a real value is successfully read.

Method	Description
mi_boolean ReadSmallInt(mi_smallint* value)	Returns mi_true if a 2-byte integer value is successfully read.
mi_boolean ReadString(const mi_string* value, size_t length)	Returns mi_true if a string of **mi_string** values, size **length**, is successfully read. If the string is longer than **length**, it is truncated and not null-terminated. To include a string-delimiter character in the string, precede it with the backslash character (\).
mi_boolean ReadUChar1(mi_unsigned_char1* value)	Returns mi_true if a 1-byte unsigned integer value is successfully read. (*Integer* is correct; the data type is misnamed.)
mi_boolean ReadUInt8(mi_unsigned_int8* value)	Returns mi_true if an 8-byte unsigned integer value is successfully read.
mi_boolean ReadUInteger(mi_unsigned_integer* value)	Returns mi_true if a 4-byte unsigned integer value is successfully read.
mi_boolean ReadUSmallInt(mi_unsigned_smallint* value)	Returns mi_true if a 2-byte unsigned integer value is successfully read.
mi_boolean ReadWChar(mi_wchar* value)	Returns mi_true if a 2-byte character is successfully read. Field and string delimiters are not skipped before reading.
mi_boolean ReadWString(mi_wchar* value, size_t length)	Returns mi_true if a string of **mi_wchar** values, size **length**, is successfully read. If the string is longer than **length**, it is truncated and not null-terminated. To include a string-delimiter character in the string, precede it with the backslash character (\).
void SetFieldDelimiters(const char* delim)	By default, the space character is a field delimiter; this method adds **delim** as another delimiter. **delim** can be a multibyte string, but it cannot be longer than **DK_MAXDELIMBYTES** (default value of 20).
void SetStringDelimiters(char begin, char end)	Sets string delimiters to two, single-byte characters (**begin** and **end**). Default values are the open-quote character (‘) and the close-quote character (’).
void Skip(char* delim)	Skips only the sequence of characters specified by **delim**.
void SkipBlanks()	Skips all space characters.
void SkipDelimiters()	Skips the characters specified by **delim** in the **SetFieldDelimiters** method and space characters.

DkOutStream

The **DkOutStream** class provides methods that write an object (an opaque type if invoked by server code or an ActiveX value object if invoked by client code) to an output stream. All of the write methods append to the output string; they do not overwrite the existing contents of the string.

The **gl_wchar** data type is configurable, but it is a 4-byte character by default. For more information on this data type, see the discussion of the IBM Informix GLS API in the *IBM Informix GLS User's Guide*.

The **DkOutStream** class provides the following methods.

Method	Description
DkOutStream(size_t initial=50, size_t increment=50)	Creates an output string of size **initial**, allocating additional memory in chunks of size **increment**.

Method	Description
mi_lvarchar* CreateLvarChar()	Returns a pointer to a new **mi_lvarchar** that holds a copy of the output string.
const char* GetBuffer()	Returns a pointer to the internal buffer that contains the output string.
void SetStringDelimiters(char begin, char end)	Sets string delimiters to two single-byte characters (**begin** and **end**). Default values are the open-quote character (“) and the close-quote character (”).
void WriteBoolean(mi_boolean value)	Writes a Boolean value of true or false to the output string.
void WriteChar(mi_char value)	Writes a value of **mi_char** to the output string.
void WriteDate(mi_date value)	Writes a date value to the output string.
void WriteDateTime(const mi_datetime& value)	Writes a datetime value to the output string.
void WriteDecimal(const mi_decimal& value)	Writes a decimal or numeric value to the output string.
void WriteDoublePrecision(mi_double_precision value)	Writes a double-precision value to the output string.
void WriteGLWChar(gl_wchar_t value)	Writes a value of type **gl_wchar** (a 4-byte character, by default) to the output string.
void WriteGLWString(const gl_wchar_t* value, size_t length)	Writes a string of **gl_wchar** values, size **length**, to the output string. This method precedes string-delimiter characters with backslash characters (\).
void WriteInt1(mi_int1 value)	Writes a 1-byte integer value to the output string.
void WriteInt8(const mi_int8& value)	Writes an 8-byte integer value to the output string.
void WriteInteger(mi_integer value)	Writes a 4-byte integer value to the output string.
void WriteInterval(const mi_interval& value)	Writes an interval value to the output string.
void WriteLiteral(const char* string)	Writes the specified string to the output string. Delimiter characters are written as is; they are neither skipped nor preceded by backslash characters. This method is the opposite of **DkInStream.Match**.
void WriteMoney(const mi_money& value)	Writes a money value to the output string.
void WriteReal(mi_real value)	Writes a real value to the output string.
void WriteSmallInt(mi_smallint value)	Writes a 2-byte integer value to the output string.
void WriteString(const mi_string* value, size_t length)	Writes a string of **mi_string** values, size **length**, to the output string. This method precedes string-delimiter characters with backslash characters (\).
void WriteUChar1(mi_unsigned_char1 value)	Writes a 1-byte unsigned integer to the output string. (*Integer* is correct; the data type is misnamed.)
void WriteUInt8(const mi_unsigned_int8& value)	Writes an 8-byte unsigned integer to the output string.
void WriteUInteger(mi_unsigned_integer value)	Writes a 4-byte unsigned integer to the output string.
void WriteUSmallInt(mi_unsigned_smallint value)	Writes a 2-byte unsigned integer to the output string.
void WriteWChar(mi_wchar value)	Writes a 2-byte character to the output string.
void WriteWString(const mi_wchar* value, size_t length)	Writes a string of **mi_wchar** values, size **length**, to the output string. This method precedes string-delimiter characters with backslash characters (\).

Memory Management Routines

These routines do not form a class. They are provided for server-project use only. It is recommended that you use the **new** and **delete** operators. Use **malloc** and **free**

only if you must; for example, if you call into a C file.

Routine	Description
void * ::operator new(size_t size)	Calls **mi_alloc** on the server side.
void ::operator delete(void *ptr)	Deletes the memory allocated with the **new** operator.
void * malloc(size_t size)	Calls **mi_alloc** on the server side.
void free(void *memblock)	Frees the memory allocated with the **malloc** routine.

Chapter 7. Using ActiveX Value Objects

In This Chapter

This chapter provides information for client application developers who are using ActiveX value objects. It includes the following sections:

- "Installing and Using ActiveX Value Objects," next
- "IRawObjectAccess Custom Interface" on page 7-2, for those using the IBM Informix ESQL/C or the Microsoft ODBC client APIs
- "ITDkValue Custom Interface" on page 7-3, for those using the IBM Informix C++ Interface client API
- "ActiveX Custom Methods" on page 7-4, for all users

The standard **ISupportErrorInfo** interface is also supported for ActiveX value objects.

Installing and Using ActiveX Value Objects

This section provides some guidelines on installing and using ActiveX value objects.

Installing ActiveX Value Objects

Use BladeManager to install the **ProjectS.bld** file on your Informix server computer and the **ProjectX.dll** file on your Windows client computer. For instructions, see the *IBM Informix DataBlade Module Installation and Registration Guide*.

The ActiveX project you install might also include the following files in the installation package to assist you in locating the CLSID (class identifier) and IID (interface identifier) information for the ActiveX value objects.

File	Contains
DkIntf_i.c	Interface identifiers (IIDs) for the ActiveX value object custom interfaces (**IRawObjectAccess** and **ITDkValue**; described in this chapter)
DkIntf.h	IID declarations for **DkIntf_i.c**
ProjectX_i.c	Class identifiers (CLSIDs) for the ActiveX value objects provided by the project named **Project**
ProjectX.h	CLSID declarations for **ProjectX_i.c**

If you are using Visual Basic, you must create a reference to the newly installed ActiveX project to start working with it.

© Copyright IBM Corp. 1996, 2008

To create a reference to an ActiveX project:

1. In Microsoft Developer Studio, choose **Project > References**.

   The Project References dialog box appears.
2. Check the check box for the project you are installing. The project is listed in the following format:

   Project`X 1.0 Type Library`
3. Click **OK**.

Using ActiveX Value Objects

Follow the Microsoft guidelines on how to invoke COM and automation objects.

If you are using the IBM Informix ESQL/C client API, it is recommended that you write the application in C++ and place only the SQL-specific code in the **.ec** files through embedded C code.

If you are using Visual Basic, you must cast the Informix **lvarchar** data type to **char** before you can work with ActiveX value objects.

Important: As you use ActiveX value objects, keep in mind that object persistence between server and client objects is not supported. In other words, although you can modify an ActiveX value object, an associated modification does not occur to the database data represented by that object unless you issue an *SQL* query.

IRawObjectAccess Custom Interface

The **IRawObjectAccess** custom interface is provided for users of the IBM Informix ESQL/C API and the Microsoft ODBC API. **IRawObjectAccess** enables you to instantiate an ActiveX value object with raw data or to extract raw data from an existing value object.

If you are using the IBM Informix ESQL/C API or the Microsoft ODBC API and a query of the database server results in an ActiveX value object, you get the raw data of the object. You can use this data and the methods of the **IRawObjectAccess** interface to instantiate the ActiveX value object and access its custom methods.

To instantiate the ActiveX value object and access its custom methods:

1. Call **CoCreateInstance()** with a CLSID of **CLSID_OPAQUE** and an IID of **IID_RawObjectAccess** to create an empty ActiveX value object.
2. Pass the raw object data to the **SetDataC()** method to fill the ActiveX value object.
3. Use **QueryInterface()** to get the IID for the **IDispatch** interface (**IID_IDispatch**).
4. Use **IDispatch::Invoke()** to access the custom methods of the ActiveX value object.

The **IRawObjectAccess** interface provides the following methods.

Method	Description
void * GetDataC()	Returns a pointer to **OpaqueStruct**, the C data structure that defines the opaque type that is encapsulated as an ActiveX value object.
SetDataC(void *struct)	Sets **OpaqueStruct** to the values specified by **struct**.
void * GetDataCpp()	Returns a pointer to **OpaqueClient**, the C++ object that represents the ActiveX value object.
SetDataCpp(void * struct)	Sets **OpaqueClient** (returned by **GetDataCpp**) to the values specified by **struct**.

ITDkValue Custom Interface

The **ITDkValue** custom interface is provided for the users of the IBM Informix C++ Interface. **ITDkValue** is a C++ class factory; when a query of the database server results in an ActiveX value object, an **ITDkValue** object is returned to you.

The **ITDkValue** object is an Object Interface for C++ **ITValue** object; thus, the **ITDkValue** interface provides the same methods as the **ITValue** interface. You can use this interface, or you can use the **QueryInterface()** routine to get the **IDispatch** interface of the object to access its custom methods.

In addition, a global function is provided that returns an **ITValue** object. It has the following syntax, where **Opaque** is the current object and **ITMVDesc** is an Object Interface for C++ descriptor structure:

```
ITValue * Opaque MakeValue(ITMVDesc *description)
```

For information on using the Object Interface for C++, see the *IBM Informix Object Interface for C++ Programmer's Guide*.

The **ITDkValue** interface provides the following methods.

Method	Description
ITBool CompatibleType(ITValue *object)	Returns TRUE if the specified object is of the same type as the current object.
ITBool Equal(ITValue *object)	Returns TRUE if the specified object is equal to the current object.
ITBool FromPrintable(const ITString &printable)	Sets the value of the current object, using a string equivalent to the one returned by the input function of the object.
ITBool IsNull()	Returns TRUE if the current object has a null value.
ITBool IsUpdated()	Returns TRUE if the current object has been updated since it was created.
ITBool LessThan(ITValue *object)	Returns TRUE if the current object is less than the specified objects and the objects are comparable.
const ITString &Printable()	Returns the value of the current object in a string equivalent to the one returned by the output function of the object.
ITBool SameType(ITValue *object)	Returns TRUE if the specified object is of the same type as the current object.
ITBool SetNull()	Sets the current object to a null value.
const ITTypeInfo &TypeOf()	Returns the type information for the current object.

ActiveX Custom Methods

This section is an alphabetic reference to all possible ActiveX custom methods for an ActiveX value object. It provides information on the usual behavior of each method.

ActiveX value objects (and the projects that provide them) can differ greatly. Thus, the set of custom methods made available to you can differ from object to object, or they can have different behaviors from what is described here (although the function headers and parameter lists of the methods do not vary).

These methods provide dual interfaces. Thus, you can either call them directly or by using **IDispatch::Invoke**.

Many of the methods compare the current ActiveX value object, named **Opaque**, to another object of the same type. All of the custom methods return `HRESULT`, with a value of `S_OK` (success) or `E_FAIL` (failure).

Method	Description
HRESULT Compare(**[in] IOpaque *other,** **[out, retval] int *relationship)**	Compares the current object to another object of the same type, returning: • 0 if the objects are equal. • -1 if the current object is less than the other object. • 1 if the current object is greater than the other object.

Method	Description
HRESULT Concat([in] IOpaque * other)	Concatenates another object of the same type to the current object. This method is usually implemented for string objects.
HRESULT Contains([in] IOpaque *other, [out, retval] BOOL *result)	Returns TRUE if the current object contains another object of the same type or FALSE if it does not.
HRESULT Divide([in] IOpaque *other, [out, retval] IOpaque **new)	Returns a new object representing the division of the current object by another object of the same type.
HRESULT Equal([in] IOpaque *other, [out, retval] BOOL *result)	Returns TRUE if the current object is equal to another object of the same type or FALSE if it is not.
HRESULT FromString([in] BSTR string)	Converts a character string to a new instance of the current object.
HRESULT GreaterThan([in] IOpaque *other, [out, retval] BOOL *result)	Returns TRUE if the current object is greater than another object of the same type or FALSE if it is not.
HRESULT GreaterThanOrEqual ([in] IOpaque *other, [out, retval] BOOL *result)	Returns TRUE if the current object is greater than another object of the same type or FALSE if it is not.
HRESULT Inter([in] IOpaque *other, [out, retval] IOpaque **new)	Returns a new object representing the points in common between the current object and another object of the same type.
HRESULT IsNull([out, retval] BOOL *result)	Returns TRUE if the current object is has a null value or FALSE if it does not.
HRESULT LessThan([in] IOpaque *other, [out, retval] BOOL *result)	Returns TRUE if the current object is less than another object of the same type or FALSE if it is not.
HRESULT LessThanOrEqual([in] IOpaque *other, [out, retval] BOOL *result)	Returns TRUE if the current object is less than or equal to another object of the same type or FALSE if it is not.
HRESULT Minus([in] IOpaque *other, [out, retval] IOpaque **new)	Returns a new object representing the current object minus another object of the same type.
HRESULT Negate()	Usually negates the current object: makes a positive object negative or a negative object positive.
HRESULT NotEqual([in] IOpaque *other, [out, retval] BOOL *result)	Returns TRUE if the current object is not equal to another object of the same type or FALSE if it is.
HRESULT Overlap([in] IOpaque *other, [out, retval] BOOL *result)	Returns TRUE if the current object has any points in common with another object of the same type or FALSE if it does not.
HRESULT Plus([in] IOpaque *other, [out, retval] IOpaque **new)	Returns a new object representing the current object plus another object of the same type.
HRESULT Positive()	Usually makes a negative object positive; a positive object remains positive.
HRESULT SetNullFlag()	Sets the current object to a null value.
HRESULT Size([out, retval] double *size)	Returns the size of the current object, in implementor-defined units.
HRESULT Times([in] IOpaque *other, [out, retval] IOpaque **new)	Returns a new object representing the current object times another object of the same type.
HRESULT ToString([out, retval] BSTR *string)	Converts the current object to a character string.

Method	Description
HRESULT Union([in] IOpaque *other, [out, retval] IOpaque **new)	Returns a new object representing the union of the current object and another object of the same type. This method is usually implemented for objects that represent areas.
HRESULT Within([in] IOpaque *other, [out, retval] BOOL *result)	Returns TRUE if the current object is within the other object or FALSE if it is not.

Chapter 8. Programming DataBlade Modules in Java

In This Chapter

This chapter contains information to help you edit and compile Java language source code generated by BladeSmith.

Prerequisite Tasks

Before you edit and compile your DataBlade module code, complete these tasks:

1. Write functional and design specifications that comply with Informix coding standards.

   See Chapter 3, "Programming Guidelines," on page 3-1, for more information.
2. Create your DataBlade module in BladeSmith.

   See "Creating DataBlade Module Objects" on page 4-8 for instructions.
3. Generate source code and SQL files in BladeSmith.

   See "Generating Files" on page 4-40 for instructions.

© Copyright IBM Corp. 1996, 2008

Important: You must use the IBM Informix Dynamic Server with J/Foundation upgrade to Informix Dynamic Server to enable services that use Java. For more information about J/Foundation, see the *J/Foundation Developer's Guide*.

Java Programming Task Overview

After you generate code with BladeSmith, complete these general tasks to finish your DataBlade module code:

1. Add code to the **ProjectUDRs.java** source code file to enable your routines to function as you intend. See "Editing Methods" on page 8-5 for instructions.
2. Compile your source code files using the generated makefile. See "Compiling Java DataBlade Module Code" on page 8-7 for instructions.
3. Debug your source code files using the Java log file. See "Debugging and Testing DataBlade Modules Written in Java" on page 8-9.
4. Execute functional tests. See "Performing Functional Tests" on page 8-11.

For a list of the Java packages, interfaces, classes, and methods you can use in Java projects, see *J/Foundation Developer's Guide*.

To avoid merging conflicts when you regenerate your code, add code only in areas marked by `TO DO:` comments or after the generated code. If you do modify code outside the designated areas, after you regenerate you might have two copies of the routine: the one you modified and the one BladeSmith generated. Although your changes remain, you must resolve conflicts in the two pieces of code.

Source Files Generated by BladeSmith

When you create new objects, BladeSmith generates the source files; some filenames are prefixed with the name of the DataBlade module (indicated by **project**). By default, BladeSmith creates the source files in the **src** and **src\java** subdirectories of the directory that contains the BladeSmith project file. Generated source files are listed in the following table.

Filename	Directory	Type of File	More Information
***Project*UDRs.java**	**src\java**	Java source code file	See "Java Source Code Files" on page 8-3.
IfmxInStream.java	**src\java**	Java source code file	See "BladeSmith Utility Classes" on page 8-4.
IfmxOutStream.java	**src\java**	Java source code file	See "BladeSmith Utility Classes" on page 8-4.
IfmxLog.java	**src\java**	Java source code file	See "BladeSmith Utility Classes" on page 8-4.
IfmxTrace.java	**src\java**	Java source code file	See "BladeSmith Utility Classes" on page 8-4.
DBDKInputException.java	**src\java**	Java source code file	See "BladeSmith Utility Classes" on page 8-4.
DBDKOutputException.java	**src\java**	Java source code file	See "BladeSmith Utility Classes" on page 8-4.
***Opaque*.java**	**src\java**	Java source code file	See "SQLData Interface Method Support Code" on page 8-3.

Filename	Directory	Type of File	More Information
readme.txt	**src\java**	Text file	This file describes the files in the **src\java** directory.
warning.txt	**src\java**	Text file	This file describes potential problems with your source code. See "Warning File" on page 8-3 for more information.
*Project*_**Java.mak**	**src\java**	Makefile	Use this file for compiling on both UNIX and Windows. See "Compiling Java DataBlade Module Code" on page 8-7 for more information.

Some of these files are described in the following subsections.

Java Source Code Files

BladeSmith generates a **ProjectUDRs.java** source code file that contains method declarations for all user-defined Java routines, cast support routines, and aggregates you defined with BladeSmith. You must edit this file to add the functionality you require. See "Editing Methods" on page 8-5 for more information.

BladeSmith generates the following utility class files that contain utility methods called by BladeSmith-generated routines:

- **IfmxInStream.java**
- **IfmxOutStream.java**
- **DBDKInputException.java**
- **DBDKOutputException.java**
- **IfmxLog.java**
- **IfmxTrace.java**

For more information on utility classes, see "BladeSmith Utility Classes" on page 8-4.

SQLData Interface Method Support Code

If you define a user-defined routine, aggregate, or cast support method that handles an opaque data type implemented in C or C++, BladeSmith generates the SQLData interface methods **readSQL()** and **writeSQL()** to translate objects from and to their internal server representation.

BladeSmith generates complete code for these methods in a file named ***Opaque*.java**, where *Opaque* is the name of the C or C++ opaque data type. You should not modify these methods.

Warning File

The **warning.txt** file includes the following types of warnings about your source code:

- **Unfinished code.** The file lists the routines to which you need to add code.

- **Other.** The **warning.txt** file might contain other warnings, as appropriate for your source code.

Using the Generated Code

This section contains the following subsections:

Comments in Generated Code

BladeSmith adds comments to the code it generates. Each routine begins with a prologue that describes the purpose of the routine, its arguments, and its return value. Comments throughout the code describe variable declarations and the results of generated Java statements and routine calls.

In comments at the beginning and end of each generated routine, BladeSmith stores information it uses when regenerating source code. The prologue includes a routine ID. A comment at the end of the routine contains a calculated checksum.

Warning: Do not modify either of these comments; BladeSmith uses them to merge your edits into the regenerated code.

Logging and Error Handling

BladeSmith adds logging and error handling code throughout the generated source code.

You can add additional logging calls using the **Log()** method from the **IfmxLog** class. The **Log()** method calls the standard Java I/O package methods **system.out.println()** and **system.err.println()**. For more information on these methods, see the *IBM Informix JDBC Driver Programmer's Guide*.

If the Java value object is used on the client, the **Log()** method writes the logging messages to the standard output. If the Java value object is used on the server, the **Log()** method writes the logging messages to the Java log file.

The Java log file is distinct from the main database server log file, **online.log**. The Java log file contains all logging and tracing messages specific to Java methods.

The Java log file is specified by the JVPLOGFILE configuration parameter, which is set in the **ONCONFIG** file. By default, the Java log file is at the following location:

```
/usr/informix/jvp.log
```

You can change the location of the Java log file by setting the JVPLOGFILE configuration parameter; see *J/Foundation Developer's Guide*.

You can use the Java log entries when you debug a Java method; see "Debugging a DataBlade Module" on page 8-10.

BladeSmith Utility Classes

BladeSmith generates the following utility classes whose methods are included in other generated code:

- **IfmxInStream and IfmxOutStream.** Provide read and write methods to convert Java value objects between a string and the internal server format. These methods perform similar tasks to the **Gen_sscanf()** utility function.
- **DBDKInputException and DBDKOutputException.** Provide exception-handling methods that are called when an exception occurs during the input or output of a Java value object to or from the database server.
- **IfmxLog.** Provides logging methods that are included throughout the source code generated by BladeSmith. For more information on using logging, see "Logging and Error Handling" on page 8-4.
- **IfmxTrace.** Not currently used.

Editing Methods

BladeSmith generates code for the following types of methods:

- "Most User-Defined Methods," next
- "Iterators" on page 8-5
- "Aggregates" on page 8-6
- "Cast Support Methods" on page 8-7

This code is generated in the in the **ProjectUDRs.java** file.

Important: To avoid code merge problems, modify only code in the sections marked with a `TO DO:` note. If you do modify code outside the designated areas, after you regenerate, you might have two copies of the routine: the one you modified and the one BladeSmith generated. Although your changes remain, you must resolve the conflicts in the two pieces of code.

Most User-Defined Methods

BladeSmith generates only minimal code for most methods you create with the Routine wizard.

The Generated Code

BladeSmith only generates templates for most user-defined methods.

The generated method declares the routine, its return type, and arguments.

Completing the Code

To complete the code for most user-defined methods, you must:

- add your declarations, if necessary.
- remove the call to **Log()**, which raises an error stating that the method is not implemented.
- compute the return value and store it in the **Gen_RetVal** argument.

For more information on programming routines, see *J/Foundation Developer's Guide*.

Example

The example JavaCircle DataBlade module has user-defined methods.

Iterators

If you create an iterator method that returns a set one row at a time, BladeSmith adds code to process the set. The Informix database server calls iterator methods repeatedly to process all of the return values.

The Generated Code

In addition to the arguments you specified when you created it, an iterator function contains an MI_FPARAM argument. The Informix database server use an MI_FPARAM structure to control iteration over the set. The generated code includes a Java **else** statement with different cases to process the set. The **else** statement uses the **getIterationState()** method to obtain the request flag from the **UDREnv** object. The Informix database server sets this flag to one of the following values before it calls the method:

- `UDR_SET_INIT`. The initial call to the iterator method. The iterator method allocates and initializes memory for state information.
- `UDR_SET_RETONE`. The iterator method is called with this request flag once for each value in the set.

  For each value in the set, the method places the address of the next value in the set in the **Gen_RetVal** argument and returns **Gen_RetVal**.

  When there are no more values to return, the iterator method must call the **setSetIterationIsDone()** function to signal the Informix database server that all of the set values have been returned.
- `UDR_SET_END`. The request flag the Informix database server sets after all values in the set have been returned. The iterator method frees allocated memory and releases any other resources it has obtained.

In the generated code, each of these sections has a `TO DO:` note. To avoid code merging problems, make changes only where indicated.

Completing the Code

To complete the iterator code, you must:

- Add information declarations.
- Initialize the iterator function.
- Allocate private state information.
- Compute the value of the iteration.
- Call **setSetIterationIsDone()** when the iteration is complete.
- Free private resources.

For more information on programming iterator methods, see *J/Foundation Developer's Guide*.

Aggregates

If you created a user-defined aggregate with the Aggregate wizard, BladeSmith generates aggregate methods in the **ProjectUDRs.java** source code file.

The Generated Code

BladeSmith only generates templates for aggregate methods.

The generated method declares the method, its return type, and arguments.

Completing the Code

To complete the code for aggregate methods, you must:

- Add your declarations, if necessary.
- Remove the call to **Log()**, which raises an error stating that the routine is not implemented.
- Compute the return value and store it in the **Gen_RetVal** argument.

For more information on programming aggregate methods, see *J/Foundation Developer's Guide*.

The Initialization Method: If you selected an initialization method, **AggregateInit()**, you must add code to it to initialize the state type required by the aggregate computation. The **AggregateInit()** method returns the state type.

The first argument of the **AggregateInit()** method is a dummy argument whose value is always `NULL`. The second argument is an optional initialization parameter to customize aggregate computation. The initialization parameter cannot be a lone host variable reference.

The Iteration Method: You must add code to the iteration method, **AggregateIter()**, to perform the aggregate computations.

Tip: Although the iteration method is called by the database server multiple times to calculate the aggregation, it is not implemented as an iterator method that returns a set of results.

The Combine Method: If you selected a combine method, **AggregateComb()**, you must add code to it to merge one partial result with another and return the updated state type.

The Final Function: If you selected a final method, **AggregateFinl()**, you must add code to convert the state type to the result type.

You can also add code to the **AggregateFinl()** method to release resources acquired by the initialization method. However, the **AggregateFinl()** method must not free the state type.

Cast Support Methods

If you specified a cast support method when you created a cast, BladeSmith generates the cast support method in the **ProjectUDRs.java** file.

The Generated Code

BladeSmith generates only templates for cast support methods.

The generated method declares the routine, its return type, and arguments.

Completing the Code

To complete the code for cast support methods, you must:

- Add your declarations, if necessary.
- Remove the call to **Log()**, which raises an error stating that the routine is not implemented.
- Convert one data type to the other.
- Store the return value in the **Gen_RetVal** argument.

In a cast support method, you might convert from one binary representation to another, if the data types involved in the cast have differing binary representations. Alternatively, you might perform a calculation to convert one data type to another.

Compiling Java DataBlade Module Code

BladeSmith generates the **Project_Java.mak** makefile in the **src\java** directory. Use this makefile to compile Java code from the command line on UNIX and Windows.

When you compile, the makefile produces a JAR file, **Project.jar**, in the source code directory **src/java**. This file is appropriate for the server and client implementations.

The makefile requires that you set the following environment variables before you compile:

- **INFORMIXDIR.** Set to the Informix database server installation directory.
- **CLASSPATH**. Set to the Java Developers Kit, the Java in the server JAR file, and the IBM Informix JDBC Driver locations:

```
.:$(JDKPATH):${INFORMIXDIR}/extend/krakatoa/krakatoa.jar:
${INFORMIXDIR}/extend/krakatoa/jdbc.jar
```

Windows NT Only

UNIX Only

- **TARGET.** Set to the path and filename of the include file for your platform. Platform-specific files are located in the directory **INFORMIXDIR/incl/dbdk**.

End of UNIX Only

End of Windows NT Only

The **BINDIR** variable in the makefile determines where the JAR files are written.

BladeSmith creates **server**, **all**, and **clean** targets in the makefile. The **server** target builds the JAR files. The **clean** target deletes the JAR files. The default **all** target is equivalent to the **server** target.

Important: When you generate code in BladeSmith, set the **Format** property of the **DataBlade** node to the correct file format for your operating system (**UNIX** or **DOS**). The default is **DOS**. See "Generating Source Files" on page 4-44 for more information.

Use the **Project_Java.mak** makefile with the JDK 1.1.*x* compiler.

To compile and link your DataBlade module JAR files:

1. If you are compiling on a different computer than the one on which DBDK is installed, copy the generated **src/java** directory with its contents to the target directory.
2. Execute the appropriate command at the UNIX C shell or the MS-DOS prompt:

UNIX Only

```
make -f Project_Java.mak
```

End of UNIX Only

Windows NT Only

```
nmake -fProject_Java.mak
```

End of Windows NT Only

Project is the name of the DataBlade module project. The **Project.jar** files are created in the **src/java** directory.

Debugging and Testing DataBlade Modules Written in Java

This section describes debugging and performing functional tests on DataBlade module routines written in Java.

This section contains the following subsections:

- "Preparing Your Environment," next
- "Debugging a DataBlade Module" on page 8-10
- "Performing Functional Tests" on page 8-11

Preparing Your Environment

Before you can debug or test your DataBlade module, you must configure your Informix database server.

Windows NT Only

UNIX Only

For information on the environment variables you must set to debug and test on UNIX, see "Preparing Your Environment" on page 9-2.

End of UNIX Only

End of Windows NT Only

Windows Only

For information on the environment variables you must set to debug and test on Windows, see "Preparing Your Environment" on page 10-2.

End of Windows Only

In addition, you must complete the following tasks to use Java with the database server:

- Create an sbspace to hold the Java JAR files.
- Create the JVP properties file.
- Add or modify the Java configuration parameters in the **ONCONFIG** file.

Windows NT Only

UNIX Only

- Install symbolic links to the Java VM libraries.

End of UNIX Only

End of Windows NT Only

For instructions on how to complete these tasks, see *J/Foundation Developer's Guide*.

Debugging a DataBlade Module

Debugging a DataBlade module is usually an iterative process, repeated several times until the code is completely debugged. The debugging process has the following general steps:

1. Compile the JAR file (if necessary).
2. Install the DataBlade module shared object and SQL scripts in the **$INFORMIXDIR/extend/project** directory.

   See "Installing a DataBlade Module" on page 8-10 for more information.
3. Start your database server while logged on as the **informix** user.

   See the *IBM Informix Dynamic Server Administrator's Guide* for more information.
4. Register the DataBlade module using BladeManager (if necessary).

   See the "Registering a DataBlade Module" on page 8-10 for more information.
5. If you are replacing an existing JAR file, shut down and restart the database server.

   See "Replacing a DataBlade Module JAR File" on page 8-11 for more information.
6. Execute a query that calls the method using an SQL query tool such as DB-Access or SQL Editor.

   See the *IBM Informix DB–Access User's Guide* for more information.
7. Examine the Java log file for errors.

   See "Logging and Error Handling" on page 8-4 for more information.
8. Edit the source code (if necessary).
9. Repeat the procedure, as necessary.

Installing a DataBlade Module

To install a DataBlade module for debugging, create a project directory and copy the necessary files to it. Create the project directory under **$INFORMIXDIR/extend**. The name of the project directory is what BladeManager uses as the DataBlade module name.

A good project naming strategy is to combine the project name and version numbers you entered in the New Project wizard in BladeSmith. For example, the **Circle** project, Version 1.0, can be in **$INFORMIXDIR/extend/Circle.1.0**. IBM Informix DataBlade modules also include a string indicating the build platform and minor release: for example, 1.0.UC1.TC2, where UC1 is the first UNIX major release, and TC2 is the second Windows minor release.

To copy the necessary files to the project directory, use one of these methods:

- Use BladePack to create an installation directory for your DataBlade module and then copy that directory into the module subdirectory under **$INFORMIXDIR/extend**. For instructions, see Chapter 11, "Using BladePack," on page 11-1.
- Copy the **project.jar** file and the contents of the **scripts** directory into the project directory.

For installation tips and solutions to common problems, see the IBM Informix Developer Zone at http://www.ibm.com/software/data/developer/informix.

Registering a DataBlade Module

You need to register your DataBlade module the first time you install it and subsequently if you change the definition of any of your DataBlade module objects

in BladeSmith and generate new SQL files. You do not have to reregister your DataBlade module when you only replace its JAR file.

Important: You must have a default sbspace defined in your database server to hold your DataBlade module JAR files. If you do not, BladeManager does not register your Java DataBlade module.

See the *IBM Informix DataBlade Module Installation and Registration Guide* for more information on registering DataBlade modules.

Replacing a DataBlade Module JAR File

When a DataBlade module is loaded onto an Informix database server, the database server stores it in the database server memory map. Therefore, if you overwrite a JAR file while it is loaded in the database server, you must stop and restart the database server to unload the old JAR file and load the new one.

Warning: If you do not stop and restart the database server after you replace a DataBlade module JAR file, the database server might fail when you call a DataBlade module routine.

To unload a module without restarting the Informix database server, you must drop all objects in the module, using the SQL DROP statement. After all objects in the module have been dropped and all instances of the methods have finished executing, the symbol references to the DataBlade module JAR file are invalidated, and a message is recorded in the log file.

After the module is unloaded, replace the JAR file and load it into the database.

Performing Functional Tests

When you generate functional tests, BladeSmith creates a set of files that include shell scripts and SQL scripts for testing opaque data type support routines, user-defined routines, and cast support functions.

Windows NT Only

UNIX Only

For instructions on how to execute functional tests on UNIX, see "Performing Functional Tests" on page 9-7.

End of UNIX Only

End of Windows NT Only

Windows Only

For instructions on how to execute functional tests on Windows, see "Performing Functional Tests on DataBlade Modules" on page 10-6.

End of Windows Only

Chapter 9. Debugging and Testing DataBlade Modules on UNIX

In This Chapter

This chapter describes how to debug and perform functional tests for DataBlade modules written in C for Dynamic Server on UNIX.

See "Debugging and Testing DataBlade Modules Written in Java" on page 8-9 for instructions on debugging DataBlade modules written in Java.

Prerequisite Tasks

Before you debug or run functional tests on your DataBlade module code, you must complete these tasks:

1. Create your DataBlade module in BladeSmith.

   See "Creating DataBlade Module Objects" on page 4-8 for instructions.
2. Add functional test data for your DataBlade module routines in BladeSmith.

   See "Adding Functional Test Data" on page 4-36 for instructions.
3. Generate source, SQL, and test files in BladeSmith.

   See "Generating Files" on page 4-40 for instructions.
4. Complete your source code.

   For instructions on completing C code, see Chapter 5, "Programming DataBlade Module Routines in C," on page 5-1

   For instructions on completing C++ and ActiveX code, see Chapter 6, "Creating ActiveX Value Objects," on page 6-1
5. Build your DataBlade module dynamic link library.

© Copyright IBM Corp. 1996, 2008

For instructions on compiling C DataBlade modules, see "Compiling on Windows" on page 5-34.

For instructions on compiling C++ and ActiveX DataBlade modules, see "Compiling Client and Server Projects" on page 6-5.

Preparing Your Environment

Test and debug your DataBlade module in a nonproduction Informix database server environment because debugging interferes with the operation of the database server.

To successfully test and debug your DataBlade module, set your environment so you can access your Informix database server installation and build your DataBlade module shared object.

To run your Informix database server, check that these environment variables are set properly: **INFORMIXDIR**, **PATH**, **LD_LIBRARY_PATH**, **ONCONFIG**, and **INFORMIXSERVER**. See the *IBM Informix Dynamic Server Administrator's Guide* for more information on configuring your Informix database server.

When testing your DataBlade module, set the **TESTDB** environment variable to the name of your test database.

To recompile your DataBlade module shared object file during debugging, also set the **TARGET** environment variable. See "Compiling on UNIX" on page 5-33 for more information on the **TARGET** environment variable.

Using the Shared Object File

A DataBlade module exists in the Informix database server as a shared object. The shared object file is loaded into the database server the first time one of its routines is executed after the database server is started. The shared object file is unloaded every time the database server is stopped.

Replacing a Shared Object File

When a DataBlade module is loaded onto an Informix database server, the database server stores it in the database server memory map. Therefore, if you overwrite a shared object file while it is loaded in the database server, you must stop and restart the database server to unload the old shared object file and load the new one.

Warning: If you do not stop and restart the database server after replacing a shared object, the database server might fail when you call a DataBlade module routine.

To unload a module without restarting the Informix database server, you must drop all objects in the module, using the SQL DROP statement. After all objects in the module have been dropped and all instances of the routines have finished executing, the symbol references to the DataBlade module shared object are invalidated, and a message is recorded in the log file.

After the module is unloaded, replace the shared object file and load it into the database.

Shared Object File Ownership and Permissions

Shared object files must be owned by the user ID that runs the Informix database server. In a production installation, the Informix database server runs as user **informix**, and shared object files are owned by user **informix**.

The Informix database server loads a shared object file only if it is marked as read-only. The **project.bld** file is marked as read-only by the makefile BladeSmith generates.

Important: If you receive a -9793 error when you try to execute a routine in the shared object file, your shared object file is not marked as read-only.

Symbols in Shared Object Files

Undefined symbols in a shared object file are resolved in the database server when the file is loaded. If a symbol is missing, the load fails on the first execution of the user-defined routine, and a message is written in the server log file.

You cannot resolve undefined symbols in a shared object file using definitions in another shared object file.

A symbol defined in a shared object file on the database server behaves in one of two ways:

- If the symbol referenced in the shared object file is in the same source file that references it, the debugger accesses the symbol in the shared object file.
- If the shared object file includes more than one source file and there is a cross-file symbol reference, the symbol is resolved in the database server. These symbols are listed as unresolved when you link the shared object file.

Important: Although most of the unresolved symbols listed when you link the DataBlade module shared object file are resolved when the database server loads the shared object, check for mistyped symbols; these are not resolved.

Installing and Registering DataBlade Modules

Installing a DataBlade module places the module's files in a subdirectory of the **$INFORMIXDIR/extend** directory; registering a DataBlade module adds the module to a database. You must install and register before you can test or debug a DataBlade module.

Installing a DataBlade Module

See "Replacing a Shared Object File" on page 9-2 for important information about updating an existing DataBlade module shared object file.

To install a DataBlade module for testing and debugging, create a project directory and copy the necessary files to it. Create the project directory under **$INFORMIXDIR/extend**. The name of the project directory is what BladeManager uses as the DataBlade module name.

A good project naming strategy is to combine the project name and version numbers you entered in the New Project wizard in BladeSmith. For example, the **Circle** project, Version 1.0, can be in **$INFORMIXDIR/extend/Circle.1.0**. IBM Informix DataBlade modules also include a string indicating the build platform

and minor release: for example, 1.0.UC1.TC2, where UC1 is the first UNIX major release, and TC2 is the second Windows minor release.

To copy the necessary files to the project directory, use one of these methods:

- Use BladePack to create an installation directory for your DataBlade module and then copy that directory into the module subdirectory under **$INFORMIXDIR/extend**. For instructions, see Chapter 11, "Using BladePack," on page 11-1.
- Copy the **project.bld** file and the contents of the **scripts** directory into the project directory.

For installation tips and solutions to common problems, see the IBM Informix Developer Zone at http://www.ibm.com/software/data/developer/informix.

Registering a DataBlade Module

You need to register your DataBlade module the first time you install it and if you change the definition of any of your DataBlade module objects in BladeSmith and generate new SQL files. You do not have to reregister your DataBlade module when you only replace its shared object file.

See the *IBM Informix DataBlade Module Installation and Registration Guide* for more information on registering DataBlade modules.

Debugging a DataBlade Module

Debugging a DataBlade module is usually an iterative process, repeated many times until the code is completely debugged. The debugging process has the following general steps:

1. Build the shared object file with debugging support while logged on as user **informix** (if necessary).

   To debug a DataBlade module, compile the shared object file with the **-g** compiler option so that debugging symbols are available to the debugger. See "Compiling DataBlade Module Code" on page 5-33 for information about compiling with debugging support.
2. Install the DataBlade module shared object and SQL scripts in the **$INFORMIXDIR/extend/project** directory.

   See "Installing a DataBlade Module" on page 9-3 for more information.
3. Start your database server with the **oninit** command, while logged on as the **informix** user.

   See the *IBM Informix Dynamic Server Administrator's Guide* for more information.
4. Register the DataBlade module, using BladeManager (if necessary).

   See "Registering a DataBlade Module" on page 9-4 for more information.
5. If you are replacing an existing shared object, shut down and restart the database server with the **onmode -yuk** and **oninit** commands.

   See "Replacing a Shared Object File" on page 9-2 for more information.
6. Load the DataBlade module by calling one of its routines.

   See "Loading the DataBlade Module" on page 9-5 for instructions.
7. Log on as user **root** in a new window to run the debugger.
8. Obtain the database server process ID for the **root** session.

   See "Identifying the Server Process" on page 9-5 for instructions.

9. Run the debugger and attach to the database server process.
   See "Running the Solaris Debugger" on page 9-6 for instructions.
10. Set any appropriate breakpoints.
    See "Setting Breakpoints" on page 9-6 for more information.
11. Issue SQL statements to call your DataBlade module routines from the **informix** session.
    See the *IBM Informix DB–Access User's Guide* for more information.
12. Edit the source code (if necessary).
13. Repeat the procedure, as necessary.

The following sections describe some of these steps.

Loading the DataBlade Module

Before you can attach to the database server process with the debugger, load your DataBlade module shared object file into the database server address space. With the shared object file loaded, set breakpoints on the routine entry points and examine local storage provided by the routines.

To load the DataBlade module into the database server address space, execute one of its routines. One technique is to call the routine with an impossible condition, as follows:

```
SELECT routine_name(column_name) FROM table_name
WHERE 1=0;
```

routine_name is the name of your routine, *column_name* is the name of a column in the table, and *table_name* is the name of the table. This statement loads your DataBlade module shared object file without executing the routine.

Identifying the Server Process

To debug a routine, you must identify the virtual processor in which that routine runs. By default, routines are assigned to the CPU virtual processor class. However, when you create a routine in BladeSmith, you can specify if it is poorly behaved and assign it to a user-defined virtual processor class.

To identify the virtual processor class assigned to a routine, look at the property page for the routine in BladeSmith. If the class field is blank, then the routine runs in the CPU VP. See "C Programming Guidelines" on page 3-6 for more information on user-defined virtual processors.

Important: If you have more than one instance of a virtual processor in a CPU or user-defined virtual processor class, threads can migrate between virtual processors, making debugging difficult. To simplify debugging, configure your database server so that there is only one instance each of the CPU VP or user-defined VP used by the routines in your DataBlade module.

To find the process ID (PID) of the CPU or user-defined virtual processor that you want to debug, execute the **onstat** command, as follows:

```
onstat -g glo
```

The last section of the output of this **onstat** command is similar to the following example.

```
Individual virtual processors:
 vp    pid       class      usercpu   syscpu     total
 1     3544      cpu        3.75      0.96       4.71
 2     3545      adm        0.05      0.03       0.08
 3     3546      lio        0.04      0.07       0.11
 4     3547      pio        0.05      0.03       0.08
 5     3548      aio        0.04      0.04       0.08
 6     3549      msc        0.39      0.19       0.58
 7     3550      aio        0.09      0.10       0.19
```

Figure 9-1. Sample onstat Command Output

Typically, the PID circled in the sample output is the one you need. In this example, there are no user-defined virtual processor classes; all the DataBlade routines are marked as well behaved and run in the single instance of the CPU VP.

Running the Solaris Debugger

To debug your DataBlade module, use a debugger that can attach to the active database server process and access the symbol tables of dynamically loaded shared object files. On Solaris, the **dbx** utility meets these criteria, as does **debugger**.

Before beginning debugging, enter the following commands to disable signal handlers in the debugger:

```
ignore SIGUSR1
ignore SIGUSR2
```

Tip: You can put these instructions in the **.dbxinit** file. Then put the file in the **$INFORMIXDIR/bin** directory. However, then you must always start **dbx** from that directory.

To start **dbx**, enter the following command at the shell prompt:

```
dbx - PID
```

PID is the process ID of the CPU VP or user-defined VP.

This command starts **dbx** on the database server virtual process without starting a new instance of the virtual processor.

When the debugger starts, it lists the loaded shared object libraries. If your DataBlade module shared object file is not on the list, load it by calling one of its routines in the database server. See "Loading the DataBlade Module" on page 9-5 for instructions.

You can set breakpoints, examine the stack, resume execution, or carry out any other normal **dbx** command. See the online **dbx** publication page for more information about available **dbx** commands.

Setting Breakpoints

You can set breakpoints in any routine with an entry point known to **dbx**.

Informix database server software is compiled with debugging support turned off, so local storage and line number information is not available for database server routines. However, after you compile the DataBlade module for debugging, you can see line number information and local storage for your functions.

When you enter a command in the client that calls one of your DataBlade module routines, the debugger stops in the routine. Then you can follow the steps of your routine. Because your DataBlade module is compiled with debugging support, you can view the local variables and stack for your routines.

Debugging a UNIX DataBlade Module with Windows

Debug a UNIX DataBlade module from your Windows computer by logging into a UNIX computer from your Windows computer and running the debugger in a telnet session or an X window emulation program.

Performing Functional Tests

When you generate functional tests, BladeSmith creates a set of files that include shell scripts and SQL scripts for testing opaque data type support routines, user-defined routines, and cast support functions. By default, these files are created in the **functest** subdirectory of the directory containing the BladeSmith project file.

Functional tests are generated only for the DataBlade module objects for which you enter test data in your BladeSmith project. See "Adding Functional Test Data" on page 4-36 for information about entering test data.

Functional testing is typically an iterative process, repeated many times until the code passes all the tests. The testing process has the following general steps:

1. Build the shared object file while logged on as user **informix**.

   See "Compiling on UNIX" on page 5-33 for instructions.
2. Install the DataBlade module shared object, SQL scripts, and test scripts in the **$INFORMIXDIR/extend/project** directory.

   See "Installing a DataBlade Module" on page 9-3 for more information.
3. Log on as the **informix** user and start your database server with the **oninit** command.

   See the *IBM Informix Dynamic Server Administrator's Guide* for more information.
4. Create a test database.

   See the *IBM Informix DB–Access User's Guide* for more information.
5. Register the DataBlade module, using BladeManager.

   See "Registering a DataBlade Module" on page 9-4 for more information.
6. If you are replacing an existing shared object, shut down and restart the database server with the **onmode -k** and **oninit** commands.

   See "Replacing a Shared Object File" on page 9-2 for more information.
7. Execute the functional tests.

   See "Executing Functional Tests" on page 9-10 for instructions.
8. Edit the source code (if necessary).
9. Regenerate the tests in BladeSmith (if necessary).
10. Repeat the procedure, as necessary.

Functional Test Overview

Functional tests include SQL scripts and shell scripts that execute the SQL scripts and determine the results. The shell scripts build test tables in a database, run the SQL test scripts, and then drop the test tables from the database.

You can create custom shell scripts to run additional tests or initialization scripts. The generated scripts include calls to your custom scripts.

Shell scripts execute SQL scripts using DB-Access. The results from the SQL statements are saved in **.log** files. When you first run functional tests, you must inspect the **.log** files and, if the results are correct, use the shell scripts to copy them to **.req** files.

When you execute functional tests after saving **.req** files, the shell script uses the UNIX **diff** command to compare the **.log** files to the **.req** files. The script prints the following messages:

- "test passed" message if the **.log** and **.req** files match
- "test failed" message if the files do not match
- "status unknown" message when a **.req** file does not yet exist

Important: There are minor formatting differences between the UNIX and Windows versions of DB–Access that can cause tests to indicate failure incorrectly.

Contents of the Functional Test Directory

The functional test directory, **functest**, includes the following subdirectories:

- **data.** Contains **.dat** files for each opaque type, user-defined routine, and cast for which you entered test data. The name of the data file is **objectname.dat**, where **objectname** is either the name of the opaque type or the name of the C routine associated with a user-defined routine or cast.
- **opaque.** Contains a subdirectory for each opaque type for which you entered test data. The subdirectory contains functional tests for the support routines defined for the opaque type.
- **udr.** Contains a subdirectory containing functional tests for each user-defined routine for which you entered test data.
- **cast.** Contains a subdirectory containing functional tests for each cast for which you entered test data.

The **functest** directory contains a master shell script, **main.sh**, for executing all of the functional tests generated for the DataBlade module. Each subdirectory in the **udr**, **opaque**, and **cast** directories also contains a **main.sh** script to execute only the functional tests in that subdirectory.

The subdirectories in the **udr**, **opaque**, and **cast** directories contain various SQL scripts. Each subdirectory has a **setup.sql** script and a **cleanup.sql** script. The **setup.sql** script creates test tables and initializes them with test data. The **cleanup.sql** script drops all of the test tables from the database.

BladeSmith creates the following SQL test scripts for the object being tested:

- **call_pos.sql**, for user-defined routines
- **call_neg.sql**, for negative tests of user-defined routines
- **cast.sql**, for casts
- Additional scripts for opaque types to test the support routines defined for the type, as described in the following table

Script Names	Support Routines Tested
textio_pos.sql	Text input/output functions for an opaque type; uses only valid test data.

Script Names	Support Routines Tested
textio_neg.sql	Text input/output functions for an opaque type; uses test data with invalid input data.
binio.sql	Binary file input/output functions; uses the valid input data for the opaque type.
textexp.sql	Text file import/export functions for opaque types; uses the UNLOAD and LOAD SQL statements.
binexp.sql	Binary file import/export functions for an opaque type; uses nested calls to the binary file import/export functions. The result of the nested calls should be equivalent to the text input format for the type.
notify.sql	The **Assign**/**Destroy** routines; inserts and deletes values in a new test table.
compare.sql	The **Compare** function for an opaque type.
equal.sql	The **Equal** function for an opaque type.
notequal.sql	The **NotEqual** function for an opaque type.
btree.sql lessthan.sql lessthanorequal.sql greaterthan.sql greaterthanorequal.sql	B-tree support functions for an opaque type.
plus.sql minus.sql times.sql divide.sql	Standard math operators for an opaque type.
positive.sql negative.sql	The **Positive** and **Negate** functions for an opaque type.
concat.sql	The concatenation operator; calls the **Concat** function for an opaque data type with two instances of the type.
hash.sql	The **Hash** support function with a SELECT...GROUP BY SQL query.

A script is generated only when the support routines it tests are defined.

Adding Custom Test Files

You can add other tests or initialization scripts to your test suite by adding your own scripts in the subdirectories of the **functest** directory and editing the sample **user.sh** shell script that BladeSmith generates. For example, you can add SQL scripts to create a test database, create special test tables in it, and execute custom tests against those tables.

Executing Functional Tests

To execute functional scripts, use the **main.sh** script. Execute a command in all test directories by executing the **main.sh** command in the top-level test directory. You can execute tests for a specific DataBlade module object by executing **main.sh** in that object's test directory.

The first time you execute the tests, initialize the reference files. See "Initializing Reference Files" on page 9-10 for instructions.

The **TESTDB** environment variable must be set to the name of the test database.

Using the Functional Test Scripts

The **main.sh** script is a Bourne shell script that accepts one of five possible command line parameters, as described in the following table.

Command	Description
main.sh build	Runs **user.sh** with a "build" target. Runs the **setup.sql** script.
main.sh clean	Deletes **.log** files. Runs **user.sh** with a "clean" target. Executes **clean.sql** in the database.
main.sh run	Uses DB-Access to run each SQL script generated by BladeSmith, saving the output in a **.log** file. If a **.req** file exists, calls **diff** to determine the test result. It prints a message telling whether the test passed or failed. After all BladeSmith-generated tests are run, executes **user.sh** with a "run" target.
main.sh save	Copies all **.log** files to **.req** files, overwriting existing **.req** files.
main.sh all	Performs the "build," "run," and "clean" actions. Use this shortcut after the **.req** files have been saved.

Initializing Reference Files

The first time you run tests, execute the "build" and "run" targets, as follows:

```
main.sh build
main.sh run
```

These two steps prepare the database, run the test scripts, and generate .**log** files. The results of all tests are `unknown (no reference file)`.

Check the results in each **.log** file to determine if the test returned the correct result. The expected result (which was entered with the test data in BladeSmith) is shown in a comment.

If the results are incorrect, you might need to fix the DataBlade module C code. In other cases, the test data can be incorrect.

When the tests return correct results, create reference files by executing the "save" target, as follows:

```
main.sh save
```

After reference files have been saved, use the "all" shortcut target to build and run the tests and clean up the database and test directory.

Chapter 10. Debugging and Testing DataBlade Modules on Windows

In This Chapter

This chapter describes how to debug and perform functional tests for DataBlade modules written in C and C++ for Dynamic Server on Windows.

See "Debugging and Testing DataBlade Modules Written in Java" on page 8-9 for instructions on debugging DataBlade modules written in Java.

Prerequisite Tasks

Before you run tests or debug your DataBlade module code, you must complete these tasks:

1. Create your DataBlade module in BladeSmith.

   See "Creating DataBlade Module Objects" on page 4-8 for instructions.
2. Optionally add functional test data for your DataBlade module routines in BladeSmith.

   See "Adding Functional Test Data" on page 4-36 for instructions.
3. Generate source, SQL, and test code in BladeSmith.

   See "Generating Files" on page 4-40 for instructions.
4. Complete your source code.

   For instructions on completing C code, see Chapter 5, "Programming DataBlade Module Routines in C," on page 5-1

   For instructions on completing C++ and ActiveX code, see Chapter 6, "Creating ActiveX Value Objects," on page 6-1
5. Build your DataBlade module dynamic link library.

   For instructions on compiling C DataBlade modules, see "Compiling on Windows" on page 5-34.

   For instructions on compiling C++ and ActiveX DataBlade modules, see "Compiling Client and Server Projects" on page 6-5.

© Copyright IBM Corp. 1996, 2008

Preparing Your Environment

Test and debug your DataBlade module in a nonproduction Informix database server environment because debugging interferes with the operation of the database server.

To use the debugging features of the DBDK Visual C++ Add-In, you must have a local database server.

Before you begin testing and debugging your DataBlade module, verify that your database server is running properly. Use the Setnet32 utility to complete the Server Information page and create a default database server. Create a database with SQL Editor; if the command succeeds, your server is properly configured and the DBDK Visual C++ Add-In should work properly.

See the *IBM Informix Dynamic Server Administrator's Guide* for more information on configuring your Informix database server.

DBDK Visual C++ Add-In and IfxQuery

The DBDK Visual C++ Add-In is a toolbar that appears in Microsoft Visual C++ after you install DBDK. The add-in aids you in debugging DataBlade modules in the following ways:

- If you are using a local database server for debugging, the add-in automates tasks from compiling through reaching the first breakpoint in your source code. In this case, you must have your database server on the same computer as the Informix DataBlade Developers Kit you used to develop your DataBlade module.
- If you are using a remote database server, the add-in automates the following tasks:
  - Installing the DataBlade module project file
  - Installing the DataBlade module SQL scripts
  - Registering the DataBlade module

The add-in toolbar contains seven buttons. To see the name of each command, position the mouse pointer over the button. Figure 10-1 shows the add-in toolbar.

Figure 10-1. DBDK Visual C++ Add-In

The primary add-in command is the **Debug DataBlade Module** command; it completes all the tasks necessary to bring your DataBlade module to the first debugging breakpoint. When you use the **Debug DataBlade Module** command to start debugging, the IfxQuery tool is launched from within Visual C++ when an SQL unit test file is the active window.

The Debug DataBlade Module Command

If you have a DataBlade module project open in Visual C++ and click the **Debug DataBlade Module** button, the **Debug DataBlade Module** command performs the following steps:

1. Checks if the DataBlade module needs to be compiled and compiles it, if necessary.
2. If necessary, creates a new directory for the DataBlade module under the **%INFORMIXDIR%\extend** directory.
3. Installs the DataBlade module dynamic link library and SQL scripts in the **%INFORMIXDIR%\extend\project.0** directory.
4. If necessary, shuts down the database server.
5. Starts Visual C++ debugger with the database server attached.

Important: The database server typically runs as a Windows service; you can start and stop it using the Services dialog box in the Control Panel. However, when the add-in starts the database server attached to the debugger, it does not run as a service and the Services dialog box does not show it running. If you attempt to start or stop the database server using the Services dialog box while it is attached to the debugger, you receive an error.

If the active window when you execute the **Debug DataBlade Module** command is an SQL file, the **Debug DataBlade Module** command launches IfxQuery, which performs the following additional tasks:

1. If necessary, creates the database you specified in the Configure DBDK Visual C++ Add-In dialog box
2. Connects to the database for the project
3. Registers the DataBlade module
4. If necessary, initializes the newly created database using the **Setup.sql** file
5. Executes the SQL statements from the active unit test SQL file until the first breakpoint is reached
6. After you pass the breakpoint, executes the next SQL statement until the next breakpoint is reached
7. After you pass all breakpoints and the routine returns, writes the results of the SQL statements to an HTML file
8. Launches the default HTML browser for your computer
9. Displays the SQL results in the HTML browser
10. Shuts down

If the active window when you execute the **Debug DataBlade Module** command is not an SQL file, you can execute SQL queries using another SQL query tool, such as SQL Editor. However, you must first explicitly register the DataBlade module by clicking the add-in **Register DataBlade Module** button or by using BladeManager (see the *IBM Informix DataBlade Module Installation and Registration Guide* for instructions). In addition, the database you specified in the Configure DBDK Visual C++ Add-In dialog box must exist.

Other Add-In Commands

The following table lists the other add-in command buttons, in addition to the **Debug DataBlade Module** command, and the tasks they complete.

Task	Button
Copy the **project.bld** file to the local or remote database server	Upload DataBlade Module
Copy the DataBlade module SQL scripts to the local or remote database server	Upload DataBlade SQL Scripts
Register the DataBlade module on the local or remote database server	Register DataBlade Module
Shut down and restart the local database server	Stops and restarts IDS on the local host
Launch the add-in help page, which is part of the DBDK InfoShelf	Launch InfoShelf
Change the database server or database for a project	Configure DBDK Visual C++ Add-In dialog box

Important: Before you can run the **Register DataBlade Module** command, you must install the DataBlade module by using the **Upload DataBlade Module** and **Upload DataBlade SQL Scripts** commands, and the database you specified in the Configure DBDK Visual C++ Add-In dialog box must exist.

Debugging a DataBlade Module

Debugging a DataBlade module is usually an iterative process, repeated many times until the code is completely debugged. The "Creating a Simple User-Define Routine" exercise in the DBDK InfoShelf tutorial guides you through this process.

The debugging process on a local database server has the following general steps:

1. Open the **project.dsw** file in Visual C++. You can do this in BladeSmith by clicking the **MSDev** button on the Generate DataBlade dialog box or by choosing **Tools > MSDev**.

   The DBDK Visual C++ Add-In toolbar should be present in the Visual C++ program if you installed DBDK after you installed Visual C++. If it is not present, you must add it manually before you continue with the next step. See "Manually Loading the Add-In" on page 10-5 for instructions.
2. The DBDK Visual C++ Add-In prompts you to configure the session for your new DataBlade project. Click **Yes** to select a local server.
3. Specify the project database server, database, and, optionally, the SQL script to initialize your database.

   See "Specifying Properties for a Project" on page 10-5 for more information.
4. Set appropriate breakpoints in one of the source code files.

   See "Setting Breakpoints" on page 10-6 for instructions.
5. Open the appropriate unit test file in Visual C++ and edit it to add appropriate SQL.

   See "Editing Unit Test Files" on page 10-6 for more information.
6. Click the **Debug DataBlade Module** button.
7. If you need to specify an executable file for the debugging session, the Executable For Debug Session dialog box will prompt you to do so. Use the browse button to select **%INFORMIXDIR%\bin\oninit.exe**.
8. If a dialog box appears, warning that **oninit.exe** does not have debugging information, click **OK** to begin debugging.

The debugger runs until the first breakpoint.

9. To resume debugging, choose **Debug > Go** from the Visual C++ menu bar.

   When you pass all breakpoints and all routines return, IfxQuery displays the SQL results in your default browser.
10. If necessary, edit and compile the source code.
11. Repeat the procedure, as necessary.

The following sections describe some of these steps.

Important: If you attempt to start or stop the database server with the Services dialog box of the Control Panel during debugging, you receive an error. When the add-in starts the database server attached to the debugger, the database server does not run as a Windows service. To stop the database server, shut down the debugger.

Manually Loading the Add-In

The DBDK Visual C++ Add-In toolbar should be present in the Visual C++ program if you installed DBDK after you installed Visual C++. If it is not present, you must add it manually.

To manually load the Visual C++ Add-In:

1. Close your **project.dsw** file.
2. Choose **Tools > Customize**.
3. On the **Add-Ins and Macro Files** page of the Customize dialog box, check the box for **DBDKAddIn.1**.

   If the box for **DBDKAddIn.1** is already checked, uncheck it, close the Customize dialog box, and then repeat Steps 2 and 3.
4. Click **Close**.
5. Open your **project.dsw** file.

Specifying Properties for a Project

To debug a project, each DataBlade module project must have an associated database server and database.

When you first open a DataBlade module project in Visual C++, the Configure DBDK Visual C++ dialog box appears, prompting you to choose a database server and database. If you choose a local database server, you can use any of the add-in commands. If you choose a remote database server, you can use only the **Upload DataBlade Module**, **Upload DataBlade SQL Scripts**, and **Register DataBlade Module** commands.

You can choose an existing database from the **DBDK Database** list or, if you are using a local database server, type in a new database name. IfxQuery creates the database you specify if it does not exist when you run the **Debug DataBlade Module** command. If the database server you specified is a remote server, you must choose an existing database name.

You can also specify an SQL file to initialize your test database for the project in the **Initialize Database File** field. You can use the generated **Setup.sql** file in the **src** directory as your initialization file after you add SQL statements to it. See "Editing Unit Test Files" on page 10-6 for a description of the **Setup.sql** file.

You can change the properties of a project at any time by clicking **Configure DBDK Visual C++ Add-In** button and completing the corresponding dialog box.

Setting Breakpoints

Before you start the debugger, set breakpoints in your source code.

To set breakpoints with Visual C++:

1. Open a source code file. You can do this by double-clicking a routine under the **Globals** node in the Class view.
2. Right-click the line of code for which you want to set a breakpoint.
3. Choose **Insert > Breakpoint**.

Editing Unit Test Files

Before you start debugging, edit the unit test files to add the SQL statements necessary to debug your DataBlade module.

When you generate unit tests for a DataBlade module, BladeSmith generates the files listed in the following table in the **src\tests** directory.

Test Name	Purpose
Setup.sql	Optionally initializes the database. You can add SQL statements to create and populate the tables necessary for your debugging tests. If you specify this file as your initialization file in the Configure DBDK Visual C++ Add-In dialog box, IfxQuery automatically runs this file after it creates a new database.
Routine.sql	Tests the user-defined routine. You can add SQL statements or modify the sample data for the routine. Use this file if you are debugging **udr.c**. IfxQuery runs this file if you click **Debug DataBlade Module** with this file in the active window.
Opaque.sql	Tests the support routines for each opaque data type. You can add SQL statements or modify the sample data for each support routine. Use this file if you are debugging **Opaque.c** or **OpaqueServer.cpp**. IfxQuery runs this file if you click **Debug DataBlade Module** with this file in the active window.
Cleanup.sql	Optionally deletes and drops tables and data in your test database. IfxQuery runs this file if you click **Debug DataBlade Module** with this file in the active window.

When you edit unit test files, add SQL statements in the areas marked with `TEST` comments. This ensures that your statements are merged when you regenerate unit tests with BladeSmith.

Performing Functional Tests on DataBlade Modules

When you have completed the code for your DataBlade module and finished debugging it, you should run functional tests to validate it.

When you generate functional tests, BladeSmith creates a set of files that include shell scripts and SQL scripts for testing extended data types, user-defined routines,

and casts. By default, these files are created in the **functest** subdirectory of the directory containing the BladeSmith project file.

Functional tests are generated only for the DataBlade module objects for which you enter test data in your BladeSmith project. See "Adding Functional Test Data" on page 4-36 for information about entering test data.

The test scripts are created to run in a UNIX shell. Therefore, you must install a UNIX-compatible toolkit on your Windows computer: for example, MKS Toolkit. For information about functional tests, see "Functional Test Overview" on page 9-7.

Although functional tests are meant to be executed after development of the DataBlade module is complete, functional testing can be an iterative process, repeated several times until the code passes all the tests. The testing process has the following general steps:

1. In Visual C++, build the **project.bld** file.

   See "Compiling on Windows" on page 5-34 for instructions.
2. Create a **project** directory under the **%INFORMIXDIR%\extend** directory for your database server.
3. Install your DataBlade module. To do this, run the **Upload DataBlade Module** and **Upload SQL Scripts** commands on the add-in or manually copy the necessary files (see "Installing a DataBlade Module" on page 9-3 for instructions).
4. Register your DataBlade module. To do this, run the **Register DataBlade Module** command on the add-in or use BladeManager (see the "Registering a DataBlade Module" on page 9-4 for instructions).
5. Execute the functional tests from a UNIX shell using MKS Toolkit.

   See "Executing Functional Tests" on page 9-10 for instructions.
6. Regenerate functional tests in BladeSmith if you change any of your test data. If you change the definition of any of your DataBlade module objects, regenerate source code and functional tests in BladeSmith.
7. Edit the source code (if necessary).
8. Repeat the procedure, as necessary.

Chapter 11. Using BladePack

In This Chapter

Refer to the online help for detailed descriptions of the BladePack user interface and screen elements.

BladePack creates installation packages for DataBlade modules and other software products. BladePack provides a visual representation of an installation package, allowing you to add files to the installation package and to customize the

© Copyright IBM Corp. 1996, 2008

installation in a variety of ways. When the installation package is defined and customizations are completed, BladePack creates the installation package in a build area.

Prerequisite Tasks

Before you package your DataBlade module code, complete these tasks:

- Create your DataBlade module in BladeSmith.

  See "Creating DataBlade Module Objects" on page 4-8 for instructions.
- Generate source, SQL, and packaging files in BladeSmith.

  See "Generating Files" on page 4-40 for instructions.
- Complete your source code.

  See Chapter 5, "Programming DataBlade Module Routines in C," on page 5-1, or Chapter 6, "Creating ActiveX Value Objects," on page 6-1, for instructions.
- Build your DataBlade module shared object or dynamic link library.

  See "Compiling DataBlade Module Code" on page 5-33 or "Compiling a Windows Server Project" on page 6-5 for instructions.

BladePack Overview

BladePack produces installation packages for installing products on UNIX and Microsoft Windows platforms. BladePack can create a simple directory tree containing files to be installed or an installation that includes an interactive user interface.

On UNIX platforms, an interactive installation includes **install** and **uninstall** shell scripts. On Windows, an interactive installation includes the **Setup** program created with InstallShield and, for InstallShield 3.1, the **Uninstall** program.

Important: You must have an InstallShield Professional 3.1 or 5.1 license to create an InstallShield installation for Windows. You specify the directory and version of InstallShield while you install the Informix DataBlade Developers Kit.

The files in an installation package can be divided into separate components, subcomponents, and shared components. You must define at least one component for an installation package. You can designate the components to include in typical, compact, or custom installations. You can also allow users to customize their installation by choosing the components they want to install.

For example, in addition to the required shared object file and SQL scripts, a DataBlade module can include example files and online help files. You can place these additional files into separate components that are included in a typical installation but excluded from a compact installation.

This section contains the following subsections:

BladePack Projects

BladePack organizes information into projects. Each project is controlled by a product file (**project.prd**), which contains entries for the component file (**project.cmp**), bill of materials file (**project.bom**), and string file (**project.str**). If you are packaging a DataBlade module created by BladeSmith, the **Generate Packaging** option creates these files in the **install** directory. The following table describes these files.

Package File	Description
project.bom	A bill of materials file. This file contains an entry for each file to be installed. The entry includes the path to the source file and the path where the file will be installed.
project.cmp	Lists the main components and subcomponents in the installation package.
project.prd	The main product file that you open with BladePack. This file lists other files that define the installation package. Initially, this file contains entries for the **.bom**, **.cmp**, and **.str** files. Add README files using BladePack.
project.str	Defines character strings used in the installation.

Important: Do not edit the generated installation package files. Instead, use BladeSmith to update the installation package files after you have added or removed DataBlade module objects in the project file.

When you build an installation package, you can include several BladePack projects. For example, you can include DataBlade modules that facilitate similar financial calculations into a single installation package.

If you include standard items in each of your installations, create a separate project for these items and include this project in every installation. For example, you can put registry changes required by all DataBlade modules in a standard project file. Include these changes in a component that is always installed.

BladePack Online Help

BladePack online help provides overview and detailed reference information for BladePack.

The "About BladePack" section contains topics that provide an overview of BladePack and installation packages.

The "BladePack Interface" section describes BladePack menus, project view pages, item view pages, dialog boxes, and the Build Installation wizard.

The "BladePack Procedures" section contains instructions for working with projects, establishing the content of the installation package, organizing components, and setting up the installation package interface.

BladePack Windows

The BladePack project window is divided into two panes. The *project view* pane displays the overall structure of the installation package. The project view contains tabbed views of the contents of the installation package arranged in hierarchical trees.

The *item view* pane contains detailed information about the object selected in the project view. You use the project view to add objects to the installation package and to organize the structure of the installation package. You use the item view to enter details about objects in the installation package. Figure 11-1 shows a BladePack project window.

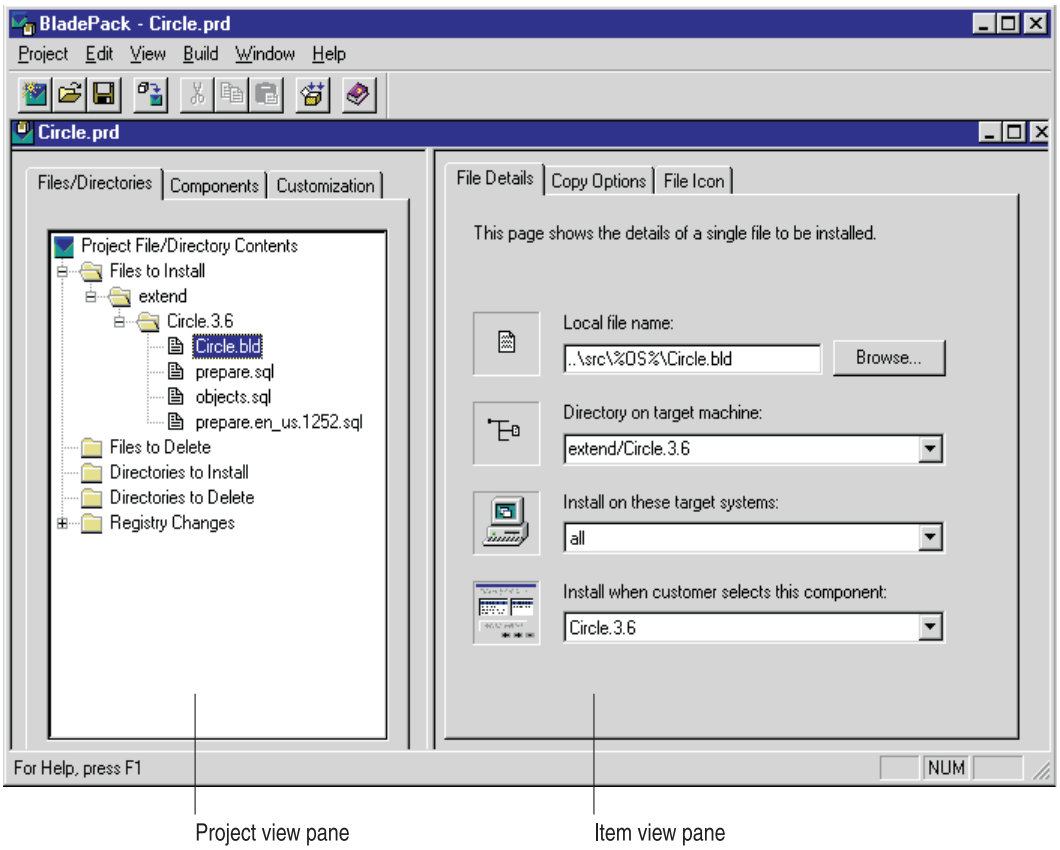


Figure 11-1. BladePack Project Window

Project View

The project view has three tabbed pages: **Files/Directories**, **Components**, and **Customization**.

Each page in the project view presents a hierarchical tree of the contents of the package. To expand or collapse a folder, click the expander button next to the folder or double-click the folder.

Important: The options you have for your installation package vary according to whether you are building a package for a UNIX installation, an InstallShield 3.1 installation for Windows, or an InstallShield 5.1 installation for Windows. To determine which options are valid for your installation package, see the appropriate section on packaging.

Files/Directories Page: When you click the **Files/Directories** tab in the project view, BladePack displays files and directories to install and files and directories to remove. BladePack also displays registry changes for Windows installations.

The **Files to Install**, **Files to Delete**, **Directories to Install**, and **Directories to Delete** folders are organized as trees that match the directories on the target computer for the installation.

The **Registry Changes** folder contains entries for the Windows registry.

Components Page: The **Components** page displays the component organization of the installation package. An installation package can have components, subcomponents, and shared components. You can create a component, subcomponent, or shared component and then drag files and directories into it. Subcomponents and shared components are subordinate to components. Shared components are useful for files that are included in more than one component.

You can organize the installation package into components to make it possible for the customer who installs the package to select portions of the DataBlade module in the **Select Components to Install** screen. For example, if your DataBlade module includes examples, you can create a component of the DataBlade module called **Examples** and then create subcomponents for each example. Then customers can choose which examples to install with the DataBlade module. Shared components do not appear in the **Select Components to Install** screen; they are installed if the component to which they belong is installed.

However, you can also ship your DataBlade module as a single component that contains all of the files. You do not have to organize your installation package into subcomponents and shared components.

Customization Page: The **Customization** page displays information that can be customized for the installation package.

The **Custom DLL Routine**, **Custom DLL Dialog**, and **Custom Program** folders contain custom routines, dialog boxes, and executable programs for InstallShield installations on Windows platforms and executable programs to run from within interactive installations on Windows or UNIX platforms. You control when routines execute by specifying the execution sequence.

Tip: To add dialog boxes and routines to your installation package, create them using Microsoft Visual C++ and then add them to a dynamic link library (DLL). For examples, see the directory **%INFORMIXDIR%\dbdk\setup\example**.

The **Readme Files** folder contains files that you want to place unpacked on the first diskette of an InstallShield installation.

The **Support Files** folder contains a list of files that are available during the installation but are not installed with the product.

Item View

The item view displays one or more tabbed pages, depending on the object you select in the project view.

If you select a folder in the project view, the first tab in the item view contains a list of the folder contents.

If you select an object, such as a file, in the project view, the first tab in the item view displays details about the object. The information displayed and the names of

the tabs depend on the type of object you are viewing. Some objects have other tabs, containing supporting information for that object.

Much of the information in the item view is editable. For example, if you add a component, you type its name in the name field that appears in the item view. Editable fields have a white background. Fields that cannot be edited have a gray background.

Registry Keys for Windows

When you install the Informix DataBlade Developers Kit, one of the installation screens allows you to specify InstallShield support, if you have InstallShield installed on your computer. The version and directory of InstallShield you choose on that screen determines the value of the following registry keys:

- **AlwaysUseInstallShield5.** Sets the version:
  - 0 indicates that you have InstallShield 3.1 or that you do not have InstallShield 5.1 on your computer.
  - 1 indicates that you have InstallShield 5.1 on your computer.
- **IShieldDir.** Sets the directory.

You can reset the version and directory of InstallShield by editing these registry keys in the registry.

The **AlwaysUseInstallShield5** key is in the following registry directory:

hkey_local_machine\software\Informix\BladePack

The **IShieldDir** key is in the following registry directory:

hkey_current_user\Environment

Packaging for UNIX Installations

To package your DataBlade module, you add content to a BladePack project, assign components, customize the installation procedure, and build the package.

Important: You cannot use all BladePack options when you create an installation package for UNIX. You can only use the options mentioned in this section.

For information on packaging DataBlade modules for InstallShield 3.1, see "Packaging for InstallShield 3.1 Installations" on page 11-12. For information on packaging DataBlade modules for InstallShield 5.1, see "Packaging for InstallShield 5.1 Installations" on page 11-20.

To build an installation package with BladePack:

1. Open a project file in one of the following ways:
   - To open the ***project*.prd** file for a project created with BladeSmith, choose **Project > Open** or launch BladePack from the BladeSmith **Tools** menu while the project is open.
   - For other projects, BladePack, choose **Project > New** to create a new BladePack project.
2. Define the content of your product, including files, directories, and registry changes.

3. Define and assign installation components.
4. Define optional customizations.
5. Build the installation package.
6. Transfer files from the build area to installation media.

Establishing Content

You can add these objects to your product in BladePack:

- Files and directories to install

  For a DataBlade module, the minimum you need is the **project.bld** or **project.jar** file and the SQL script files. These files are automatically added to the **Files to Install** folder when you open a **project.prd** file.

  In addition, consider adding documentation, help, applications, and other files to support your DataBlade module.
- Files and directories to delete

  If you are packaging an upgrade, you might need to specify files or directories to delete in the old installation.

To add a file or directory:

1. Choose **Edit > Insert > object**, where *object* is **File to Install**, **File to Delete**, **Directory to Install**, or **Directory to Delete**.

   The object appears on the **Files/Directories** page.
2. Specify the properties of the object on the **Details** and other pages in the item view.

The following sections describe the properties of the objects on the **Files/Directories** page.

Files and Directories to Be Installed or Deleted

The following table lists properties you define when you add files and directories to install or delete.

Property	Description
Local name	The local name of the file or directory to be installed or deleted. Choose **Browse** to select a file from the Open dialog box. You can have multiple operating system-specific files to install. See "Local Paths for Files for Multiple Operating Systems" on page 11-8 for more information.
Target directory	For files and directories to install only. The directory in which the file or directory is installed. See "Specifying a Target Directory" on page 11-8 for more information.
Target operating system	The operating system on which to install the file or directory.
Component	The component, subcomponent, or shared component to which the file or directory is assigned.

	See "Assigning to Components" on page 11-10 for more information on assigning to components.
File copy options	For files and directories to install only. Options for copying files, including whether the file is installed only if it has the same or later date or version than the existing file. The default is **None**.

Local Paths for Files for Multiple Operating Systems: If you have files that are operating-system-specific, put them in a directory structure that is the same except for one directory, which is named for the operating system. When you add the file to your BladePack project, replace the directory named for the operating system with %OS%.

For example, if you compile your C or C++ DataBlade module on Sun Solaris and Windows, you have two **project.bld** files, one in each of these directories:

- **project/src/c/Solaris-sparc**
- **project\src\c\WinNT-i386**

Add the **project/src/c/Solaris-sparc/project.bld** file to the **Files to Install** folder and then replace **Solaris-sparc** with %OS%. When you build the BladePack project for Sun Solaris and Windows, BladePack adds the appropriate **project.bld** file to each project.

For a Java DataBlade module, you have only one version of the **project.jar** file, which is in the **project/src/java** directory.

Specifying a Target Directory: For UNIX, the only option in the list for the **Directory on target machine** field is **__home**. This is the directory the installer chooses during the installation process. By default this directory is **$INFORMIXDIR**.

For the DataBlade module files (**project.bld** or **project.jar** and the SQL scripts), you should specify the **extend/project** directory as the target directory under the **$INFORMIXDIR** directory.

Managing Components

BladePack allows you to organize your product installation package into three layers: component, subcomponent, and shared component. To see the component hierarchy for your product, click the **Components** tab in the project view.

Organizing an installation package into a component structure allows you to define Typical, Compact, and Custom installations. You specify whether each component or subcomponent is included in the Typical and Compact installations, and whether it is initially selected when users choose the Custom installation.

Use a shared component for those portions of your product that are shared by more than one component. A shared component is always installed with the subcomponent with which it is associated.

During a Custom installation, users can choose to install any components or subcomponents. When you mark a component Custom, the component is initially selected. The user can choose to include or exclude any components, except shared components, from the installation.

In most cases, the component level is sufficient to create Typical, Compact, and Custom installation options. For example, suppose you have created the following components (and no subcomponents) in your installation package and marked them as shown:

- **DataBlade module.** Typical, Compact, Custom.
- **Help.** Typical.
- **Examples**. Typical.
- **Debugging Support.** Custom.

In this scheme, users install the DataBlade module, help files, and examples if they choose the Typical installation. If they choose the Compact installation, they install the DataBlade module only. If they choose the Custom installation, the DataBlade module and debugging support are preselected. They can choose to add help and examples.

To create a component:

1. Choose **Edit > Insert > Component**.
2. Complete the properties on the **Component Details** page in the item view.

To create a subcomponent:

1. Select the component to which you want the subcomponent to be subordinate.
2. Choose **Edit > Insert > Subcomponent**.
3. Complete the properties on the **Component Details** page in the item view.

To create and copy a shared component:

1. Select the subcomponent to which you want the shared component to be subordinate.
2. Choose **Edit > Insert > Shared Component**.
3. With the shared component still selected, choose **Edit > Copy**.
4. Select another component to which you want to add the shared component.
5. Choose **Edit > Paste**.

Component Properties

The following table lists the properties of components and subcomponents you define when you create them.

Property	Description
Name	The name of the component or subcomponent that appears in the left column on the **Select Installation Components** screen during a custom installation
Description	The description of the component or subcomponent that appears in the right column when the item is selected on the **Select Installation Components** screen during a custom installation
Inclusion	What type of install the component or subcomponent is included in: • Compact • Typical • Custom

Shared components have one property: an identifier that is assigned by BladePack. You can edit the identifier; it can be an alphanumeric string up to 128 characters.

Make sure it is unique among shared components. If you change an identifier, be sure to update it for every instance of that shared component.

Assigning to Components

You must assign every item on the **Files/Directories** and **Customization** pages to a component, subcomponent, or shared component. If you try to build the project with unassigned items, the build fails and you receive an error message telling you which item is not assigned to a component.

Initially, all files and directories that appear on the **Files/Directories** page are listed under the **Unassigned Files and Directories** folder on the **Components** page. Custom extensions are not shown on the **Components** page.

To assign an item to a component, use one of these methods:

- On the **Components** page, drag an item out of the **Unassigned Files and Directories** folder into the folder of the correct component. This process is not valid for custom extensions.
- On the **Details** page for the item in the item view, select a component from the **Install when customer selects this component** list. You must have already defined the component.

Customizing the Installation

Custom extensions for the installation program are optional. Your customization options depend on your operating system.

For UNIX installation packages, you can add custom programs to call from the installation program and README files.

In addition to adding the custom programs in their respective folders, you must also add the file containing the custom extension to the **Support Files** folder. However, if you have more than one program in a file, you need only add that file to the **Support Files** folder once.

To add a custom program:

1. Choose **Edit > Insert > Custom Program**.
2. Complete the **Details** page in the item view.
3. Choose **Edit > Insert > Support File**.
4. Type the path and filename of the file containing the custom routine, dialog box, or program or click **Browse** to select the file from the Open dialog box.

The following table lists the properties of custom programs you specify when you add them to your installation project.

Property	Description
Name	The filename of the program. Type the name or click **Browse** to select the file in the Open dialog box.
ID or command line arguments	The command line arguments you want to use for the program during the installation process.
Target operating system	The operating system on which the custom extension runs.

Component	The component, subcomponent, or shared component to which the file or directory is assigned. See "Assigning to Components" on page 11-10 for more information on assigning to components.
When to run	When the custom extension is executed during the installation procedure: • Before the installation program begins • Before the project files are copied • After the project files are copied • Before the installation program exits

To add a README file:

1. Choose **Edit > Insert > Readme File**.
2. Type the path and filename or click **Browse** to select the file from the Open dialog box.

Building the Installation

When the content and organization of your installation package are complete, build and test it.

To build, choose **Build > Build Installation**. The Build Installation wizard is launched and prompts you for the following information:

- Installation type: interactive or file tree
  See "Installation Type," next, for more information.
- Platform for which to build the installation package
  Choose one from the list. Build a separate package for every platform.
- The target directory in which to build the installation
  This can be any directory. By default it is the **project/install** directory.
- List of BladePack **project.prd** files to include in the installation package
  You can bundle more than one project in a single installation.

Warning: If the combined length of the path and filename of any file is longer than 255 characters, the build fails. This is due to a Windows limitation. To solve this problem, you can select a shorter staging directory.

Installation Type

BladePack creates a directory structure in the target directory and copies files into the tree. When you build an interactive installation for a UNIX platform, BladePack includes **install** and **uninstall** shell scripts.

When you build a file tree installation, BladePack creates the file tree specified in the project in the target directory. A file tree build is useful for debugging the BladePack project.

After you successfully build an interactive installation, the target directory contains the subdirectories described in the following table.

Directory	Description
cdrom	Contains an image of the installation package that can be

transferred to distribution media. DataBlade module developers can rename this directory to the name of the DataBlade module before creating a **.tar** file.

This directory contains an **install** shell script and other files that the **install** script uses during installation.

support — Contains copies of files to support the installation, such as project files. Its contents are not distributed with your installation package.

folder_tree — Contains a directory tree containing the contents of all of the directories to be included in the installation package. The contents of this directory are not distributed with your installation package.

tree — The root of a directory tree containing files to be included in the installation package. The **tree** directory can be used to debug problems in the installation package.

The contents of this directory are not distributed with your installation package.

Creating Distribution Media

To ensure that customers can install DataBlade modules and other IBM Informix products using common instructions, the product you distribute must conform to the IBM Informix DataBlade module installation standard. BladePack creates an interactive installation that ensures a consistent user interface.

Important: If you create a *UNIX* **.tar** file, rename the **cdrom** directory to the name of your product before you copy the directory to media or the release area. For example, for the Circle DataBlade module, rename the **cdrom** directory to **circle**. (Do not include the version number in the directory name.)

Copy the renamed directory and its contents to the media or into the archive file. This makes it possible to distribute multiple products with their own installations on a CD-ROM or tape.

For example, to install the Circle DataBlade module from CD-ROM, the installer mounts the CD-ROM, changes to the **circle** subdirectory, and executes the install script.

To install the Circle DataBlade module from a file named **circle3.6.tar**, retrieved through a local network or the Internet, the installer extracts the file into a temporary directory, changes to the **circle** subdirectory, and executes **install**. When the installation has finished, the **circle** subdirectory can be removed.

Tip: BladePack does not compress **.tar** files. If you want to distribute your DataBlade module as a compressed file, you must compress it yourself.

Packaging for InstallShield 3.1 Installations

To package your DataBlade module, you add content to a BladePack project, assign components, customize the installation procedure, and build the package.

For information on packaging DataBlade modules for UNIX, see "Packaging for UNIX Installations" on page 11-6. For information on packaging DataBlade modules for InstallShield 5.1, see "Packaging for InstallShield 5.1 Installations" on page 11-20.

To build an installation package with BladePack:

1. Open a project file:
   - For DataBlade modules created using BladeSmith, open the **project.prd** file by choosing **Project > Open** or launch BladePack from the BladeSmith **Tools** menu while the project is open.
   - For other products, create a new BladePack project by choosing **Project > New**.
2. Define the content of your product, including files, directories, and registry changes.
3. Define and assign installation components.
4. Define optional customizations.
5. Build the installation package.
6. Transfer files from the build area to installation media.

Establishing Content

You can add these objects to your product in BladePack:

- Files and directories to install

  For a DataBlade module, you need the **project.bld** or **project.jar** file and the SQL script files. If you have an ActiveX client project, you also need the **project.dll** file. These files are automatically added to the **Files to Install** folder when you open a **project.prd** file.

  In addition, consider adding documentation, help, applications, and other files to support your DataBlade module.

  If you have an ActiveX client implementation, consider including CLSID (class identifier) and IID (interface identifier) information by including C++ client support library files (see "Support Library Files" on page A-2) in the installation package.
- Files and directories to delete

  If you are packaging an upgrade, you might need to specify files or directories to delete in the old installation.
- Registry changes

  You might have to specify registry changes for your DataBlade module.

To add a file or directory or a change to the registry:

1. Choose **Edit > Insert > object**, where **object** is **File to Install**, **File to Delete**, **Directory to Install**, **Directory to Delete**, or **Change to Registry**.

   The object appears on the **Files/Directories** page.
2. Specify the properties of the object on the **Details** and other pages in the item view.

The following sections describe the properties of the objects on the **Files/Directories** page.

Files and Directories to Be Installed or Deleted

The following table lists properties you define when you add files and directories to install or delete.

Property	Description
Local name	The local name of the file or directory to be installed or deleted. Choose **Browse** to select a file from the Open dialog box.

	You can have multiple operating-system-specific files to install. See "Local Paths for Files for Multiple Operating Systems" on page 11-14 for more information.
Target directory	For files and directories to install only. The directory in which the file or directory is installed. See "Specifying a Target Directory" on page 11-15 for more information.
Target operating system	The operating system on which to install the file or directory.
Component	The component, subcomponent, or shared component to which the file or directory is assigned. See "Assigning to Components" on page 11-17 for more information on assigning to components.
File copy options	For files and directories to install only. Options for copying files, including whether the DLL is installed only if it has the same or later date or version than the existing DLL. The default is **None**.
File sharing options	For files and directories to install only. Whether and how a file can be shared. The default is **None**. See "File Sharing Options" on page 11-15 for more information.
Icon options	For files to install only. Whether the file has an associated icon and information about that icon. The icon appears in the program group. Icons are typically used for applications, read me files, or help files; DataBlade modules do not require icons. The default is no icon.

Local Paths for Files for Multiple Operating Systems: If you have files that are operating-system-specific, put them in a directory structure that is the same except for one directory, which is named for the operating system. When you add the file to your BladePack project, replace the directory named for the operating system with `%OS%`.

For example, if you compile your C or C++ DataBlade module on Sun Solaris and Windows, you have two **project.bld** files, one in each of these directories:

- **project/src/c/Solaris-sparc**
- **project\src\c\WinNT-i386**

Add the **project\src\c\WinNT-i386\project.bld** file to the **Files to Install** folder and then replace **WinNT-i386** with `%OS%`. When you build the BladePack project for Sun Solaris and Windows, BladePack adds the appropriate **project.bld** file to each project.

For a Java DataBlade module, you have only one version of the **project.jar** file, which is in the **project/src/java** directory.

Specifying a Target Directory: You have the following options in the list for the **Directory on target machine** field:

- **__home.** The directory the installer chooses during the installation process. By default this directory is **$INFORMIXDIR**.
- **__system.** The Windows **system** directory.
- **__windows.** The Windows directory.

The target directory for the DataBlade module files (**project.bld** and the SQL scripts) should be the **extend\project** directory under the **$INFORMIXDIR** directory.

File Sharing Options: For files in the **Files to Install** folder, choose one of these file sharing options from the **Copy Options** page:

- **File is system shared DLL.** Indicates that the file can be used by more than one program on the target computer. The file is marked to prevent it from being removed in an uninstallation process.
- **File may be in use on target system.** Indicates that the file can be in use when the program is installed. If the file exists on the target computer and is in use during the installation process, the installation continues, but the computer must be rebooted before the program is run. In this case, the installation program displays the **Setup Complete, Reboot Required** screen.
- **None.** Default. Indicates that your files are not shared and cannot be in use during the installation process. In this case, the installation program displays the **Setup Complete** screen without a prompt to reboot the computer.

Registry Changes

You can add entries to the Windows registry for the initialization and configuration of your DataBlade module and its associated programs.

Refer to your Microsoft Developer Studio documentation for information about the registry.

To add registry changes:

1. Choose **Edit > Insert > Change to Registry**.
2. Complete the **Registry Changes Details** page in the item view.

The following table lists the properties you define when adding registry changes.

Property	Description
Registry hive	The standard primary registry keys under which you want to add a key: • HKEY_CLASSES_ROOT • HKEY_CURRENT_USER • HKEY_LOCAL_MACHINE
Registry path	The key you want to add, expressed as a path.
Key name	The name of the key.
Key value	The value of the key.

Component — The component, subcomponent, or shared component to which the file or directory is assigned.

See "Assigning to Components" on page 11-17 for more information on assigning to components.

Managing Components

BladePack allows you to organize your product installation package into three layers: component, subcomponent, and shared component. To see the component hierarchy for your product, click the **Components** tab in the project view.

Organizing an installation package into a component structure allows you to define Typical, Compact, and Custom installations. You specify whether each component or subcomponent is included in the Typical and Compact installations, and whether it is initially selected when users choose the Custom installation.

Use a shared component for those portions of your product that are shared by more than one component. A shared component is always installed with the subcomponent with which it is associated.

During a Custom installation, users can choose to install any components or subcomponents. When you mark a component Custom, the component is initially selected. The user can choose to include or exclude any components, except shared components, from the installation.

In most cases, the component level is sufficient to create Typical, Compact, and Custom installation options. For example, suppose you have created the following components (and no subcomponents) in your installation package and marked them as shown:

- **DataBlade module.** Typical, Compact, Custom.
- **Help.** Typical.
- **Examples**. Typical.
- **Debugging Support.** Custom.

In this scheme, users install the DataBlade module, help files, and examples if they choose the Typical installation. If they choose the Compact installation, they install the DataBlade module only. If they choose the Custom installation, the DataBlade module and debugging support are preselected. They can choose to add help and examples.

To create a component:

1. Choose **Edit > Insert > Component**.
2. Complete the properties on the **Component Details** page in the item view.

To create a subcomponent:

1. Select the component to which you want the subcomponent to be subordinate.
2. Choose **Edit > Insert > Subcomponent**.
3. Complete the properties on the **Component Details** page in the item view.

To create and copy a shared component:

1. Select the subcomponent to which you want the shared component to be subordinate.
2. Choose **Edit > Insert > Shared Component**.

3. With the shared component still selected, choose **Edit > Copy**.
4. Select another component to which you want to add the shared component.
5. Choose **Edit > Paste**.

Component Properties

The following table lists the properties of components and subcomponents you define when you create them.

Property	Description
Name	The name of the component or subcomponent that appears in the left column on the **Select Installation Components** screen during a custom installation
Description	The description of the component or subcomponent that appears in the right column when the item is selected on the **Select Installation Components** screen during a custom installation
Inclusion	What type of install the component or subcomponent is included in: • Compact • Typical • Custom

Shared components have one property: an identifier that is assigned by BladePack. You can edit the identifier; it can be an alphanumeric string up to 128 characters. Make sure it is unique among shared components. If you change an identifier, be sure to update it for every instance of that shared component.

Assigning to Components

You must assign every item on the **Files/Directories** and **Customization** pages to a component, subcomponent, or shared component. If you try to build the project with unassigned items, the build fails and you receive an error message telling you which item is not assigned to a component.

Initially, all files and directories that appear on the **Files/Directories** page are listed under the **Unassigned Files and Directories** folder on the **Components** page. Custom extensions are not shown on the **Components** page.

To assign an item to a component, use one of these methods:

- On the **Components** page, drag an item out of the **Unassigned Files and Directories** folder into the folder of the correct component. This process is not valid for custom extensions.
- On the **Details** page for the item in the item view, select a component from the **Install when customer selects this component** list. You must have already defined the component.

Customizing the Installation

Custom extensions for the installation program are optional. For InstallShield 3.1 installation packages, you can add these custom extensions:

- Routines to call from the installation program
- InstallShield dialog boxes
- Programs to call from the installation program
- README files for the installation program

Adding Custom Extensions

In addition to adding the custom routines, dialog boxes, and programs in their respective folders, you must also add the file containing the custom extension to the **Support Files** folder. However, if you have more than one routine, dialog box, or program in a file, you need only add that file to the **Support Files** folder once.

To add a custom routine, dialog box, or program:

1. Choose **Edit > Insert > Item**, where **Item** is **Custom DLL Routine**, **Custom DLL Dialog**, or **Custom Program**.
2. Complete the **Details** page in the item view.
3. Choose **Edit > Insert > Support File**.
4. Type the path and filename of the file containing the custom routine, dialog box, or program or click **Browse** to select the file from the Open dialog box.

To add a README file:

1. Choose **Edit > Insert > Readme File**.
2. Type the path and filename or click **Browse** to select the file from the Open dialog box.

The following table lists the properties of custom routines, dialog boxes, and programs you specify when you add them to your installation project.

Property	Description
Name	The filename of the routine, dialog box, or program. Type the name or click **Browse** to select the file in the Open dialog box.
ID or command line arguments	For a custom routine, the ID string is an identifier you can use to determine which routine to call if you have more than one routine in a single DLL. For a custom dialog box, the resource ID that you specified when creating it in Microsoft Developer Studio. For a custom program, the command-line arguments you want to use for the program during the installation process.
Target operating system	The operating system on which the custom extension runs.
Component	The component, subcomponent, or shared component to which the file or directory is assigned. See "Assigning to Components" on page 11-17 for more information on assigning to components.
When to run	When the custom extension is executed during the installation procedure: • Before the installation program begins • Before the project files are copied • After the project files are copied • Before the installation program exits

Building the Installation

When the content and organization of your installation package are complete, build and test it.

To build, choose **Build > Build Installation**. The Build Installation wizard is launched and prompts you for the following information:

- Installation type: interactive or file tree

  See "Installation Type" on page 11-19 for more information.
- Platform for which to build the installation package

  Choose one from the list. Build a separate package for every platform.
- Installation screen text

  See "Installation Screen Display Text" on page 11-20 for more information.
- The target directory in which to build the installation

  This can be any directory. By default it is the **project\install** directory.
- List of BladePack **project.prd** files to include in the installation package

  You can bundle more than one project in a single installation.

Warning: If the combined length of the path and filename of any file is longer than 255 characters, the build fails. This is due to a Windows limitation. To solve this problem, you can select a shorter staging directory.

Installation Type

BladePack creates a directory structure in the target directory and copies files into the tree. When you build an interactive installation package for Windows, BladePack calls InstallShield to process the files and create CD-ROM and diskette images.

When you build a file tree installation, BladePack creates the file tree specified in the project in the target directory. A file tree build is useful for debugging the BladePack project.

After you successfully build an interactive installation, the target directory contains the subdirectories described in the following table.

Directory	Description
tree	The root of a directory tree containing files to be included in the installation package. The **tree** directory can be used to debug problems in the installation package. This directory is compressed into a single archive called **files.z** in the **cdrom** directory.
cdrom	Contains an image of the installation package that can be transferred to distribution media. DataBlade module developers can rename this directory to the name of the DataBlade module before creating a **.tar** file. This directory contains **Setup.exe** and other files that support an InstallShield installation.
disk1, disk2...disk*n*	Contain files needed for an InstallShield installation, with files split to fit on 1.4 MB

	diskettes. The **disk1** directory contains **Setup.exe** and the files required to begin a diskette installation.
support	Contains copies of files to support the installation, such as project files, **.dll** files, and bitmap images. Its contents are not distributed with your installation package.
folder_tree	Contains a directory tree containing the contents of all of the directories to be included in the installation package. The contents of this directory are not distributed with your installation package.

Installation Screen Display Text

BladePack provides default text strings for the InstallShield installation wizard screens for Windows. You can override some of these text strings. For example, in the Select Installation Type wizard, you can change the text that appears next to the words Typical, Compact, and Custom to provide your own definitions for these three types of installations. When you save a BladePack project, BladePack saves any new string definitions in the appropriate string files.

Creating Distribution Media

To ensure that customers can install DataBlade modules and other IBM Informix products using common instructions, the product you distribute must conform to the IBM Informix DataBlade module installation standard. BladePack creates an interactive installation that ensures a consistent user interface.

To install a product from diskettes on a Windows platform, the installer executes the **Setup.exe** program on the first diskette. To create diskettes, copy the contents of the **disk1** ... **diskn** directories to formatted 1.4 MB diskettes.

To create all other types of media, use the **cdrom** directory in the build area.

To distribute multiple products with their own installations on a CD-ROM or tape, rename the **cdrom** directory to the name of the DataBlade module before you copy the directory and its contents to the media or into the archive file.

For example, to install the Circle DataBlade module from CD-ROM, the installer mounts the CD-ROM, changes to the **circle** subdirectory, and executes **Setup**.

Important: Put the **Setup.exe** program in a short path. If the combined length of the path and filename of any file is longer than 255 characters, the program will not execute. This is due to a Windows limitation.

Packaging for InstallShield 5.1 Installations

To package your DataBlade module, you add content to a BladePack project, assign components, customize the installation procedure, and build the package.

Important: You cannot use all BladePack options when you create an installation package for InstallShield 5.1. You can only use the options mentioned in this section.

For information on packaging DataBlade modules for UNIX, see "Packaging for UNIX Installations" on page 11-6. For information on packaging DataBlade modules for InstallShield 3.1, see "Packaging for InstallShield 3.1 Installations" on page 11-12.

To build an installation package with BladePack:

1. Open a project file:
   - For DataBlade modules created using BladeSmith, open the **project.prd** file by choosing **Project > Open** or launch BladePack from the BladeSmith **Tools** menu while the project is open.
   - For other products, create a new BladePack project by choosing **Project > New**.
2. Define the content of your product, including files, directories, and registry changes.
3. Define and assign installation components.
4. Define optional customizations.
5. Build the installation package.
6. Transfer files from the build area to installation media.

Establishing Content

You can add these objects to your product in BladePack:

- Files and directories to install

  For a DataBlade module, you need the **project.bld** or **project.jar** file and the SQL script files. If you have an ActiveX client project, you also need the **project.dll** file. These files are automatically added to the **Files to Install** folder when you open a **project.prd** file.

  In addition, consider adding documentation, help, applications, and other files to support your DataBlade module.

  If you have an ActiveX client implementation, consider including CLSID (class identifier) and IID (interface identifier) information by including C++ client support library files (see "Support Library Files" on page A-2) in the installation package.
- Registry changes

  You might have to specify registry changes for your DataBlade module.

To add a file or directory or a change to the registry:

1. Choose **Edit > Insert > object**, where **object** is **File to Install**, **Directory to Install**, or **Change to Registry**.

   The object appears on the **Files/Directories** page.
2. Specify the properties of the object on the **Details** and other pages in the item view.

The following sections describe the properties of the objects on the **Files/Directories** page.

Files and Directories to Be Installed

The following table lists properties you define when you add files and directories to install.

Property	Description

Local name	The local name of the file or directory to be installed. Choose **Browse** to select a file from the Open dialog box. You can have multiple operating-system-specific files to install. See "Local Paths for Files for Multiple Operating Systems" on page 11-22 for more information.
Target directory	The directory in which the file or directory is installed. See "Specifying a Target Directory" on page 11-22 for more information.
Target operating system	The operating system on which to install the file or directory.
Component	The component, subcomponent, or shared component to which the file or directory is assigned. See "Assigning to Components" on page 11-24 for more information on assigning to components.
Icon options	Whether the file has an associated icon and information about that icon. The icon appears in the program group. Icons are typically used for applications, README files, or help files; DataBlade modules do not require icons. The default is no icon.

InstallShield 5.1 does not support deleting files and directories, file copy options, or file sharing options.

Local Paths for Files for Multiple Operating Systems: If you have files that are operating-system-specific, put them in a directory structure that is the same except for one directory, which is named for the operating system. When you add the file to your BladePack project, replace the directory named for the operating system with `%0S%`.

For example, if you compile your C or C++ DataBlade module on Sun Solaris and Windows, you have two **project.bld** files, one in each of these directories:

- **project/src/c/Solaris-sparc**
- **project\src\c\WinNT-i386**

Add the **project\src\c\WinNT-i386\project.bld** file to the **Files to Install** folder, then replace **WinNT-i386** with `%0S%`. When you build the BladePack project for Sun Solaris and Windows, BladePack adds the appropriate **project.bld** file to each project.

For a Java DataBlade module, you have only one version of the **project.jar** file, which is in the **project/src/java** directory.

Specifying a Target Directory: You have the following options in the list for the **Directory on target machine** field:

- **__home.** The directory the installer chooses during the installation process. By default this directory is **$INFORMIXDIR**.
- **__system.** The Windows **system** directory.

- **__windows.** The Windows directory.

The target directory for the DataBlade module files (**project.bld** and the SQL scripts) should be the **extend/project** directory under the **$INFORMIXDIR** directory.

For InstallShield 5.1, BladePack puts files that are installed in different directories into different subcomponents; user files are put in the main component, while system files are put in a subcomponent.

Registry Changes

You can add entries to the Windows registry for the initialization and configuration of your DataBlade module and its associated programs.

Refer to your Microsoft Developer Studio documentation for information about the registry.

To add registry changes:

1. Choose **Edit > Insert > Change to Registry**.
2. Complete the **Registry Changes Details** page in the item view.

The following table lists the properties you define when you add registry changes.

Property	Description
Registry hive	The standard primary registry keys under which you want to add a key: • HKEY_CLASSES_ROOT • HKEY_CURRENT_USER • HKEY_LOCAL_MACHINE
Registry path	The key you want to add, expressed as a path.
Key name	The name of the key.
Key value	The value of the key.
Component	The component, subcomponent, or shared component to which the file or directory is assigned. See "Assigning to Components" on page 11-24 for more information on assigning to components.

Managing Components

BladePack allows you to organize your product installation package into three layers: component, subcomponent, and shared component. To see the component hierarchy for your product, click the **Components** tab in the project view.

Organizing an installation package into a component structure allows you to define Typical, Compact, and Custom installations. You specify whether each component or subcomponent is included in the Typical and Compact installations, and whether it is initially selected when users choose the Custom installation.

During a Custom installation, users can choose to install any components or subcomponents. When you mark a component Custom, the component is initially selected. The user can choose to include or exclude any components, except shared components, from the installation.

In most cases, the component level is sufficient to create Typical, Compact, and Custom installation options. For example, suppose you have created the following components (and no subcomponents) in your installation package, and marked them as shown:

- **DataBlade module.** Typical, Compact, Custom.
- **Help.** Typical.
- **Examples**. Typical.
- **Debugging Support.** Custom.

In this scheme, users install the DataBlade module, help files, and examples if they choose the Typical installation. If they choose the Compact installation, they install the DataBlade module only. If they choose the Custom installation, the DataBlade module and debugging support are preselected. They can choose to add help and examples.

To create a component:

1. Choose **Edit > Insert > Component**.
2. Complete the properties on the **Component Details** page in the item view.

To create a subcomponent:

1. Select the component to which you want the subcomponent to be subordinate.
2. Choose **Edit > Insert > Subcomponent**.
3. Complete the properties on the **Component Details** page in the item view.

Component Properties

The following table lists the properties of components and subcomponents you define when you create them.

Property	Description
Name	The name of the component or subcomponent that appears in the left column on the **Select Installation Components** screen during a custom installation
Description	The description of the component or subcomponent that appears in the right column when the item is selected on the **Select Installation Components** screen during a custom installation
Inclusion	What type of install the component or subcomponent is included in: • Compact • Typical • Custom

Assigning to Components

You must assign every item on the **Files/Directories** and **Customization** pages to a component, subcomponent, or shared component. If you try to build the project with unassigned items, the build fails and you receive an error message telling you which item is not assigned to a component.

Initially, all files and directories that appear on the **Files/Directories** page are listed under the **Unassigned Files and Directories** folder on the **Components** page. Custom extensions are not shown on the **Components** page.

To assign an item to a component, use one of these methods:

- On the **Components** page, drag an item out of the **Unassigned Files and Directories** folder into the folder of the correct component. This process is not valid for custom extensions.
- On the **Details** page for the item in the item view, select a component from the **Install when customer selects this component** list. You must have already defined the component.

Customizing the Installation

Custom extensions for the installation program are optional.

For InstallShield 5.1 installation packages, you can add README files for the installation program. However, after you export your project to InstallShield 5.1, you can add custom extensions to your project using the InstallShield 5.1 project wizard.

To add a README file:

1. Choose **Edit > Insert > Readme File**.
2. Type the path and filename or click **Browse** to select the file from the Open dialog box.

Building the Installation

When the content and organization of your installation package are complete, build and test it.

To build, choose **Build > Build Installation**. The Build Installation wizard is launched and prompts you for the following information:

- Installation type: interactive or file tree

  See "Installation Type" on page 11-25 for more information.
- Platform for which to build the installation package

  Choose one from the list. Build a separate package for every platform.
- Installation screen text

  See "Installation Screen Display Text" on page 11-26 for more information.
- The target directory in which to build the installation

  This can be any directory. By default it is the **project\install** directory.
- List of BladePack **project.prd** files to include in the installation package

  You can bundle more than one project in a single installation.

Warning: If the combined length of the path and filename of any file is longer than 255 characters, the build fails. This is due to a Windows limitation. To solve this problem, you can shorten the names of your components and subcomponents or select a shorter staging directory.

Installation Type

BladePack creates a directory structure in the target directory and copies files into the tree. When you build an interactive installation package for Windows, BladePack calls InstallShield to process the files and create CD-ROM and diskette images.

When you build a file tree installation, BladePack creates the file tree specified in the project in the target directory. A file tree build is useful for debugging the BladePack project.

When you build the installation package with BladePack, you specify a staging directory to hold the installation files. By default, the staging directory is the **project\install\InstallShield5.1\project** directory. In addition to putting the **Setup.exe** file in the staging directory, BladePack also puts it in the **project\install\cdrom** directory.

BladePack creates the InstallShield 5.1 project file, **project.ipr**, in the staging directory. To open the project file in InstallShield 5.1, double-click it.

After you successfully build an interactive installation, the staging directory contains the subdirectories described in the following table.

Directory	Description
cdrom	Contains an image of the installation package that can be transferred to distribution media. DataBlade module developers can rename this directory to the name of the DataBlade module before creating a **.tar** file. This directory contains **Setup.exe** and other files that support an InstallShield installation.
Component Definitions	Contains the component definitions. The **Default.cdf** file contains the component-to-subcomponent relationships.
File Groups	Contains all files to be installed. The **component.fgl** file describes which files are in which components and subcomponents.
Media	Contains directories for each media configuration you specify. The **Default.mda** file describes where to build the media files.
Registry Entries	Contains the **Default.rge** file, which describes the registry entries created in the installation process. Registry entries are associated with components.
Script Files	Contains custom setup files and the **setup.rul** custom setup script.
Setup Files	Not currently used by BladePack.
Shell Objects	Contains icons registered during installation.
String Tables\0009-English	Contains custom installation screen strings in the **value.shl** file.
Text Substitutions	Not currently used by BladePack.

Installation Screen Display Text

BladePack provides default text strings for the InstallShield installation wizard screens for Windows. You can override some of these text strings. For example, in the Select Installation Type wizard, you can change the text that appears next to the words **Typical**, **Compact**, and **Custom** to provide your own definitions for these three types of installations. When you save a BladePack project, BladePack saves any new string definitions in the appropriate string files.

Creating Distribution Media

To ensure that customers can install DataBlade modules and other IBM Informix products using common instructions, the product you distribute must conform to

the IBM Informix DataBlade module installation standard. BladePack creates an interactive installation that ensures a consistent user interface.

To distribute multiple products with their own installations on a CD-ROM or tape, rename the **cdrom** directory to the name of the DataBlade module before you copy the directory and its contents to the media or into the archive file.

For example, to install the Circle DataBlade module from CD-ROM, the installer mounts the CD-ROM, changes to the **circle** subdirectory, and executes the setup program.

Important: Put the **Setup.exe** program in a short path. Due to a Windows limitation, if the path for **Setup.exe** is too long, it fails to execute.

Appendix A. Source Files Generated for DataBlade Modules

You can use the tables in this appendix to find a brief description of the following types of files BladeSmith generates for your DataBlade project:

You can find these same descriptions in a comprehensive table that is ordered alphabetically; see "Alphabetical List of Generated Files" on page A-8.

C Source Code Files

BladeSmith generates Visual C++ project and workspace files and a UNIX makefile into the **\project\src** directory.

Project.dsp	Visual C++ project file
Project.dsw	Visual C++ workspace file
ProjectU.mak	Combination C and C++ makefile for use from the UNIX command line

BladeSmith generates the following C source files into the **project\src\C** directory. You can modify only the **Opaque.c**, **udr.c**, and **statistics.c** files.

Opaque.c	Source code file generated for each opaque type; the file contains the support functions for that opaque type.
Project.def	Definitions file listing all exported C routines; for use by Visual C++ 6.0 or later.
Project.h	Header file that contains project definitions, including the C data structures that define your opaque types.
readme.txt	Text file providing short descriptions of the files in this directory.
statistics.c	Source code file that contains statistics support functions.
support.c	Source code file that contains utility functions and **#include** directives for header files.
udr.c	Source code file that contains user-defined routines, cast support functions, and aggregates.
warning.txt	Text file providing warnings about potential source code problems.

ActiveX/C++ Source Code Files

BladeSmith generates Visual C++ project and workspace files and a UNIX makefile into the **\project\src** directory.

Project.dsp	Visual C++ project file
Project.dsw	Visual C++ workspace file

© Copyright IBM Corp. 1996, 2008

ProjectU.mak Combination C and C++ makefile for use from the UNIX command line

The following sections provide a brief description of the ActiveX/C++ source files that BladeSmith generates into the **project\src\ActiveX** directory:

This appendix lists the files generated for an ActiveX value object project called **Project** that consists of a single ActiveX value object with an underlying opaque type called **Opaque**.

Important: In addition to adding logic to the opaque support routines (see "Adding Project-Specific Logic to the Source Code" on page 6-3), you can add your own functions to the C++ classes in the **OpaqueCommon**, **OpaqueClient**, and **OpaqueServer .cpp** and **.h** files. Do not modify any other of the generated source files.

Client Project Files

For each project, BladeSmith generates client-specific support library files and project files.

Support Library Files

For each project, BladeSmith generates the following client-specific support library files. Do not modify these files.

DkClient.cpp Client-specific support library functions

DkIntf.h Support library header file that defines the ActiveX value object custom interfaces (**IRawObjectAccess** and **ITDkValue**)

DkIntf_i.c Support library file that contains IIDs (interface identifiers) for interfaces defined in **DkIntf.h**

DkIntfImpl.h Support library C++ template implementations for custom interfaces defined in **DkIntf.h**

Project Files

For each project named *Project*, BladeSmith generates the following client project files. Do not modify these files.

ProjectX.cpp Object map entry, DLL entry points, and so on

ProjectX.def Definitions file

ProjectX.idl IDL file that Visual C++ uses to generate **ProjectX.h** and **ProjectX.tlb**

ProjectX.rc Resource file

ProjectXps.def Generated by ATL

ProjectXps.mk Generated by ATL

Resource.h Header file that contains definitions, including **IDR_OPAQUE**

StdAfx.cpp	For precompiled header
StdAfx.h	Standard header file

Client Files

For each opaque type/ActiveX value object named *Opaque*, BladeSmith generates the following client files. Only the **OpaqueClient.cpp** and **OpaqueClient.h** files can be modified.

Opaque.cpp	C++ file that contains the methods for the ActiveX value object, that call into the C++ class (**OpaqueClient.cpp**).
Opaque.h	Header file that contains the ActiveX value object definition.
Opaque.rgs	Instructions for registering the ActiveX value object on the client computer.
OpaqueClient.cpp	C++ class file that contains placeholders (function definitions and null bodies) for the methods for **OpaqueClient**. This file can be modified.
OpaqueClient.h	Header file that contains the **OpaqueClient** class definition. This file can be modified.

Common Files

For each project, BladeSmith generates support library files and object files that are used to compile both the client project and the server project.

Support Library Files

For each project, BladeSmith generates the following support library files, which are used by both the client project and the server project. Do not modify these files.

StdDbdk.cpp	Support library file that provides the server and, with the **DkClient.cpp** file (see "Client Project Files" on page A-2), the client library functions
StdDbdk.h	Support library header file for client and server; contains class and function definitions

Object Files

For each opaque type/ActiveX value object named *Opaque*, BladeSmith generates the following files, which are used by both the client project and the server project. Do not modify the **OpaqueStruct.h** file.

OpaqueCommon.cpp	C++ file that contains the logic for all ActiveX custom methods and their server-project equivalents. This file can be modified.
OpaqueCommon.h	Header file that contains the **OpaqueCommon** class definition. This file can be modified.
OpaqueStruct.h	C header file that contains the **OpaqueStruct** definition (the C structure representing the opaque type)

Server Project Files

For each project named *Project*, the following server project files are generated. Do not modify these files.

ProjectWrap.cpp	C++ file that contains the interfaces to the server-side support routines
ProjectWrap.h	Header file for the server-side interfaces

Server Files

For each opaque type/ActiveX value object named *Opaque*, BladeSmith generates the following server files. You can modify these files.

OpaqueServer.cpp	C++ class file that contains placeholders (function definitions and null bodies) for the methods for **OpaqueServer**. This file can be modified.
OpaqueServer.h	Header file that contains the **OpaqueServer** class definition. This file can be modified.

Java Source Code Files

BladeSmith generates the following source files in the **project\src\java** directory.

DBDKInputException.java	Utility class file that provides exception-handling methods that are called when an exception occurs during input of a Java value object to or from the database server.
DBDKOutputException.java	Utility class file that provides exception-handling methods that are called when an exception occurs during output of a Java value object to or from the database server.
IfmxInStream.java	Utility class file that provides read methods to convert Java value objects between a string and the internal server format.
IfmxLog.java	Utility class file that provides logging methods that are included throughout the source code generated by BladeSmith.
IfmxOutStream.java	Utility class file that provides write methods to convert Java value objects between the internal server format and a string.
IfmxTrace.java	Not currently used.
***Opaque*.java**	Provides SQLData read and write methods to support opaque types written in C or C++.
Project_Java.mak	Use this file for compiling on either UNIX or Windows.
ProjectUDRs.java	Contains method declarations for all user-defined Java routines, cast support routines, and aggregates in the BladeSmith project. You must edit this file to add the functionality you require.
readme.txt	This file describes the files in the **src\java** directory.

warning.txt	This file describes potential problems with your source code.

SQL Script Files

BladeSmith generates the following SQL scripts in the **project\scripts** directory.

errors.locale	Contains locale-specific error messages. For example, the file for the default U.S. English locale is **errors.en_us.1252**. This file is only generated if you define new error messages for your DataBlade module.
objects.sql	Contains SQL statements that update the **sysbldobjects** system table with information about the DataBlade module objects that are created in a database. BladeManager uses the information in the table to register, unregister, and upgrade DataBlade modules.
prepare.locale.sql	Contains SQL statements for locale-specific objects. For example, the file for the default U.S. English locale is **prepare.en_us.sql**.
prepare.sql	Contains SQL statements that describe the DataBlade module to BladeManager.

Unit Test Files

When you generate unit tests for a DataBlade module, BladeSmith generates the files listed in the following table in the **src\tests** directory.

Setup.sql	Optionally initializes the database. You can add SQL statements to create and populate the tables necessary for your debugging tests. If you specify this file as your initialization file in the Properties dialog box, IfxQuery automatically runs this file after it creates a new database.
Routine.sql	Tests the user-defined routine. You can add SQL statements or modify the sample data for the routine. Use this file if you are debugging **udr.c**. IfxQuery runs this file if you click **Debug DataBlade Module** with this file in the active window.
Opaque.sql	Tests the support routines for each opaque data type. You can add SQL statements or modify the sample data for each support routine. Use this file if you are debugging **Opaque.c** or **OpaqueServer.cpp**. IfxQuery runs this file if you click **Debug DataBlade Module** with this file in the active window.
Cleanup.sql	Optionally deletes and drops tables and data in your test database. IfxQuery runs this file if you click **Debug DataBlade Module** with this file in the active window.

Functional Test Files

The functional test directory, **project\functest**, includes the following subdirectories:

- **cast.** Contains a subdirectory that contains functional tests for each cast for which you entered test data.
- **data.** Contains **.dat** files for each opaque type, user-defined routine, and cast for which you entered test data. The name of the data file is **objectname.dat**, where *objectname* is either the name of the opaque type or the name of the C routine associated with a user-defined routine or cast.
- **opaque.** Contains a subdirectory for each opaque type for which you entered test data. The subdirectory contains functional tests for the support routines defined for the opaque type.
- **udr.** Contains a subdirectory that contains functional tests for each user-defined routine for which you entered test data.

The **project\functest** directory contains a master shell script, **main.sh**, for executing all of the functional tests generated for the DataBlade module. Each subdirectory in the **udr**, **opaque**, and **cast** directories also contains a **main.sh** script to execute only the functional tests in that subdirectory.

Casting Function Tests

BladeSmith generates the following files in the **project\functest\cast\castfunction** directory for every cast function for which you entered test data.

cast.sql	Tests for *castfunction.*
cleanup.sql	Drops all of the test tables from the database.
main.sh	Executes the tests in this directory (**.sh** indicates this is a shell script).
setup.sql	Creates test tables and initializes them with test data.

Opaque Data Type Support Routines Tests

BladeSmith generates the following files in the **project\functest\opaque\opaque** directory for every support routine for which you entered test data.

binexp.sql	Tests binary file import/export functions for an opaque type; uses nested calls to the binary file import/export functions. The result of the nested calls should be equivalent to the text input format for the type.
binio.sql	Tests binary file input/output functions; uses the valid input data for the opaque type.
btree.sql	Tests the B-tree index support routine.
cleanup.sql	Drops all of the test tables from the database.
compare.sql	Tests the SQL **Compare** function.
concat.sql	Tests the concatenation operator; calls the SQL **Concat** function for an opaque data type with two instances of the type.
contains.sql	Tests the SQL **Contains** function.
divide.sql	Tests the SQL **Divide** function.

equal.sql	Tests the SQL **Equal** function.
greaterthan.sql	Tests the SQL **GreaterThan** function.
greaterthanorequal.sql	Tests the SQL **GreaterThanOrEqual** function.
hash.sql	Tests the SQL **Hash** support function with a SELECT...GROUP BY SQL query.
inter.sql	Tests the SQL **Inter** function.
lessthan.sql	Tests the SQL **LessThan** function.
lessthanorequal.sql	Tests the SQL **LessThanOrEqual** function.
main.sh	Executes the tests in this directory (**.sh** indicates this is a shell script).
minus.sql	Tests the SQL **Minus** function.
negative.sql	Tests the SQL **Negative** function.
notequal.sql	Tests the SQL **NotEqual** function.
notify.sql	Tests the SQL **Assign** and **Destroy** routines; inserts and deletes values in a new test table.
overlap.sql	Tests the SQL **Overlap** function.
plus.sql	Tests the SQL **Plus** function.
positive.sql	Tests the SQL **Positive** function.
setup.sql	Creates test tables and initializes them with test data.
size.sql	Tests the SQL **Size** function.
textexp.sql	Tests text file import/export functions; uses the UNLOAD and LOAD SQL statements.
textio_neg.sql	Tests text input/output functions; uses test data with invalid input data.
textio_pos.sql	Tests text input/output functions; uses only valid test data.
times.sql	Tests the SQL **Times** function.
union.sql	Tests the SQL **Union** function.
within.sql	Tests the SQL **Within** function.

User-Defined Routine Tests

BladeSmith generates the following files in the **project\functest\udr\routine** directory for every user-defined routine for which you entered test data.

call_neg.sql	Negative tests for *routine*.
call_pos.sql	Positive tests for *routine*.
cleanup.sql	Drops the test tables from the database.
main.sh	Executes the tests in this directory (**.sh** indicates this is a shell script).
setup.sql	Creates test tables and initializes them with test data.

Installation Packaging Files

BladeSmith generates the following installation packaging files that you can modify with BladePack in the **project\install** directory.

project.bom A bill of materials file. This file contains an entry for each file to be installed. The entry includes the path to the source file and the path where the file will be installed.

project.cmp Lists the main components and subcomponents in the installation package.

project.prd The main product file that you open with BladePack. This file lists other files that define the installation package. Initially, this file contains entries for the **.bom**, **.cmp**, and **.str** files. Add README files using BladePack.

project.str Defines character strings used in the installation.

Alphabetical List of Generated Files

The following table provides an alphabetical list of the files generated by the Informix DataBlade Developers Kit.

Filename	Directory	Description
binexp.sql	**project\functest\opaque\opaque**	Tests binary file import/export functions for an opaque type; uses nested calls to the binary file import/export functions. The result of the nested calls should be equivalent to the text input format for the type.
binio.sql	**project\functest\opaque\opaque**	Tests binary file input/output functions; uses the valid input data for the opaque type.
btree.sql	**project\functest\opaque\opaque**	Tests the B-tree index support routine.
call_neg.sql	**project\functest\udr\routine**	Negative tests for *routine*.
call_pos.sql	**project\functest\udr\routine**	Positive tests for *routine*.
cast.sql	**project\functest\cast\castfunction**	Tests for *castfunction*.
cleanup.sql	**project\functest\opaque\opaque**	Drops the test tables from the database.
cleanup.sql	**project\src\tests**	Unit test script file. Optionally deletes and drops tables and data in your test database. IfxQuery runs this file if you click **Debug DataBlade Module** with this file in the active window.
cleanup.sql	**project\functest\cast\castfunction**	Drops the test tables from the database.
cleanup.sql	**project\functest\udr\routine**	Drops the test tables from the database.
compare.sql	**project\functest\opaque\opaque**	Tests the SQL **Compare** function.
concat.sql	**project\functest\opaque\opaque**	Tests the concatenation operator; calls the SQL **Concat** function for an opaque data type with two instances of the type.
contains.sql	**project\functest\opaque\opaque**	Tests the SQL **Contains** function.

Filename	Directory	Description
DBDKInputException.java	**project\src\java**	Utility class file that provides exception-handling methods that are called when an exception occurs during input of a Java value object to or from the database server.
DBDKOutputException.java	**project\src\java**	Utility class file that provides exception-handling methods that are called when an exception occurs during output of a Java value object to or from the database server.
divide.sql	**project\functest\opaque\opaque**	Tests the SQL **Divide** function.
DkClient.cpp	**project\src\ActiveX**	Client-specific support library functions.
DkIntf.h	**project\src\ActiveX**	Client-specific support library header file that defines the ActiveX value object custom interfaces (**IRawObjectAccess** and **ITDkValue**).
DkIntf_i.c	**project\src\ActiveX**	Client-specific support library file that contains IIDs (interface identifiers) for interfaces defined in **DkIntf.h**
DkIntfImpl.h	**project\src\ActiveX**	Client-specific support library C++ template implementations for custom interfaces defined in **DkIntf.h**.
equal.sql	**project\functest\opaque\opaque**	Tests the SQL **Equal** function.
errors.locale	**project\scripts**	Locale-specific error messages file.
greaterthan.sql	**project\functest\opaque\opaque**	Tests the SQL **GreaterThan** function.
greaterthanorequal.sql	**project\functest\opaque\opaque**	Tests the SQL **GreaterThanOrEqual** function.
hash.sql	**project\functest\opaque\opaque**	Tests the SQL **Hash** support function with a SELECT...GROUP BY SQL query.
IfmxInStream.java	**project\src\java**	Utility class file that provides read methods to convert Java value objects between a string and the internal server format.
IfmxLog.java	**project\src\java**	Utility class file that provides logging methods that are included throughout the source code generated by BladeSmith.
IfmxOutStream.java	**project\src\java**	Utility class file that provides write methods to convert Java value objects between the internal server format and a string.
IfmxTrace.java	**project\src\java**	Not currently used.
inter.sql	**project\functest\opaque\opaque**	Tests the SQL **Inter** function.
lessthan.sql	**project\functest\opaque\opaque**	Tests the SQL **LessThan** function.
lessthanorequal.sql	**project\functest\opaque\opaque**	Tests the SQL **LessThanOrEqual** function.
main.sh	**project\functest**	Executes the all of the tests generated for the DataBlade module (**.sh** indicates this is a shell script).

Filename	Directory	Description
main.sh	**project\functest\cast\castfunction**	Executes the tests in this directory (**.sh** indicates this is a shell script).
main.sh	**project\functest\opaque\opaque**	Executes the tests in this directory (**.sh** indicates this is a shell script).
main.sh	**project\functest\udr\routine**	Executes the tests in this directory (**.sh** indicates this is a shell script).
minus.sql	**project\functest\opaque\opaque**	Tests the SQL **Minus** function.
negative.sql	**project\functest\opaque\opaque**	Tests the SQL **Negative** function.
notequal.sql	**project\functest\opaque\opaque**	Tests the SQL **NotEqual** function.
notify.sql	**project\functest\opaque\opaque**	Tests the SQL **Assign** and **Destroy** routines; inserts and deletes values in a new test table.
object.sql	**project\scripts**	SQL script file that contains the SQL statements to create DataBlade module objects in the database.
Opaque.c	**project\src\C**	A C file is generated for each opaque type; it contains the support functions for that opaque type.
Opaque.cpp	**project\src\ActiveX**	Client C++ file that contains the methods for the ActiveX value object that call into the C++ class (**OpaqueClient.cpp**).
Opaque.h	**project\src\ActiveX**	Client header file that contains the ActiveX value object definition.
Opaque.**java**	*project***src\java**	Provides SQLData read and write methods to support opaque types written in C or C++.
Opaque.rgs	**project\src\ActiveX**	Client file with instructions for registering the ActiveX value object on the client computer.
Opaque.sql	**project\src\tests**	Unit test script file. Tests the support routines for each opaque data type. You can add SQL statements or modify the sample data for each support routine. Use this file if you are debugging **Opaque.c** or **OpaqueServer.cpp**. IfxQuery runs this file if you click **Debug DataBlade Module** with this file in the active window.
Opaque_proxy.java	**project\src\java**	Contains value object proxy methods for the value object named **Opaque**. Do not edit this file.
OpaqueClient.cpp	**project\src\ActiveX**	Client C++ class file that contains placeholders (function definitions and null bodies) for the methods for **OpaqueClient**. This file can be modified.
OpaqueClient.h	**project\src\ActiveX**	Client header file that contains the **OpaqueClient** class definition. This file can be modified.

Filename	Directory	Description
OpaqueCommon.cpp	**project\src\ActiveX**	C++ file that contains the logic for all ActiveX custom methods and their server-project equivalents. This file can be modified.
OpaqueCommon.h	**project\src\ActiveX**	Server and client header file that contains the **OpaqueCommon** class definition. This file can be modified.
OpaqueServer.cpp	**project\src\ActiveX**	Server-side C++ class file that contains placeholders (function definitions and null bodies) for the methods for **OpaqueServer**. This file can be modified.
OpaqueServer.h	**project\src\ActiveX**	Server-side header file that contains the **OpaqueServer** class definition. This file can be modified.
OpaqueStruct.h	**project\src\ActiveX**	Server and client C header file that contains the **OpaqueStruct** definition (the C structure representing the opaque type).
overlap.sql	**project\functest\opaque\opaque**	Tests the SQL **Overlap** function.
plus.sql	**project\functest\opaque\opaque**	Tests the SQL **Plus** function.
positive.sql	**project\functest\opaque\opaque**	Tests the SQL **Positive** function.
prepare.sql	**project\scripts**	SQL script file that contains SQL statements that describe the DataBlade module to BladeManager.
Project.bom	**project\install**	A bill of materials file that contains an entry for each file to be installed.
Project.cmp	**project\install**	Component file listing main components and subcomponents in the installation package.
Project.def	**project\src\C**	Definitions file listing all exported C routines; for use by Microsoft Developer Studio Visual C++ 4.2 or above.
Project.dsp	**project\src**	Visual C++ project file.
Project.dsw	**project\src**	Visual C++ workspace file.
Project.h	**project\src\C**	Header file that contains project definitions, including the C data structures that define your opaque types.
Project.ibs	*project*	BladeSmith project file.
Project.prd	**project\install**	BladePack product file listing the other files that define the installation package.
Project.str	**project\install**	Character string file for interactive installations.
Project_Java.mak	**project\src\java**	Use this file for compiling on either UNIX and Windows.
ProjectU.mak	**project\src**	Combination C and C++ makefile for use from the UNIX command line.

Filename	Directory	Description
ProjectUDRs.java	**project\src\java**	Contains method declarations for all user-defined Java routines, cast support routines, and aggregates in the BladeSmith project. You must edit this file to add the functionality you require.
ProjectWrap.cpp	**project\src\ActiveX**	C++ file that contains the interfaces to the server-side support routines.
ProjectWrap.h	**project\src\ActiveX**	Header file for the server-side interfaces.
ProjectX.cpp	**project\src\ActiveX**	Client project file with object map entry, DLL entry points, and so on.
ProjectX.def	**project\src\ActiveX**	Client project definitions file.
ProjectX.dsp	**project\src\ActiveX**	Client project Microsoft Developer Studio Visual C++ 5.0 ATL project file.
ProjectX.idl	**project\src\ActiveX**	Client project IDL file that Visual C++ 5.0 uses to generate **ProjectX.h** and **ProjectX.tlb**.
ProjectX.mak	**project\src\ActiveX**	Client project Windows makefile for scripts or for command-line users.
ProjectX.rc	**project\src\ActiveX**	Client project resource file.
ProjectXps.def	**project\src\ActiveX**	Client project file generated by ATL.
ProjectXps.mk	**project\src\ActiveX**	Client project file generated by ATL.
readme.txt	**project\src\C**	Describes the files in this directory.
readme.txt	**project\src\java**	Describes the files in the **src\java** directory.
Resource.h	**project\src\ActiveX**	Client project header file that contains definitions, including **IDR_OPAQUE**.
Routine.sql	**project\src\tests**	Unit test script file. Tests the user-defined routine. You can add SQL statements or modify the sample data for the routine. Use these file if you are debugging **udr.c**. IfxQuery runs this file if you click **Debug DataBlade Module** with this file in the active window.
setup.sql	**project\functest\cast\castfunction**	Creates test tables and initializes them with test data.
setup.sql	**project\functest\opaque\opaque**	Creates test tables and initializes them with test data.

Filename	Directory	Description
setup.sql	**project\src\tests**	Unit test script file. Optionally initializes the database. You can add SQL statements to create and populate the tables necessary for your debugging tests. If you specify this file as your initialization file in the Properties dialog box, IfxQuery automatically runs this file after it creates a new database.
setup.sql	**project\functest\udr\routine**	Creates test tables and initializes them with test data.
size.sql	**project\functest\opaque\opaque**	Tests the SQL **Size** function.
StdAfx.cpp	**project\src\ActiveX**	Client project file for precompiled header.
StdAfx.h	**project\src\ActiveX**	Client project standard header file.
StdDbdk.cpp	**project\src\ActiveX**	Support library file that provides the server and, with the **DkClient.cpp** file (see "Client Project Files" on page A-2), the client library functions.
StdDbdk.h	**project\src\ActiveX**	Support library header file for client and server; contains class and function definitions.
support.c	**project\src\C**	C file that contains utility functions and **#include** directives for header files.
textexp.sql	**project\functest\opaque\opaque**	Tests text file import/export functions; uses the UNLOAD and LOAD SQL statements.
textio_neg.sql	**project\functest\opaque\opaque**	Tests text input/output functions; uses test data with invalid input data.
textio_pos.sql	**project\functest\opaque\opaque**	Tests text input/output functions; uses only valid test data.
times.sql	**project\functest\opaque\opaque**	Tests the SQL **Times** function.
udr.c	**project\src\C**	C file that contains user-defined routines, cast support routines, and aggregates.
union.sql	**project\functest\opaque\opaque**	Tests the SQL **Union** function.
warning.txt	*project***src\c**	Describes potential problems with your source code.
warning.txt	**project\src\java**	Describes potential problems with your source code.
within.sql	**project\functest\opaque\opaque**	Tests the SQL **Within** function.

Appendix B. Completing BladeSmith-Generated Code

This appendix provides tables that list the types of objects BladeSmith generates, indicate whether BladeSmith generates complete code or template code you must complete, and provide a reference to the instructions in this guide for completing the code.

Opaque Data Type Support Routines in C

The following table lists the opaque data type support routines BladeSmith generates for C language DataBlade module projects and provides a reference to the sections in this guide that explain how to complete or customize code for each type of object.

Opaque Support Routine	Complete Code?	Customizable Code?	How to Complete or Customize the Code
Text input/output	Yes	Yes	See "Text Input and Output Functions" on page 5-16.
Binary send/receive	Yes	Yes	See "Binary Send and Receive Functions" on page 5-19.
Text file import/export	Yes		See "Text File Import and Export Functions" on page 5-20.
Binary file import/export	Yes	No	See "Binary File Import and Export Functions" on page 5-21.
Assign/destroy	Yes	Yes	See "The Assign and Destroy Routines" on page 5-22.
LOhandles()			See "LOhandles() Function" on page 5-23.
Compare()	Yes	Yes	See "Compare Function" on page 5-23.
B-tree comparison functions: • **Equal()** • **LessThan()** • **LessThanOrEqual()** • **GreaterThan()** • **GreaterThanOrEqual()** • **NotEqual()**	Yes	No	See "B-Tree Comparison Functions" on page 5-25.
R-tree comparison functions	No	No	See "R-Tree Comparison Functions" on page 5-25.
Mathematic functions: • **Plus()** • **Minus()** • **Times()** • **Divide()** • **Positive()** • **Negate()**	No	Yes	See "Mathematical Functions" on page 5-26.
Concat()	No	Yes	See "Concat() Function" on page 5-26.
Hash()	No	Yes	See "Hash() Function" on page 5-27.

© Copyright IBM Corp. 1996, 2008

Opaque Support Routine	Complete Code?	Customizable Code?	How to Complete or Customize the Code
OpaqueStatCollect()	Yes	Yes	See "The Statistics Collection Function" on page 5-27.
OpaqueStatPrint()	Yes	Yes	See "The Statistics Print Function" on page 5-28.
Opaque_SetMinValue()	No	Yes	See "The Statistics Minimum, Maximum, and Distribution Functions" on page 5-28.
Opaque_SetMaxValue()	No	Yes	See "The Statistics Minimum, Maximum, and Distribution Functions" on page 5-28.
Opaque_SetHistogram()	No	Yes	See "The Statistics Minimum, Maximum, and Distribution Functions" on page 5-28.

User-Defined Routines in C

The following table provides references to sections in this guide that explain how to complete or customize code for C language user-defined routines that BladeSmith generates.

C Object	Complete Code?	Customizable Code?	How to Complete or Customize the Code
User-defined routines (general)	No	Yes	See "Most User-Defined Routines" on page 5-29.
Cast support functions	No	Yes	See "Cast Support Functions" on page 5-29.
Aggregate functions: • **AggregateInit()** • **AggregateIter()** • **AggregateComb()** • **AggregateFinl()**	No	Yes	See "Aggregate Functions" on page 5-30.
Selectivity functions	No	Yes	See "Selectivity Functions" on page 5-31.
Iterator functions	No	Yes	See "Iterator Functions" on page 5-32.

Opaque Data Type Support Routines in C++

The following table lists the opaque data type support routines BladeSmith generates for C++/ActiveX DataBlade module projects and provides a reference to the sections in this guide that explain how to complete or customize code for each type of object.

C++ Method	Complete Code?	Customizable Code?	How to Complete or Customize the Code
Binary send/receive	Yes	Yes	See "Implementing ActiveX Value Objects" on page 6-2 for an overview of the programming tasks. See "Support Methods Reference" on page 6-7 for a description of the internal object and support library methods that you can use.

C++ Method	Complete Code?	Customizable Code?	How to Complete or Customize the Code
Binary file import/export	Yes	Yes	See "Implementing ActiveX Value Objects" on page 6-2 for an overview of the programming tasks. See "Support Methods Reference" on page 6-7 for a description of the internal object and support library methods that you can use.
Text input/output: • **FromString()** • **ToString()**	Yes	Yes	See "Implementing ActiveX Value Objects" on page 6-2 for an overview of the programming tasks. See "Support Methods Reference" on page 6-7 for a description of the internal object and support library methods that you can use.
Text import/export: • **TextImport()** • **TextExport()**	Yes	Yes	See "Implementing ActiveX Value Objects" on page 6-2 for an overview of the programming tasks. See "Support Methods Reference" on page 6-7 for a description of the internal object and support library methods that you can use.
Type compare: • **Compare()** • **Equal()** • **NotEqual()**	Yes	Yes	See "Implementing ActiveX Value Objects" on page 6-2 for an overview of the programming tasks. See "Support Methods Reference" on page 6-7 for a description of the internal object and support library methods that you can use.
B-tree comparison methods: • **Equal()** • **LessThan()** • **LessThanOrEqual()** • **GreaterThan()** • **GreaterThanOrEqual()** • **NotEqual()**	Yes	No	See "Implementing ActiveX Value Objects" on page 6-2 for an overview of the programming tasks. See "Support Methods Reference" on page 6-7 for a description of the internal object and support library methods that you can use.
R-tree comparison methods	No	No	See "Implementing ActiveX Value Objects" on page 6-2 for an overview of the programming tasks. See "Support Methods Reference" on page 6-7 for a description of the internal object and support library methods that you can use.
Mathematic methods: • **Plus()** • **Minus()** • **Times()** • **Divide()** • **Positive()** • **Negate()**	No	Yes	See "Implementing ActiveX Value Objects" on page 6-2 for an overview of the programming tasks. See "Support Methods Reference" on page 6-7 for a description of the internal object and support library methods that you can use.

C++ Method	Complete Code?	Customizable Code?	How to Complete or Customize the Code
Concat()	No	Yes	See "Implementing ActiveX Value Objects" on page 6-2 for an overview of the programming tasks. See "Support Methods Reference" on page 6-7 for a description of the internal object and support library methods that you can use.
Hash()	No	Yes	See "Implementing ActiveX Value Objects" on page 6-2 for an overview of the programming tasks. See "Support Methods Reference" on page 6-7 for a description of the internal object and support library methods that you can use.
IsNull()	Yes	Yes	See "Implementing ActiveX Value Objects" on page 6-2 for an overview of the programming tasks. See "Support Methods Reference" on page 6-7 for a description of the internal object and support library methods that you can use.
SetNullFlag()	Yes	Yes	See "Implementing ActiveX Value Objects" on page 6-2 for an overview of the programming tasks. See "Support Methods Reference" on page 6-7 for a description of the internal object and support library methods that you can use.

User-Defined Routines in Java

The following table provides references to sections in this guide that explain how to complete or customize code for Java language user-defined routines that BladeSmith generates.

Java Method	Complete Code?	Customizable Code?	How to Complete or Customize the Code
User-defined methods (general)	No	Yes	See "Most User-Defined Methods" on page 8-5.
Cast support methods	No	Yes	See "Cast Support Methods" on page 8-7.
Aggregate methods: • **AggregateInit()** • **AggregateIter()** • **AggregateComb()** • **AggregateFinl()**	No	Yes	See "Aggregates" on page 8-6.
Iterators	No	Yes	See "Iterators" on page 8-5.

Appendix C. Testing for an Sbspace

If your DataBlade module contains data types that contain smart large object data (BLOB and CLOB data types), an sbspace to store the smart large object must exist for each database in which users register the DataBlade module. If the required sbspace does not exist, registration fails.

You can test for the existence of a particular sbspace when BladeManager prepares your DataBlade module for registration by using the following procedure. If you test for the sbspace and it does not exist, registration fails and BladeManager writes an error message to the registration log. If you do not test for the sbspace and it does not exist, registration fails with an obscure error message.

To implement a test for a particular sbspace, use BladeSmith to add a custom SQL statement to your DataBlade module that executes the **SYSBldTstSBSpace()** function. This is the syntax of the EXECUTE FUNCTION statement that executes **SYSBldTstSBSpace()**:

```
EXECUTE FUNCTION SYSBldTstSBSpace("opt_name");
```

opt_name is the name of the required sbspace. To indicate the default sbspace, replace `"opt_name"` with `" "`.

To add this user-defined routine to your registration script:

1. In BladeSmith, define the DataBlade module object that has a data type of BLOB or CLOB.

   For example, create an opaque type called **BigType** that has a member of type BLOB.

   See "Creating Data Types" on page 4-22 for instructions.
2. Choose **Edit > Insert > SQL File**.

   The New SQL File wizard appears.
3. In the **Descriptive name for SQL** text box, type a name for the SQL file. For example, type `SbspaceTest`.
4. In the **Custom SQL create text** text box, type the following statement:

   ```
   EXECUTE FUNCTION SYSBldTstSBSpace("opt_name");
   ```
1. Click **Next**.
2. To specify which data types in your DataBlade module contain smart large objects, click the appropriate objects in the **These objects require this SQL** box. For example, click **BigType**.

   For more information on dependencies, see "Object Dependencies" on page 4-39.
3. Click **Finish**.

When BladeManager prepares the DataBlade module for registration, the database server executes the **SYSBldTstSBSpace()** function before the SQL statement to create the **BigType** data type. If the sbspace specified in **SYSBldTstSBSpace()** exists, the database server creates the dependent data type (**BigType**). If the sbspace does not exist, the database server writes an error to a BladeManager log file.

© Copyright IBM Corp. 1996, 2008

For more information on BladeManager, see the *IBM Informix DataBlade Module Installation and Registration Guide*.

Appendix D. Accessibility

IBM strives to provide products with usable access for everyone, regardless of age or ability.

Accessibility features for IBM Informix Dynamic Server

Accessibility features help a user who has a physical disability, such as restricted mobility or limited vision, to use information technology products successfully.

Accessibility Features

The following list includes the major accessibility features in IBM Informix Dynamic Server. These features support:

- Keyboard-only operation.
- Interfaces that are commonly used by screen readers.
- The attachment of alternative input and output devices.

Tip: The IBM Informix Dynamic Server Information Center and its related publications are accessibility-enabled for the IBM Home Page Reader. You can operate all features using the keyboard instead of the mouse.

Keyboard Navigation

This product uses standard Microsoft Windows navigation keys.

Related Accessibility Information

IBM is committed to making our documentation accessible to persons with disabilities. Our publications are available in HTML format so that they can be accessed with assistive technology such as screen reader software. The syntax diagrams in our publications are available in dotted decimal format.

You can view the publications for IBM Informix Dynamic Server in Adobe Portable Document Format (PDF) using the Adobe Acrobat Reader.

IBM and Accessibility

See the *IBM Accessibility Center* at http://www.ibm.com/able for more information about the commitment that IBM has to accessibility.

© Copyright IBM Corp. 1996, 2008

Notices

IBM may not offer the products, services, or features discussed in this document in all countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:
INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created

© Copyright IBM Corp. 1996, 2008

programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
J46A/G4
555 Bailey Avenue
San Jose, CA 95141-1003
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

All IBM prices shown are IBM's suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

> © (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. (enter the year or years). All rights reserved.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. These and other IBM trademarked terms are marked on their first occurrence in this information with the appropriate symbol (® or ™), indicating US registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of IBM trademarks is available on the Web at http://www.ibm.com/legal/copytrade.shtml.

Adobe, Acrobat, Portable Document Format (PDF), and PostScript are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, other countries, or both.

Intel, Intel logo, Intel Inside, Intel Inside logo, Intel Centrino, Intel Centrino logo, Celeron, Intel Xeon, Intel SpeedStep, Itanium, and Pentium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Java and all Java-based trademarks and logos are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, or service names may be trademarks or service marks of others.

Index

Special characters

A

B

© Copyright IBM Corp. 1996, 2008

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

Printed in USA

G229-6366-01

Spine information:

IBM Informix **Version 4.20**

IBM Informix DataBlade Developers Kit User's Guide