

VisualAge Pacbase



eBusiness & Pacbench C/S Applications Graphic Presentation

Version 3.5



Note

Before using this document, read the general information under "Notices"

You may consult or download the complete up-to-date collection of the VisualAge Pacbase documentation from the VisualAge Pacbase Support Center at:

<http://www.ibm.com/software/awdtools/vapacbase/productinfo.htm>

Consult the Catalog section in the Documentation home page to make sure you have the most recent edition of this document.

1st Edition (May 2004)

This edition applies to the following licensed program:
VisualAge Pacbase Version 3.5

Comments on publications (including document reference number) should be sent electronically through the Support Center Web site at:

<http://www.ibm.com/software/awdtools/vapacbase/productinfo.htm>

or to the following postal address:

IBM Paris Laboratory
Support VisualAge Pacbase
1 place J.B. Clément
93881 Noisy-le-Grand Cedex FRANCE

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 1983, 2003. All rights reserved.

Note to U.S. Government Users – Documentation related to restricted rights – Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.

Contents

See Detailed Contents next page.

Chapter 1: Introduction.....11

Chapter 2: Generating Proxies.....13

Delivered Generic classes (Java).....13

Launching the Generator.....15

Generation Results.....16

Chapter 3: Development Principles.....25

Visual Representation of Proxies in the Target Environment 25

Use of Properties.....26

Use of Methods.....29

Use of Events.....48

Error Management.....50

Chapter 4: Developing a Java Client.....59

Example of an Applet.....59

Specificities of a Standalone Application.....78

Error Management.....80

Communication Management.....86

Testing the Generated Application – Packaging.....90

Application Deployment.....94

Chapter 5: Developing a COM Client.....95

Visual Basic Example of a COM Proxy Use.....95

Error Management.....99

Communication Management.....100

Application Deployment.....102

Chapter 6: Index.....103

Detailed Contents

Chapter 1: Introduction.....11

Chapter 2: Generating Proxies.....13

Delivered Generic classes (Java).....	13
Online Documentation of Generic Classes	14
Launching the Generator.....	15
From the eBusiness Module of Developer workbench	
From WSAD (or Eclipse)	15
From VisualAge for Java	15
From the .exe File.....	15
From a Java Virtual Machine	16
Generation Results.....	16
Java.....	16
Introduction.....	16
Generated Classes.....	17
COM.....	22
Introduction.....	22
Generated Classes.....	23
Compilation Results.....	24
Compiling with Visual C++ Version 5.0 and 6.0..	24

Chapter 3: Development Principles25

Visual Representation of Proxies in the Target Environment	25
Java Environment	25
COM Environment.....	26
Use of Properties	26
Local Checks.....	26
Check of the Length of the <code>detail</code> Property Fields.....	27
Selecting the Local or Server Sort Criterion on a List of Instances	27
Local Sort	27
Server Sort.....	27
Specification of the Local Sort Criterion (Java Only) ..	28
Table Model (Java Only).....	28
Sub-schema Management	29
Use of Methods.....	29
The Different Types of Server Methods.....	29
Managing Folder Reading.....	30
Provisional Large Reading of Dependent Nodes.	30
Transferring an Instance Between the rows and detail Properties.....	31
Large Reading and Transferring an Instance Between Rows and Detail Properties: Working Mechanism.....	31
Large Reading of Reference Nodes	36
Principle of Paging in a Folder's Nodes.....	36
Selection Criteria Associated with Large Reading Methods.....	36
Limitation of the Scope of Large Reading	37
Reading of a Root Node or Dependent Instance..	37
Reading of a Reference Node Instance.....	37
Selection Criteria Associated with Instance Reading	38
Folder Update Management	38
Local Updates.....	38
Server Updates.....	38

Management of Effective Transactions	39
Re-initializing instances in the local cache.....	39
Managing collections of instances.....	39
Load of the local cache with no server access	39
Asynchronous Methods	40
Principles	40
Global Methods or Methods Associated with an Instance	40
Examples.....	41
Storing the Proxy Context.....	42
Externalization of the Management of Requests.....	42
User Service.....	43
Java	43
COM	43
Database Logical Lock.....	44
Customization of the Columns of a Jtable (Java Only)	45
Management of Data Element Presence	46
Java	46
COM	46
Management of Data Element Check	47
Java	47
COM	47
Sub-Schema Management.....	48
Use of Events	48
Java	48
Event-driven Management of Large Reading	48
Event-driven Management of Instance Reading..	49
COM	49
Event-driven Management of Large Reading	49
Event-driven Management of Instance Reading..	50
Error Management	50
Introduction	51
Local Errors	51
Server Errors	54
System Errors.....	54
System Errors Received from the Elementary Component.....	54
System Errors Received from the Communications Monitor	55
System Errors Received from the Services Manager.....	55
Communication Errors.....	57

Chapter 4: Developing a Java Client..... 59

Example of an Applet	59
Introduction	59
Presentation of the End User Interface	60
Developing the End User Interface with VisualAge Java V1	61
Implementing the Example and Creating the Applet	61
Developing the Customers Window	64
Developing the Orders Window	69
Developing the End User Interface with VisualAge Java V2	70
Implementing the Example and Creating the Applet	70
Developing the Customers Window	73
Developing the Orders Window	77
Specificities of a Standalone Application	78
Introduction	78
Example	79

Error Management.....	80
Principles.....	80
Introduction.....	80
Programming	80
Local Errors.....	80
Server Errors.....	81
System Errors	81
Communication Errors	81
Example of Error Management	81
Introduction.....	81
Presentation of the Non-Visual Classes in Use.....	81
Presentation of the ErrorManagerExample Visual Class	83
Code for Displaying the Error Window	85
Communication Management	86
Processing a Request	86
Direct Access to the Middleware	87
Access via a Gateway.....	87
Access via a Particular Adapter	87
Dynamic Change of the Middleware Access Parameters	88
Definition of the Use Context via the Location Editor.....	89
Launching from the eBusiness Module of Developer workbench.....	89
Launching from VisualAge for Java.....	89
Launching from the .exe File	89
Launching from a Java Virtual Machine.....	89
Testing the Generated Application – Packaging	90
Testing the Generated Application.....	90
Testing Server Components with the Services Test Facility	90
Version Compatibility Check	91
Packaging.....	91
Reminder: Prerequisites	91
Export	91
Application Deployment	94
Chapter 5: Developing a COM Client.....	95
Visual Basic Example of a COM Proxy Use.....	95
Presentation of the End User Interface.....	95
Visual Basic Development Example.....	96
Inserting the COM Proxy in the Visual Basic Project	96
Setting of the Proxy in the Application Design Mode.....	97
Selecting and Filling the Grid Representing the rows Attribute	98
Error Processing.....	98
Filling of the detail Attribute	99
Error Management.....	99
Communication Management	100
Processing a Request	100
Definition of the Use Context via the Location Editor.....	101
Launching from the eBusiness Module of Developer workbench.....	Error! Bookmark not defined.
Launching from the .exe File	101
Application Deployment	102
Chapter 6: Index	103

NOTICES

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. 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. Subject to IBM's valid intellectual property or other legally protectable rights, any functionally equivalent product, program, or service may be used instead of the IBM product, program, or service. The evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, are the responsibility of the user.

IBM may have patents or pending patent applications covering subject matter 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:

Intellectual Property and Licensing
International Business Machines Corporation
North Castle Drive, Armonk, New-York 10504-1785
USA

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of information which has been exchanged, should contact:

IBM Paris Laboratory
Département SMC
1 place J.B. Clément
93881 Noisy-le-Grand Cedex
France

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

IBM may change this publication, the product described herein, or both.

TRADEMARKS

IBM is a trademark of International Business Machines Corporation, Inc. AIX, AS/400, CICS, CICS/MVS, CICS/VSE, COBOL/2, DB2, IMS, MQSeries, OS/2, PACBASE, RACF, RS/6000, SQL/DS, TeamConnection, and VisualAge are trademarks of International Business Machines Corporation, Inc. in the United States and/or other countries.

Java and all Java-based trademarks and logos are trademarks of Sun Microsystems, Inc. in the United States and/or other countries.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States and/or other countries.

UNIX is a registered trademark in the United States and/or other countries licensed exclusively through X/Open Company Limited.

All other company, product, and service names may be trademarks of their respective owners.

Foreword

In this volume, we assume that you are familiar with the the contents of the *eBusiness & Pacbench C/S Applications: Concepts & Architecture* manual.

Since this manual cannot include all the information related to the development of applications with VisualAge Pacbase, you will also find useful information in the:

- *eBusiness & Pacbench C/S Applications: Proxy Programming Interface* manual,
- Developer workbench online help,
- *Pacbench C/S Applications: Business Logic* manual,
- eBusiness Tools online help,
- *Middleware User's Guide*,
- Documentation associated with WSAD or VisualAge for Java, including HTML online documentation or the documentation associated with any other Java tool used.
- or Documentation associated with your COM development environment.

Typographical conventions in use

The Courier New font is used for any character strings you can enter, displayed by the product or corresponding to generated codes.

Italics is used for titles of publications or Chapter in cross-references.

The following symbols are used to point out:



a note, a remark.



cross-reference to another location in the documentation.



a helpful hint or tip, a useful piece of information.



an action to be carried out in a Tool or an Editor.



that you must proceed with caution (risky or irreversible action, etc.).

Terminology standards

- FVP refers to a Folder View Proxy.
- LVP refers to a Logical View Proxy.
- Folder View Proxy, Elementary Proxy and Logical View Proxy are usually referred to as Proxy objects and Proxy components.
- For convenience, in the sections that are common to the Java and COM environments, the term *property* is used to designate both a Java property and a COM attribute and the term *method* is used to designate both a Java method and a COM action.

Chapter 1: Introduction

This manual intends to guide you in creating graphic applications in a Java or COM environment, using the server components developed with the VisualAge Pacbase eBusiness (Developer workbench) or Pacbench C/S module (VisualAge Pacbase WorkStation).

The generated applications can be executed on a PC but can also be accessed from a Web browser (Intranet or Internet).

Contents of this manual

This manual contains the following chapters:

- The introduction gives you a brief description of the different steps of a graphic client development, from the generation of components extracted from the VA Pac Repository to the graphic construction of applications.
- The next chapter presents the generation step and details its results, according to the generation target (Java or COM).
- The next chapter presents the public interface of the generated components as well as error and communication management.
- The next two chapters provide examples with detailed comments on the development of standard graphic and Web applications. They also provide information on how to test and package an application.
- Finally, an index lists the actions/methods, attributes/properties, events and classes whose names are mentioned in this manual.

Development steps

Developing a graphic application consists in:

- Developing the server components with the VisualAge Pacbase eBusiness (Developer workbench) or Pacbench C/S module (VisualAge Pacbase WorkStation).

☞ For information on the development of server components, refer to the Developer workbench online help if you use Developer workbench, or to the *Pacbench C/S Applications: Business Logic* manual if you use the VisualAge Pacbase WorkStation.

- Generating Proxies from these server components, using the eBusiness tools, and then testing these Proxies.


The generation phase produces classes which will execute services associated with the server components.

The generator takes as input the extraction file initially transferred onto the client development workstation.

☞ For more details on the generation features, refer to *Chapter 2: Generating Proxies*.

- Developing a client application.

Developing a client application implies integrating Proxies and calling their services. You can develop the client application with WSAD, the VisualAge for Java workstation (or any development tool which uses JDK version 1.1 and onwards) or any development tool which uses the COM technology.

 You will find an example with comments on the development of a standard and web application in *Chapter 4: Developing a Java Client* and *Chapter 5: Developing a COM Client*.

Compatibility of Elementary Components / Proxy Objects

Elementary Components and Proxy objects must be generated with the same version number, for a difference of version can produce discrepancies showing inconsistency in the client application.

To avoid possible discrepancies, you must implement a version control by setting an option in the Elementary Component. This option sends an error message if a discrepancy between the version numbers is detected when the Client calls the Elementary Component.



In Developer workbench, fill in the **Version** field in the Definition tab of the Elementary Component.



If you use VA Pac via its VisualAge Pacbase WorkStation interface, indicate the **NUVERS** option. For more details, refer to the *Pacbench C/S Applications: Business Logic* manual.

Chapter 2: Generating Proxies

Delivered Generic classes (Java)

To develop a Client, you will use generated classes and a great deal of generic classes, which are delivered with the product to avoid the multiplication of elements at each new generation. Unlike the generated classes documented further on, generic classes do not depend on the characteristics of the processed Logical View.



You must make sure that generic classes are installed on your workstation before starting developing your application.

Generic classes are delivered in the following packages:

- `com.ibm.vap.generic`

This package contains:

- The Parent classes of the `ProxyLv` generated classes:
 - ♦ `ProxyLv`
 - ♦ `HierarchicalProxyLv`
 - ♦ `DependentProxyLv`
 - ♦ `Folder`
 - ♦ `ReferenceProxyLv`
- The `Data` generic classes (Parent of `selectionCriteria` and `DataDescription` generated classes)
- The classes of the local cache:
 - ♦ `Node`
 - ♦ `HierarchicalNode`
 - ♦ `DependentNode`
 - ♦ `RootNode`
 - ♦ `ReferenceNode`
- Exceptions and errors:
 - ♦ `VapException`
 - ♦ `LocalException`
 - ♦ `ServerException`
 - ♦ `CommunicationError`
 - ♦ `SystemError`
- The layout classes in the form of a list of `DataDescription` generated classes (handled by the `rows` property of Proxy objects).
- The classes describing the properties of the Proxy objects as defined in VisualAge Pacbase:
 - ♦ `VapProxyProperties`
 - ♦ `VapHierarchicalProxyProperties`
 - ♦ `VapDependentProxyProperties`
 - ♦ `VapFolderProperties`
 - ♦ `VapReferenceProxyProperties`

- The classes related to the handling of XML streams:
 - ♦ `XMLMapping`
 - ♦ `XMLWrapper`
- `com.ibm.vap.exchange`

This package contains the classes handling the Exchange Manager.

Online Documentation of Generic Classes

An HTML-formatted documentation is delivered with the eBusiness or Pacbench C/S runtime. This documentation presents:

- Generic classes, parent of the `ProxyLv` and `data` generated classes used on execution,
- Errors and exceptions raised by these execution classes,
- The beans associated with VA Pac Data Element-type properties, used in the Composition Editor for a quick mapping of the data.

In the `awt` palette, these beans are:

- ♦ `Pacbase Text Field`
- ♦ `Pacbase Integer Field`
- ♦ `Pacbase Decimal Field`
- ♦ `Pacbase Date Field`
- ♦ `Pacbase Time Field`
- ♦ `Pacbase Long Field`
- ♦ `Pacbase Text Choice`
- ♦ `Pacbase Integer Choice`
- ♦ `Pacbase Decimal Choice`
- ♦ `Pacbase Date Choice`
- ♦ `Pacbase Time Choice`
- ♦ `Pacbase Long Choice`

In the `swing` palette, these beans are:

- ♦ `Pacbase Swing Text Field`
- ♦ `Pacbase Swing Integer Field`
- ♦ `Pacbase Swing Decimal Field`
- ♦ `Pacbase Swing Date Field`
- ♦ `Pacbase Swing Time Field`
- ♦ `Pacbase Swing Long Field`
- ♦ `Pacbase Swing Text ComboBox`
- ♦ `Pacbase Swing Integer ComboBox`
- ♦ `Pacbase Swing Decimal ComboBox`
- ♦ `Pacbase Swing Date ComboBox`
- ♦ `Pacbase Swing Time ComboBox`
- ♦ `Pacbase Swing Long ComboBox`
- ♦ `Pacbase Swing Date RadioButtonGroup`
- ♦ `Pacbase Swing Decimal RadioButtonGroup`
- ♦ `Pacbase Swing Integer RadioButtonGroup`

Launching the Generator

From the eBusiness Module of Developer workbench

You launch the Proxy Generator from the 'Applications' or 'Folders' tab of the workbench.

☞ For explanations on the Proxy Generator's interface, refer to the eBusiness Tools' online help.

From WSAD (or Eclipse)

To launch the Proxy Generator from WSAD or Eclipse,

- select 'File' → 'New' → 'Other'.
- or right click on a Java project and select 'New' → 'Other'.

Then select 'VisualAge Pacbase eBusiness' → 'eBusiness Proxy Generator'.

From VisualAge for Java

To launch the Proxy Generator from VisualAge for Java directly, select, in the 'Workspace' menu, 'Tools' → 'VisualAge Pacbase eBusiness' → 'Proxy Generator'. In this case:

- The generation type is forced to Java.
- You must indicate a VisualAge for Java project into which the generated classes will be imported.

☞ For explanations on the Proxy Generator's interface, refer to the eBusiness Tools' online help.

If you want to parameterize the launching of the Proxy Generator, select, in the 'Workspace' menu, 'Tools' → 'VisualAge Pacbase eBusiness' → 'Proxy Generator Properties'.

From the .exe File

Execute the **vapgen.exe** file.

You can parameterize the launching of the Proxy Generator via the following parameters in order to prepare and speed up the generation process:

- lang<LANGUAGE>**: This parameter enables you to select the language of the generator's graphic interface: **fr** for French, **en** for English. The language of the system is used by default.
- input<INPUT_FILE>**: this parameter enables you to indicate the extraction file which contains the Folders to be generated.
- output<OUTPUT_DIR>**: this parameter enables you to indicate the generation output directory.
- java**: with this parameter, you indicate that the generation is for a Java target.
- com**: with this parameter, you indicate that the generation is for a COM target.
- i18n**: this parameter enables you to activate the internationalization option for the generation.

Moreover, the following parameters are specific to a generation for Java:

- proxy**<PROXY_PACKAGE> (only for Proxies originating from entities of the Pacbench C/S module): this option allows you to specify the Java package in which the classes corresponding to the Proxy components will be generated.
- data**<DATA_PACKAGE> (only for Proxies originating from entities of the Pacbench C/S module): this option allows you to specify the Java package in which the data classes will be generated.
- config**<CONFIGURATION_FILE>: this parameter enables you to indicate the Java configuration file which is to be used.
- classpath**<PATH>: this parameter enables you to indicate the path that will be used for the compilation of the generated Java proxies.

From a Java Virtual Machine

To launch the Proxy Generator from a Java Virtual Machine, execute the **java_vapGen.bat** file.

This .bat is an example which you must modify according to the location of your JDK (Java Developer ToolKit) or JRE (Java Runtime Environment).

You can indicate the generation options indicated above to prepare and speed up the generation process.

Generation Results

The generated elements always depend on the type of service carried out by the Elementary Component. They will not be the same depending on whether the Elementary Component updates or just reads.

The generated file only contains the classes which depend on the characteristics of the processed Logical View.


Java

Introduction

Once generation is over, the following files are created:

- The **VAPLOCAT.INI** location file is created in the generation output directory.

You complete this file by using the Location Editor tool. This tool enables you to parameterize the location file, thus avoiding syntax errors which might inhibit the communication between the Client (Proxy) component and the servers.

 For more information, refer to the online help of the Location Editor.

This file must be entered at the gateway start.

- The source files of the generated classes and resources required for editing them.
 - ☞ If you checked the **Generate beans** option, a **BeanInfo** class is generated for each execution class (**ProxyLv** and **data**). The **BeanInfo** class associated with a class (bean) bears the name of this class, with a **BeanInfo** suffix at the end.

For example, when a `CustomerProxyLv` class (Root Proxy in our example) is generated, a `CustomerProxyLvBeanInfo`, is systematically generated.

A `BeanInfo` class contains editing information (label, icon, etc.) for the associated execution class. It contains public methods that give information on the associated bean class, such as the class name, properties, methods and events available on the bean.

☞ If you checked the `Generate XML schema and data` option, an `XMLMapping` class and an `XMLWrapper` class are generated for the root proxy class (`ProxyLv`).

For example: `CustomerProxyLvXMLMapping` and `CustomerProxyLvXMLWrapper`.

Generated Classes

The elements generated by the eBusiness or Pacbench C/S module correspond to classes whose coding consists of a prefix and a hard-coded part assigned by the generator. The prefix corresponds to the Logical View's name, whose maximum length is 36 characters.

☞ If you are not satisfied with the server's prefix, you can change it in the generator.

- `[Prefix]Data` Class

This class represents the description of a Logical View instance. It contains a set of properties which correspond to the Logical View Data Elements.

- `[Prefix]SelectionCriteria` class

This class represents the description of the selection criteria. It contains a set of properties which correspond to identifier Data Elements and extraction parameters associated with the Logical View.

- `[Prefix]Buffer` Class

This class represents the description of the contextual information. It contains a set of properties which correspond to the user buffer Data Elements.

- `[Prefix]DataUpdate` class

This class inherits from the `[Prefix]Data` class. Compared to its parent class, it contains two additional properties whose values vary according to the modifications in progress. These two properties are:

- ♦ `action` : update action which can be `Read`, `Modified`, `Created` or `Deleted`. This property is only visible in the list of the `UpdatedFolders` property.
- ♦ `updatedInstancesCount` : number of useful server updates associated with the relevant Folder instance. Its value can be 1 to n.

- `[Prefix]UserData` class

This class inherits from the `[Prefix]Data` class. It contains an additional property which corresponds to the key Data Element of the parent instance.

- `[Prefix]TableModel` class

This class is generated only if you checked the **Use Swing** option at generation time. It inherits from the **Pacbase TableModel** class and implements the **TableModel** swing component, used to fill in a swing table with data. This generated class is used to display the list of instances of the **[Prefix]Data** class generated in the swing table.

- **[Prefix]UpdateTableModel** class

This class is generated only if you checked the **Use Swing** option at generation time. It inherits from the **Pacbase UpdateTableModel** class and implements the **TableModel** swing component, used to fill in a swing table with data. This generated class is used to display the list of instances of the **[Prefix]DataUpdate** class generated in the swing table.

If you checked the **Generate EJB Proxies Classes** option, the following classes are generated:

- **[Prefix]Session** class

This class represents the Remote Interface of the generated EJB Session.

- **[Prefix]SessionBean** class

This class corresponds to the generated EJB Session.

- **[Prefix]SessionHome** class

This class is the Home class for the generated EJB Session.

If you checked the **Generate XML schema and data** option, two classes and an XML schema are generated:

- **[Prefix]ProxyLvXMLMapping** class

This class represents the description of the mapping specific to a Folder or a Folder View. It inherits from the **XMLMapping** class.

- **[Prefix]ProxyLvXMLWrapper** class

This class offers methods which enable each node to identify the request. It inherits from the **XMLWrapper** class.

For example, the `getCustomerProxyLvHierarchicalDetailXML()` method returns `super.getXML(getCustomerProxyLv(), false, true);`

- **[Folder_Name].xsd** XML schema

This schema corresponds to the complete structure of a Folder or a Folder View. It is constituted of a structure (**ComplexType**) corresponding to the root node, which calls the dependent nodes' structures (**ComplexType**), etc. as they are described in the Folder or the Folder View.

Additionally, according to the **Deployment Descriptor** type chosen, one of the following files can be generated:

- **[Prefix].xml**

This file is generated if the deployment descriptor type is **XML**.

- **[Prefix].ser**

This file is generated if the deployment descriptor type is **Serialized**. This type is used for an EJB generation that complies with the EJB 1.0 specification.

Import

If the Proxy Generator has been launched from VisualAge for Java, you do not need to perform this step.

However if the Proxy Generator has not been launched from VisualAge for Java, you must specify the project into which the generated classes will be imported.

To do so, proceed as follows:

- From the VisualAge for Java Workbench, select the **Import...** choice in the **File** menu.
- In the **SmartGuide - Type of import** window, enter the name of the project into which classes are imported or select, via the **Browse** button, a project which has already been created.
 - ☞ We advise you not to import generated classes in a VisualAge Package or Java standard project.
- As for the import type, select the **Entire Directory (Including resources)** option.
 - ☞ The **Java files** option can also be selected. In this case, the icons specific to the eBusiness module will not be available.
- Click **Next** to get to the following detail. Select, via the **Browse** button, the directory in which classes have been generated.
 - ☞ Use the directory which was selected for generation in the **Output directory** field, in the generation option detail. In our example, it is **C:\vap\gen\java**.

Online Documentation of the Generated Classes

The documentation on the generated classes' public interface is directly integrated into the code in the form of comments.

You can therefore consult this documentation from the source of the desired element in the Workbench or a VisualAge browser.

You can also generate this documentation by using the Javadoc Facility delivered with the JDK.

• Generating Documentation

You can generate documentation associated with a project, a package, a class or even a method.

To start Javadoc, you have two possibilities:

- From the Workbench or a browser in VisualAge:
 - ♦ select the desired project, package, class or method.
 - ♦ open the associated pop-up menu and select the **Generate javadoc** choice. The documentation is generated by default in the VisualAge directory, ...**Ide\javadoc** Subdirectory.
- From a DOS or OS/2 window, by using the default parameters of the Proxy components generator, execute the following command:

```
javadoc -classpath c:\vap\generated\java -d c:\doc -public com.ibm.vap.generated.data com.ibm.vap.generated.proxies
```

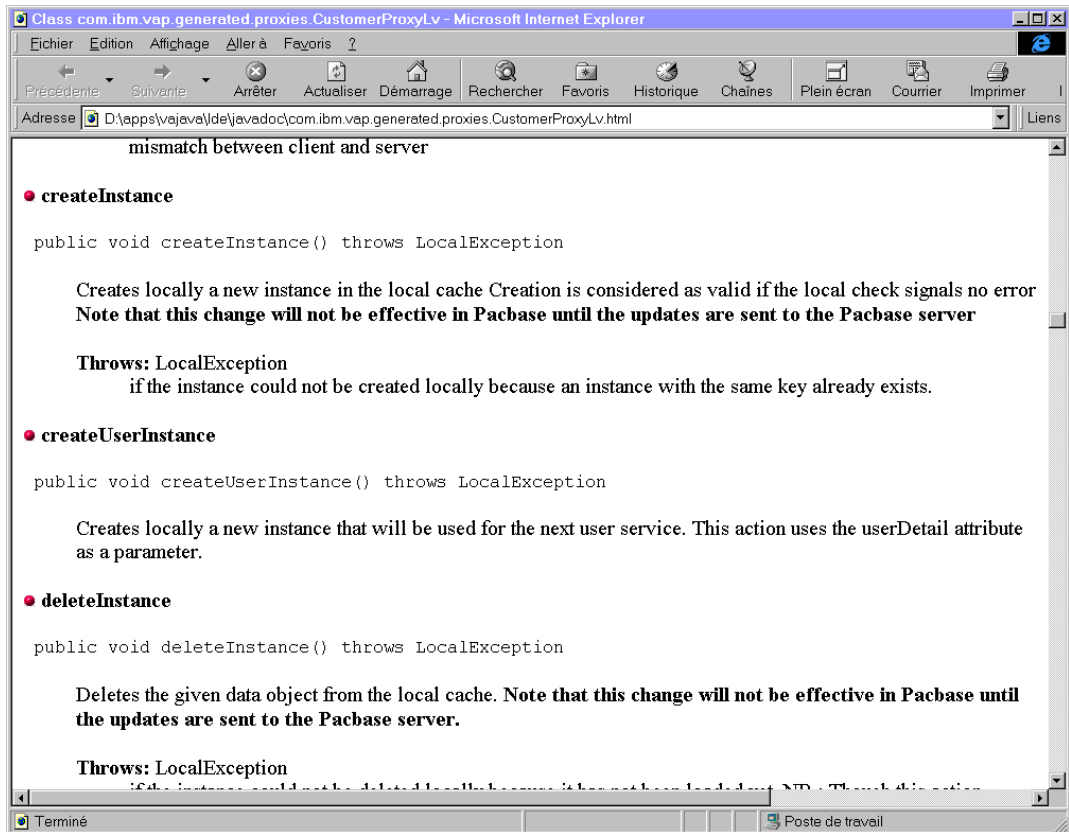
- **Results**

The generated documentation only contains information useful to the developer, for it corresponds exactly to each generated class.

Only the documentation of the public interface elements - properties or methods –**actually generated**– is extracted. It is an HTML-formatted documentation including hypertext links.

For each method, besides the comments, its signature – parameter(s), return code and exceptions – is extracted as well.

For information only, an example is presented:



You can refer to the *eBusiness & Pacbench C/S Applications: Proxy Programming Interface* manual where properties, methods and events are documented thematically.

Customizing Classes

All generated classes inherit from generic classes.

Generic classes are loaded upon the installation of the product.

You can implement new functions in the generated Proxy Components. You have two ways for customizing the generated classes. Do not modify parent and generated classes.

- **By following the inheritance system**

You can follow this inheritance system to create new classes allocated to the new functions to be implemented.

- Customization of ProxyLv Classes

You can add a behavior common to all Root, Reference or Dependent Proxy objects.

You just have to create a class inheriting from **DependentProxyLv**, **FolderProxyLv** or **ReferenceProxyLv**, then implement a new functionality and modify the parent class of the generated Proxy objects.

☞ You need to repeat this modification at each generation.

- Customization of Data Classes

You can change the **Data**, **UserData** and **SelectionCriteria** classes used by a given Proxy.

To do so, you have to define:

- ♦ a class inheriting from the **DataDescription** class, then perform the required implementation, and modify the **newData()** and **newData(String[] values)** methods of the Proxy.
- ♦ a class inheriting from the **UserDataDescription** class, then perform the required implementation, and modify the **newUserData()** and **newUserData(String[] values)** methods of the Proxy.
- ♦ a class inheriting from the **SelectionCriteria** class, then perform the required implementation, and modify the **newSelectionCriteria()** and **newSelectionCriteria(String[] values)** methods of the Proxy.

☞ The **UserData** class is generated only if a user service has been specified for the Proxy.

- **By Modifying the Generated Classes' Hierarchy**

When launching the generator, you can specify a configuration file. This configuration file enables you to modify the standard generation for the Java Proxies. For example, it allows you to modify the hierarchy of the generated classes by indicating the parent class of each type of generated class. This file format is the standard format of Java **.properties** file.

Example of configuration file:

```
#
# Default Configuration File for the VisualAge Pacbase for
# Java proxy generator.

#
# Folder proxy superclass
Folder.superclass = Folder

# DependentProxy superclass
DependentProxyLv.superclass = DependentProxyLv

# ReferenceProxyLv superclass
ReferenceProxyLv.superclass = ReferenceProxyLv

# UserBuffer superclass
UserBuffer.superclass = DataGroup
```

```

# DataDescription superclass
DataDescription.superclass = DataDescription

# DataDescriptionUpdate superclass
DataDescriptionUpdate.superclass = DataDescriptionUpdate

# SelectionCriteria superclass
SelectionCriteria.superclass = DataGroup

# PacbaseTableModel superclass
PacbaseTableModel.superclass =
com.ibm.vap.beans.swing.PacbaseTableModel

# PacbaseUpdateTableModel superclass
PacbaseUpdateTableModel.superclass =
com.ibm.vap.beans.swing.PacbaseUpdateTableModel

# Locale language
#Locale.language = fr
# Locale country
#Locale.country = FR
# Locale variant
#Locale.variant = EURO

# Default VapConfigurator class
Configurator.class = com.ibm.vap.generator.VapConfigurator

# Default PacbaseUpdateTableModel Status Column header
PacbaseUpdateTableModel.status.header = Status

# Default PacbaseUpdateTableModel Update Count Column
header
PacbaseUpdateTableModel.update_count.header = Update Count
# Default PacbaseUpdateTableModel LockTimestamp Column
header
PacbaseUpdateTableModel.lock_timestamp.header = Lock
Timestamp

# Default DataDescriptionUpdate Read Status Label
PacbaseUpdateTableModel.read.label = Read
# Default DataDescriptionUpdate Created Status Label
PacbaseUpdateTableModel.created.label = Created
# Default DataDescriptionUpdate Modified Status Label
PacbaseUpdateTableModel.modified.label = Modified
# Default DataDescriptionUpdate Deleted Status Label
PacbaseUpdateTableModel.deleted.label = Deleted

```

COM

Introduction

Once generation is over, the following files are created:

- The **VAPLOCAT.INI** location file is created in the generation output directory.

You complete this file by using the Location Editor tool. This tool enables you to parameterize the location file, thus avoiding syntax errors which might inhibit the communication between the Client (Proxy) component and the servers.

☞ For more information, refer to the online help of the Location Editor.

This file must be entered upon the gateway start.

- The source files of generated classes and resources required for editing them in the directory `<Foldername><Version>` (in our example: `VDCLINT2.0`).

Generated Classes

The elements generated by the eBusiness or Pacbench C/S module correspond to classes whose coding consists of a prefix and a hard-coded part assigned by the generator. The prefix corresponds to the Logical View's name, whose maximum length is 36 characters.

- **[Prefix]Data** Class
This class represents the description of a Logical View instance. It contains a set of properties which correspond to the Logical View Data Elements.
- **[Prefix]SelectionCriteria class**
This class represents the description of the selection criteria. It contains a set of properties which correspond to identifier Data Elements and extraction parameters associated with the Logical View.
- **[Prefix]Buffer** Class
This class represents the description of the contextual information. It contains a set of properties which correspond to the user buffer Data Elements.
- **[Prefix]DataUpdate** class
This class inherits from the **[Prefix]Data** class. Compared to its parent class, it contains two additional properties whose values vary according to the modifications in progress. These two properties are:
 - ♦ **action** : update action which can be **Read, Modified, Created** or **Deleted**. This property is only visible in the list of the **UpdatedFolders** property.
 - ♦ **updatedInstancesCount** : number of useful server updates associated with the relevant Folder instance. Its value can be 1 to n.
- **[Prefix]UserData** class
This class inherits from the **[Prefix]Data** class. It contains an additional property which corresponds to the key Data Element of the parent instance.

If you checked the **Generate XML schema and data** option, two classes and an XML schema are generated:

- **[Prefix]ProxyLvXMLMapping** class
This class represents the description of the mapping specific to a Folder or a Folder View. It inherits from the **XMLMapping** class.
- **[Prefix]ProxyLvXMLWrapper** class
This class offers methods which enable each node to identify the request. It inherits from the **XMLWrapper** class.
- **[Folder_Name].xsd** XML schema

This schema corresponds to the complete structure of a Folder or a Folder View. It is constituted of a structure (**ComplexType**) corresponding to the root node, which calls the dependent nodes' structures (**ComplexType**), etc. as they are described in the Folder or the Folder View.



For more information, refer to the *eBusiness & Pacbench C/S Applications: Proxy Programming Interface* manual.



A brief description of the public interface is available in online mode. You can view it if your client workstation enables it.

Compilation Results

If compilation has ended normally, the compiled elements are located in the **<Foldername><version>\Release** directory (in our example **VDCLNT2.0\Release**).

A report is available in the **<Foldername><version>\Resume.txt** file.

Once generation and compilation are completed, the Proxy can be immediately used on the workstation where the generator is installed.

Compiling with Visual C++ Version 5.0 and 6.0

At the end of the generation, you get pieces of code which you will use to compile your Proxy.

Two make files are generated:

- **C<Foldername>.mak** for the C++ static library,
- **<Foldername>.mak** for the COM component (DLL or OCX)

You can compile each component by launching, in a DOS window, the **compil_C<Foldername>.bat** and **compil_<Foldername>.bat** files. These files are processed by the Visual Studio C++ compiler.

Chapter 3: Development Principles

This chapter introduces you to the main concepts of development: presentation of some methods, properties and events, error management, and communication management.








- ☞ If your Proxy contains only one Elementary Proxy (a Root Proxy), the properties, methods and events associated with large reading, reference or dependent nodes are not available.
- ☞ For convenience, the term *property* designates both a Java property and a COM attribute, and the term *method* designates both a Java method and a COM action.

Visual Representation of Proxies in the Target Environment

Java Environment

Once it has been imported in the VisualAge workstation, a Folder View Proxy can be handled through a graphic bean.

You will find below a presentation of the icons relating to the Proxy objects provided at the installation.

Icons	Types of Elementary Proxy objects
	Root Proxy
	Dependent Proxy 0,N
	Dependent Proxy 0,1
	Dependent Proxy 1,N
	Dependent Proxy 1,1
	Reference Proxy 0,1
	Reference Proxy 1,1

COM Environment

Once the ActiveX Proxy is generated and compiled, it can be graphically integrated in any client language supporting the COM standard. The Proxy is graphically represented by the following icon:



Use of Properties

A property corresponds to a piece of information handled by a Proxy object. This piece of information defines an elementary data item, a list of elementary data or a list of composite data instances. A property corresponds either to a constant, a parameter or an action result. According to the context, it is initialized by the graphic application or the Proxy.

Two kinds of properties are found:

- those standing for technological variables. They enable to adjust the behavior of the Proxy objects in the target environment.
- Those corresponding to the Logical View data.



The availability of a property depends on the Proxy type. All the public interface properties are documented in the *eBusiness & Pacbench C/S Applications: Proxy Programming Interface* manual.

Local Checks

The Elementary Proxy automatically performs local checks if an instance is created or modified via the `createInstance` or `modifyInstance` method. Each Data Element belonging to the Logical View is checked.

The following checks are performed:

- Checks on the value lists defined in the Data Element description
- Checks on the value ranges defined in the Data Element description
- Checks on the presence of the mandatory Data Elements called in a Logical View. The presence of identifier-type Data Elements or foreign key-type Data Elements is automatically checked for a reference relation with a minimum cardinal value of 1.

If local checks detect an error, an error message is set via the Error Manager.

These checks can be selectively triggered for each root or dependent-type node in the Folder concerned. The property used to activate or deactivate the check of Data Elements on the server is `serverCheckOption`.

Whether the message is sent or received, the detection of an empty Data Element is automatic.

However, the Elementary Proxy does not perform numeric or date checks ; these are performed by the graphic application.

Check of the Length of the `detail` Property Fields

For each field of the `detail` property, the checks performed upon the local creation or modification of an instance ensure that the length of the value contained in the property does not exceed the maximum length of the value for this property.

The length is checked systematically (except if the property does not belong to the current sub-schema), even if the property has been defined as 'not to be checked in the client'.

A local error is sent if the length is excessive.

- ☞ The length of the fields included in user buffers is not checked. If it is excessive, the length is truncated to its maximum value.

Selecting the Local or Server Sort Criterion on a List of Instances

The `localSort` property enables you to specify whether the Proxy sorts the instances of the `rows` property according to the local sort (`true`) or keeps the instances in the order they were sent by the server (`false`).

You can change the sort type at any moment.

- ☞ This property is not effective in user services.

Local Sort

In standard, the instances of the `rows` property are sorted according to the local criterion if the parameter has not been changed after the generation. In this context, two sort types exist:

- If no sort criterion has been locally defined (see paragraph *Specification of the Local Sort Criterion (Java Only)*), the Proxy implicitly sorts the instances in the increasing order of the identifiers defined on the Logical View.
- If a local criterion has been locally defined, the Proxy sorts the instances in the order defined by this criterion.

In all cases, when an instance is created locally, it is inserted according to the current sort criterion applied in the `rows` property.

If you change dynamically or cancel the local sort criterion, the instances contained in the `rows` property are immediately sorted according to the new criterion.

Server Sort

The instances of the `rows` property are sorted according to the server criterion if the `localSort` property is set to `false`. In this context, the instances contained in the `rows` property are displayed in the order sent by the server, without taking the local sort criterion into account.

If collections are managed manually or in the case of a paging in extend mode, the instances sent by the server are added at the end of the existing collection in the `rows` property.

All the locally-created instances are added at the end of the existing collection in the **rows** property. In this context, an instance which is not positioned at the end of a collection but which is deleted and created again locally is transferred to the end of the collection contained in the **rows** property.

Specification of the Local Sort Criterion (Java Only)

You can dynamically change the sort criterion used to present instances in the **rows** property. To do this, use the **dataComparator** property of each Proxy: **void setDataComparator(Comparator c)**.

The required parameter is an instance of the class implementing the following interface **com.ibm.vap.generic.Comparator**.

This interface consists of a method representing the following relation:

```
int compare(Object a, Object b).
```

This method must return:

- a negative number if $a < b$
- 0 if $a = b$
- and a positive number if $a > b$.

For example:

```
import com.ibm.vap.generic.Comparator ;
public final class CustomerComparator implements Comparator {
public static final int NAME = 0 ;
public static final int COMPANY = 1 ;
public int criteria ;
public int compare(Object a, Object b) {
try {
switch(criteria) {
case NAME:
return ((CustomerData) a).getName().compareTo(
(CustomerData) a).getName());
break;
case COMPANY:
return ((CustomerData) a).getComp().compareTo(
(CustomerData) a).getComp());
break;
}
} catch (IllegalAccessException ice) {
return 0;
}
}
}
```

Table Model (Java Only)

This property is available in read/write mode on all types of nodes, provided you chose the generation option **Use Swing**.

This property enables you to insert a JTable in your application (a JTable is a swing component made up of rows and columns).

By default this property is initialized with a new instance of a generated TableModel.

Sub-schema Management

The sub-schemas specified in the Logical View's description can be taken into account when selection, read or update methods are executed, provided Elementary Components manage the presence of Data Elements (activation of parameters **Presence vectors generation** or **Data control** in the Definition tab of the Elementary Component in Developer workbench, or options **VECTPRES=YES** or **CHECKSER=YES** in the Va Pac WorkStation).

Each node has two properties:

- **subSchema**, via which you can assign the desired sub-schema when a selection, read or update method is executed by the Elementary Component of the node. The value of this property can be assigned via the **subSchemaList** property.
- **subSchemaList**, via which all the sub-schemas available on a node are listed. Since a sub-schema cannot be given a name in the VisualAge Pacbase Repository, each sub-schema is designated by **SubSchema<n>** (with **n** = 01 to 10).

☞ In a COM environment, the **subSchema** attribute cannot be accessed directly but only through the **getSubSchemaCount** and **getSubSchemaElementAt (Int index)** actions.

Use of Methods

A method corresponds to a process which can be executed by a Proxy objet. It is triggered using a connection between an event of the graphic application and the code of a method available in a Proxy.

☞ The availability of a method depends on the Proxy type. All the methods of the public interface are documented in the *eBusiness & Pacbench C/S Applications: Proxy Programming Interface* manual.

The Different Types of Server Methods

The server methods execute procedures implemented in one or more Elementary Components associated with the Folder. These methods send a request to the Elementary Components which send back a result on the workstation. The queries and responses generally contain technical parameters, Logical View instances associated with one or more nodes and contextual data defined in a user buffer.

You must distinguish two types of server methods

- Those which systematically access the server:
 - ♦ **selectInstances**
 - ♦ **readInstance**: This method retrieves only one instance.

- ♦ **readInstances**: This method retrieves more than one instance, according to specified selection keys.
- ♦ **readInstanceAndLock**
- ♦ **readInstanceWithFirstChildren** (Java) or **readWithFirstChildren** (COM)
- ♦ **readInstanceWithAllChildren** (Java) or **readWithAllChildren** (COM)
- ♦ **readInstanceWithFirstChildrenAndLock** (Java) or **readWithFirstChildrenAndLock** (COM)
- ♦ **readInstanceWithAllChildrenAndLock** (Java) or **readWithAllChildrenAndLock** (COM)
- ♦ **readAllChildrenFromDetail.**
- ♦ **readAllChildrenFrom** (Java) or **readWithAllChildrenFrom** (COM):
- Those which do not systematically access the server:
 - ♦ **readNextPage**: The server is accessed except if the event (**noPageAfter** for Java or **NO_PAGE_AFTER** for COM) indicating there is no page after the current page was sent back during the previous selection
 - ♦ **readPreviousPage**: The server is accessed except if the event (**noPageBefore** for Java or **NO_PAGE_AFTER** for COM) indicating there is no page after the current page was sent back during the previous selection.
 - ♦ **readFirstChildrenFromDetail**: the server is accessed, except if the property controlling the maximum number of requested instances (**maximumNumberOfRequestedInstances** for Java or **maxNumberOfRequestedInstances** for COM) of the Dependent Proxy objects are set to 0 and if their **globalSelection** properties are set to false.
 - ♦ **checkExistenceOfDependentInstances** for Java or **checkExistenceOfDependencies** for COM: The server is accessed except if the existence of dependent instances can be checked locally.
 - ♦ **updateFolder**: The server is only accessed if there is at least one instance of the node concerned, modified in its **updatedFolders** property.

Managing Folder Reading

Large reading of a Folder root enables to read from a client component all the instances of the Folder's root node existing in the database. The methods concerned are **selectInstances** and **readNextPage**.

Provisional Large Reading of Dependent Nodes

In the context of client/server architectures, a graphic application handling a Folder strives to anticipate data reading to minimize exchanges with the servers.

In a hierarchical network, various provisional reading methods are possible:

- The '**allChildren**' type method reads *all* the dependent instances of the instance selected in the **detail** property of the parent Elementary Proxy.

- The `'firstChildren'` type method only reads the instances which are *immediately* dependent of the selected instance in the `detail` property of the parent Elementary Proxy.

The first provisional large reading method is available on the Root Proxy only.

The second provisional large reading method is available on the Root Proxy or on the Dependent Proxy which themselves hold Dependent Proxy objects.

Transferring an Instance Between the rows and detail Properties

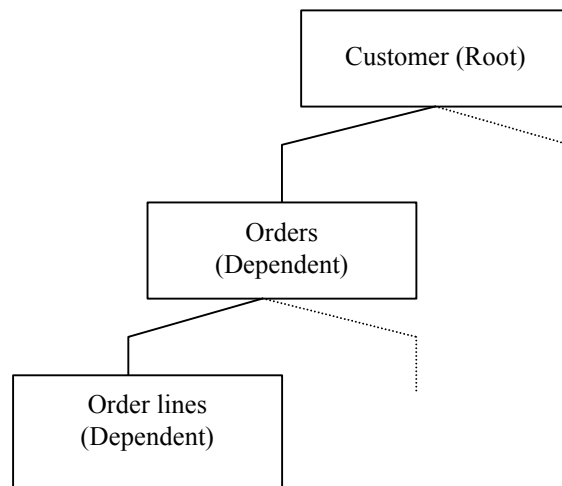
The transfer of an instance between the `rows` and the `detail` properties enables the `detail` property to be loaded with an instance initially retrieved by a large reading method.

This transfer is only available for Root and Dependent Proxy. It corresponds to a local reading method which also loads all the local instances of Dependent Proxy known by the Folder View Proxy. The transfer is made using the method `getDetailFromDataDescription` for Java or `getDetailFromData` for COM.

Large Reading and Transferring an Instance Between Rows and Detail Properties: Working Mechanism

This example illustrates the loading of the `detail` property with an instance previously retrieved in the `rows` property by a large reading method on a root or dependent node, and the principle of the provisional large reading of dependent nodes.

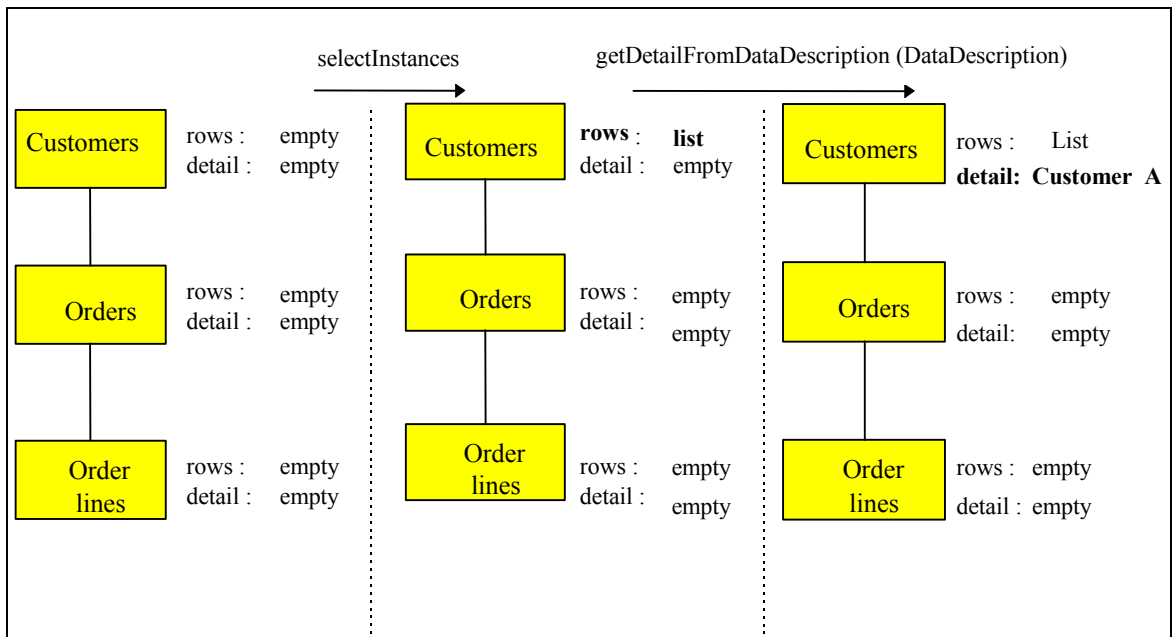
It is based on a Folder View Proxy, which consists on three Elementary Proxy objects:



The schemas below introduce the working mechanism based on this principle.

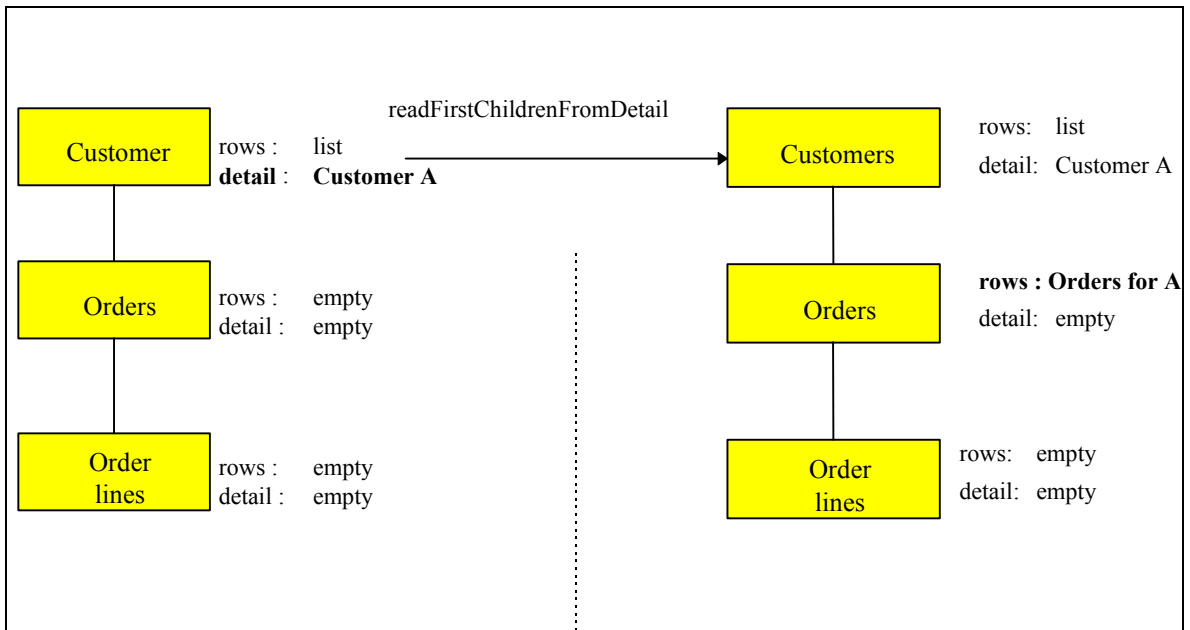


The method codes used in this example correspond to the Java codes but the same principle applies to the COM environment.



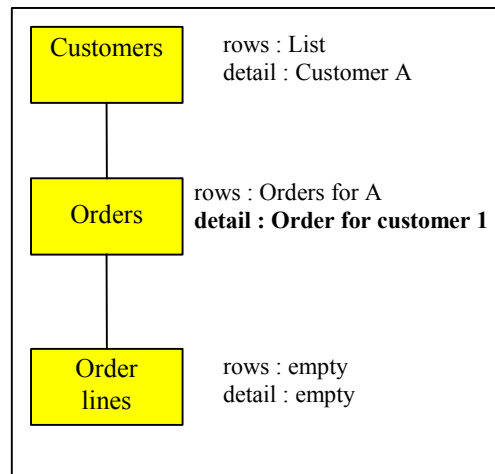
The **detail** property of the customer node now contains Customer A. There are three solutions to read Customer A's dependent instances (i.e. his associated orders).

SOLUTION 1



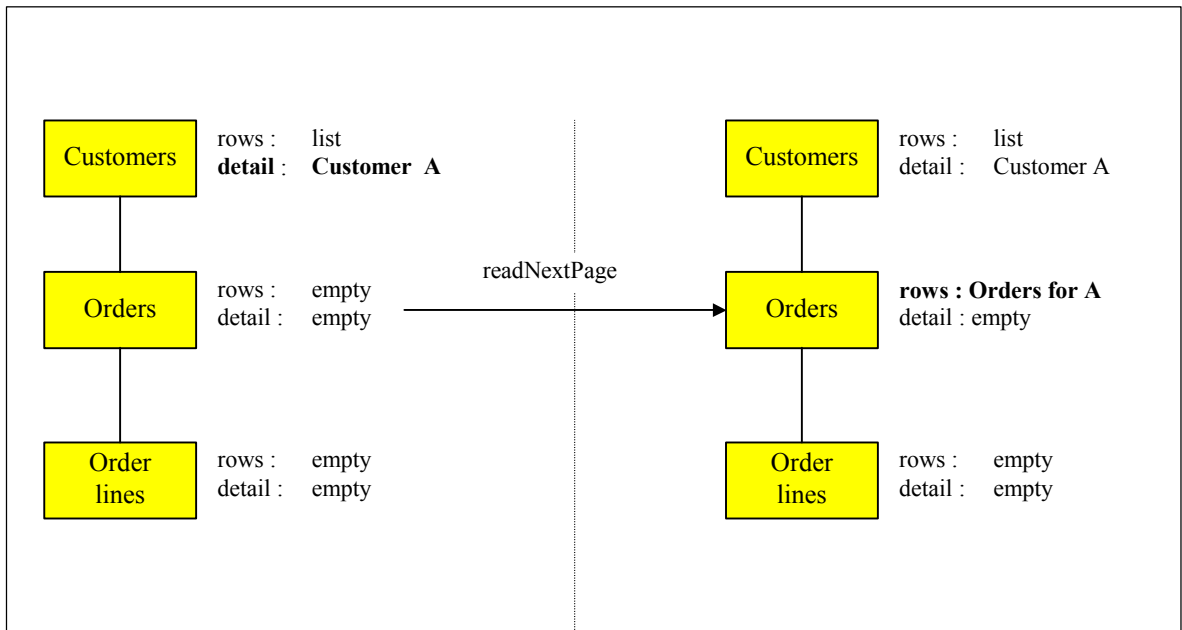
The **readFirstChildFromDetail** method on the Customers Root Proxy not only reads the **rows** property of Order Lines Dependent Proxy for Customer A, but it also reads the **rows** property of other possible Elementary Proxy objects directly dependent of the Root Proxy.

In this context, the **getDetailFromDataDescription (DataDescription)** method on the Orders Dependent Proxy initiates:



Then, to read the order lines of the Order 1 of Customer A, use the **readFirstChildFromDetail** method on Orders or **readNextPage** method on Order Lines.

SOLUTION 2

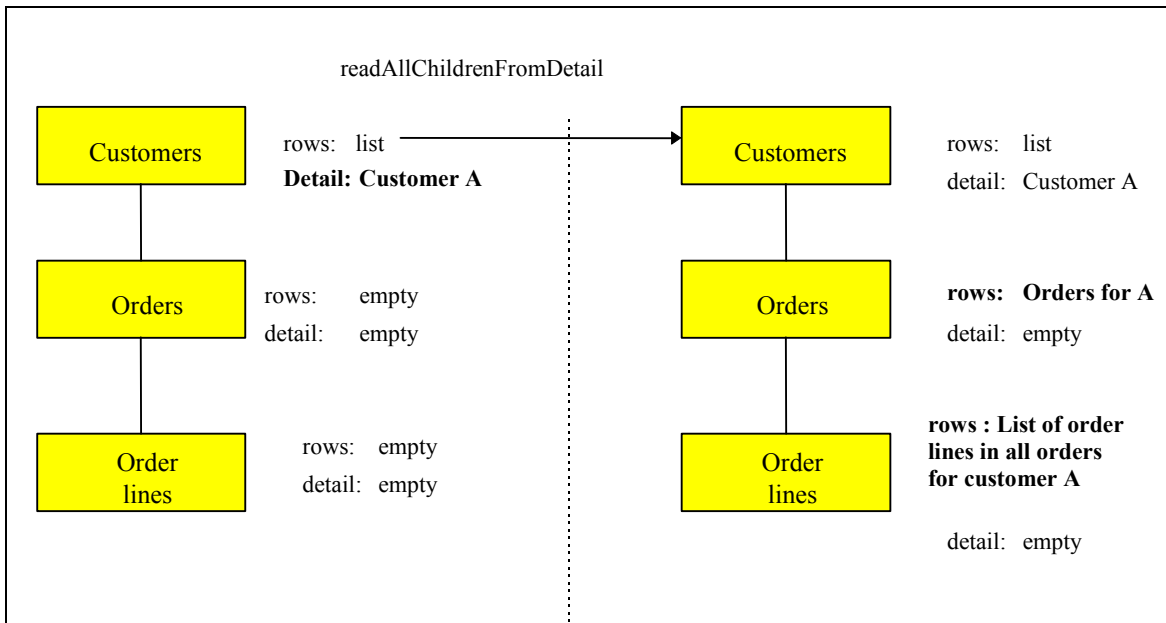


The `readNextPage` method on the Orders Dependent Proxy loads its `rows` property. The instances of other possible Elementary Proxy object, which are directly dependent of the Root Proxy, are not read.

In this context, the `getDetailFromDataDescription (DataDescription)` method on the Orders Dependent Proxy initiates the same result as in solution 1.

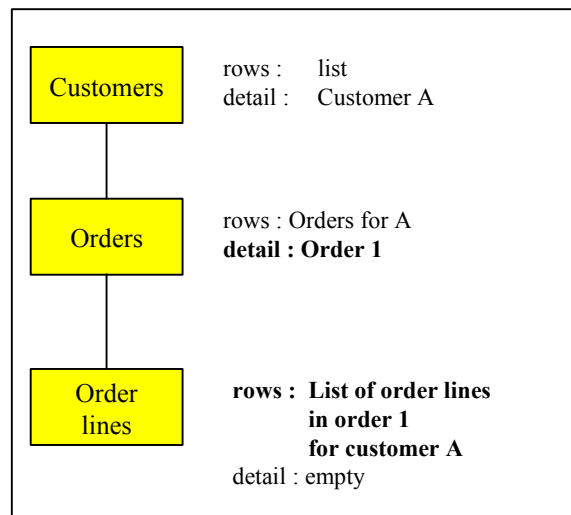
To read subsequently the order lines of the Order 1 of Customer A, proceed as in solution 1.

SOLUTION 3



The `readAllChildrenFromDetail` method on the Customers Root Proxy reads not only all the orders for A but also all the other possible instances dependent of A whatever their hierarchical level can be. In our example, all the order lines for all the orders of A are therefore read.

In this context, the `getDetailFromDataDescription (DataDescription)` method on the Orders Dependent Proxy initiates:



The `getDetailFromDataDescription (DataDescription)` method automatically loads the `rows` property of the Order lines Dependent Proxy with the order lines contained in Order 1 for Customer A. These order lines have been previously transferred to the workstation using the `readAllChildrenFromDetail` method.

Large Reading of Reference Nodes

The reading of a reference node is considered as aid on criteria. It shows the end user a list of information which can be referred to for a dependent node.

The information presented to the end user is both necessary and sufficient to assist him in making a choice.

To optimize the volume of characters sent for this type of service, the Logical Views have a « aid on criteria »-type subschema used to select the concerned Data Elements at the server level.

The large reading of reference nodes is executed on request and cannot be involved in provisional large reading. The methods concerned are **selectInstances** and **readNextPage**.

Principle of Paging in a Folder's Nodes

Two types of paging are offered on a Folder's nodes:

- The first, called *non-extend* paging, is used to paginate forwards and backwards on a predefined collection through specific methods. Each method executes a read request to the server and its result overwrites that of the previous read. This type of paging is available only on root or reference nodes.
- The second type of paging, called *extend* paging, is used to gradually retrieve the instances of a defined collection as read requests for following pages are made. In this context, the backwards paging function disappears and is performed locally by the scroll box of the graphic control which presents the list of instances. This type of paging is available to all the nodes of a Folder.

Selection Criteria Associated with Large Reading Methods

The selection criteria associated with large reading methods are elementary or composite properties associated with each node of a Folder. They are split into two types:

- Functional selection criteria corresponding to the identifier and to elements required for defining extraction methods for the Logical View associated with the node. These criteria are the following ones:
 - ♦ **selectionCriteria** which defines identifier Data Elements and parameters by value.
 - ♦ **extractMethodCode** which defines the code for the extraction method desired.
 - ♦ **extractMethodCodes** for Java contains the list of the available extraction methods.
 - ☞ In the COM target, the **getExtractMethodCodesCount** and **getExtractMethodCodesElementAt(Int i)** actions enable you to access the list of the available extraction methods.
- Organic selection criteria corresponding to information used to control the volume of instances selected for each node.
 - ♦ **globalSelection** is a Boolean property which, when set to true, is used to retrieve all instances of the node via a selection request.

- ♦ **maximumNumberOfRequestedInstances** (Java) or **maxNumberOfRequestedInstances** (COM) is a numerical property which specifies, when the **globalSelection** property is set to false, the number of instances to be read for a node via a selection request. This property can hold the value 0. In this case, the concerned part of the tree structure is not read during a provisional large reading.

Limitation of the Scope of Large Reading

During a large reading using selection criteria, only the instances corresponding to these criteria are read.

However, in the case of a large reading ordered by an **allChildren** provisional reading type, the **globalSelection** property is considered to be set to true on each node.

Reading of a Root Node or Dependent Instance

The reading of a dependent or root node instance enables you to retrieve an instance of the node without previously making a large selection of a collection of instances for this node. It directly loads the node's **detail** property.

This type of reading is considered as a collection selection and therefore cancels the previous selection even if it was the result of a large reading method. The loading of the **detail** property therefore initializes the **rows** property.

The methods which implement this selection function are used to:

- Retrieve the instance without its dependencies (**readInstance**).
- Retrieve the instance with its first level dependencies (**readInstanceWithFirstChildren**) for Java or **readWithFirstChildren** for COM.
- Retrieve the instance with all its dependencies (**readInstanceWithAllChildren**) for Java or **readWithAllChildren** for COM.

Reading of a Reference Node Instance

The reading of a reference node instance cannot activate the provisional large reading process. It is used to retrieve the entire description of the instance of the node called in its **detail** property.

Only the **readInstance** method is therefore available on a reference node.

Update methods are not available for reference nodes. Their **rows** and **detail** properties are independent. The result of the **readInstance** method does not therefore initialize the **rows** property.

The **rows** property is used to display sufficient information to assign one of the reference node instances to the referencing node instance.

☞ In the COM environment, the **rows** attribute cannot be accessed directly but only through the actions **getRowCount()** and **getRowsElementAt(Int i)**.

The **detail** property is used to view the entire description of a referenced instance.

The structure of these two properties can thus be different.

Selection Criteria Associated with Instance Reading

The identifier of the node instance to be read is specified in the functional selection criteria associated with the node.

For dependent nodes, the identifier of the node corresponds to the identifier of the Logical View associated with the node, discarded from the Data Elements which are the identifiers of Logical Views higher in the hierarchy. These hierarchical identifiers are automatically initialized by the Folder View Proxy according to the navigation in the Folder.

Folder Update Management

Local Updates

Local update services are available on each root or dependent node in the Folder.

- Create a node instance
- Modify a node instance
- Delete a node instance

However, there are certain rules specific to Folder management:

- The creation of a dependent node instance is only authorized if the hierarchy of the instances contained in the **detail** properties of higher nodes exists.
- Deleting a node instance initiates the recursive deletion of local instances of dependent nodes.

To allow the developer to manage messages which can warn users of the impact of a cascade deletion, a method which checks the existence of dependent instances is available on root or dependent nodes.

This method (**checkExistenceOfDependentInstances** for Java or **checkExistenceOfDependencies** for COM) sends a Boolean result which is either true or false. If no dependency is found in the Folder's local cache and the instance concerned is not created locally, this method sends a check request to the server.

To enable a user to undo local updates on a Folder instance, an **undoAllLocalFolderUpdates** method can be used to discard all local updates on all Folder nodes applied since the last server update method. This method is only available on the Folder's root node. Another method **undoLocalFolderUpdates** can be used to eliminate all updates associated with the Folder's Root node you have parameterized.

Server Updates

Only the Root Proxy provides server update methods.

Server updates correspond to methods which enable a client component to send all local updates made since the last server update method.

These updates concern all the modified dependent instances. They can concern several Folder instances.

When a server update method sends back errors, the Folder remains with the « Modified Locally » status; new collection selections can be made only by correcting the errors and sending back the updates, or by using the `undoAllLocalFolderUpdates` method or the `undoLocalFolderUpdates` method.

Before the request is sent to the server, the server update methods check the Folder integrity for locally created instances. For each locally created instance of the node, this method checks the minimum cardinal values of each link and sends an error if the number of dependent instances does not respect the properties of the associated links.

A server update method can be accompanied by a request to refresh the updated instances if some of their Data Elements, such as the identifiers, are calculated by the server. This refreshment request is made using the `refreshOption` property.

Management of Effective Transactions

The management of effective transactions is automatically carried out by the local cache.

It consists in calculating the resulting update of various local updates made on the same instance of the Folder's node. It controls the creation of duplicate instances. If several local updates have been made on the same instance of the node, only the last one will be sent to the server.

Re-initializing instances in the local cache

The `resetCollection` method is used to remove all the instances from the cache of a Folder View Proxy before initializing a new collection of instances. This method can be executed by all types of nodes in a Folder View Proxy containing a `rows` property.

☞ The `rows` attribute cannot be accessed directly but only through the methods `getRowCount()` and `getRowElementAt(int i)`.

Managing collections of instances

The management of collections of instances can be carried out automatically, or manually by positioning the `manualCollectionReset` Boolean property which is available for all types of nodes in a Folder View Proxy. The manual management mode is used to create heterogeneous collections through a series of selection and paging methods.

Load of the local cache with no server access

The `initializeInstance` method allows to store, in the local cache, a Logical View instance that has not been either read by the server or created locally. It allows the update of the Logical View instance though it has not been previously read from the server. This method is available for all types of nodes and is valid when the Logical View instance does not exist locally.

Asynchronous Methods

Principles

The asynchronous programming is used to dissociate the method used to send a request from the method used to retrieve its response. You can use this type of programming whether you use an asynchronous communication protocol or not.

You can use the Proxy components in asynchronous mode independently of the middleware used. As for the Proxy, you can work with an asynchronous mode, by using a *location* whose middleware is synchronous, and the other way round.

In this context, the end user will be more efficient as he can send a request in advance, and retrieve the response when he needs it. This method allows you to optimize your working time.

Furthermore, the communication protocols are used to make the requests or the responses in the local messages threads more secure by allowing the message to be conveyed, whichever the network situation is.

The communication mode is defined by the **Asynchronous** Boolean property at the Folder level.

The **getLastReplyContext** method enables you to retrieve the sending context of the request (**serverActionContext**).

In Java, you can also use **com.ibm.vap.generic.AsynchronousRequestException** to retrieve the sending context of the request.

If no error has been detected and if the response has been processed, **getReply(context)** returns true.

Global Methods or Methods Associated with an Instance

Some server methods, labeled as global methods, are independent of any selection, whereas others depend on any instance included in the local cache. In asynchronous mode, global methods store the response identifiers in a collection and each executed request adds its identifier to this collection. The methods associated with a Logical View instance store their response identifiers in a collection associated with the concerned instance. The collections of response identifiers associated with the global methods are lost when the application using the Proxy is closed. A collection of response identifiers associated with an instance contained in the local cache is lost when this instance is locally deleted or after a change of collection.

- **global methods**

The responses associated with the following methods can be executed independently of the current collection:

- **executeUserService ()**
- **readInstance ()** (ROOT)
- **readInstances ()** (ROOT)
- **readInstanceWithFirstChildren ()** (ROOT)
- **readInstanceWithAllChildren ()** (ROOT)

- `readInstanceAndLock ()` (ROOT)
- `readInstanceWithFirstChildrenAndLock ()` (ROOT)
- `readInstanceWithAllChildrenAndLock ()` (ROOT)
- `readNextPage ()` (ROOT)
- `selectInstances ()`
- `lock ()`
- `unlock ()`
- `readPreviousPage ()`

- **Methods associated with an instance**

The following methods depend either on the `detail` instance, or on the instance passed as a parameter, or on the `detail` parent instance for dependent nodes:

- `checkExistenceOfDependentInstances ()`
- `readAllChildren (data)`
- `readFirstChildren (data)`
- `readAllChildrenFromCurrentInstance ()`
- `readAllChildrenFrom ()` (DEP)
- `readFirstChildrenFromCurrentInstance ()`
- `readFirstChildrenFrom ()` (DEP)
- `readInstance ()` (DEP)
- `readInstances ()` (DEP if maximum cardinality is 'n')
- `readInstanceWithFirstChildren ()` (DEP)
- `readInstanceWithAllChildren ()` (DEP)
- `readInstanceAndLock ()` (DEP)
- `readInstanceWithFirstChildrenAndLock ()` (DEP)
- `readInstanceWithAllChildrenAndLock ()` (DEP)

Examples

There are two ways to use asynchronous methods:

- **Polling**

This system consists in 'watching for' a response in the *thread* with no risk of blocking the application while waiting for the information display. The response code is stored in a *thread*, which is different from the one in the main application, but it points on the Root Proxy. We also assume that it knows a context associated with an asynchronous method.

```
while (!myFolder().getReply(context)) {
    wait(1000);
}
```

- **Background job access**

The following example describes how to store information on the dependent instances before the end user explicitly requests it, and with no risk to block the application.

- When the user chooses a collection of radical instances:

```
myFolder.setAsynchronous(false);
myFolder.selectInstances();
```

☞ As its result is required for the continuation of the operation, the **selectInstances** method is used with a synchronous mode.

- Then, **rows** will be browsed so as to find all the dependent instances of each instance read by the **selectInstances** method.

You need first to switch to the asynchronous mode:

```
myFolder.setAsynchronous(true);
Enumeration rows = myFolder.rows.elements();
Vector contexts = new Vector();
while (rows.hasMoreElements()) {
    Data currentData = (Data)rows.nextElement();
    try {
        myFolder.readAllChildren(currentData);
    } catch (VapException ve) {
    } catch (AsynchronousRequestException are) {
        contexts.addElement(are.getContext());
    }
}
myFolder.setAsynchronous(false);
```

- Then, when the user wants to work in a data:

```
try {
    int index = myFolder.rows().indexOf(data);
    myFolder.getReply(contexts.elementAt(index));
} catch (VapException) {} //Everything is OK //
myFolder.getDetailFromDataDescription(data);
```

This way, the dependent instances are immediately displayed in the selection tree. Note that the methods' responses are processed at the last moment, but this is not compulsory.

Storing the Proxy Context

You can store the reading context of a Proxy.

Using the **getProxyContext** method on the root node will store the current keys on each node, the next and previous keys, the selection criteria, the current detail and the local updates.

Using the **initFromProxyContext** method and giving a previously-stored context will restore all the reading keys of your Proxy. You will then be able to read from where you stopped before storing the context.

Externalization of the Management of Requests

You can manage the services requests in a specific object which is an instance of the **MainRequest** class. You can then post services requests which are sent by different Proxies before sending only one request to the server.

Proxies then share the same execution context.

You initialize the request on any root Proxy with the `createRequest` method and you indicate which root Proxies participate in the request with the `setRequest` method. The request is sent to the server via the `sendRequest` method.

The Proxies which participate in the request must belong to the same eBusiness Application as the Proxy which creates the request.

User Service

Java

Each root or dependent type node contains the following elements to implement a user service:

- A property used to obtain the list of user services available on this node plus `nil`. This property is `userServiceCodes`.
- A property used to initialize the user service code to be executed. This property is `userServiceCode`.
- A property used to locally store Logical View instances to be processed for the next user service. This property is `userServiceInputRows`.
- A property used to present various Logical View instances sent back by a user service. This property is `userServiceOutputRows`.
- A property which presents the Logical View instances which are candidates for the execution of the next user service. This property is `userDetail`.
- Local methods used to memorize each Logical View instance to be sent to the server to execute a user service. These methods are `createUserInstance`, `modifyUserInstance` and `deleteUserInstance`.

The root node of the Folder also has the following elements:

- A method used to execute all the user services parameterized on each Folder node. This method is `executeUserServices`.
- A method used to delete all the local instances stored for all the Folder nodes. This method is `resetUserServiceInputInstances`.
- A method used to delete the current instance stored locally. This method is `resetUserServiceCodes`.

This principle means that 1 to n user services can be executed, in the same request, distributed on the different nodes. The execution sequence of these services corresponds to the hierarchical order of nodes, browsing the tree from top to bottom and from left to right.

COM

Each root or dependent type node contains the following elements to implement a user service:

- Two actions enable you to obtain the list of available user services on the node. These actions are `getUserServiceCodesCount()` and `getUserServiceCodesElementAt(int i)`.

- Two actions enable you to locally store Logical View instances to be processed for the next user service. These actions are `getUserInputRowCount ()` and `getUserInputRowsElementAt (Int i)`.
- Two actions enable you to present various Logical View instances sent by a user service. These actions are `getUserOutputRowCount ()` and `getUserOutputRowsElementAt (Int i)`.
- An attribute which presents the Logical View instances which are candidates for the execution of the next user service. This attribute is `userDetail`.
- Local actions used to memorize each Logical View instance to be sent to the server for the execution of a user service. These actions are `createUserInstance`, `modifyUserInstance` and `deleteUserInstance`.

The root node of the Folder also has the following elements:

- An action used to execute all the user services on each Folder node. This action is `executeUserServices`.
- An action used to delete all the local instances stored for all the Folder nodes. This action is `resetUserRows`.
- An action used to delete the current instance stored locally. This action is `resetUserServiceCodes`.

This principle means that 1 to n user services can be executed, in the same request, distributed on the different nodes. The execution sequence of these services corresponds to the hierarchical order of nodes, browsing the tree from top to bottom and from left to right.

Database Logical Lock

The upload-download mechanisms associated with a Folder increase the elapsed time between reading the initial Folder image and displaying the result of an update.

In this context, with no lock mechanism, two users can modify the same Folder instance. The result of accumulated updates are therefore difficult to manage.

To enable the user to use a Folder in an exclusive appropriation mode, two types of locks for a node instance are available:

- The optimistic lock which works on the principle of verifying the change of a `TimeStamp` before executing the update procedure.
- The pessimistic lock which uses an entity for exclusive update by recording a specific resource. In this case, the Folder update procedure is carried out before freeing up the exclusive resource.

The server lock procedure is triggered by the explicit execution of a specific method available on the Root Proxy. This method is `lock`.

The server unlock procedure is triggered automatically with the execution of a server update method or explicitly by the execution of a specific method available on the Root Proxy. This specific method is `unlock`.

The developer is responsible for writing the lock processing in the root Elementary Component.

This processing receives the identifier of the Logical View instance to be locked as well as the request type (lock or unlock) to be executed.

In return, it must set a status used to grant or refuse the lock and the **TimeStamp** or the name of the resource used.

If the lock is refused, the Root Proxy sends an event whereby all subsequently executed local or server update methods are disabled for the specified instance of the Folder. This event is **lockFailed** for Java and **LOCK_FAILED** for COM. In this case, the Folder changes to the « Read only » status.

The concept of the logical lock is defined in the Folder entity or in the Elementary Component for a single-view development.

When the logical lock method is active on a Folder, all read requests for the **detail** property of the Root Proxy can be accompanied by a logical lock request on the server.

Customization of the Columns of a Jtable (Java Only)

A JTable is a swing component available from VisualAge Java version 2.0 onwards.

If you insert this component as is, it will display, when the application is executed, all the columns which correspond to all the Data Elements of the Logical View, with the clear names defined in the Logical View.

To select the columns to be displayed, to change their heading or to create a new column which will display data locally computed, you must customize the JTable.

To do so, you must first create a new public class which inherits either from the generated TableModel (this is useful to retrieve part of its implementation) or directly from **PacbaseTableModel** (**com.ibm.vap.beans.swing** package).

Then you simply have to customize the following methods:

- **public int getColumnCount()** : retrieves the number of columns to be displayed.
- **public String getColumnName(int col)** retrieves the heading of the **col** column (starting with column 0).
- **public Object getValueAt(int row, int column)** : retrieves the object (generally String) to be displayed on row **row**, column **column**.

The following example inherits from a generated TableModel. It reduces the number of columns to be displayed from 7 to 3. The first two columns represent two standard Data Elements (Client's Id and name). The third Data Element represents the client's address, i.e. the concatenation of the street, zip code and town.

```
package test.swing;

import com.ibm.vap.generated.reuse.CustomerData;
public class NewCustomerTableModel extends
com.ibm.vap.generated.reuse.CustomerTableModel {

    public int getColumnCount () {
```

```

        return 3;
    }

    public String getColumnName (int i){
        if (i == 0) return "Id";
        if (i == 1) return "Name";
        if (i == 2) return "Address";
        return "";
    }

    public Object getValueAt(int row, int column){
        try {
            CustomerData data = (CustomerData) getRows().elementAt(row);
            if (column == 0) return data.getCusId();
            if (column == 1) return data.getCusNam();
            if (column == 2) {
                String result = "" +
                    data.getStreet() + "." +
                    data.getZipcod() + "-" +
                    data.getTown();
                return result;
            }
        } catch (Throwable t) {}
        return null;
    }
}

```

Management of Data Element Presence

Java

The two following methods enable you to manage the presence of the Logical View's Data Elements at the Proxy level.

The **is<delco>Present** method enables you to test the presence or absence of the **delco** Data Element. It is generated for all **DataDescription** and **UserDataDescription** classes.

The **setNull<delco>Present(boolean aBoolean)** method enables you to specify the presence or absence of the **delco** Data Element before a local update method. It is generated for all **DataDescription** and **UserDataDescription** classes.

By default, all the Data Elements are considered to be absent, except if a default value has been indicated in the VisualAge Pacbase description.

COM

The two following methods enable you to manage the presence of the Logical View's Data Elements at the Proxy level.

The `is<delco>Present` action enables you to test the presence or absence of the `delco` Data element. It is generated for all `DataDescription` and `UserDataDescription` classes.

The `set<delco>Present(aBoolean)` action enables you to specify the presence or absence of the `delco` Data element. It is generated for all `DataDescription` and `UserDataDescription` classes.

By default, all the Data Elements are considered to be absent, except if a default value and/or a list of values have been indicated in the VisualAge Pacbase description.

Management of Data Element Check

Java

The three following actions enable you to manage the check of the Logical View's Data Elements at the Proxy level.

The `get<delco>Index` method indicates the index of the `delco` Data element in the `DataDescription` class. This index is used in the activation of the server check on the `delco` Data element.

The `setCheck(int index, boolean aBoolean)` method enables you to activate or inhibit the server checks on a Data Element (pointed by the index) before any local update method. It is generated for all the `DataDescription` classes of the root or dependent nodes whose Elementary Components include the `Presence vectors generation` parameter in Developer workbench or the `NULLMNGT=YES` and `CHECKSER=YES` options in the VA Pac WorkStation.

By default, all the Data Elements are to be checked (if the `serverCheckOption` property is set to `true`).

COM

The following two actions enable you to manage the check of the Logical View's Data Elements at the Proxy level.

The `setCheck<fieldIndex, aBoolean>` action enables you to activate or inhibit the server checks on a Data Element before any local update action.

The `getCheck<fieldIndex>` action enables you to test whether the server checks are activated on a Data Element.

Both these actions are generated for all the `DataDescription` classes of the root or dependent nodes whose Elementary Components include the `Presence vectors generation` parameter in Developer workbench or the `NULLMNGT=YES` and `CHECKSER=YES` options in the VA Pac WorkStation.

By default, all the Data Elements are to be checked (if the `serverCheckOption` attribute is set to `true`).

Sub-Schema Management

The server selection or read methods take into account the sub-schema present in the `subSchema` property and return the values of the Data Elements belonging to the sub-schema. If a selection method is followed by a paging method, the sub-schema taken into account is that associated with the selection method.

The local creation methods do not refer to any sub-schema.

The local modification/deletion methods refer to the sub-schema associated with the instance, that is:

- if the modification/deletion is performed on an instance which was created locally, the sub-schema is empty.
- if the modification/deletion is performed on a read instance, the sub-schema is that associated with the selection of this instance.

Moreover the following methods are specific to the sub-schema management.

The `resetSubSchema` method enables you to reset the `subSchema` property, that is to select no sub-schema.

The `completeInstance` method enables you to retrieve the values of the Data Elements which do not belong to the sub-schema, by calling the Elementary Component associated with the Logical View.

The `belongsToSubSchema` method enables you to know whether the Data Element passed as a parameter belongs to the sub-schema associated with the `detail` attribute.

Use of Events

Java

The events sent by an Elementary Proxy are used to trigger application methods belonging to the graphic application. This processing is performed by connecting a Proxy event to one or more methods in the graphic application. The conditional execution of methods is facilitated by the fact that an event is always accompanied by its opposite event; both events cannot be sent at the same time.



The availability of an event depends on the type of Proxy. All the Public Interface events are documented in the *eBusiness & Pacbench C/S Applications: Proxy Programming Interface* manual.

Event-driven Management of Large Reading

Event-driven management of large reading provides the developer with information on the state of the collection of instances contained in a node. Each available paging action offers its own event-driven paging system.

The paging action in non-extend mode can send the following four events:

- **noPageBefore**: This event is sent by a root or reference node at the end of the execution of a collection selection or paging action when it does not return any error and when the read page is the first in the current collection.

- **pageBefore:** This event is sent by a root or reference node at the end of the execution of a collection selection or paging action when it does not return any error and when the read page is not the first in the current collection.
- **noPageAfter:** This event is sent by a root or reference node at the end of the execution of a collection selection or paging action when it does not return any error and when the read page is the last in the current collection.
- **pageAfter:** This event is sent by a root or reference node at the end of the execution of a collection selection or paging action when it does not return any error and when the read page is not the last in the current collection.

The paging method in extend mode can send the following two events:

- **pageAfter:** This event is sent by any type of node at the end of the execution of a collection selection or forwards paging action when it does not return any error and when the number of instances contained in the node is not the total number of instances contained in the database.
- **noPageAfter:** This event is sent by any type of node at the end of the execution of a collection selection or forwards paging action when it does not return any error and when the number of instances contained in the node is the total number of instances contained in the database when the request is made.

Event-driven Management of Instance Reading

A Logical View can be mapped on one or more physical storage entities. In this context, the event-driven management of a Logical View instance reading can send the following event:

- **notFound** when the instance searched for is not found in the database. This event can be sent when the Logical View is mapped on one or more tables.

COM

The events sent by an Elementary Proxy are used to trigger application actions belonging to the graphic application. This processing is performed by connecting a Proxy event to one or more actions in the graphic application. The conditional execution of actions is facilitated by the fact that an event is always accompanied by its opposite event ; both events cannot be sent at the same time. After being sent, the event is stored in a stack. Therefore, the graphic application must access the stack regularly using the **getServerEventsCount** and **popServerEvent** methods.

☞ The availability of an event depends on the type of Proxy. All the Public Interface events are documented in the *eBusiness & Pacbench C/S Applications: Proxy Programming Interface* manual.

Event-driven Management of Large Reading

Event-driven management of large reading provides the developer with information on the state of the collection of instances contained in a node. Each available paging action offers its own event-driven paging system.

The paging action in non-extend mode can send the following four events:

- **NO_PAGE_BEFORE:** This event is sent by a root or reference node at the end of the execution of a collection selection or paging action when it does not return any error and when the read page is the first in the current collection.
- **PAGE_BEFORE:** This event is sent by a root or reference node at the end of the execution of a collection selection or paging action when it does not return any error and when the read page is not the first in the current collection.
- **NO_PAGE_AFTER:** This event is sent by a root or reference node at the end of the execution of a collection selection or paging action when it does not return any error and when the read page is the last in the current collection.
- **PAGE_AFTER:** This event is sent by a root or reference node at the end of the execution of a collection selection or paging action when it does not return any error and when the read page is not the last in the current collection.

The paging action in extend mode can send the following two events:

- **PAGE_AFTER:** This event is sent by any type of node at the end of the execution of a collection selection or forwards paging action when it does not return any error and when the number of instances contained in the node is not the total number of instances contained in the database.
- **NO_PAGE_AFTER:** This event is sent by any type of node at the end of the execution of a collection selection or forwards paging action when it does not return any error and when the number of instances contained in the node is the total number of instances contained in the database when the request is made.

Event-driven Management of Instance Reading

A Logical View can be mapped on one or more physical storage entities. In this context, the event-driven management of a Logical View instance reading can send the following event:

- **NOT_FOUND** when the instance searched for is not found in the database. This event can be sent when the Logical View is mapped on one or more tables.

Error Management

There is no error message displayed during the extraction and generation phases.

To be extracted, an Elementary Component must have been compiled correctly, because:

- the GVC command in the GPRT procedure extracts the required data without displaying any error messages, even if the Elementary Component does not contain any Logical View,

- The generator does not control the input file.

However, some error messages can be displayed during the development, test and execution phases of the application. These messages result from local, server or communication errors.

Introduction

There are four types of errors:

- Local errors, sent by the Client Component, which correspond to manipulation or input errors in the Client Component.
- Server errors, sent by the Elementary Component, which are data access errors and user errors set in the server.
- System errors, sent by the Elementary Component, which corresponds to a discrepancy between the Proxy and the Elementary Components,
- Communication errors.

You will find below the list of the local and communication errors which might occur, as well as the structure of the error key for the server and system errors.



The error management specific to the Java environment is documented in *Chapter 4: Developing a Java Client*, subchapter *Error Management*, and the error management specific to the COM environment is documented in *Chapter 5: Developing a COM Client*, subchapter *Error Management*.

Local Errors

Local errors are listed below:

- **ASYNCHRONOUS_VIOLATION**
This error occurs when a lock, unlock or existence check of dependent instances in asynchronous mode is requested.
- **CARDINALITY_VIOLATION**
This error occurs if the cardinalities are not respected when an update method is activated.
- **CREATION_CONTEXT_INVALID**
This error occurs when trying to save a Proxy context while a request exists on that Proxy.
- **CURRENT_INSTANCE_MISSING**
This error occurs when a method is applied to the **detail** property whereas the latter does not contain any instance.
- **CURRENT_USER_INSTANCE_MISSING**
This error occurs when a user method is applied to the **userDetail** property whereas this property does not contain any instance.
- **FOLDER_USER_CONTEXT_LENGTH_ERROR**
This error occurs when the length of the value of a Data Element which belongs to the user buffer exceeds the authorized length for this Data Element.

- **INSTANCE_ALREADY_LOCKED**
This error occurs when a lock action is requested on an instance, which has already been locked on the server.
- **INSTANCE_NOT_LOCKED**
This error occurs when an unlock action is requested on an unlocked instance on the server.
- **INVALID_CHANGE**
This error occurs when the instance to be modified does not exist in the local cache.
- **INVALID_CREATION**
This error occurs when an instance is created though it already exists in the local cache.
- **INVALID_DELETION**
This error occurs when the instance to be deleted does not exist in the local cache.
- **INVALID_INITIALIZATION**
This error occurs when there is an attempt to initialize an instance which is already known by the local cache, whatever the instance status may be (READ, CREATED, MODIFIED ; DELETED).
- **INVALID_INSTANCE**
This error occurs when a primary key of the current instance is not valid.
- **LENGTH_ERROR**
This error occurs when the length of the value of a Data Element which belongs to the current instance exceeds the authorized length for this Data Element.
- **LOCK_SERVICE_ALREADY_REQUESTED**
This error occurs when trying to lock a record which is already locked in a request.
- **NO_SERVER_RESPONSE_REQUESTED**
This error occurs when the user action has been sent to the server because no response is expected.
- **PARENT_INSTANCE_MISSING**
This error occurs when a dependent node instance is selected though the parent instance does not exist.
- **READ_SERVICE_ALREADY_REQUESTED**
This error occurs when trying to do a read action which is already in the current request.
- **REFERENCE_USER_CONTEXT_LENGTH_ERROR**
This error occurs when the length of the value of a Data Element belonging to the user buffer associated with a reference node exceeds the authorized length for this Data Element.
- **REFERING_INSTANCE_MISSING**
This error occurs when the **transferReference** method does not find any instance in the **detail** of the referring node.

- **REQUEST_ALREADY_EXISTS**
This error occurs when trying to create an external request while one already exists.
- **REQUEST_BAD_APPLICATION**
This error occurs when a node is trying to link to a request which does not belong to the same C/S Application.
- **REQUEST_BAD_USER_BUFFER**
This error occurs when trying to link to a request whose buffer is different.
- **REQUEST_NOT_ACTIVE**
This error occurs when trying to act on an inactive request.
- **REQUEST_TOO_LARGE**
This error occurs when trying to add a service to a request which is already full (9999 services).
- **SERVER_UPDATE_REQUIRED**
This error occurs when a method is applied to an instance whereas the parent instance created locally does not exist in the database yet. A server update is previously required for the parent instance.
- **SUBSCHEMA_ERROR**
This error occurs when a Data Element belonging to the current instance is updated whereas it was not filled in for this instance on a server access parameterized with a sub-schema.
- **UNKNOWN_CONTEXT**
This error occurs when the context is not known.
- **UNKNOWN_INSTANCE**
This error occurs when the selected instance is not known by the local cache.
- **UPDATE_CURRENTLY_POSTED**
This error occurs when trying to locally update a record which is already posted in a request.
- **UNLOCK_SERVICE_ALREADY_REQUESTED**
This error occurs when trying to unlock a record whose unlocking has been already requested.
- **VALUE_ERROR**
This error occurs when the contents of a Data Element belonging to the current instance are not valid.
- **VALUE_REQUIRED**
This error occurs when the contents of a Data Element belonging to the current instance are considered to be absent whereas they are required.

Server Errors

You must know the error key if you wish to display customized labels in the Client component.

For details on how the access keys to the local labels of server errors are obtained, refer to the *eBusiness & Pacbench C/S Applications: Proxy Programming Interface* manual.

Col 1-3	Col 4-9	Col 10-13	Col 14-19	Col 20	Col 21	Col 22-25	
lib	ser	seg			E	DUPL	Invalid creation
lib	ser	seg			E	NFND	Invalid deletion/modification
lib	ser				E	Error code	User error
lib	ser	vie	dte	2	E		Required Data Element
lib	ser	vie	dte	5	E		Value error
lib	ser		LOCKED		E		Already locked instance
lib	ser		NTLOCK		E		Instance not locked

Legend:

lib = library code vie = Logical View code ser = server code
dte = Data Element code seg = physical access segment code
E = Exception (Server error)

System Errors

You must know the error key if you wish to display customized labels in the Client component.

For details on how the access keys to the local labels of system errors are obtained, refer to the *eBusiness & Pacbench C/S Applications: Proxy Programming Interface* manual.

System Errors Received from the Elementary Component

Col 1-3	Col 4-9	Col 10-13	Col 14-19	Col 20	Col 21	Col 22-25	
			MISPCV		S		Components out of phase
lib	ser		LKABSC		S		No timestamp set on Lock
lib	ser	vie	ACCESS		S		Data access error
lib	ser	LTH			S		View length error
lib	ser	SERV			S		Unknown Service
lib	ser	STRU			S		View structure error
lib	ser	VERS			S		Version error
lib	ser	VIEW			S		Unknown View

Legend:

lib = library code vie = Logical View code ser = server code
dte = Data Element code seg = physical access segment code
[S] = system error

The type of error **Unknown service** is displayed when the service requested by the Proxy is not recognized by the Elementary Component.

The type of error **View length error** is displayed when the format of a Logical View associated with the Proxy changes and when this Proxy cannot be regenerated.

To solve this problem, you must regenerate the Proxy.

The type of error **Components out of phase** occurs when the Client and Elementary Components are out of phase.

To solve this problem, you must regenerate the Proxy.

System Errors Received from the Communications Monitor

Col 1-3	Col 4-9	Col 10-13	Col 21	
lib	mon	LSRV	S	Erroneous length of the message received
lib	mon	NPSA	S	Erroneous structure of the parameters of the next service
lib	mon	PCOD	S	Erroneous structure of the "service code" parameter
lib	mon	PCVS	S	Erroneous structure of a service request from the client
lib	mon	PCVF	S	Erroneous structure of the message from the client
lib	mon	PNOD	S	Unknown Folder code
lib	mon	PNUM	S	Erroneous structure of the "service number" parameter
lib	mon	TAND	S	Tandem/Pathway error
lib	mon	WF00	S	Erroneous access to the work file or to the database (open/close)

Legend lib = library code mon = Communications Monitor code S = System error

System Errors Received from the Services Manager

Most of these errors are internal errors that you can solve only by contacting the VisualAge Pacbase support.

Col 1-3	Col 4-9	Col 10-13	Col 21	
lib	serv	BUF1	S	Erroneous structure of user buffer
lib	serv	CHK1	S	Missing "Check Option" parameter
lib	serv	CHK2	S	Erroneous length of the "Check Option" parameter
lib	serv	CP01	S	Erroneous structure of a "Selection Criteria" field from the Elementary Component
lib	serv	CP02	S	Erroneous structure of a field of a Logical View instance from the Elementary Component
lib	serv	DANA	S	Erroneous structure of the erroneous field code in the user error message
lib	serv	DOS1	S	Missing "Folder name" parameter
lib	serv	DOS2	S	Erroneous length of the "Folder name" parameter
lib	serv	ERK1	S	Erroneous structure of the user error message key
lib	serv	ERKY	S	Erroneous structure of the "Selt Message" key from the Elementary Component
lib	serv	ERL1	S	Erroneous structure of the user error message label
lib	serv	ERLA	S	Erroneous structure of the "Selt Message" label from the Elementary Component
lib	serv	EXT1	S	Erroneous structure of the "extraction method" parameter
lib	serv	EXT2	S	Unknown extraction method in the Folder

Col 1-3	Col 4-9	Col 10-13	Col 21	
lib	serv	FRRE	S	Erroneous read access to the work file before update
lib	serv	FRRD	S	Erroneous read access to the work file
lib	serv	FRW2	S	Erroneous writing of the last record of the work file
lib	serv	FRWR	S	Erroneous write access to the work file
lib	serv	FRRW	S	Erroneous update access to the work file
lib	serv	LCK1	S	Missing "Lock Timestamp" parameter
lib	serv	LCK2	S	Erroneous length of the "Lock Timestamp" parameter
lib	serv	LNG1	S	Erroneous conversion of service length on a multi-messages request
lib	serv	LNG2	S	Erroneous conversion of service length on a single-message request
lib	serv	LTH	S	Erroneous "length" parameter in the elementary Elementary Component
lib	serv	NOCP	S	Erroneous structure of the "number of occurrences" parameter
lib	serv	NOC1	S	Erroneous length of the " number of occurrences" parameter
lib	serv	NOC2	S	Erroneous conversion of the "number of occurrences" parameter "
lib	serv	NOD1	S	Missing "node name" parameter
lib	serv	NOD2	S	Erroneous length of the "node name" parameter
lib	serv	NOD3	S	Unknown node name in the Folder
lib	serv	NOS1	S	Erroneous structure of the "service number" parameter
lib	serv	NOS2	S	Erroneous conversion of the "service number" parameter
lib	serv	NUVE	S	Erroneous "version number" parameter in the elementary Elementary Component
lib	serv	OCNB	S	Erroneous structure, in the user error message, of the field code which bears the error
lib	serv	PC01	S	Erroneous conversion of the "Selection Criteria" parameter
lib	serv	PC02	S	Erroneous conversion of a field of the user buffer
lib	serv	PC03	S	Erroneous conversion of a field of a Logical View instance from the client
lib	serv	PC05	S	Erroneous conversion of a "Selection Criteria" field from the client
lib	serv	PC06	S	Erroneous conversion of a field of a Logical View instance to be updated
lib	serv	PCV1	S	Erroneous structure of the "Selection Criteria" parameter
lib	serv	PCV3	S	Erroneous structure of a field of a Logical View instance from the client
lib	serv	PCV4	S	Erroneous structure of a "Selection Criteria" field from the client
lib	serv	PCV5	S	Erroneous structure of the action code of a Logical View instance to be updated
lib	serv	PCV6	S	Erroneous structure of a field of a Logical View instance to be updated from the client
lib	serv	PCV7	S	Erroneous structure of the presence indicator of a field of a Logical View instance to be updated
lib	serv	PCVF	S	Erroneous structure of the message from the client
lib	serv	PCVS	S	Erroneous structure of a service request from the client
lib	serv	PILO	S	Erroneous access to the main record of the work file
lib	serv	RFH1	S	Missing "Refresh Option" parameter
lib	serv	RFH2	S	Erroneous length of the "Refresh Option" parameter
lib	serv	SCH1	S	Erroneous structure of the "sub-schema code" parameter
lib	serv	SCH2	S	Unknown sub-schema code in the Folder
lib	serv	SERV	S	Erroneous "operation code" parameter in the elementary Elementary Component
lib	serv	SRV1	S	Service not found in the work file

Col 1-3	Col 4-9	Col 10-13	Col 21	
lib	serv	SRV2	S	Unknown service code in the Folder
lib	serv	SRV3	S	Erroneous structure of the "service code" parameter
lib	serv	STRU	S	Erroneous "structure" parameter in the elementary Elementary Component
lib	ges	TAND	S	Tandem/Pathway error
lib	serv	TYNO	S	Unauthorized service on the node
lib	serv	USR1	S	Missing "User service" parameter
lib	serv	USR2	S	Erroneous length of the "user service" parameter
lib	serv	USR3	S	Unknown user service in the Folder
lib	serv	VER2	S	Erroneous length of the "version number" parameter
lib	serv	VIEW	S	Erroneous "Logical View code" parameter in the elementary Elementary Component
lib	serv	WF00	S	Erroneous access to the work file

Legend lib = library code serv = Services Manager code S = system error

Communication Errors

If a communication error message is displayed, inform the person in charge of the communication because the line may be blocked or defective, or a Server may be busy, etc.

Three communication error messages are likely to be displayed:

- **Open server error**
- **Call server error**
- **Close server error**

Chapter 4: Developing a Java Client

Once you have generated and imported the Proxy objects, you just need to integrate them into the graphic application.

This chapter gives you a detailed description of a client development (applet and standalone) in VisualAge for Java, including the following steps: insertion of Proxy objects with programming links involving methods, properties and events. This chapter also presents error management, communication management, as well as the test and deployment of the application.

Example of an Applet

Introduction

This example describes a VisualAge Java applet, i.e. a program meant to be used in a browser.

This applet is developed first with VisualAge Java version 1 and then with VisualAge Java version 2 to illustrate the specificities of both versions.



To facilitate the development and reusability of clients implemented with VisualAge, we advise you to use a project for each functional application. The project must contain one or more graphic class packages and a generated class package.

The Proxy components inserted in the applet have been generated with the **Generate Beans** option and:

- The **Use IBM Enterprise Access Builder classes** option for version 1.
- The **Use Swing** option for version 2.

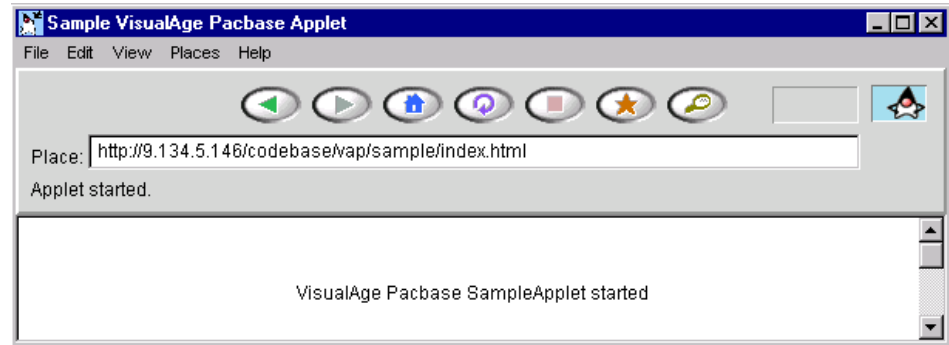
In the example, three Elementary Proxies of the FVP are used:

- The Root Proxy corresponding to the **Customers** node which manages the customers in the information system described by the Folder.
- The Dependent Proxy corresponding to the **Orders** node which manages the orders in the information system described by the Folder.
- The Dependent Proxy corresponding to the **Order lines** node which manages the order lines in the information system described by the Folder.

Presentation of the End User Interface

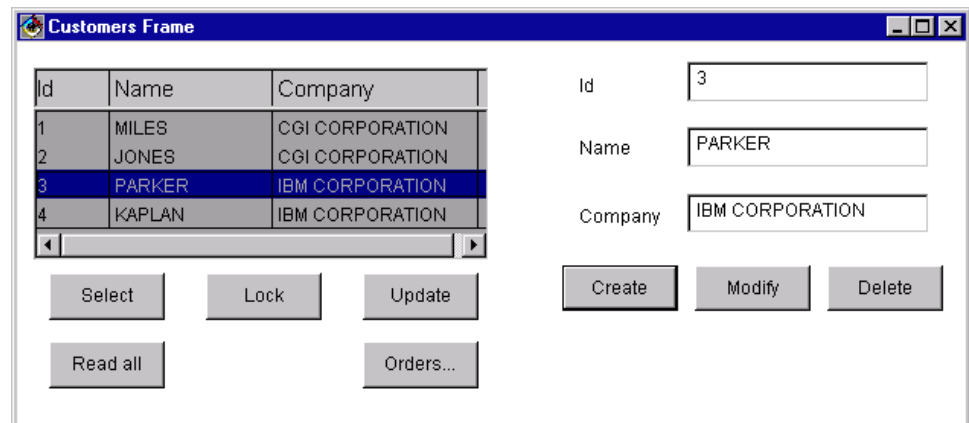
The graphic user interface consists of the applet itself and two windows.

- **The applet**



The applet does not have any function in the user application. It is the starting point and enables the application to be accessed via the web.

- **The Customers Window**

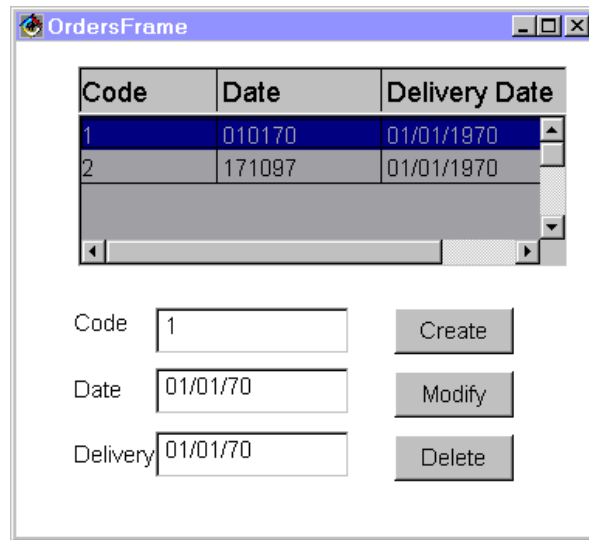


This window opens automatically at the applet start. It contains the following functionalities:

- The **Select** button is used to display a list of customers.
- The **Read all button** is used to submit a reading request of the selected customer 's orders from this window.
- As a lock option has been specified in the Folder, the user must click on the **Lock** button to lock the instance selected in the list before starting any updating process.
- The **Update** button is used to update the database.
- The **Orders...** button is used to display the Orders Frame window.
- The **Modify** button is used to modify customers in the detail after locking the instance.

When a customer is selected in the list, the user must click on **Lock** to appropriate this customer for a short time. Then, he can enter modifications and click on **Modify**.

- **The Orders Window**



The **Orders** window opens when the user clicks on the **Orders...** button in the **Customers** window.

For each customer selected in the detail of the **Customers** window, this window allows you to:

- view this customer's list of orders.
- select an order in the list and display it in the detail.
- create, modify, delete orders.

Developing the End User Interface with VisualAge Java V1

Developing the graphic user interface consists in programming the applet and in building the **Customers** and **Orders** windows.

You will not find here detailed information on how to use VisualAge tools and functions. If you are not familiar with them, refer to the appropriate documentation.

We try as much as possible to describe the development steps sequentially, but we sometimes need to organize the description of the windows programming into different consistent groups of functionalities.

At the end of each programming step, the Composition Editor is shown as it should appear on your screen, sometimes the connections previously developed are hidden so that you can see the newly created ones.

Implementing the Example and Creating the Applet

- In a project, create a **vap.sample** package meant to contain the application components.
- In this package, create a **SampleApplet** applet. In the **SmartGuide - Create Applet** window, check the **Design the applet visually** option, so that the classes browser opens directly on the **Visual Composition** tab.

- **Displaying the Text**

In the Visual Composition Editor, execute the following operations:

- Resize the applet.
- Place a **Label** bean (**Data Entry** category) in the applet. Enter **VisualAge Pacbase SampleApplet started** in the **Properties** window of the bean, in the **text** field.

- **Integrating the Root Proxy**

To place the Root Proxy on the Free Form Surface, execute the following operations:

- In the **Options** menu, select the **Add Bean** choice.
- In the **Add Bean** window, which is opening, you must enter **com.ibm.vap.generated.proxies.CustomerProxyLv** in the appropriate field (you can use the **Browse...** Button).
The class name (**CustomerProxyLv** in our example) must be preceded by the package name selected for the generation. In our example, the package name is **com.ibm.vap.generated.proxies**.

- **Defining the Communication with the Gateway**

For more information on this subject, refer to subchapter *Communication Management*.

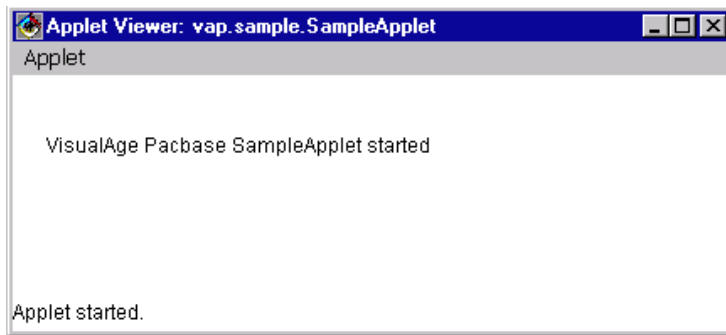
In this example, we assume that the Folder View Proxy communicates with its host, that is to say, with the HTTP server where the applet is stored. To enable the communication, follow these steps:

- Open the pop-up menu from the Free Form Surface.
- Select **Tear-Off Property**, then **codeBase (URL)**.
- Click on the Free Form Surface. A variable bean named **codeBase1** is displayed.
- Open the pop-up menu of the **codeBase1** bean, select **Connect**, and then **All Features...**.
- In the **Start connection from** window, select the **host** property. A dotted link is displayed.
- Click the Root Proxy. The pop-up menu is displayed.
- Select **All Features...**. The **Connect property named:** window opens.
- Display all the properties available for the Root Proxy by checking the **Show expert features** box.
- Select the **Host** property, then click on **OK**.

The **host** property of **codeBase1** is now connected to the **Host** property of the Root Proxy.

These connections are equivalent to the following code line in the applet:
getCustomerProxyLv1 () .setHost (getCodeBase () .getHost ());

Now, you can test your applet. The **Applet Viewer** window opens:



- **Programming the Opening of the CustomersFrame Window from the Applet**

This phase consists of two parts ; both must be done after the operations described in the *Creating the Window and Integrating the FVP in the Composition Editor* and *Promoting the Root Proxy* paragraphs:

- Calling the Root Proxy in the applet
 - ♦ In the applet's Composition Editor, place a constant bean of **CustomersFrame** type on the Free Form Surface.
 - ♦ Connect the **this** property of the Root Proxy to the **customerProxyLv1This** property of the **CustomersFrame** bean.
- Programming the opening of the window from the applet.

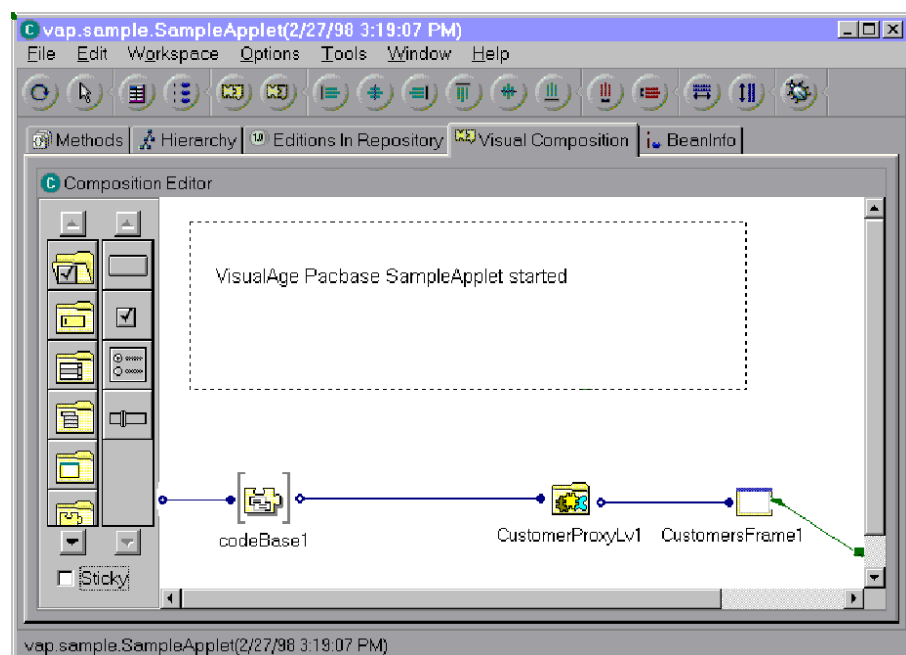
Now we want the **CustomersFrame** window to open immediately after the applet starts.

To do so, connect the **componentShown** event of the applet to the **show** method of the **CustomersFrame** bean.

- **Result in the Composition Editor**

The applet is now completed.

The Composition Editor should look like this at this stage:



Developing the Customers Window

Mapping Rows and Detail

Once the Root Proxy has been inserted and promoted, this phase describes the mapping of its **rows** and **detail** properties.

• **Creating the Window and Integrating the FVP in the Composition Editor**

Creating a window for the management of customers consists in creating the corresponding class in the Workbench.

- In the Workbench, create a **CustomersFrame** class in the **vap.sample** package.
- In the **SmartGuide -Create Class or Interface** window which opens then:
 - ♦ Enter **java.awt.Frame** in the **Superclass** field. You specify this way that the **CustomersFrame** class inherits from the **java.awt.Frame** class.
 - ♦ Check the **Design the class visually** option so that the class browser directly opens on the **Visual Composition** tab.
- To integrate the Folder View Proxy in the Visual Composition Editor, execute the steps detailed in the *Creating the Window and Integrating the FVP in the Composition Editor* paragraph. But this time, you must select the **Variable** option in the **Bean Type** field of the **Add Bean** window.
 - ☞ The Root Proxy placed here is a Proxy of a variable type because it must represent the same Proxy instance as the one called in the applet.

• **Promoting the Root Proxy**

Now the purpose is to ensure the permanent identity of this variable Proxy and the constant Proxy instantiated in the applet. To do so, the variable Proxy must be public outside the **CustomersFrame** class, so that a property connection – property between the constant and variable Proxy objects can be performed in the applet.

Proceed as follows:

- From the Proxy's pop-up menu, choose **Promote Bean feature**.
- In the **Property** column, choose **this** and click **Promote**.

The **CustomersFrame** class now has a readable/writable property named **customerProxyLvl1This** and typed Root Proxy.
- Save the window (**File** menu, **Save Bean** or **CTRL-F2** choice).

• **Mapping the Rows Property**

☞ This mapping uses an EAB container; so it is possible only if you checked the **Use the IBM Enterprise Access Builder classes** for the generation.

This step consists in the following operations:

- In the **CustomersFrame** bean, place a **Multi Column List Box** bean, located in the **Access Enterprise** category. Resize the bean.

- You have now to map, in the container, the columns corresponding to the Data Elements associated with the root node.

Open the **Properties** window of the container. From the **columns** field of this window, add a column for each Data Element of the Logical View, in the order they are called in the Repository. You can map as many columns as properties held by the Proxy **DataDescription** or the first ones only.

☞ The columns to be mapped correspond to the values returned by the **getAttributeStrings()** method of the **DataDescription**.

- Once the columns have been created, connect the **IRows** property of the Proxy to the **elements** property of the container.

☞ **IRows** is identical to **Rows** but it returns an instance of the **IVector** class, which, contrary to the **DataDescriptionVector** class returned by **Rows**, is compatible with the **elements** property of the **Multi Column List Box** bean.

This operation is equivalent to the mapping of the **Rows** property of the Proxy.

• Mapping the Detail Property

This step requires the following operations:

- From the Root Proxy, tear-off the **detail** property.
- Insert a **label** bean for each property to be mapped.
- You must now map an input field for each property to be displayed.

To do this, use the **Pacbase Text Field**, **Pacbase Integer Field**, **Pacbase Decimal Field**, **Pacbase Date Field** and **Pacbase Time Field** beans provided by the eBusiness or Pacbench C/S module when the runtime is imported.

These beans are located in the **VisualAge Pacbase** category, in the palette.

You can also insert these beans by selecting the **Add bean** choice in the **Options** menu; enter the full name of the package followed by the bean name. For example: **com.ibm.vap.beans.PacbaseDateField**.

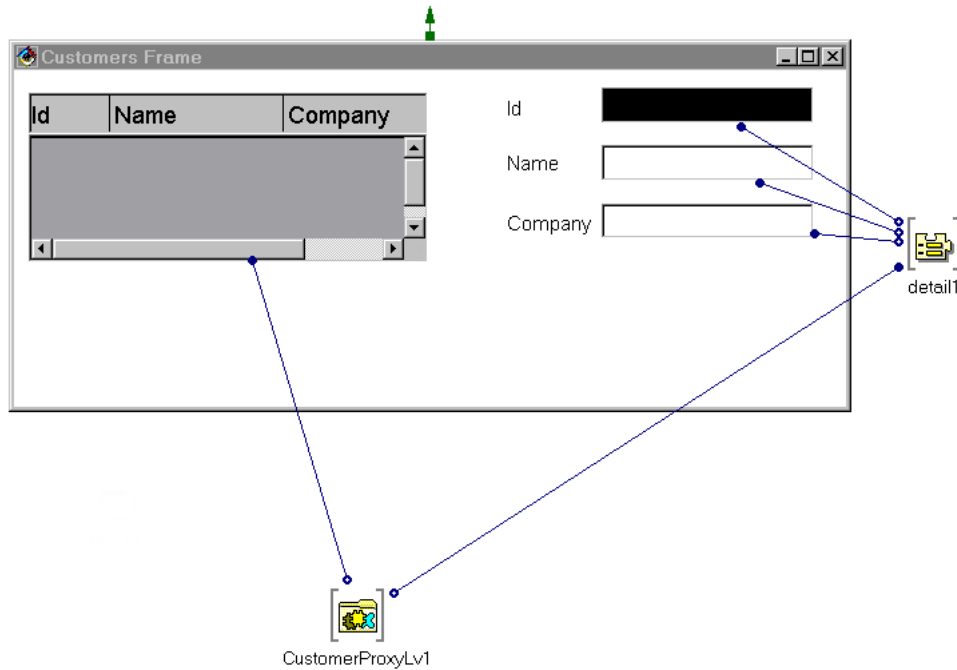
☞ For more details on these beans, refer to the online documentation of the generic classes.

- Insert the type of bean required for each property to be mapped.
- Then, for each mapped field, connect the property of the **detail** bean to its corresponding **String**, **Int**, **Decimal**, **Date** or **Time** property.

In our example, for the input field corresponding to the customer number, you must connect the **Customer Number** property of the **detail** bean to the **Int** property of the field.

- **Result in the Composition Editor**

After all these steps, the Composition Editor should look like this:



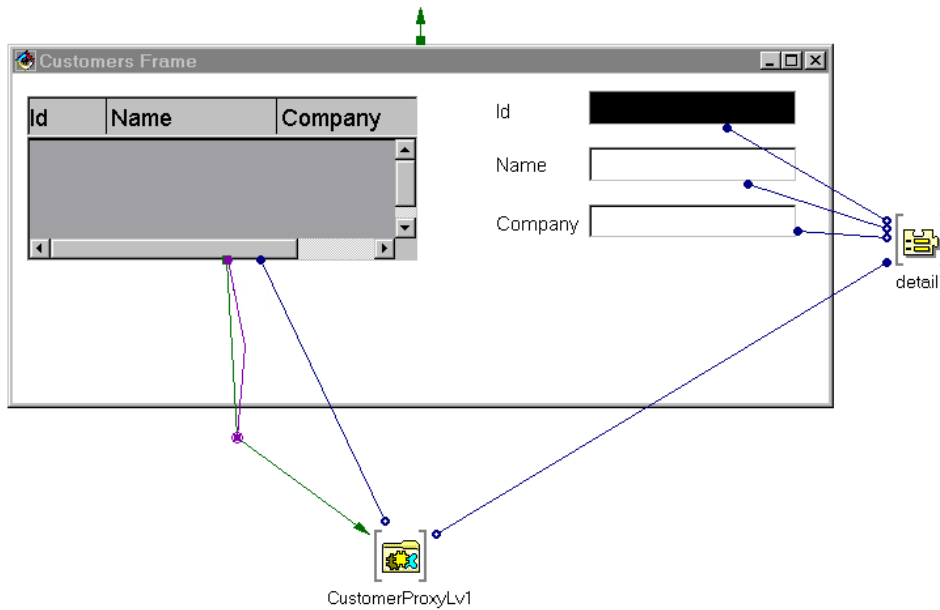
Selection of an Instance in Rows and Transfer in Detail

Now the purpose is to program the transfer of the instance selected by the user, from the container into the **detail** property of the Proxy.

To do this, proceed as follows:

- Connect the **itemStateChanged** event of the MultiColumnListBox to the Root Proxy's **Get Detail From DataDescription** method.
- The link appears as a dotted line because the **Get Detail From DataDescription** method requires a parameter.
- To define this parameter, connect the **selectedObject** property of the MultiColumnListBox to the data-type parameter of the link (**customerData** in our example). This parameter is available when the cursor is placed in the middle of the connection line.

The Composition Editor should look like this:



Activation of the Proxy's Methods and Navigation towards the Orders Window

- **Activation of the Proxy's Methods**

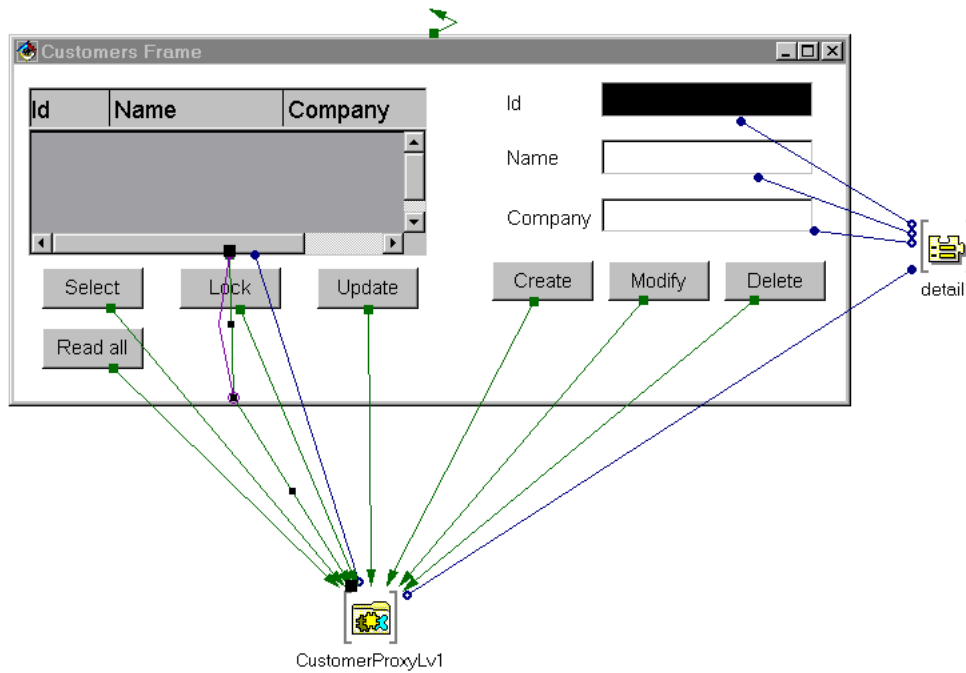
For each method you wish programming, place a push-button in the window. You can modify the label in the **Properties** window of the button.

To activate a method, connect the **actionPerformed** event of a given button to the appropriate method of the Proxy. If this method is not available, check the **Show expert features** option.

For example:

- To program the activation of the **Update** button, you must connect its **actionPerformed** event to the **Update Folder** method of the Proxy.
- To program the activation of the **Read all** button, you must connect its **actionPerformed** event to the **Read All Children From Detail** method of the Proxy.

After these steps, the Composition Editor should look like this:

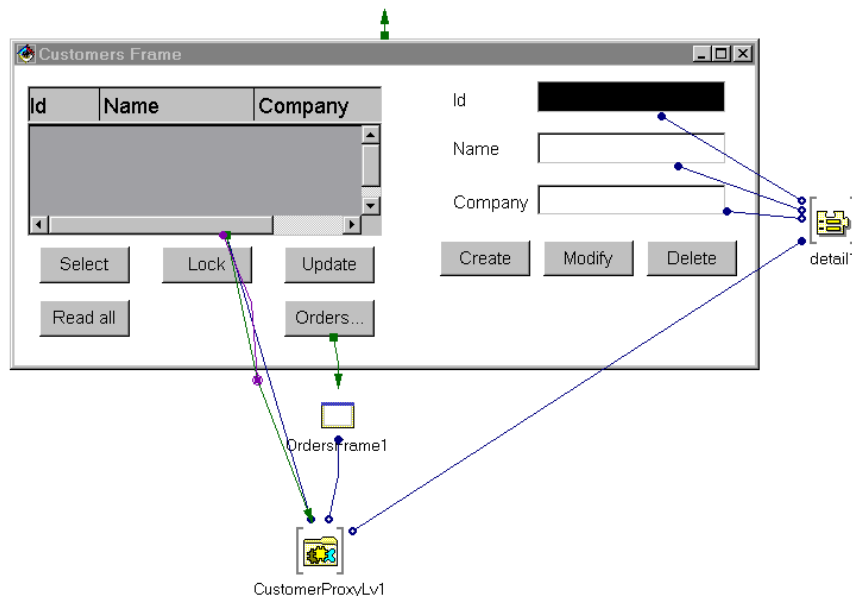


- **Navigation management**

This step must be executed after promoting the Orders Dependent Proxy.

- On the Free Form Surface, place an **OrdersFrame** constant bean.
- Connect the **Order Proxy** of the Root Proxy to the **orderProxyLv1This** property of the **OrdersFrame** bean. This connection ensures the link of the two Proxy objects between the two windows.
- Place an **Orders...** button in the **CustomersFrame** bean.
- Connect the **actionPerformed** event of the button to the **show** method of the **OrdersFrame** bean.

At the end of this step, the Composition Editor should look like this:



For better readability, the links between the other buttons and the Root Proxy are hidden.

Developing the Orders Window

This window gives information on the orders. It opens when the user clicks on the **Orders...** button in the **Customers** window.

The container automatically displays the orders of the selected client in the **Customers** window if the user clicks **Read all** before clicking **Orders...**

The development of this window using the **OrderProxyLv** Dependent Proxy consists of the stages described below.

Creating the Orders Window

In the Workbench, create an **OrdersFrame** class which inherits from **java.awt.Frame** in the **vap.sample** package, as you did when you created the **Customers** window.

Integrating and Promoting the Dependent Proxy

In the Visual Composition tab, place an **OrderProxyLv** variable bean on the Free Form Surface.

For more details on the proxy integration, refer to paragraph *Developing the Customers Window*.

Now we have to promote this variable Proxy so that it can be public from the outside.

To do this, proceed as follow:

- From the Proxy's pop-up menu, choose **Promote Bean feature**.
- In the **Property** column, choose **this** and click on **Promote**.
The **OrdersFrame** class has a public property now, in a read/write mode, named **OrderProxyLv1This** and typed Dependent Proxy.
- Save the window (**File** menu, **Save Bean** or **CTRL-F2** choice).

Mapping the Dependent Proxy's Rows and Detail

The operations to be executed here are identical to those required for the mapping of the **rows** and **detail** properties of the Root Proxy.

The same EAB container is used for the mapping of **rows**.

For more details, refer to paragraph *Developing the Customers Window*.

Selection of an Instance in Rows and Transfer in Detail

The operations to be executed here are identical to those required for the **rows** and **detail** properties of the Root Proxy.

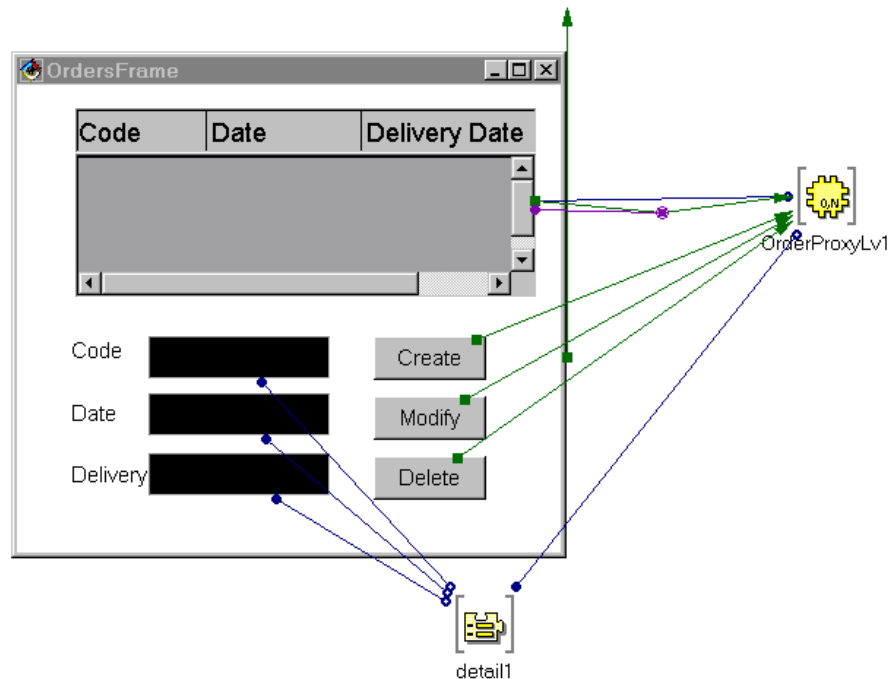
For more details, refer to paragraph *Developing the Customers Window*.

Activation of the Dependent Proxy's Methods

Use the same principles as those described in paragraph *Developing the Customers Window*.

Result in the Composition Editor

After all these steps, the Composition Editor should look like this:



Developing the End User Interface with VisualAge Java V2

This section details the development, with VisualAge Java V2, of the application whose end-user interface is presented in section *Presentation of the End User Interface*.

The Proxy components inserted in the application have been generated with the **Generate Beans** et **Use Swing** options. All the components inserted here are Swing beans.

We develop the application from scratch, without using the application developed with VisualAge Java V1.



If you have already developed an application with VisualAge Java V1, you can save it and resume its development with VisualAge Java V2. However if you want to insert Swing components, you must re-generate the Proxy components with the **Use Swing** option to ensure a correct communication between the Proxy components and the Swing beans.



VisualAge Java V2 does not recognize EAB containers, which are components specific to VisualAge Java V1. If the application developed with the Version 1 includes such components, you must replace them with other components (swing JTable component for example).

Implementing the Example and Creating the Applet

- In a project, create an **example.swing** package meant to contain the application components.

- In this package, create a **SwingApplet** applet. In the **SmartGuide - Create Applet** window, select **JApplet** in the **SuperClass** field, via the **Browse** button, and check the **Compose the applet visually** option, so that the class browser opens directly on the **Visual Composition** tab.

- **Displaying the Text**

In the Visual Composition Editor, execute the following operations:

- Resize the applet.
- Place a **JLabel** bean in the applet. Enter **VisualAge Pacbase SwingSampleApplet started** in the **Properties** window of the bean, in the **text** field.

- **Integrating the Root Proxy**

To place the Root Proxy on the Free Form Surface, execute the following operations:

- Click the **Choose Bean** icon located above the palette.
- Select the type of bean **Class**.
- With the Browse button, select class name **CustomerProxyLv**.

- **Defining the Communication with the Gateway**

For more information on this subject, refer to subchapter *Communication Management*.

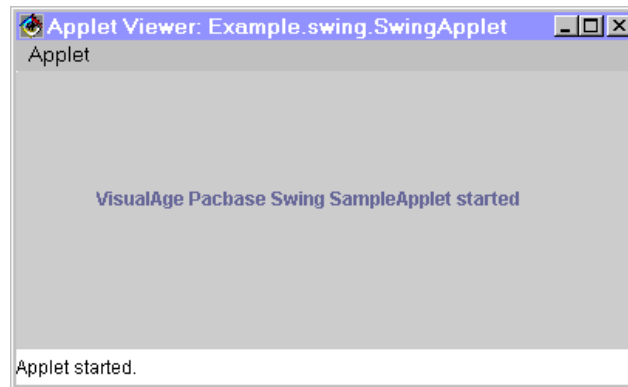
In this example, we assume that the Folder View Proxy communicates with its host, that is to say with the HTTP server where the applet is stored. To enable this communication, follow these steps:

- Open the pop-up menu in the Free Form Surface.
- Select **Tear-Off Property**, then **codeBase (URL)**.
- Click in the Free Form Surface. A variable bean named **codeBase1** is displayed.
- Open the pop-up menu of the **codeBase1** bean, select **Connect**, and then **Connectable Features...**
- In the **Start connection from** window, select the **host** property. A dotted link is displayed.
- Click on the Root Proxy. The pop-up menu is displayed.
- Select **Connectable Features...** The **End Connection to** window opens.
- Display all the properties available for the Root Proxy by checking **Show expert features**.
- Select the **Host** property, then click **OK**.

The **host** property of **codeBase1** is now connected to the **Host** property of the Root Proxy.

These connections are equivalent to the following code line in the applet:
`getCustomerProxyLv1 () .setHost (getCodeBase () .getHost ()) ;`

Now you can test your applet. The **Applet Viewer** window opens:



- **Programming the opening of the CustomersFrame Window from the applet**

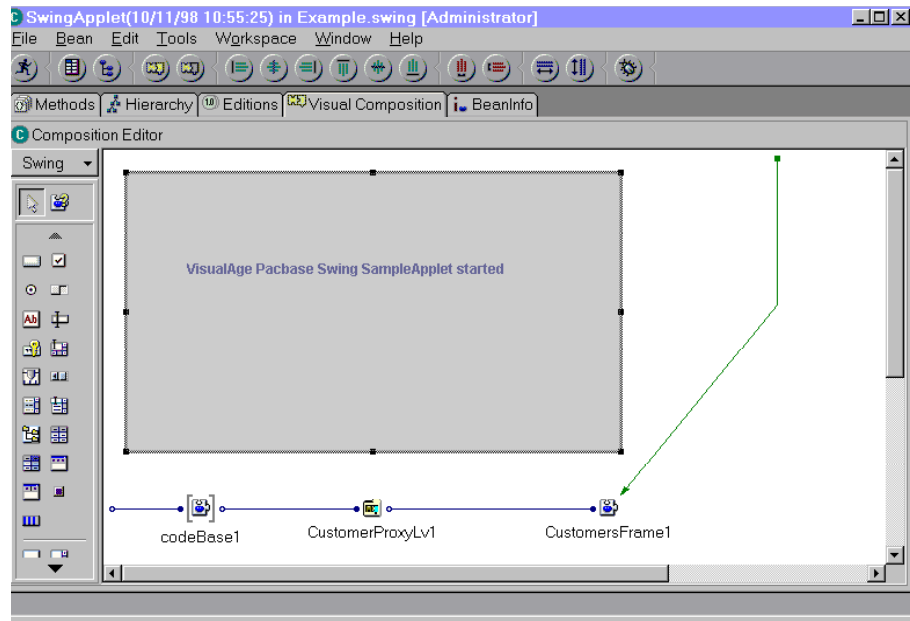
This step consists of two parts. Both must be done once the operations described in paragraphs *Creating the Window and Integrating the FVP in the Composition Editor* and *Promoting the Root Proxy* have been executed :

- Call of the Root Proxy in the applet
 - ♦ In the applet's Composition Editor, place a constant bean with the **CustomersFrame** type on the Free Form Surface.
To do so, select the **Choose Bean** icon located above the palette, select the **Class** type of bean and select, with the Browse button, the **CustomersFrame** class.
 - ♦ Connect the **this** property of the Root Proxy to the **customerProxyLv1This** property of the **CustomersFrame** bean.
- Programming the opening of the window from the applet
Now we want the **CustomersFrame** window to open immediately after the applet starts.
To do so, connect the **componentShown** event of the applet to the **show** method of the **CustomersFrame** bean.

- **Result in the Composition Editor**

The applet is now completed.

The Composition Editor should look like this as this stage:



Developing the Customers Window

Mapping Rows and Detail

Once the Root Proxy has been inserted and promoted, this phase describes the mapping of its **rows** and **detail** properties.

- **Creating the Window and Integrating the FVP in the Composition Editor**

To create a window for the management of customers consists in creating the corresponding class in the Workbench.

- In the Workbench, create a **CustomersFrame** class in the **example.swing** package.
- In the **SmartGuide -Create Class or Interface** window which opens then:
 - ♦ select **JFrame**, with the **Browse** button, in the **Superclass** field: you specify this way that the **CustomersFrame** class inherits from the **com.sun.java.swing.JFrame** class.
 - ♦ Check the option **Compose the class visually** so that the class browser opens directly on the **Visual Composition** tab.
- To insert the Folder View Proxy in the Visual Composition Editor:
- Click on the **Choose Bean** icon located above the palette.
- Select the **Variable** type of bean.
- With the Browse button, select the **CustomerProxyLv** class.

☞ The Root Proxy inserted here is of a variable type because it must represent the same Proxy instance as the one called in the applet.

- **Promoting the Root Proxy**

Now the purpose is to ensure the permanent identity of this variable Proxy and the constant Proxy instantiated in the applet. To do so, the variable Proxy must be public outside the **CustomersFrame** class, so that a property – property connection between the constant Proxy and the variable Proxy can be performed in the applet.

Proceed as follows:

- Open the Proxy's pop-up menu. Select **Promote Bean feature**.
- In the **Property** column, choose **this** then click **>>**.

The **CustomersFrame** class now has a readable/writable public property named **customerProxyLv1This** and typed Root Proxy.

- Save the window (**Bean** menu, **Save Bean** choice).

- **Mapping the Rows Property**

In the **CustomersFrame** bean, place a **JTable** bean and resize it.

To see the columns of the JTable, you must run the JFrame. The columns of the JTable are initialized with the Data Elements of the Logical View and the lines represent the instances included in **Rows**. The content of the JTable is refreshed each time **Rows** is updated (selections, creations...).

Two mappings are possible:

- If you want to display all the columns which correspond to all the Data Elements of the Logical View with the clear names defined in the Proxy, you simply have to connect the **Table model** property of the Proxy to the **model** property of the JTable.
- However if you want to select the columns to be displayed, modify their heading or create a new column to display data locally computed, you must customize the JTable.

To do so, you must create a new **TableModel** class and customize its methods.

This new class must be public and must inherit:

- ♦ either from **CustomerTableModel**, located in the **com.ibm.vap.generated.reuse** package. This way, this method automatically inherits all the existing methods of **CustomerTableModel** and you will have to modify only the methods that do not suit your application.
- ♦ or directly from **PacbaseTableModel**, located in the **com.ibm.vap.beans.swing** package. In this case, you will have to re-write all the methods you want to use since they are not retrieved automatically.

To create the new class, select the **Add Class** choice in the pop-up menu of the package (**com.ibm.vap.generated.reuse** or **com.ibm.vap.beans.swing**). Name it (**NewCustomerTableModel** for example) and, in the **Superclass** field, select the class (**CustomerTableModel** or **PacbaseTableModel**) from which the new class will inherit.

You must then customize the methods of this new class by inserting code directly in the **source** part. In our example, we chose to make the new **NewCustomerTableModel** class inherit from the **CustomerTableModel** class.

The **public int getColumnCount()** method must be customized to limit the number of columns to 3 in the JTable, whereas the Logical View contains 7 Data Elements.

```
public class NewCustomerTableModel extends
com.ibm.vap.generated.reuse.CustomerTableModel
{public int getColumnCount (){ return 3;}
}
```



The other methods via which the columns of the JTable can be customized are documented in *Chapter 3: Development Principles, Use of Methods, Customization of the Columns of a Jtable (Java Only)*.

In the Composition Editor, you simply have to:

- Place an instance of the new **TableModel**,
- Connect this instance, via its **this** property, to the **TableModel** property of the Proxy,
- Connect this instance, via its **this** property, to the **model** property of the JTable.

• Mapping the Detail Property

This step requires the following operations:

- From the Root Proxy, tear-off the **detail** property.
- Insert a **JLabel** bean for each property to be mapped.
- you must now map an input field for each property to be displayed.

To do so, use the beans **Pacbase Swing Text Field** and **Pacbase Swing Integer Field** provided by the eBusiness or Pacbench C/S module when the runtime is imported.

You can also insert these beans by clicking on the **Choose Beans** icon located above the palette. Select the type **Class** or **Variable**. Then select, with the Browse button, the name of the bean: for example: **PacbaseJTextField**.



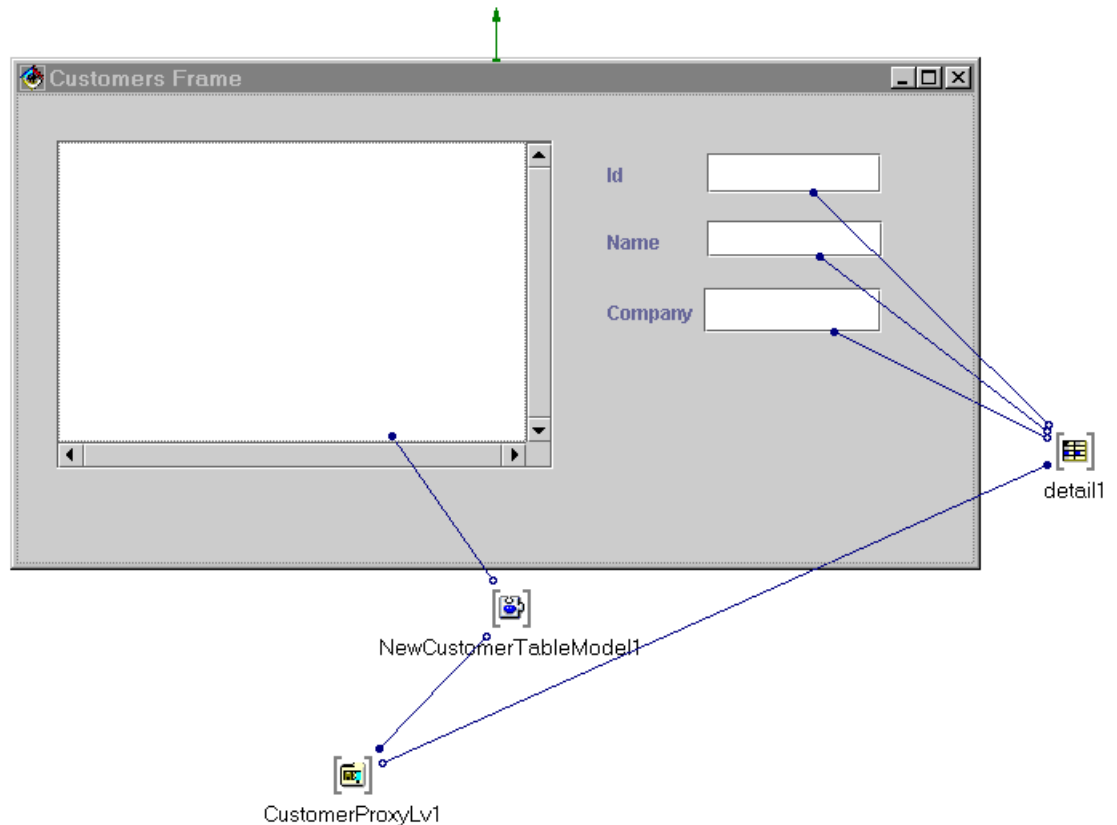
For more details on these beans, refer to the online documentation of the generic classes.

- Insert the type of bean required for each property to be mapped.
- then, for each mapped field, connect the property of the **detail** bean to the corresponding **String** or **Int** type property.

In our example, for the input field which corresponds to the customer number, you must connect the **Customer Number** property of the **detail** bean to the **Int** property of the field.

- **Result in the Composition Editor**

After all these steps, the Composition Editor should look like this:



We chose here to customize the JTable. This is the reason why a **NewCustomerTableModel1** bean is inserted.

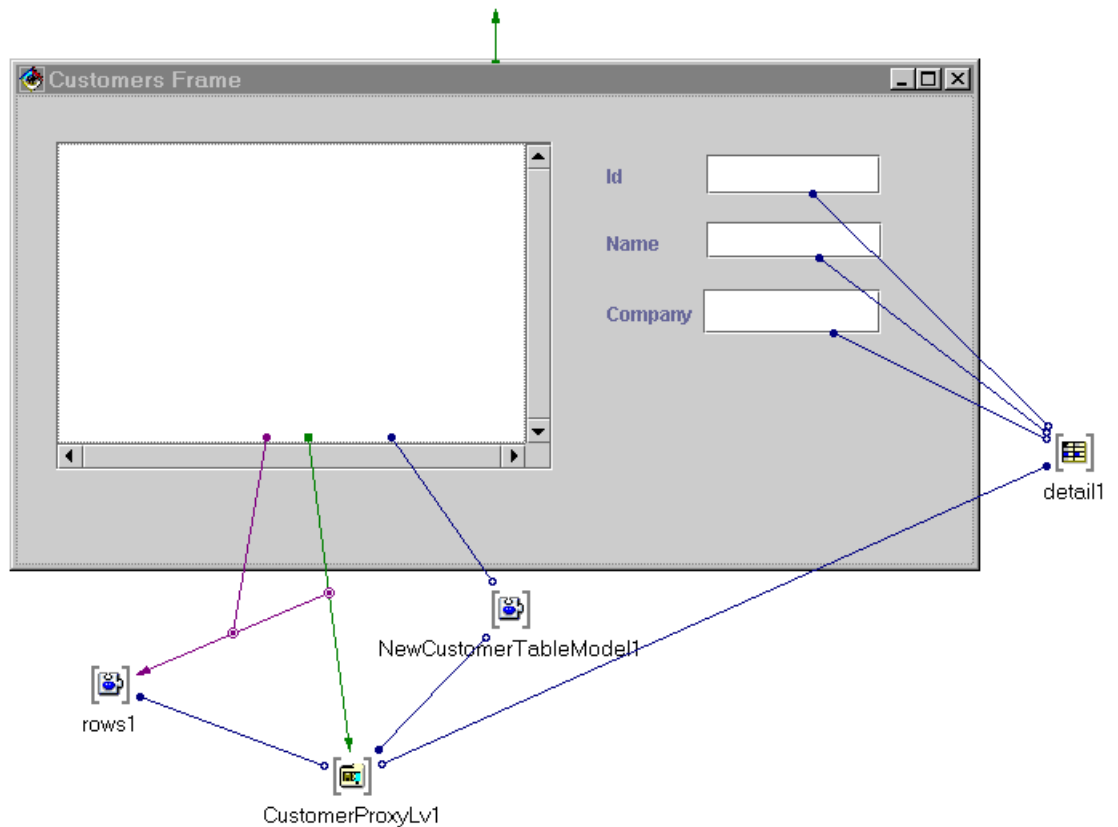
Selection of an Instance in Rows and Transfer in Detail

Now the purpose is to program the transfer of the instance selected by the user from the table into the **detail** property of the Proxy.

To do so, proceed as follows:

- Connect the **keyEvents** and **mouseEvents** events of the JTable to the **Get Detail From DataDescription** method of the Root Proxy.
- Two links appear in dotted lines since the **Get Detail From DataDescription** method requires a parameter.
- Make a **Tear-Off** of the **Rows** property of the Root Proxy. You obtain a variable bean named **rows1**.
- Select the dotted links one at a time. To specify the parameter required by the **Get Detail From DataDescription** method, connect the data-type parameter of the link (here **customerData**) to the **elementAt(int)** method of **rows1**.
- Then connect the **SelectedRow** property of the JTable to the **Arg1** parameter of the link created in the preceding step. This parameter is available when the cursor is placed in the middle of the connection link.

The Composition Editor should look like this:



To avoid complexity, we display here the connection of only one of the two events sent by the JTable.

Activation of the Proxy's Methods and Navigation towards the Orders Window

- **Activation of the Proxy's Methods**

Methods are activated in the same way as in the Root Proxy of the VisualAge Java V1 example.

For more details, refer to *Developing the End User Interface with VisualAge Java V1, Activation of the Proxy's Methods and Navigation towards the Orders Window*.

- **Navigation Management**

Navigation is managed in the same way as in the Root Proxy of the VisualAge Java V1 example.

For more details, refer to *Developing the End User Interface with VisualAge Java V1, Activation of the Proxy's Methods and Navigation towards the Orders Window*

Developing the Orders Window

The Orders window is identical to that developed in the VisualAge Java V1 example.

For more details, refer *Developing the End User Interface with VisualAge Java V1, Developing the Orders Window*.

Creating the Orders Window

In the Workbench, create an **OrdersFrame** class which inherits from **JFrame** in the **example.swing** package, as you did when you created the **Customers** window.

☞ For more details, refer, in this section, to paragraph *Developing the Customers Window, Creating the Window and Integrating the FVP in the Composition Editor*.

Integrating and Promoting the Dependent Proxy

In the Visual Composition tab, place an **OrderProxyLv** variable bean on the Free Form Surface.

☞ For more details on the integration of the Proxy, refer, in this section, to paragraph *Developing the Customers Window, Creating the Window and Integrating the FVP in the Composition Editor*.

You must then promote this variable Proxy so that it can be public from the outside.

To do so, proceed as follows:

- From the Proxy's pop-up menu, choose **Promote Bean feature**.
- In the **Property** column, choose **this** and click on **>>**.
The **OrdersFrame** class has a public property now, in read/write mode, named **OrderProxyLv1This** and typed Dependent Proxy.
- Save the window (**Bean** menu, **Save Bean** choice).

Mapping the Dependent Proxy's rows and detail

The operations to be executed are identical to those required for the Root Proxy.

☞ For more details, refer, in this section, to *Developing the Customers Window*.

Selection of an Instance in rows and Transfer in detail

The operations to be executed are identical to those required for the **rows** and **detail** properties of the Root Proxy.

☞ For more details, refer, in this section, to *Developing the Customers Window*.

Activation of the Proxy's Methods

Use the same principles as those described in *Developing the End User Interface with VisualAge Java V1, Developing the Customers Window, Activation of the Proxy's Methods and Navigation towards the Orders Window*.

Specificities of a Standalone Application

Introduction

This subchapter puts the emphasis on the differences between the development of an applet and the development of a *standalone* application.

☞ A *standalone* application, unlike an applet application, is not meant to be executed by a Web browser.

The main differences between the two types of applications are the following ones:

- For the development of a *standalone* application, the base class in use has a **Frame** type (for awt) or a **JFrame** type (for swing). This class inherits from **java.awt.Frame** (for awt) or **com.sun.java.swing.JFrame** (for swing) whereas an applet inherits from **java.applet.Applet** (for awt) or from **java.applet.JApplet** (for swing).
- As for the development of a *standalone* application, the instantiation of the Root Proxy is conditioned by the use of the **setLocationsFile** method, and not by the use of the **Host** and **Port** properties. This method is used to position the **VAPLOCAT.INI** *locations* file so that the application can directly get to the middleware without passing through the gateway.

Example

This example shows you how to transform an applet previously developed in this subchapter into a *standalone* application. In the example, the **CustomersFrame** and **OrdersFrame** windows are re-used. Now, the **CustomersFrame** window is the starting point of the application.

☞ For the windows developed with the Swing option of VisualAge Java V2, you simply have to replace the awt classes present in the code by the corresponding Swing classes.

No changes are required in the Composition Editor. You just have to insert specific code in the **main** method of the **CustomersFrame** class. The **main** method is the starting point of a *standalone* application.

The specific code is presented below, between the **//begin** and **//end** comment lines.

```
/**
 * main entrypoint - starts the part when it is run as an application
 * @param args java.lang.String[]
 */
public static void main(java.lang.String[] args) {
    try {
        vap.sample.CustomersFrame aCustomersFrame = new vap.sample.CustomersFrame();
        try {
            Class aCloserClass = Class.forName(« uvm.abt.edit.WindowCloser »);
            Class parmTypes[] = { java.awt.Window.class };
            Object parms[] = { aCustomersFrame };
            java.lang.reflect.Constructor aCtor = aCloserClass.getConstructor(parmTypes);
            aCtor.newInstance(parms);
        } catch (java.lang.Throwable exc) {};
        //begin
        aCustomersFrame.setCustomerProxyLv1(new
        com.ibm.vap.generated.proxies.CustomerProxyLv());
        aCustomersFrame.getCustomerProxyLv1().setLocationsFile(« d:\\user\\vapb\\vaplocat.ini
        »);
        //end
        aCustomersFrame.setVisible(true);
        catch (Throwable exception) {
            System.err.println(« Exception occurred in main() of java.awt.Frame »);
        }
    }
}
```

The first instruction specifies the creation of a new Root Proxy. The former Proxy has already been instantiated in the applet and it cannot be re-used here as a constant bean.

The next instruction positions the *locations* file.

Error Management

Principles

Introduction

The management of the errors associated with the handling of VisualAge Pacbase Proxies is based on the 'raise of exceptions' principle specific to the Java language.

The Proxy objects can raise four types of errors from the eBusiness or Pacbench C/S module and also the whole set of Java errors.

For the management of the eBusiness or Pacbench C/S errors, the user is provided with four generic classes whose methods enable to convey the errors raised by the Proxy objects. Each class corresponds to a type of errors:

- Local errors
- Server errors
- System errors
- and Communication errors

All these classes inherit from `java.lang.Throwable` and are stored in the `com.ibm.vap.generic` package.

See also *Chapter 3: Development Principles*, subchapter *Error Management* for the list of the possible local and communication errors, as well as the structure of the error key for the server and system errors.

Programming

Exceptions must be intercepted by the programming in the Client component.

Programming errors requires writing in Java on the one hand, and on the other hand, creating the window used to display these errors.

An example of error management is provided in section *Example of Error Management*.

For more information on the exceptions that can be possibly raised by the Proxies, refer to the *eBusiness & Pacbench C/S Applications: Proxy Programming Interface* manual or, in your VisualAge station, the corresponding method signature.

Local Errors

Local errors produce the `com.ibm.vap.generic.LocalException` exception. This exception contains a property of `int` type that enables the error identification.

The errors which are responsible for this exception are listed in section *Local Errors (Chapter 3: Development Principles)*. They are also described in the HTML documentation associated with the generic classes: Package `com.ibm.vap.generic`.

- ☞ These error messages correspond to constants of the `com.ibm.vap.generic.LocalException` class and represent types of errors. Prefixed with `LOCAL_`, each constant makes up an error key. This error key allows to identify the associated label in the local error labels file `vaperror.properties`.

Server Errors

Server errors produce the `com.ibm.vap.generic.ServerException` exception.

This exception is raised upon the receiving of a logical error message detected by the server. The exception holds the key and associated message.

- ☞ The structure of the error key for the server errors are listed in section *Server Errors (Chapter 3: Development Principles)*.

System Errors

System errors (physical errors) produce a `com.ibm.vap.generic.SystemError` error. This type of error represents an internal and irretrievable error.

- ☞ The structure of the error key for the system errors are listed in section *System Errors (Chapter 3: Development Principles)*.

Communication Errors

Communication errors with the Server produce the `com.ibm.vap.generic.CommunicationError` error. Like any Java error, VisualAge Pacbase communication error returns no key. To know the cause of the error, you must retrieve the message associated with the error (`getMessage()` method of the class).

- ☞ Communication errors are listed in section *Communication Errors (Chapter 3: Development Principles)*.

Example of Error Management

Introduction

The Java code of this example is provided with the Java version of the eBusiness or Pacbench C/S module. It shows a method allowing to manage all the types of errors that are likely to be raised by the Proxies. To do so, three classes have been defined:

- `StandardErrorMessageFactory`: class allowing to create a vector of `StandardErrorMessage` from the exception raised
- `StandardErrorMessage`: class unifying the characteristics of the various types of errors
- `ErrorManagerExample`: graphic class used to display the errors

Presentation of the Non-Visual Classes in Use

- `StandardErrorMessageFactory` class

This class provides the `public static java.util.Vector getStandardErrorMessages (Throwable th)` method allowing to create a vector of `StandardErrorMessage` from a given exception whatever its type may be.

- `StandardErrorMessage` class

Each `StandardErrorMessage` object has a number of properties that are initialized or not according to the error type (local, server, system, communication error, etc.) :

- the **hierarchical Proxy associated with the error** if the error is related to a particular instance of Logical View.

- the **error key** (for local, server or system errors):

It is a character string returned by the server, in the case of a server or system error.

In the case of a local error, it is a character string corresponding to the name of the constant associated with the error type and defined in the `LocalException` class, prefixed with `LOCAL_`.

☞ For the list of all local error types, refer to *Chapter 3: Development Principles, Local Errors*.

☞ To know the error keys corresponding to the server or system errors, refer respectively to *Chapter 3: Development Principles, Server Errors* and *System Errors*.

- the **local label of the error**:

- ♦ in the case of a local, server, or system error, the error key is interpreted by the Client component to get the local label. In the `vaperror.properties` file, a correspondence table allows to determine this label from a given error key.

☞ For information on the `vaperror.properties` file, refer to the *eBusiness & Pacbench C/S Applications: Proxy Programming Interface* manual.

- ♦ for the other error types, the local label corresponds directly to the message associated with the Java exception.

- the **server label of the error**, for the server and system errors, if a error label server has been coded using the Business Logic function.

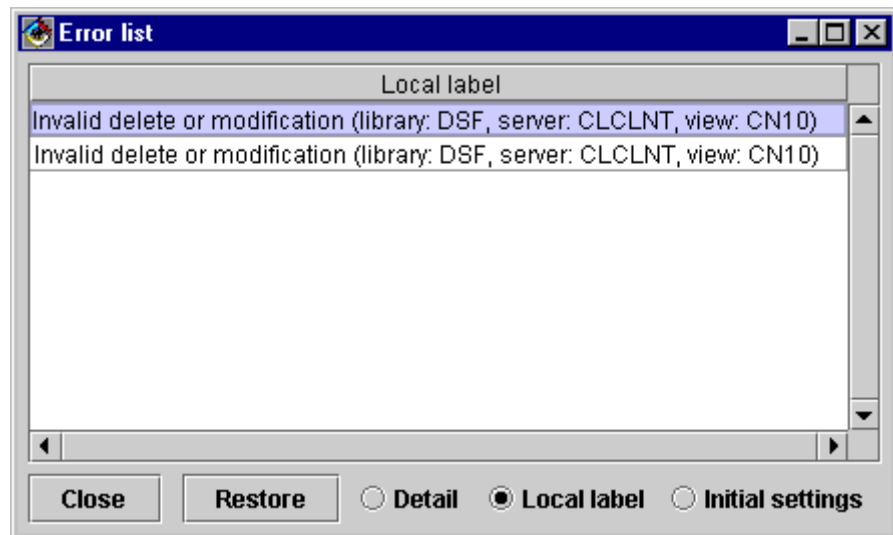
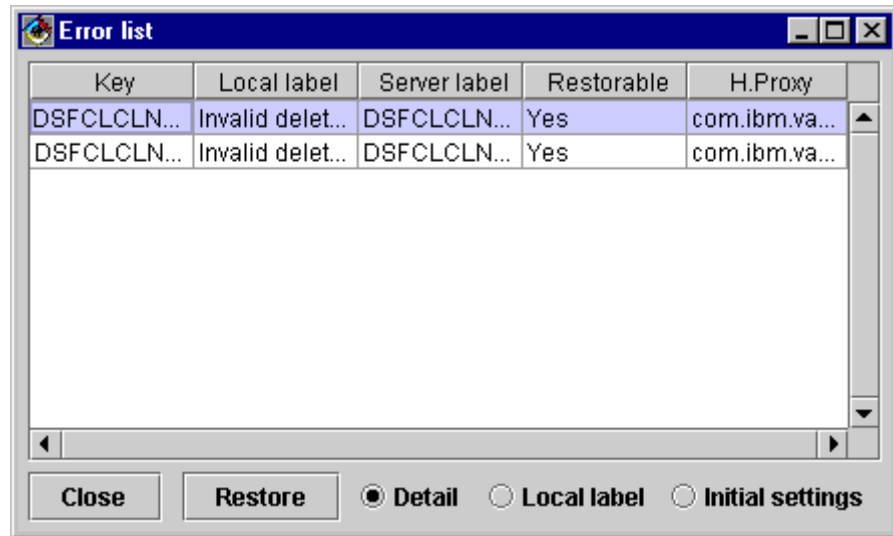
- the boolean property `Restorable`. This property is true:

- ♦ when a Proxy is associated with the error.
- ♦ when it is possible to display again the instance that caused the error in the detail of the window calling the Proxy.

☞ For additional details on error management, refer to the *Developer's Documentation / eBusiness Applications Series: Proxy Programming Interface*.

Presentation of the ErrorManagerExample Visual Class

Graphic Interface



Class Functionalities

This class is used to display a list of error messages corresponding to instances of the `StandardErrorMessage` class.

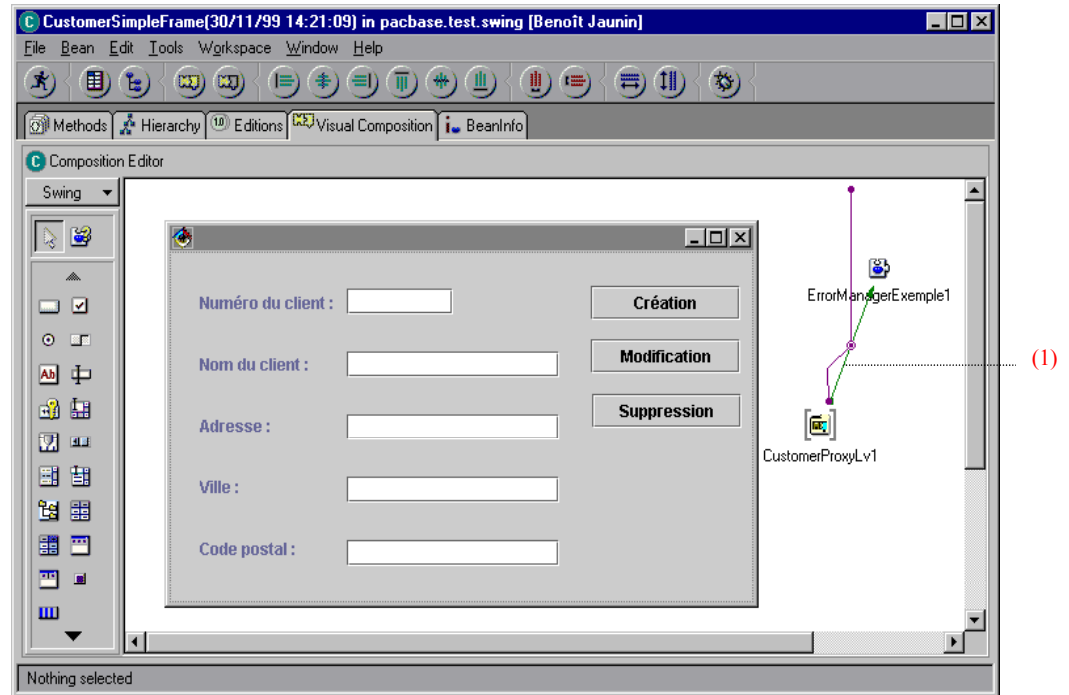
All the graphic controls used by this window are proportional to its size.

The window's functionalities are the following:

- The instances of **StandardErrorMessage** inferred by the **StandardErrorMessageFactory** class are transferred to this window in a vector using the **addStandardErrorMessages (Vector)** method. As long as the window is not closed, the transferred messages are added to the messages already displayed. The messages displayed can be deleted using the **resetCurrentStandardErrorMessages ()** method.
- The properties of **StandardErrorMessage** instances that can be viewed are defined in the **visibleColumns** property of the **ErrorManagerExample** window. This property corresponds to a table of **int** whose values can be the following:
 - **1** to display the error message key
 - **2** to display the local label of the error message
 - **3** to display the server label of the error message
 - **4** to display the restoration status of the error
 - **5** to display the name of the class associated with the hierarchical Proxy
- Three radio-buttons have been inserted to dynamically select different presentations for the list of **StandardErrorMessage** instances:
 - **Detail** is used to display all the properties of each **StandardErrorMessage** instance.
 - **Local label** is used to display only the local label of the **StandardErrorMessage** instance.
 - **Initial settings** is used to display the properties defined in the **visibleColumns** attribute.
- The **Restore** button is enabled when an instance of **StandardErrorMessage** is selected and if the error context can be restored. When the user clicks on the button, the error context is restored and the window that manages the Logical View instance that caused the error is displayed.

To implement this functionality, the **ErrorManagerExample** window must know each window managing the **detail** property of a Proxy node using the **addProxyManagingbyWindow** method. This method takes as parameters the Proxy node (**HierarchicalProxyLv**) and the error management window (**Jframe**).

Example



This example illustrates the principle allowing to record the association of a hierarchical Proxy and the window that calls it, and inform the error management window of this association.

The link (1) connects the event `this` of the Proxy to the method `addProxyManagingbyWindow` of the error management window (`ErrorManagerExemple1`). In this context, this method will be executed on the Proxy instantiation.

The two other links allow to transfer the hierarchical Proxy instance (`this` parameter of the Proxy to the `hp` parameter of the connection) and the window instance (`this` parameter of `CustomerSimpleFrame` to the `wi` parameter of the connection).

Code for Displaying the Error Window

The solution proposed is adapted to applications developed in VisualAge Java with the visual composition editor.

It assumes that the error management window (`ErrorManagerExemple1` class) has been inserted as a *class*-type or *variable*-type bean in the visual composition editor containing the window that calls the error management window.

The exceptions associated with the calling window are handled by inserting in the method `handleException(Throwable)` of the window, the following:

```
private void handleException(Throwable exception) {
    java.util.Vector standardErrorMessages =
    pacbase.test.swing.ErrorManager.StandardErrorMessageFactory.getStandardErrorMessages(exception);
    if (standardErrorMessages != null) {
        if (standardErrorMessages.size() >= 0) {
            getErrorManagerExemple1().addStandardErrorMessages(standardErrorMessages);
            getErrorManagerExemple1().showStandardErrorMessages();
        }
    }
}
```

Communication Management

This subchapter contains all the information needed by the developer to implement the middleware used by the applications generated with the eBusiness or Pacbench C/S module.

☞ The information related to the operation of the middleware used by these applications when they are deployed is documented in the *Middleware User's Guide*.

Processing a Request

Middleware services are executed from a set of specific communication classes provided upon the installation of the product.

The communication with the Server is executed via the **ServerAdapter** interface. There are two ways of implementing this interface:

- **MiddlewareAdapter** which directly accesses the middleware's native DLLs (in C++) which are locally installed. It also allows to parameterize the communication context (**location**, **userId**, **password**, **clientEncoding**...) and its operating mode (**traceLevel**, **nbMaxConnection**, **connectionCleaningInterval**...).
- **GatewayAdapter** which uses a Gateway or a Relay via TCP/IP. This type of implementation is dedicated to applets. The parameters (**host**, **port**, **userid**, **password**, **clientEncoding**...) which define the communication with **VapGateway** or **vaprelay** must be specified. However the parameters which define the communication with the application server must be specified in the **VapGateway** module.

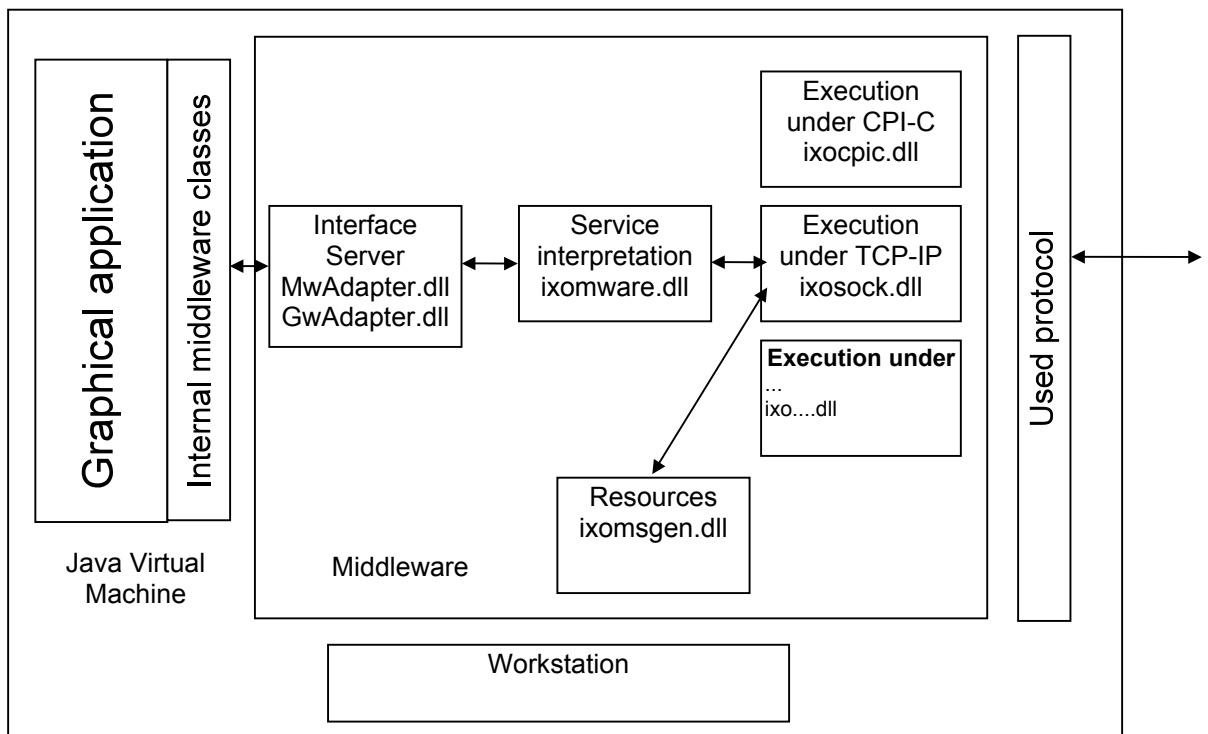
☞ For more information, refer to the *Middleware User's Guide*.

When a **ServerAdapter** object is instantiated, it systematically creates another object of the **Requester** class which implements the technical conditions to communicate with the Server, e.g. the methods for sending and receiving messages. There are two ways of implementing the **Requester** class. These two implementations are associated, respectively, with the **MiddlewareAdapter** and **GatewayAdapter** classes :

- **NativeRequester** which directly accesses (via **JNI**) the C functions of the middleware DLLs.
- **GatewayRequester** which implements the messages via **vaprelay** or **VapGateway**.

Direct Access to the Middleware

The following diagram shows you the processing of a request:



To instantiate a Folder in direct mode, you can use the default builder `CustomerProxyLv()`

For example:

```
myProxy = new CustomerProxyLv() ;
myproxy.setLocationsFile("c:\vap\gen\java\VAPLOCAT.INI") ;
```

The second instruction specifies which location file must be used. By default, we search for a `VAPLOCAT.INI` file located in the current directory.

Access via a Gateway

Two builders can be used to instantiate a Folder via the Gateway:

- `CustomerProxyLv(String host)`
For example: `new CustomerProxyLv("9.134.5.146")`
- `CustomerProxyLv(String host, int port)`
For example: `new CustomerProxyLv("9.134.5.146", 6001)`

☞ If `host` is equivalent to null, we consider that it is a local middleware.

Access via a Particular Adapter

You access the middleware via a particular adapter by using the `setServerAdapter(ServerAdapter)` or `setServerAdapterName(String)` property.

For example: `proxy.setServerAdapterName("GwAdapter")`

Dynamic Change of the Middleware Access Parameters

Different methods of the Root Proxy allow dynamic changes of these parameters:

- `void setHost(String host)`
 - ☞ If `host` is equivalent to null, we consider that it is a local middleware.
- `void setPort(int port)`
- `void setLocationsFile(String filename)` (direct mode)
- `void setTraceFile(String filename)`
- `void setTraceLevel(int traceLevel)` (direct mode)

Definition of the Use Context via the Location Editor

The communication management requires the definition of the middleware use context. This context corresponds to a location.

You define locations, via the Location Editor tool, in a specific file named `vaplocat.ini`.



See the online help available from the Location Editor. See also the *Middleware User's Guide*, chapter *Protocols Description & Configuration* for the list and the meaning of the parameters you must specify for each location, depending on the protocol in use.

There are various ways of launching the Location Editor:

Launching from the eBusiness Module of Developer workbench

You launch the Location Editor from the 'Applications' or 'Folders' tab of Developer workbench.

Launching from VisualAge for Java

To launch the Location Editor from VisualAge for Java directly, select, in the 'Workspace' menu, 'Tools' → 'VisualAge Pacbase eBusiness' → 'Location Editor'.

Launching from the .exe File

Execute the `vapLocationEditor.exe` file.

You can parameterize the launching of the Location Editor via the following option:

`-inputfile<INPUT_FILE>`: this parameter enables you to indicate the path of the file which will initialize the Editor. This file is either an existing location file or a `.gvc` file (which contains the extraction of eBusiness proxies). All the characteristics of the Communication Monitors are present in this input file.

Instead of initializing the Editor by an input file, you can choose to launch the Editor in expert mode in order to create and modify Communication Monitors. In this case, use the `-expertmode` option.

Launching from a Java Virtual Machine

To launch the Location Editor from a Java Virtual Machine, execute the `java_vapLocationEditorTool.bat` file.

This `.bat` is an example which you must modify according to the location of your JDK (Java Developer ToolKit) or JRE (Java Runtime Environment).

You can indicate the option indicated above to parameterize the launching of the Location Editor.

Testing the Generated Application – Packaging

Testing the Generated Application

Testing Server Components with the Services Test Facility

You can test the server-side components of your application without having to develop the application's graphic interface. You can do that via the Services Test Facility.

This Facility enables you to see the proxy's attributes, test the available methods and the communication.

You can then identify problems related to:

- The design and implementation of Folders and Elementary Components,
- Proxies generation,
- Communication and middleware.



See the online help available from the Services Test Facility.

There are different ways of launching the Services Test Facility:

Launching from the eBusiness Module of Developer Workbench

You launch the Services Test Facility from the 'eBusiness Applications' tab of the workbench.

Launching from WSAD (or Eclipse)

To launch the Services Test Facility from WSAD (or Eclipse), right click on:

- a Java project which contains VisualAge Pacbase proxies,
- or one or more packages which contain VisualAge Pacbase proxies,
- or one or more Java classes which correspond to VisualAge Pacbase proxies.

Launching from VisualAge for Java

To launch the Services Test Facility from VisualAge for Java directly, select, in the 'Workspace' menu, 'Tools' → 'VisualAge Pacbase eBusiness' → 'Services Test Facility'.

Launching from the .exe File

Execute the `vapServicesTestFacility.exe` file.

You can parameterize the launching of the Services Test Facility via the following options:

`-classpath<PATH>`: this parameter enables you to indicate the path of the proxies classes that you want to test.

`-folders<PROXY_CLASS_NAME>`: this parameter enables you to indicate the proxies that you want to test.

Launching from a Java Virtual Machine

To launch the Services Test Facility from a Java Virtual Machine, execute the `java_vapServicesTestFacility.bat` file.

This .bat is an example which you must modify according to the location of your JDK (Java Developer ToolKit) or JRE (Java Runtime Environment).

You can indicate the options indicated above to parameterize the launching of the Services Test Facility.

Version Compatibility Check

If the version control option detects any discrepancy, it means that the Elementary Component and the Proxy object were not generated with the same version number. So you must:

- regenerate the Proxy object if you have regenerated only a new version of the Elementary Component,
- or implement, in VisualAge, the generated graphic application including the new proxy component if it has not already been done,
- or implement the generated Elementary Component if it has not already been done.

Packaging

Packaging consists in making a developed and tested application or applet available for use outside the development environment. To do so, you must export this applet or application.

Reminder: Prerequisites

In the operating phase, the **VAPLOCAT.INI** file must be located in the same directory as the final applications. The developer responsible for the installation of these applications must make sure of it.

- Applet

For the implementation of a Java applet, the following elements should be installed on the user's station:

- An HTTP server
- A 1.1 *enabled* Java Web browser
- The gateway and the relay are installed on the HTTP server station

- *Standalone* application

For the implementation of a Java *standalone* application, the Java Runtime Environment (JRE) should be installed on the user's workstation.

☞ For more details on the execution environment, refer to the *eBusiness & Pacbench C/S Applications: Concepts & Architecture* manual.

Export

What will You Export ?

You must export all the execution classes used by the application which do not belong to the base classes. The editing classes such as the **BeanInfo** classes or the beans used for a quick mapping of VA Pac Data Element-type properties are optional.

So you must export:

- All the packages of the project which contains the runtime, except the packages `com.ibm.vap.beans` and/or `com.ibm.vap.beans.swing` (depending on the package used in your application). For those two packages, export only the classes actually used by your application.
- The project containing the generated Proxy components,
- The applet or application itself, that is to say the whole project containing the applet or the application or the package(s). In our example, it is the `vap.sample` package (for the V1 example) or `example.swing` (for the V2 example).
- possibly, the external beans: in the V1 example, we use the `ImulticolumnListbox` bean. Therefore, it is necessary to export the whole package, which is the `com.ibm.ivj.javabeans` package.

Implementation

- **For an applet**

Before exporting an applet, you must create a directory in the HTTP server root, in which will be stored the export result. For example `c:\www\html\codebase`. This directory constitutes the Root of Java classes in the HTTP server.

☞ You can export the `class` files directly in the HTTP server tree structure, in `codebase` in our example, or in another directory. In this case, copy these files in the directory before implementing the applet, and respect the packages' tree structure.

- **For a standalone application**

In this case, the location of the directory in which the result of export will be stored does not matter. You just have to declare this directory in the `CLASSPATH` variable.

How to export?

- **From WSAD**

In the **File** menu, choose **Export**. The export wizard opens.

☞ For more information, refer to the WSAD online help.

- **From VisualAge for Java**

To export, follow these steps:

- In the VisualAge for Java Workbench, select all the elements that you want to export,
- In the **File** menu, select the **Export** choice,
- In the **SmartGuide - Type of Export** window, select the **Class Files** option, and then click on **Next**,
- In the **SmartGuide - Export to files** window, enter the output directory name or select it with the **Browse** button, by using the **Create package subdirectories** option.



For more information, refer to the online documentation accessible from the VisualAge for Java workstation online help (**Help** menu, **Tasks**. Choice, then **Exporting to the file system**) or from Windows Explorer (open the **Tasks** directory, select an HTML file in the **Export** director, the **overview.htm** file being a suggested entry point).

Optimizing the Downloading Time of the .class File

For this option, the JDK should be installed in the execution environment.

Once the export has been executed in the form of **.class** files, you can optimize the downloading time of classes by saving all these files into a single archived file **.jar**.

In the DOS window, put your cursor in the export output directory, then enter the following command:

```
jar cvf sample.jar com vap
```

where:

- **sample.jar** is the name of the archived file,
- **com** and **vap** represent two directories which contain all the **.class** files required for the execution of the final application.

For an applet, the **.jar** file obtained must be copied in the HTTP server tree structure.

Writing an HTML File (Applet Only)

Finally, it is necessary to write an HTML file containing the applet to be able to execute it in a Web browser. This file is used to set some parameters, such as the applet width and length.

To apply this procedure to our example, you must create, in the **c:\www\html\codebase\vap\sample** directory, an **index.html** file containing the following text:

```
<HTML>
<TITLE>
Sample Applet
</TITLE>
<BODY>
<CENTER>
<APPLET      code="vap.sample.SampleApplet.class"      WIDTH=1000"
HEIGHT=1000
codebase="/codebase"></APPLET>
</CENTER>
</BODY>
</HTML>
```

or the following text, if you have created a **.jar** archived file:

```
<HTML>
<TITLE>
Sample Applet
```

```

</TITLE>
<BODY>
<CENTER>
<APPLET      code="vap.sample.SampleApplet.class"      WIDTH=1000
HEIGHT=1000
archive="/codebase/sample.jar"
codebase="/codebase"></APPLET>
</CENTER>
</BODY>
</HTML>

```

Application Deployment

☞ To deploy an application, follow the operations described in the WSAD or VisualAge for Java documentation.

But you must also install some files specific to the use of the eBusiness or Pacbench C/S module.

- **For an applet:**

The end-user workstation must be equipped with a browser.

- **For a *standalone* application:**

- If no gateway is used, you must install, on the end-user workstation:
 - ♦ the middleware package,
 - ♦ **VAPLOCAT.INI**,
 - ♦ **VAPRUN.JAR**,
 - ♦ **VAPSWING.JAR** if the application uses **Swing** or **VAPAWT.JAR** if the application uses **AWT**.
- If a gateway is used, you must install, on the server where the gateway is installed:
 - ♦ **GATEWAY.EXE**,
 - ♦ the middleware package,
 - ♦ **VAPLOCAT.INI**,

In this case, you must not install any of these files on the end-user workstation. However, on this workstation, you must install:

- ♦ **VAPSWING.JAR** if the application uses **Swing** or
- ♦ **VAPAWT.JAR** if the application uses **AWT**.

Chapter 5: Developing a COM Client

Once you have generated and compiled the Proxy objects, you just need to use them in a client language able to manage ActiveX objects. In the example described hereafter, the client language used is Visual Basic.

This chapter gives you a detailed description of a client development, including the following steps: insertion of Proxy objects with programming links involving actions, attributes and events, error management, communication management and test of the application.

Visual Basic Example of a COM Proxy Use

This example presents the development of a Visual Basic executable program using a COM Proxy. This Proxy has been generated then compiled on a workstation with Visual Studio 6.0.

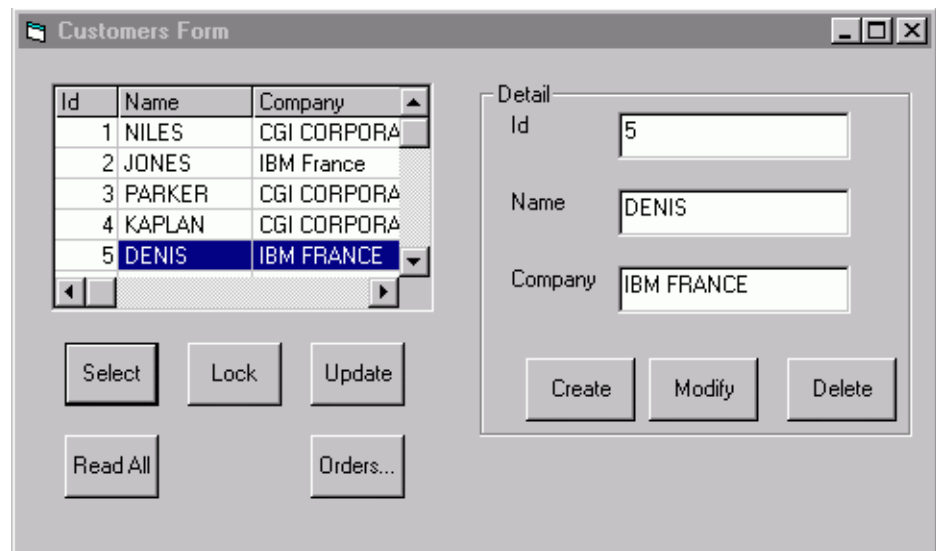
In the example, three Elementary Proxies of the FVP are used:

- The Root Proxy corresponding to the **Customers** node which manages the customers in the information system described by the Folder.
- The Dependent Proxy corresponding to the **Orders** node which manages the orders in the information system described by the Folder.
- The Dependent Proxy corresponding to the **Order lines** node which manages the order lines in the information system described by the Folder.

Presentation of the End User Interface

The graphic user interface consists of two windows:

- **The Customers Window**



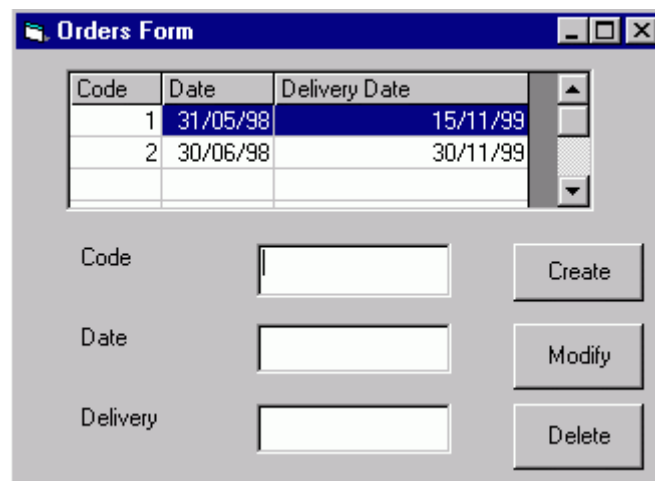
This window opens automatically at the Visual Basic executable program start. It contains the following functionalities:

- The **Select** button is used to display a list of customers.

- The **Read all button** is used to submit a reading request of the selected customer 's orders from this window.
- As a lock option has been specified in the Folder, the user must click on the **Lock** button to lock the instance selected in the list before starting any updating process.
- The **Update** button is used to update the database.
- The **Orders . . .** button is used to display the Orders Frame window.
- The **Modify** button is used to modify customers in the detail after locking the instance.

When a customer is selected in the list, the user must click on **Lock** to appropriate this customer for a short time. Then, he can enter modifications and click on **Modify**.

- **The Orders Window**



The **Orders** window opens when the user clicks on the **Orders . . .** button in the **Customers** window.

For each customer selected in the detail of the **Customers** window, this window allows you to:

- view this customer's list of orders.
- select an order in the list and display it in the detail.
- create, modify, delete orders.

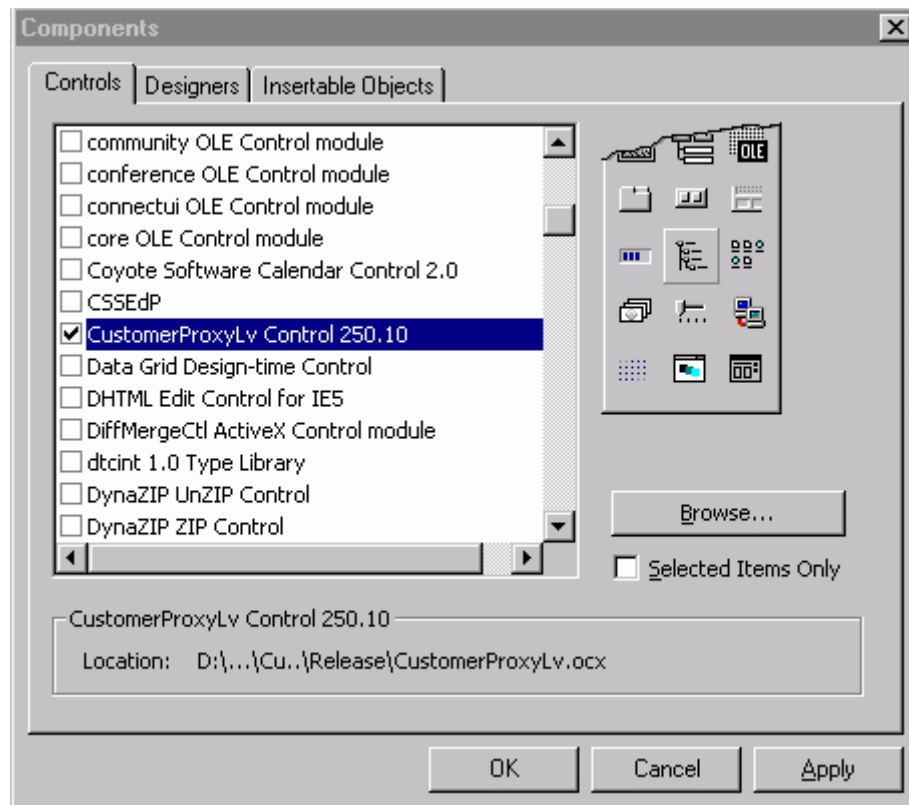
Visual Basic Development Example

Developing the end user interface includes the construction of the **Customers** and **Orders** windows as well as the interfacing with the COM Proxy.

You will not find here detailed information on how to use Visual Basic and functions. If you are not familiar with them, refer to the appropriate documentation.

Inserting the COM Proxy in the Visual Basic Project

To graphically integrate a Proxy to the Visual Basic toolbar, select the **Project** menu, then **Components**. Then in the dialog box that opens up, check the Proxy to be inserted.



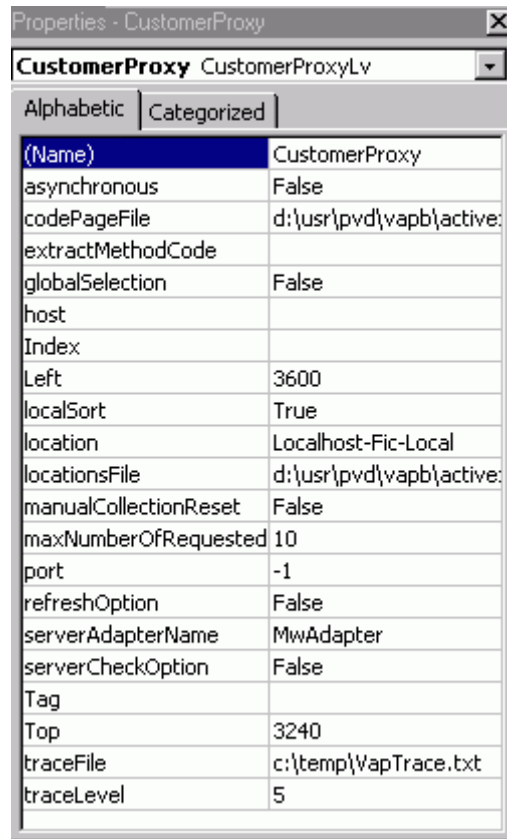
If the Proxy you want is not in the list of objects, you can directly use the **Browse** button to find it by specifying the object *nameProxyLV.ocx* that is in the generation directory **Release**.

The selected Proxy is then displayed in the toolbar. You just need to insert it in your Form.

Setting of the Proxy in the Application Design Mode

Some Proxy attributes such as the location type, the trace level or the trace file, etc., can be parameterized when the application is designed.

You can view and modify these attributes using the **Properties** window that can be accessed via the **View** menu, **Properties window** choice. Select the Proxy you need in the list of objects.



Selecting and Filling the Grid Representing the rows Attribute

You find hereafter an example of code used to fill in a MSFlexGrid-type grid resulting from a selection action, once the user has clicked on the **Select** button.

```
Private Sub CmdSelect_Click()
    CustomerProxy.selectInstances
    'check if no error has occurred
    processErrors

    ' Put the customer rows in the grid
    For i = 0 To CustomerProxy.getRowsCount - 1
        Set customerDetail = CustomerProxy.getRowsElementAt(i)
        GrdCustomer.AddItem customerDetail.Nuclie & vbTab &
            customerDetail.Nomcli & vbTab &
customerDetail.Raisoc, i + 1
        Set customerDetail = Nothing
    Next i
End Sub
```

Error Processing

You find below an example that enables you to manage the possible errors sent by the Proxy. This function can be called after each call of an action that can send an error.

```
Sub processErrors()
    Dim i As Integer
    Dim vap_errors As VapCollection
    Dim vap_error As VapError

    Set vap_errors = CustomerProxy.getErrors
```

```

    i = vap_errors.Count

    While i > 0
        Set vap_error = vap_errors.Item(i - 1)
        MsgBox vap_error.getLabel, vbInformation, "Error has
occurred"
        Set vap_error = Nothing
        i = i - 1
    Wend

    Set vap_errors = Nothing
End Sub

```

- ☞ The objects **VapCollection** and **VapError** are provided by the **VapTools.dll** utility. They must be referenced in the Visual Basic application. To do so, select the **Project**, then **References...** menus, then check **VapTools**.

Filling of the detail Attribute

The example below shows how the **detail** attribute of the Proxy is filled in when a line is selected from the grid of the **rows** attribute.

```

Private Sub GrdCustomer_Click()
    ' Retrieve the current select row in the detail
    Set customerDetail =
CustomerProxy.getRowsElementAt(GrdCustomer.Row - 1)
    TxtId = customerDetail.Nuclie
    TxtName = customerDetail.Nomcli
    TxtCompany = customerDetail.Raisoc
    CustomerProxy.getDetailFromData customerDetail
    Set customerDetail = Nothing

```

Error Management

There are four error types:

- local errors,
- server errors,
- system errors,
- communication errors.

- ☞ The list of all the errors is available in *Chapter 3: Development Principles Error Management*.

In a COM environment, you manage errors via the **VAPERROR** interface. This interface contains attributes, actions and events which enable you to manage all error types.

The **VAPERROR** interface is available in the **VapTools** library delivered with the generator.

- ☞ The attributes, actions and events available through the **VAPERROR** interface are documented in the *eBusiness & Pacbench C/S Applications: Proxy Programming Interface manual, Error Management* chapter.

Communication Management

This subchapter contains all the information needed by the developer to implement the middleware used by the applications generated with the eBusiness or Pacbench C/S module.

☞ The information related to the operation of the middleware used by these applications when they are deployed is documented in the *Middleware User's Guide*.

Processing a Request

Middleware services are executed from a set of specific communication classes provided upon the installation of the product.

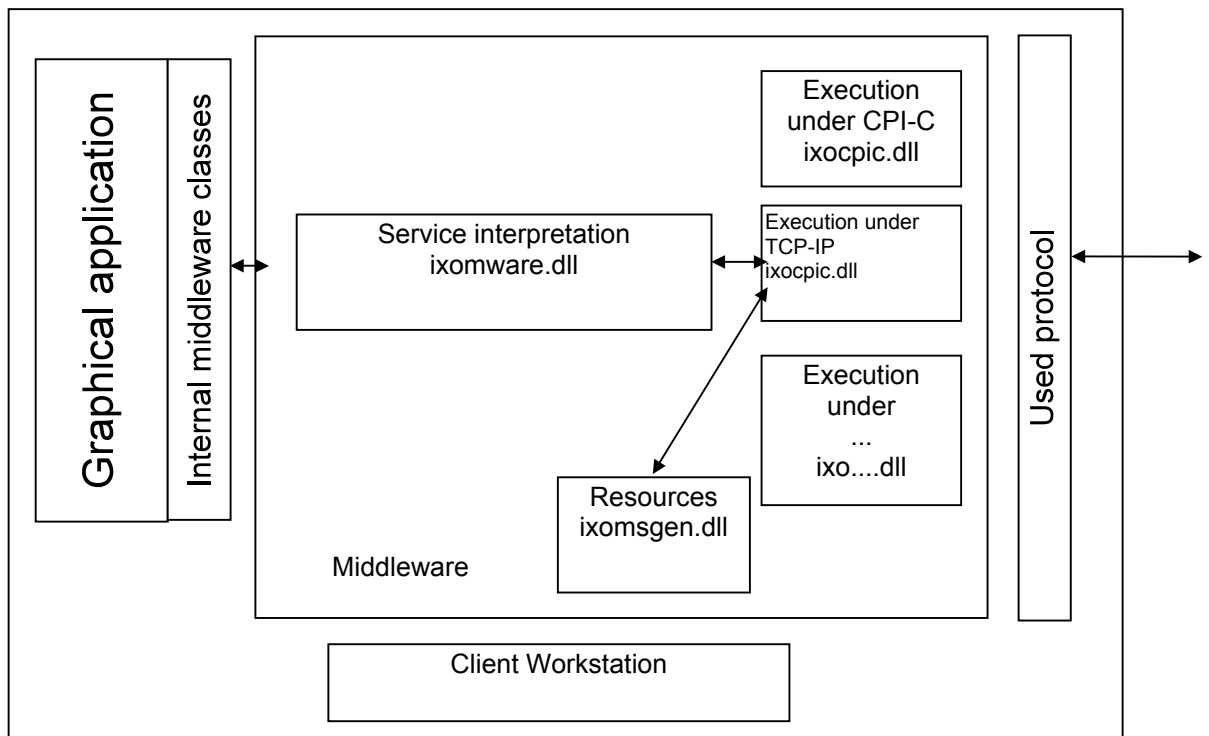
The communication with the Server is executed via the **ServerAdapter** interface. There are two ways of implementing this interface:

- **MiddlewareAdapter** which directly accesses the middleware's native DLLs (in C++) which are locally installed. It also allows to parameterize the communication context (**location**, **userId**, **password**, **clientEncoding**...) and its operating mode (**traceLevel**, **nbMaxConnection**, **connectionCleaningInterval**...).
- **GatewayAdapter** which uses a Gateway or a Relay via TCP/IP. This type of implementation is dedicated to applets. The parameters (**host**, **port**, **userid**, **password**, **clientEncoding**...) which define the communication with **VapGateway** or **vaprelay** must be specified. However the parameters which define the communication with the application server must be specified in the **VapGateway** module.

☞ For more information, refer to the *Middleware User's Guide*.

When a **ServerAdapter** object is instantiated, it systematically creates another object of the **Requester** class which implements the technical conditions to communicate with the Server, e.g. the methods for sending and receiving messages. There are two ways of implementing the **Requester** class. These two implementations are associated, respectively, with the **MiddlewareAdapter** and **GatewayAdapter** classes :

- **NativeRequester** which directly accesses (via **JNI**) the C functions of the middleware DLLs.
- **GatewayRequester** which implements the messages via **vaprelay** or **VapGateway**.



Definition of the Use Context via the Location Editor

The communication management requires the definition of the middleware use context. This context corresponds to a location.

You define locations, via the Location Editor tool, in a specific file named **vaplocat.ini**.

See the online help available from the Location Editor. See also the *Middleware User's Guide*, chapter *Protocols Description & Configuration* for the list and the meaning of the parameters you must specify for each location, depending on the protocol in use.

There are various ways of launching the Location Editor:

Launching from the eBusiness Module of Developer workbench

You launch the Location Editor from the 'Applications' or 'Folders' tab of Developer workbench.

Launching from the .exe File

Execute the **vapLocationEditor.exe** file.

You can parameterize the launching of the Location Editor via the following option:

-inputfile<INPUT_FILE>: this parameter enables you to indicate the path of the file which will initialize the Editor. This file is either an existing location file or a **.gvc** file (which contains the extraction of eBusiness proxies). All the characteristics of the Communication Monitors are present in this input file.

Instead of initializing the Editor by an input file, you can choose to launch the Editor in expert mode in order to create and modify Communication Monitors. In this case, use the **-expertmode** option.

Application Deployment

Deploying a client application using an ActiveX Proxy consists in installing the runtimes and files required for the operation of the developed client application on all the workstations, i.e. to make sure that this application can be used outside its development environment, and on other workstations. So you must install:

- **MFC42.DLL** (Microsoft),
- **MSVCRT.DLL** (standardly installed with Windows NT - Microsoft),
- **OLEAUT32.DLL** (standardly installed client languages supporting the COM standard - Microsoft),
- the middleware package,
- **VAPLOCAT.INI**,
- **VAPRUNTIME.DLL** (runtime of the generated COM Proxies),
- **VAPTOOLS.DLL** (utility of the generated COM Proxies).

☞ To enable the multilingualism function of the Proxies, the error message file (**VapErrorMsg.pro**) and the label and authorized values file specific to each Proxy (**<folder>_Labels.pro**) must be stored on the end-user workstation.

Chapter 6: Index

A

Actions/Methods	
belongsToSubSchema	48
checkExistenceOfDependencies	30, 38
checkExistenceOfDependentInstances	30, 38, 41
completeInstance	48
createInstance	26
createRequest	43
createUserInstance	43
createUserServiceInstance	44
deleteUserInstance	43, 44
executeUserService	40
executeUserServices	43, 44
get<delco>Index	47
getCheck(int index)	47
getCheck<fieldIndex>	47
getDetailFromData	31
getDetailFromDataDescription	31, 33, 35
getProxyContext	42
getRowCount	37, 39
getRowsElementAt(Int i)	37, 39
getUserInputRowCount	44
getUserInputRowsElementAt(Int i)	44
getUserOutputRowCount	44
getUserOutputRowsElementAt(Int i)	44
getUserServiceCodesCount	43
getUserServiceCodesElementAt(Int i)	43
initFromProxyContext	42
initializeInstance	39
is<delco>Present	46, 47
lock	41, 44
modifyInstance	26
modifyUserInstance	43, 44
readAllChildren(data)	41
readAllChildrenFrom	30, 41
readAllChildrenFromCurrentInstance	41
readAllChildrenFromDetail	30, 35
readFirstChildren(data)	41
readFirstChildrenFrom	41
readFirstChildrenFromCurrentInstance	41
readFirstChildrenFromDetail	30, 33
readInstance	30, 37, 40, 41
readInstanceAndLock	30, 41
readInstances	30, 40, 41
readInstanceWithAllChildren	30, 37, 40, 41
readInstanceWithAllChildrenAndLock	30, 41
readInstanceWithFirstChildren	30, 37, 40, 41
readInstanceWithFirstChildrenAndLock	30, 41
readNextPage	30, 33, 34, 36, 41
readPreviousPage	30, 41
readWithAllChildren	30
readWithAllChildrenAndLock	30
readWithFirstChildren	30
readWithFirstChildrenAndLock	30
resetCollection	39
resetSubSchema	48
resetUserRows	44
resetUserServiceCodes	43, 44
resetUserServiceInputInstances	43
selectInstances	30, 36, 41
sendRequest	43
set<delco>Present(a Boolean)	47
setCheck<fieldIndex,a Boolean>	47
setNull<delco>Present(boolean a Boolean)	46
setRequest	43
undoAllLocalFolderUpdates	38, 39
undoLocalFolderUpdates	38, 39
unlock	41
updateFolder	30
Attributes/Properties	

action	17, 23
detail	31, 37, 38, 45, 51, 52
extractMethodCode	36
extractMethodCodes	36
globalSelection	30, 36, 37
localSort	27
manualCollectionReset	39
maximumNumberOfRequestedInstances	30, 37
maxNumberOfRequestedInstances	30
refreshOption	39
rows	13, 27, 31, 37
selectionCriteria	36
serverCheckOption	26
serverCheckOption	47
subSchema	29
subSchema	48
subSchemaList	29
updatedFolders	30
UpdatedFolders	17, 23
updatedInstancesCount	17, 23
userDetail	43, 44
userServiceCode	43
userServiceCodes	43
userServiceInputRows	43
userServiceOutputRows	43

E

Events	
LOCK_FAILED	45
lockFailed	45
NO_PAGE_AFTER	30, 50
NO_PAGE_BEFORE	30, 50
noPageAfter	30, 49
noPageBefore	30, 48
NOT_FOUND	50
notFound	49
PAGE_AFTER	50
PAGE_BEFORE	50
pageAfter	49
pageBefore	49

G

Generated classes	
[Prefix]Buffer	17, 23
[Prefix]Data	17, 23
[Prefix]DataUpdate	17, 23
[Prefix]ProxyLvXMLMapping	18, 23
[Prefix]ProxyLvXMLWrapper	18, 23
[Prefix]SelectionCriteria	17, 23
[Prefix]Session	18
[Prefix]SessionBean	18
[Prefix]SessionHome	18
[Prefix]TableModel	18
[Prefix]UpdateTableModel	18
[Prefix]UserData	17, 23
DataDescription	13
selectionCriteria	13
Generated XML Schema	
[Folder_Name].xsd	18, 23
Generic classes	
CommunicationError	13
DependentNode	13
DependentProxyLv	13
Folder	13
HierarchicalNode	13
HierarchicalProxyLv	13
LocalException	13
Node	13
Pacbase Date Choice	14

Pacbase Date Field	14
Pacbase Date Swing Field	14
Pacbase Decimal Choice	14
Pacbase Decimal Field	14
Pacbase Integer Choice	14
Pacbase Integer Field	14
Pacbase Long Choice	14
Pacbase Long Field	14
Pacbase Swing Date ComboBox	14
Pacbase Swing Date RadioButtonGroup	14
Pacbase Swing Decimal ComboBox	14
Pacbase Swing Decimal Field	14
Pacbase Swing Decimal RadioButtonGroup	14
Pacbase Swing Integer ComboBox	14
Pacbase Swing Integer Field	14
Pacbase Swing Integer RadioButtonGroup	14
Pacbase Swing Long ComboBox	14
Pacbase Swing Long Field	14
Pacbase Swing Text ComboBox	14
Pacbase Swing Text Field	14
Pacbase Swing Time ComboBox	14
Pacbase Swing Time Field	14
Pacbase Text Choice	14
Pacbase Text Field	14
Pacbase Time Choice	14

Pacbase Time Field	14
ProxyLv	13
ReferenceNode	13
ReferenceProxyLv	13
RootNode	13
ServerException	13
SystemError	13
VapDependentProxyProperties	13
VapException	13
VapFolderProperties	13
VapHierarchicalProxyProperties	13
VapProxyProperties	13
VapReferenceProxyProperties	13
XMLMapping	14
XMLWrapper	14

M

Methods..... See Actions/Methods

P

Properties..... See Attributes/Properties

This index is not an exhaustive list of the public interface elements.



To obtain the list of elements, refer to the *Graphic Presentation: Public Interface of Generated Components Reference Manual*.