Application Development with
VisualAge for Smalltalk and MQSeries

May 1997

IBM
International Technical Support Organization
Raleigh Center
International Technical Support Organization

Application Development with
VisualAge for Smalltalk and MQSeries

May 1997
Take Note!

Before using this information and the product it supports, be sure to read the general information in Appendix F, “Special Notices” on page 159.

First Edition (May 1997)

This edition applies to the following products:

- VisualAge for Smalltalk for OS/2 Version 4
- VisualAge for Smalltalk for WindowsNT Version 4
- MQSeries for AIX Version 2 Release 2.1
- MQSeries for OS/2 Version 2 Release 2.0.1
- MQSeries for Windows NT Version 2 Release 0

Comments may be addressed to:
IBM Corporation, International Technical Support Organization
Dept. HZ8 Building 678
P.O. Box 12195
Research Triangle Park, NC 27709-2195

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

© Copyright International Business Machines Corporation 1997. 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.
3.5.3 How to Test a Client/Server Connection .......................... 28
3.6 Communication between Queue Managers .......................... 29
  3.6.1 How to Define a Connection between Two Systems ............ 30
  3.6.2 How to Start Communication .................................. 32
  3.6.3 How to Start Channels ........................................ 32
3.7 How to Trigger Applications ...................................... 34
  3.7.1 How a Client Sends a Request and Receives a Reply ........... 34
  3.7.2 How a Server Triggers an Application ........................ 36
3.8 Message Queuing Interface (MQI) .................................. 37

Chapter 4. How VisualAge and MQSeries Work Together ............... 41
  4.1 The VisualAge MQSeries Parts .................................. 41
    4.1.1 MQSeries Procedure Dialog Part ............................ 41
    4.1.2 MQSeries Connection Part .................................. 42
    4.1.3 MQSeries Connection Specifications Part .................. 42
    4.1.4 MQSeries Message Part .................................... 42
  4.2 How VisualAge and MQSeries Interact ............................ 43
  4.3 More about the Background Job .................................. 44
  4.4 About Reply and Request Queues ................................ 45
  4.5 About Syncpoint Processing ..................................... 45
  4.6 About Data Conversion and Record Parser ........................ 45

Chapter 5. The First VisualAge for Smalltalk MQ Application .......... 47
  5.1 What the Application Does ...................................... 47
  5.2 How to Set Up MQSeries ........................................ 48
  5.3 How to Define a Message ....................................... 49
  5.4 How to Set Up VisualAge for Smalltalk .......................... 50
  5.5 How to Develop an Application ................................ 51
    5.5.1 How to Create a VisualAge Application .................... 51
    5.5.2 How to Create a Visual Part ................................ 52
    5.5.3 How to Create Controls .................................... 53
    5.5.4 How to Use the MQSeries Procedure Dialog ................. 55
  5.6 How to Test the Application .................................... 58

Chapter 6. A Quick Tour of a VisualAge for Smalltalk MQ Application .. 61
  6.1 How to Run the Program ........................................ 61
  6.2 How to Create the User Interface ................................ 63
  6.3 How to Connect and Disconnect .................................. 64
    6.3.1 How to Connect to MQSeries Parts ........................... 65
    6.3.2 How to Display Error Messages .............................. 66
    6.3.3 The First Test ............................................. 67
  6.4 How to Put a Message Into a Queue ............................. 68
  6.5 How to Get a Message from a Queue ............................. 68
  6.6 How to Add a Log Window ...................................... 70
Index ............................................................ 169

ITSO Redbook Evaluation ........................................ 171
<p>| Figures |
|------------------|-------------|
| 1. VisualAge Overview | 2 |
| 2. Enabling Technologies for VisualAge | 3 |
| 3. Public Interface of Parts | 5 |
| 4. VisualAge Notification Framework | 6 |
| 5. VisualAge Development Environment | 7 |
| 6. VisualAge Install - directories | 10 |
| 7. VisualAge Client Configuration (EMSRV) | 11 |
| 8. VisualAge Client Configuration (LAN) | 11 |
| 9. VisualAge for Smalltalk Pro - Icon View | 12 |
| 10. VisualAge Team Environment | 13 |
| 11. VisualAge System Transcript | 14 |
| 12. VisualAge Image Owner | 14 |
| 13. VisualAge Quick Start | 15 |
| 14. VisualAge Organizer | 16 |
| 15. MQSeries at Run Time | 17 |
| 16. Message Queuing: Principle | 18 |
| 17. RUNMQSC - Interactive | 22 |
| 18. RUNMQSC - Using Command File | 22 |
| 19. RUNMQSC - Input File | 22 |
| 20. RUNMQSC - Output File | 23 |
| 21. MQSeries Channels | 25 |
| 22. Client/Server Connection | 26 |
| 23. Definitions for Server Connection (vaqmgr.tst) | 27 |
| 24. Testing Client/Server Connection | 29 |
| 25. Listener Window (RUNMQLSR) | 29 |
| 26. Communication between Two Queue Managers | 30 |
| 27. Triggering Channels | 33 |
| 28. Triggering Applications | 35 |
| 29. Fragments of an MQSeries Program | 38 |
| 30. Logic of VisualAge MQSeries Parts | 44 |
| 31. First Application: Design | 48 |
| 32. First Application: Queues | 48 |
| 33. First Application: Message Structure | 49 |
| 34. First Application: New Application | 51 |
| 35. First Application: VisualAge Organizer Window | 52 |
| 36. First Application: Create New Part | 52 |
| 37. First Application: Composition Editor | 53 |
| 38. First Application: Push Button Settings | 54 |
| 40. First Application: ProcDialog Settings - Records | 55 |
| 41. First Application: Visual Parts Pop-Up Menu | 56 |</p>
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>First Application: Event to Action Connection</td>
<td>57</td>
</tr>
<tr>
<td>First Application: Composition Editor Window</td>
<td>58</td>
</tr>
<tr>
<td>Using AMQSPUT</td>
<td>59</td>
</tr>
<tr>
<td>First Application: GUI</td>
<td>59</td>
</tr>
<tr>
<td>Using AMQSGET</td>
<td>59</td>
</tr>
<tr>
<td>Second Application: Two Instances at Run Time</td>
<td>62</td>
</tr>
<tr>
<td>Second Application: User Interface</td>
<td>63</td>
</tr>
<tr>
<td>Second Application: MQ Series Connection Spec Settings</td>
<td>64</td>
</tr>
<tr>
<td>Second Application: Connect to Queue Manager</td>
<td>65</td>
</tr>
<tr>
<td>Second Application: Connect and Disconnect</td>
<td>67</td>
</tr>
<tr>
<td>Second Application: Complete GUI</td>
<td>69</td>
</tr>
<tr>
<td>Second Application: Method</td>
<td>71</td>
</tr>
<tr>
<td>Second Application: Start Connection From ...</td>
<td>72</td>
</tr>
<tr>
<td>Second Application: Connect Event ... to Method</td>
<td>73</td>
</tr>
<tr>
<td>Second Application: Constant Parameter Value Settings</td>
<td>73</td>
</tr>
<tr>
<td>Second Application: Start Connection from Combo Box</td>
<td>75</td>
</tr>
<tr>
<td>Second Application: Parameters for Combo Box</td>
<td>75</td>
</tr>
<tr>
<td>Second Application: Completed in Composition Editor</td>
<td>77</td>
</tr>
<tr>
<td>To-Do List: Message Flow</td>
<td>79</td>
</tr>
<tr>
<td>To-Do List: GUIs at Run Time</td>
<td>80</td>
</tr>
<tr>
<td>To-Do List: Client GUI</td>
<td>81</td>
</tr>
<tr>
<td>To-Do List: Connect to Queue Manager</td>
<td>84</td>
</tr>
<tr>
<td>To-Do List: Processing Reply Message</td>
<td>87</td>
</tr>
<tr>
<td>To-Do List: Add an Item</td>
<td>88</td>
</tr>
<tr>
<td>To-Do List: Completed Client View</td>
<td>89</td>
</tr>
<tr>
<td>To-Do List: Server Interface and Parts</td>
<td>90</td>
</tr>
<tr>
<td>To-Do List: Server Code</td>
<td>93</td>
</tr>
<tr>
<td>To-Do List: Complete Server View</td>
<td>94</td>
</tr>
<tr>
<td>Extended To-Do List: GUIs at Run Time</td>
<td>96</td>
</tr>
<tr>
<td>Extended To-Do List: Composition Editor - Client View</td>
<td>98</td>
</tr>
<tr>
<td>Getting a Particular Message: Client View</td>
<td>106</td>
</tr>
<tr>
<td>Getting a Particular Message: Server Program</td>
<td>107</td>
</tr>
<tr>
<td>Getting a Particular Message: Windows at Run Time</td>
<td>108</td>
</tr>
<tr>
<td>Syncpoint Processing Example</td>
<td>110</td>
</tr>
<tr>
<td>Hotel Reservation: Objects</td>
<td>114</td>
</tr>
<tr>
<td>Hotel Reservation: Public Interface Editor</td>
<td>115</td>
</tr>
<tr>
<td>Hotel Reservation: Base Form for MQHotelReqView</td>
<td>116</td>
</tr>
<tr>
<td>Hotel Reservation: MQHotelReqForm</td>
<td>117</td>
</tr>
<tr>
<td>Hotel Reservation: HotelRsvForm</td>
<td>118</td>
</tr>
<tr>
<td>Hotel Reservation: Non-Visual Server Part</td>
<td>119</td>
</tr>
<tr>
<td>Hotel Reservation: Server - Public Interface Editor</td>
<td>119</td>
</tr>
<tr>
<td>Hotel Reservation: MQHotelRsvServer</td>
<td>120</td>
</tr>
<tr>
<td>Hotel Reservation: Contents Method</td>
<td>121</td>
</tr>
<tr>
<td>Hotel Reservation: putRequest Method</td>
<td>122</td>
</tr>
</tbody>
</table>
# Tables

1. MQSeries Objects for MQM-to-MQM Connection ............................. 31
2. Start MQM-to-MQM Connection .................................................. 32
3. MQSeries Objects to Start Channels .......................................... 33
4. To-Do List: Parts ................................................................. 82
5. To-Do List: Part Names ......................................................... 82
6. Hotel Reservation: HotelReq Attributes .................................... 114
7. Hotel Reservation: HotelRsv Attributes .................................... 115
8. Hotel Reservation Application Versions ..................................... 127
9. Files for the Examples ............................................................ 157
10. Applications and Classes ....................................................... 157
Preface

This redbook helps you design and develop applications using the MQSeries Parts of VisualAge for Smalltalk Version 4.0. You can use VisualAge for Smalltalk to develop client/server applications that run under OS/2, AIX and Windows NT operating systems.

When selecting a tool for object-oriented application programming, VisualAge for Smalltalk has been chosen for the gains in development productivity, for its wealth of features, and for its portability among different operating systems.

MQSeries provides a powerful messaging system with an uncomplicated application programming interface for computers and networks from multiple vendors. MQSeries connectivity provides high integrity with assured message delivery and time independence.

This redbook presents several practical examples to demonstrate how to invoke MQSeries functions from programs written in VisualAge for Smalltalk. It is intended for VisualAge programmers who want to learn how to utilize MQSeries, and also for novices who are taking their first steps into the world of visual programming.

Some knowledge of MQSeries, VisualAge and Smalltalk is assumed.

The Team That Wrote This Redbook

This redbook was produced by a team of specialists from around the world working at the Systems Management and Networking ITSO Center, Raleigh.

Walter Fang currently is the IBM Asia Pacific VisualAge Market Manager, and a certified VisualAge Smalltalk developer. He has published several redbooks on VisualAge and Visual Model Technique (VMT), and has taught object technology workshops worldwide during his assignment at the International Technical Support Organization, San Jose Center, from 1995 to 1996. Before joining the ITSO, Walter worked in the Canadian Network Computing Solutions Center as a Senior I/T Architect, where he helped customers develop object-oriented client/server architectures and solutions.

Yuriko Sawatani is an Advisory I/T Specialist at the Asia Pacific Software Technical Support and Solution Technology, Yamato Laboratory. She has worked in Smalltalk since 1989 and taught many VisualAge and Object Technology workshops in AP. She holds a Master of Science degree from Tokyo Institute of Technology. Her areas of expertise include C++, Java
and Smalltalk, and she is a member of the ISO C++ working group. Her current interests are in applying object technology and framework solutions to application development projects.

**Tzue-Ing Shieh** is a Staff Applications Developer from the VisualAge for Smalltalk product team in Research Triangle Park, North Carolina. He has four years of experience in object-oriented programming using Smalltalk and Java focusing on Communication API, such as MQSeries, RPC, TCP/IP, and NetBIOS. He has worked at IBM for ten years. His areas of expertise include National Language Support, DBCS enabling, VSAM programming, and MVS/ESA system programming. Prior to joining IBM, he worked for Associate National Bank for two years where he gained extensive knowledge on banking software systems for revolving credit and credit card processing.

**Dieter Wackerow** is an Advisory ITSO Specialist for MQSeries in the Systems Management and Networking ITSO Center, Raleigh. His areas of expertise include application design and development for various industries, performance evaluations, capacity planning, and modelling of computer systems and networks. He also wrote a simulator for banking hardware and software. He teaches classes and has written on performance issues, application development for the banking industry, and about MQSeries.

Thanks to the following people for their invaluable contributions to this project:

Linda Robinson  
Shawn Walsh  
International Technical Support Organization, Raleigh Center

Dave Salkeld  
John De Binder  
VisualAge for Smalltalk Service Offerings, RTP, North Carolina

---

**Comments Welcome**

**Your comments are important to us!**

We want our redbooks to be as helpful as possible. Please send us your comments about this or other redbooks in one of the following ways:

- Fax the evaluation form found in “ITSO Redbook Evaluation” on page 171 to the fax number shown on the form.
- Use the electronic evaluation form found on the Redbooks Home Pages at the following URLs:
For Internet users http://www.redbooks.ibm.com
For IBM Intranet users http://w3.itso.ibm.com/redbooks

• Send us a note at the following address:
  redbook@vnet.ibm.com
Chapter 1. The VisualAge Visual Programming Tool

When selecting a tool for object-oriented application programming, VisualAge for Smalltalk has been chosen for the gains in development productivity, for its wealth in features, and for its portability among different operating systems, such as NT, WIN95, OS/2, AIX, HP/UX and Sun Solaris.

1.1 Product Overview

IBM VisualAge is addressing the most advanced methodologies in application development:

- Object orientation
- Visual programming
- Rapid application development
- Software construction from parts
- Reuse of legacy code and existing data

Extensions for “distributed Smalltalk,” for Smalltalk on server systems, and for integration with workflow control open new ways for true client/server programming and peer-to-peer technology. Client/server applications are developed rapidly, efficiently, and with high quality; simply select and connect prefabricated reusable software components on the screen. All the programming, for example, designing the graphical user interface and the appropriate program logic, is intuitive. VisualAge for Smalltalk offers the following components, as shown in Figure 1 on page 2.

The Composition Editor plays a central role in visual programming. It serves both as the workplace for GUI design, and for visual programming by visually constructing from parts.

The library of parts is a repository of reusable components. VisualAge includes prefabricated parts that support graphical user interfaces as well as generic parts for database queries, transactions, multimedia, and remote and local functions. It is easily expandable by developing new parts yourself, or by obtaining them from vendors.

VisualAge’s Communications subsystem supports most communications protocols and application programming interfaces. It enables you to: access remote applications across platforms, establish communications with remote program logic using visual connections, enrich legacy systems by adding workstation-based graphical user interfaces (GUls).
VisualAge for Smalltalk provides seamless support to access a broad range of relational databases.

Transaction management provides components for access to CICS and IMS based transaction systems.

VisualAge for Smalltalk does not come with build-in OO Analysis and Design tools, but it works nicely together with the VMT (Visual Modeling Technique) methodology, and with several tools provided by third-party vendors.

Distributed objects can send standard Smalltalk messages to one another, regardless of their physical location. They can also freely send other Smalltalk objects as arguments, and receive objects as results. The different parts of an application can be located on any computer in the network that is running with the Distributed feature. Using the Distributed feature, applications may be split in many ways, to support both client/server and true peer-to-peer design. Among others, distributed Smalltalk provides:

- Messaging: Communication logic is provided, down to the low-level task of passing Smalltalk objects across a network.
- Activation support: Any remote Smalltalk image can be started as required by the application.
Name server support: You can update object location information without having to change your Smalltalk code.

Distribution Toolkit: Tools are provided that can help you design, build, debug, optimize, and configure distributed applications from a single physical location.

VisualAge’s visual language support parts enable generic access to business-critical code that are available in C or COBOL dynamic-link libraries (DLLs). System Object Model (SOM) and Distributed System Object Model (DSOM) support in VisualAge allows you to reuse and subclass object classes developed in other languages and packaged as SOM classes and objects. In addition, support is provided for Dynamic Data Exchange (DDE) which allows two applications running on the same machine to exchange data dynamically. VisualAge provides advanced and comprehensive support for team programming, with concurrent access to a central library of classes, parts, and subsystems in a networked development environment. In addition, VisualAge provides support for tracking code, application and configuration management, and version and release control.
Some visual programming tools on market are not fully object-oriented but are object-based or class-based only. Truly object-oriented visual programming tools, as VisualAge for Smalltalk, are completely based on object technology, make and allow full use of inheritance, and integrate with a complete object-oriented application development environment. See Figure 2 on page 3 for the enabling technologies for VisualAge. All components at every stage of development are objects. Thus, there is a single view from the problem domain to the implementation.

Because they provide options for a repository and for team programming support, VisualAge for Smalltalk and IBM Smalltalk suit the end user in a single-user environment as well as a team of developers on large client/server platforms.

1.2 Scripting Language

The ability to switch smoothly between visual programming techniques and scripting language is an important advance. Visual programming has some limitations and needs a language environment to complete a job.

The scripting language of VisualAge is IBM Smalltalk, a robust implementation of the object-oriented Smalltalk language. Following industry standards ANSI X3J20, POSIX, X-Windows and OSF/Motif, Smalltalk and VisualAge applications can easily be ported to different platforms. IBM Smalltalk is fully contained within the VisualAge product.

1.3 Parts and Public Interfaces

VisualAge uses visual “construction from parts” techniques. A part is a software object with a standardized set of interfaces that allow it to be connected to other parts to build an application. A part can be primitive or a composite, that is, composed of other parts. Parts can further be grouped into visual and nonvisual.

Visual parts can present a graphical view to the end user at run time, and are used to compose the graphical user interface (GUI). A visual part may also contain nonvisual parts.

Nonvisual parts do not have a run-time view. Non-visual parts are used to implement business logic objects or generic behavior, or to wrap database queries or generic DLLs. They do not have a run-time view, and are—from a modeling point of view—regarded more stable than GUI parts.

The VisualAge public interface exposes three clearly defined features:
• Attributes, which are the logical properties of a part. Attributes are objects which a part can return or set on request.
• Actions, which are the behaviors of the part the services the part may provide to others.
• Events, which provide a notification mechanism, signaling that something has happened to the part. Events are most often used to trigger some action.

The public interface of parts refers to the features used to connect parts, as shown in Figure 3.

1.4 Notification Framework

IBM VisualAge provides a notification framework which consists of notifiers and observers. A notifier maintains a list of objects that depend on the occurrence of specific events. To register itself to a notifier, an object adds an observer to the notifier’s list. An object can then signal the occurrence of
each of its events to the notifier which in turn broadcasts the signal to its list of dependent observers.

Figure 4 shows how this notification framework is used as the controller for the model and views in a VisualAge environment. In visual programming with VisualAge, the notification framework is implicitly used with the connection from event to action, event to attribute, attribute to attribute, and attribute to action.

![Figure 4. VisualAge Notification Framework](image)

### 1.5 Packaging and Distribution

In VisualAge for Smalltalk, an application defines a set of parts that work together to fulfill a common task. An application can be managed as a whole. One or more applications may be packaged together as a Smalltalk image to produce the run-time application for distribution to the end users. This Smalltalk image together with a set of run-time DLLs may become the unit of execution of a workflow activity.
Chapter 2. Installation and Configuration

This chapter provides a brief set of guidelines on how to install VisualAge for Smalltalk and how to configure your workstations.

VisualAge for Smalltalk provides you with the flexibility to configure your working environment either as a stand-alone visual programming workstation for a single programmer or as a multi-developer environment where many client workstations share one central manager library.

![Figure 5. VisualAge Development Environment](image)

You can choose from the following list which products of VisualAge for Smalltalk Pro you want to install:
We recommend you install a stand-alone product for simplicity if you have about 200 MB free disk space in your workstation. If not, you can install the Manager Library product on one workstation, then install the client product on your own workstation and connect it with the Manager Library. If your team members want to share code with you, they can install the client product on their own workstations then connect with the same Manager Library as yours.

**Note:** In addition to the client or stand-alone version, you must install on your workstation any other products you may use. Since this book uses the MQSeries parts, you must install the Communications/Transactions product, too.

In the next sections, we guide you step-by-step to install VisualAge for Smalltalk products for whatever configuration you choose. For further instructions, please install the VisualAge for Smalltalk Documentation product and Acrobat, then reference the Installation Guide.

### 2.1 Installing VisualAge for Smalltalk Stand-Alone

**Step 1.** Put the VisualAge for Smalltalk product CD in your CD-ROM drive.

**Step 2.** Make the CD-ROM drive your current drive and change directory to \\OS2\INSTALL.

**Step 3.** To start the installation process type `install`.

**Step 4.** On the Welcome screen click on **Install**.

**Step 5.** On Installation screen select **VisualAge for Smalltalk Pro - Stand Alone** and click on **Install**.

**Step 6.** When the License Agreement window appears, click **Yes**.

**Step 7.** In the Install window mark Update CONFIG.SYS and click on **OK**.
Step 8. In the Install - directories window (Figure 6 on page 10):
- Select the components you want to install.
  The VisualAge Stand Alone Base is sufficient when you want to try out the product.
- If you wish, accept the default directory name \VAST.
- Click on Install....

Step 9. A window informs you about the progress of the installation. This process takes several minutes.

Step 10. On the IBM Software Registration screen click Cancel.

Step 11. Click OK in the window that tells you that the installation is completed.

Step 12. Re-boot your system or install more products and re-boot after that.

Note: At this time you may want to save your image, just in case it gets corrupted later. Change the directory to \VAST and execute this command:

```
COPY abt.icx pure.icx
```
2.3 Installing VisualAge for Smalltalk Client

Note: This only works if your Manager Library is installed on OS/2 Warp 3.0 (or higher) or UNIX systems supported by VisualAge for Smalltalk.

Step 1. Put the VisualAge for Smalltalk product CD in your CD-ROM drive.

Step 2. Make the CD-ROM drive your current drive and change the directory to \OS2\INSTALL.

Step 3. To start the installation process type install.

Step 4. On the Welcome screen click on Install.

Step 5. On the Installation screen select VisualAge for Smalltalk Pro - Client and click on Install.

Step 6. When the License Agreement window appears, click Yes.

Step 7. In the Install window mark Update CONFIG.SYS and click on OK.

Step 8. In the Install - directories window:
   - Select the components you want to install.
     The VisualAge Client Base is sufficient in most cases.
Step 9. A window informs you about the progress of the installation. This process takes several minutes.

Step 10. In the Client Configuration window you specify how to connect to the VisualAge server. You have two options:

1. Using TCP/IP (see Figure 7):
   - Select the connection type **EMSRV** (Envy Manager Server).
   - Select OS/2 as operating system.
   - Enter the path and name of the manager library in the server.
     **Note:** Use the *real* drive letter of the server disk.
   - Enter the IP address of the Visual Age server.
   - Click on **OK**.

2. Using LAN Server (see Figure 8):
   - Select the connection type **File I/O**.
   - Enter the path and name of the manager library in the server.
     **Note:** Use the drive letter you see when you type `net use`.

---

*Figure 7. VisualAge Client Configuration (EMSRV)*

*Figure 8. VisualAge Client Configuration (LAN)*
- Click on OK.

Step 11. On the IBM Software Registration screen click Cancel.

Step 12. Click on OK in the window that tells you that the installation is completed.

Step 13. Re-boot your system. After the installation you see the icons in Figure 9.

![VisualAge for Smalltalk Pro - Icon View](image)

Figure 9. VisualAge for Smalltalk Pro - Icon View

### 2.4 Installing the Communication/Transaction Product

Step 1. Put the VisualAge for Smalltalk product CD in your CD-ROM drive.

Step 2. Make the CD-ROM drive your current drive and change directory to \OS2\INSTALL.

Step 3. To start the installation process type install.

Step 4. On the Welcome screen click on Install.

Step 5. On the Installation screen select Communication/Transaction and click on Install.

Step 6. When the License Agreement window appears, click Yes.

Step 7. In the Install window click on OK. The CONFIG.SYS remains unchanged.

Step 8. In the Install - directories window select Communications/Transactions and click on Install.

You have no other choice but to install the product into your VisualAge Pro directory, here D:\VAST.

Step 9. Click on OK when the completion message appears.

Step 10. In the Installation window click Cancel.

Step 11. In the Welcome window click Exit.

Step 12. If you install the VisualAge for Smalltalk Client product, you need to copy all the *.dat files under your installed directory, C:\VAST\IMPORT, to your Manager Library \IMPORT directory.
after you install all the needed VisualAge for Smalltalk products. If you are using the Stand-alone product, skip this step.

2.5 Using VisualAge for Smalltalk for the First Time

After all required products have been installed, you have to select the features you need for the image you want to develop. The features are in the manager library. The library can be in a server (if you are a client) or in your stand-alone system.

When you invoke the product for the first time, follow these steps to load features into your image:

Step 1. You will see two windows:

a. The System Transcript window in Figure 11 on page 14 is always there. It contains a log of your activities.

b. In the Selection Required window in Figure 12 on page 14 select the owner of the image.

If you get a walkback or error message saying that Emlibrary is not found, make sure you net use to the Manager Library disk if you choose File I/O to connect to it. If you select EMSRV, then make sure your emsrv.exe is active on your manager library.
Step 2. If you installed the stand-alone version, you will be the library supervisor. Select Library Supervisor and click on OK.

Step 3. Now you must connect the image to the current library. Since this may take some time, the installation program wants you to confirm this step. Click on OK.

Step 4. After the image is connected to the library (here D:\VAST\manager.dat) a window is displayed. Click on OK.
Step 5. Next you see the VisualAge Quick Start window shown in Figure 13.
   - Select Go to the VisualAge Organizer.
   - Eliminate the check mark from Show this window at startup.
   - Click on OK.

Step 6. Next you see the Organizer window shown in Figure 14 on page 16.
   Before you can develop your first application, you have to load VisualAge features into your image.

Step 7. In the System Transcript window, click on Options and select Load/unload features... from the menu.

Step 8. From the list of available features select:
Figure 14. VisualAge Organizer

- VisualAge: Communications, MQ (includes TCP/IP)
- Visual Age: VisualAge Base
- Visual Age: Language Interface C
- Visual Age: Notebook Style Settings Views V4.0

Move the selected one to the right side of the window by clicking the >> push button. It takes a while to load those features.

Step 9. Then click on OK. The load process takes several minutes.

Step 10. After the features are loaded, save your image.

2.6 Create and Change Users

When you first bring up the VisualAge image, the default user is Library Supervisor. If you are in a multi-developer environment, your VA administrator should have created your user name for you already. If not, you will need to use the default Library Supervisor to create your own user name. In the stand-alone product, you can use Library Supervisor to create your new user name and then change your user name to it.

In the System Transcript screen, select Tools, System, then Users to get to the menu for Creating Users. For more information, please refer to the IBM Smalltalk User’s Guide, Chapter 30, “Library Operations.”
Chapter 3. MQSeries Overview

MQSeries is IBM’s award-winning middleware for commercial messaging and queuing. It runs on a variety of platforms. The MQSeries products enable programs to communicate with each other across a network of unlike components, such as processors, subsystems, operating systems and communication protocols. MQSeries programs use a consistent application program interface (API) across all platforms.

Figure 15. MQSeries at Run Time

Figure 15 shows the parts of an MQSeries application at run time. Programs use MQSeries API calls, that is the Message Queue Interface (MQI), to communicate with a queue manager (MQM), the run-time program of MQSeries. For the queue manager to do its work, it refers to objects, such as queues and channels. The queue manager itself is an object as well.

The following sections provide a brief overview of MQSeries.

Note: For the examples discussed in this book, we used MQSeries for OS/2 and MQSeries for Windows NT.
3.1 What Is Messaging and Queuing?

Message queuing is a method of program-to-program communication. Programs within an application communicate by writing and retrieving application-specific data (messages) to/from queues, without having a private, dedicated, logical connection to link them.

*Messaging* means that programs communicate with each other by sending data in messages and not by calling each other directly.

*Queuing* means that programs communicate through queues. Programs communicating through queues need not be executed concurrently.

With asynchronous messaging, the sending program proceeds with its own processing without waiting for a reply to its message. In contrast, synchronous messaging waits for the reply before it resumes processing.

![Figure 16. Message Queuing: Principle](image)

MQSeries is used in a client/server or distributed environment. Programs belonging to an application can run in one workstation or in different machines on different platforms.

3.1.1 Messages

A message consists of two parts: data that is sent from one program to another and a message descriptor. The message descriptor identifies the message (message ID) and contains control information, also called attributes, such as message type, expiry time, correlation ID, priority, and the name of the queue for the reply.

MQSeries knows four types of messages:
Datagram A message containing information for which no response is expected.

Request A message for which a reply is requested.

Reply A reply to a request message.

Report A message that describes an event such as the occurrence of an error.

Note: VisualAge for Smalltalk uses request and reply messages.

There are persistent and non-persistent messages. Persistent messages are written to logs on a hard drive and survive system failures. Non-persistent messages cannot be recovered after a system restart.

3.1.2 Queue Manager

The heart of MQSeries is its run-time program, the queue manager (MQM). Its job is to manage queues of messages. Application programs invoke functions of the queue manager by issuing API calls. For example, the MQPUT API puts a message on a queue to be read by another program using the MQGET API. This scenario is shown in Figure 16 on page 18.

A program may send messages to another program that runs in the same machine as the queue manager, or to a program that runs in a remote system, such as a server or a host. The remote system has its own queue manager with its own queues.

Application programmers do not need to know where the program runs they send messages to. They put their message on a queue and let the queue manager worry about the destination machine and how to get it there.

For the queue manager to do its work, it refers to objects that are defined by an administrator, usually when the queue manager is created or when a new application is added. The objects are described in the next section.

The functions of a queue manager can be defined as follows:

- It manages queues of messages for application programs.
- It provides an application programming interface, the Message Queue Interface (MQI).

Note: The Networking Blueprint identifies three communication styles:

- Common Programming Interface - Communications (CPI-C)
- Remote Procedure Call (RPC)
- Message Queue Interface (MQI)
• It uses networking facilities to transfer messages to another queue manager when necessary.

• It provides additional functions that allow administrators to create and delete queues, alter the properties of existing queues, and control the operation of the queue manager. These functions are invoked through the utility RUNMQSC, which stands for run MQSeries commands.

3.1.3 Queue Manager Objects

The queue manager itself is an object. Usually, an administrator creates it with the command crtmqm, either from the command line or from an icon. You can create several queue managers in one system. One of them should be the default queue manager. The following command creates the default queue manager VAQMGR:

crtmqm /q VAQMGR

The /q makes it the default MQM. The name is case-sensitive. To start the default queue manager issue the command:

strmqm

Before the queue manager can do any messaging and queueing the administrator has to define objects, such as queues. There are some default definitions for objects every queue manager needs. They are defined in a file provided with MQSeries. To define these default objects use the utility RUNMQSC, also provided with the product. The command to create these objects is:

runmqsc < c:\mqm\mqsc\amqscoma.tst > out.lst

The queue manager must be running to create the objects defined in the file amqscoma.tst. Check the last lines of the output file, here out.lst, for any errors.

The queue manager can own objects of the following types:

• Queues
• Process definitions
• Channels

The objects are common across different MQSeries platforms. There are other objects that apply to MVS systems only, such as the buffer pool, PSID, and the storage class.

3.1.3.1 Queues

Message queues are used to store messages sent by a program. There are local queues that are owned by the local queue manager, and remote...
queues that belong to a different queue manager. Queues are described in more detail in 3.3, “Message Queues” on page 23.

### 3.1.3.2 Channels

A channel is a logical communication link. In MQSeries, there are two different kinds of channels:

- **Message channel:**
  
  A message channel connects two queue managers via message channel agents (MCA). Such a channel is unidirectional. It comprises two message channel agents, a sender and a receiver, and a communication protocol. An MCA is a program that transfers messages from a transmission queue to a communication link or vice versa. For bidirectional messaging you have to define two channels, a sender channel and a receiver channel.

- **MQI channel:**
  
  A Message Queue Interface (MQI) channel connects an MQI client to a queue manager in a server machine. MQI clients don’t have a queue manager of their own. An MQI channel is bidirectional.

Figure 21 on page 25 shows the use of both channel types. For more detailed information refer to the *Distributed Queuing Guide*.

**Channels for Testing Your VisualAge Programs:** If your workstation has a queue manager installed, you don’t need to create a channel unless your VisualAge program communicates with a remote machine. If you don’t have a queue manager installed, you need an MQI channel. You also have to change the VisualAge default platform library MQM to MQI. Refer to section 5.4, “How to Set Up VisualAge for Smalltalk” on page 50 for more information.

### 3.1.3.3 Process Definitions

A process definition object defines an application to a queue manager. For example, it contains the name of the program to trigger when a message arrives.

### 3.2 Manipulating MQM Objects

MQSeries provides the utility RUNMQSC to create and delete queue manager objects and to manipulate them. The queue manager must be running when you use the utility. RUNMQSC works in two ways:

- You can type the commands.
- You can create a file containing a list of commands and use this list as input.

The commands in Figure 17 start the default queue manager and create a local queue for it.

```bash
[C:\]strmqm
MQSeries queue manager running.
[C:\]runmqsc
33H2205,5622-908 (C) Copyright IBM Corp. 1994,1995. ALL RIGHTS RESERVED.
Starting MQSeries Commands.
define qlocal('REQUEST') replace descr ('request queue')
 1 : define qlocal('REQUEST') replace descr ('request queue')
AMQ8006: MQSeries queue created.
Ctrl + C  --- ends RUNMQSC
1 MQSC commands read.
0 commands have a syntax error.
0 commands cannot be processed.

Figure 17. RUNMQSC - Interactive

Another way to create MQSeries objects is by using an input file instead of typing the commands.

```bash
[C:\]strmqm
MQSeries queue manager running.
[C:\]runmqsc < mycoma.tst > a.a
```  

Figure 18. RUNMQSC - Using Command File

The input file contains the following lines. The + indicates that the command continues on the next line.

```
********************************************************************
* File: MYCOMA.TST *
********************************************************************

DEFINE QLOCAL('REQUEST') REPLACE +
  DESCRIPTOR('request queue')
```

Figure 19. RUNMQSC - Input File
3.3 Message Queues

Queues are defined as objects belonging to a queue manager. MQSeries knows a number of different queue types, each with a specific purpose. Some of them are described in the following sections.

3.3.1 Local Queue

A queue is local if it is owned by the queue manager to which the application program is connected. They are used to store messages for programs that use the same queue manager. For example, each program A and program B has a queue for incoming messages and another queue for outgoing messages. Since the queue manager serves both programs, all four queues are local.

**Note:** Both programs do not have to run in the same workstation. Client workstations usually use a queue manager in a server machine.

3.3.2 Remote Queue

A queue is remote if it is owned by a different queue manager. It is the local definition of a remote queue. Remote queues are associated with a transmission queue.
Applications do not need to know the location of the remote queue. Programs write messages to queues. The local queue manager is responsible for forwarding the messages to the remote queue manager.

Note: A program cannot read messages from a remote queue.

3.3.3 Transmission Queue
A remote queue is associated with a transmission queue. Transmission queues are used as an intermediate step when sending messages to remote queues.

Typically, there is only one transmission queue for each remote queue manager. All messages written to queues owned by a remote queue manager are actually written to the transmission queue for this remote queue manager. The messages will then be read from the transmission queue and sent to the remote queue manager.

Transmission queues are transparent to the application. They are used internally by the queue manager.

Note: When a program opens a remote queue, the attributes of the queue are obtained from the transmission queue. Therefore, the results of a program writing messages to a queue will be affected by the transmission queue characteristics.

3.3.4 Alias Queue
Alias queues are not real queues but definitions. They are used to assign different names to the same physical queue. This allows multiple programs to work with the same queue, accessing it under different names and with different definitions.

3.3.5 Initiation Queue
An initiation queue is a local queue to which the queue manager writes a trigger message when certain conditions are met on another local queue, for example, when a message is put into an empty message queue. Trigger messages are read by the trigger monitor, an MQSeries application. The trigger monitor then starts the application that will process the message.

Note: Applications do not need to be aware of initiation queues, but the triggering mechanism implemented through them is a powerful tool to design and write asynchronous applications.

3.3.6 Reply-To Queue
A request message must contain the name of the queue into that the responding program must to put the reply message.
3.3.7 Dead-Letter Queue

A queue manager must be able to handle situations when it cannot deliver a message. Here are some examples:

- The destination queue is full.
- The destination queue does not exist.
- Message puts have been inhibited on the destination queue.
- The sender is not authorized to use the destination queue.
- The message is too large.
- The message contains a duplicate message sequence number.

These messages are written by the queue manager to a dead-letter queue. A dead-letter queue is defined when the queue manager is created. It will be used as a repository for all messages that cannot be delivered.

3.4 Clients and Servers

Before you install MQSeries you have to decide if the workstation will become an MQ client or an MQ server. In MQSeries, a server has a queue manager installed; a client does not. A server can also be used as a client.

When a client cannot connect to its server it cannot work, because queue manager and queues for a client reside in the server. Several MQSeries clients share MQSeries objects in the server they are attached to. The queue manager is one of them.

In some cases, it may be advantageous to have queues in the end user’s workstation, especially in a mobile environment. That allows you to run your application when a connection between client and server does (temporarily) not exist.

The difference between an end user’s workstation that is a client and one that has a queue manager is the way messages are sent.

Figure 21. MQSeries Channels
Figure 21 shows the use of MQI and message channels.

- MQI channels connect clients to a queue manager in a server machine.
- A message channel connects a queue manager to another queue manager in another system.

**Note:** MQI channels are faster than message channels.

The following sections describe what you have to do to define and test the connection between an MQ client and an MQ server. A more detailed description is in the publication *MQSeries Clients*.

**Note:** The VisualAge for Smalltalk MQSeries parts support both client and server. Define your platform library (MQM or MQIC) accordingly.

### 3.5 Communication between Client and Server

This section provides information on how to set up and use the message queue interface (MQI) between MQSeries clients and an MQSeries server.

**Note:** If you use this interface, link your programs with the MQIC library.
3.5.1 How to Define a Client/Server Connection

The following describes MQM objects and other definitions needed for the VisualAge for Smalltalk examples developed throughout this publication.

To define the connection, you have to know the transmission protocol and the addresses of the systems. We use TCP/IP.

3.5.1.1 On the Server

Define the queues that the application needs and a channel of the type server connection. The queue manager definitions are in the file vaqmgr.tst, shown in Figure 23.

```
DEFINE QLOCAL(′REQUEST′) REPLACE +
    DESCR(′request queue′)

DEFINE QLOCAL(′REPLY′) REPLACE +
    DESCR(′reply queue′)

DEFINE CHANNEL(′VACH1′) CHLTYPE(SVRCONN) REPLACE +
    TRPTYPE(TCP) MCAUSER(′′)
```

Figure 23. Definitions for Server Connection (vaqmgr.tst)

We define a queue for the application to put messages in and an MQI channel of the type server connection. Create the objects by issuing the command:

```
runmqsc < vaqmgr.tst > a.a
```

3.5.1.2 On the client

Define an environment variable for the MQSeries client that defines the connection on the client side. Set the variable with the following command or place the command in the CONFIG.SYS.

```
set MQSERVER=VACH1/TCP/9.24.104.206(1414)
```

Notes:

1. MQSERVER is the name of the environment variable.
2. VACH1 is the name of the channel to be used for communication between client and server. The channel must be defined in the server.
3. TCP denotes that TCP/IP is to be used to connect to the machine with the address following the parameter.
4. (1414) is the default port number for MQSeries. You may omit this parameter if the listener on the server side uses this default, too.

3.5.2 How to Start a Client/Server Connection

Before you can start the application in the client you have to start in the server a program that listens to the communication link between client and server. MQSeries provides a program that does just that. You start the listener with the following command:

```
start runmqlsr /t tcp /m VAQMGR /p 1414
```

Notes:

1. start creates a new window for the listener.
2. runmqlsr is the name of the listener.
3. /t tcp defines that there is a TCP/IP connection between client and server.
4. /m VAQmgr specifies the name of the queue manager the client connects to. If omitted, the default queue manager is used.
5. /p 1414 defines the TCP/IP port number. 1414 is the default assigned to MQSeries applications hence, you may omit this parameter.

The server is now ready to process MQI calls from the application running in the client.

3.5.3 How to Test a Client/Server Connection

With the steps described above communication between client and server is established. You can test the connection using programs provided with MQSeries. They are in the directory:

```
c:\mqm\tools\c\samples\bin
```

• AMQSPUTC puts messages on a queue.
• AMQSGETC gets messages from a queue.

On the client machine, type the commands shown in bold in Figure 24 on page 29.

• After AMQSPUTC is started type a few messages and then press Enter twice to end it.
• AMQSGETC times out after a few seconds.

On the server machine, you see the listener window shown in Figure 25 on page 29 after the two programs have completed.
3.6 Communication between Queue Managers

In this section, we discuss what you have to define to send messages to a queue manager that resides in another system. We use message channels for communication between queue managers (see Figure 21 on page 25).

Each machine has a queue manager installed and each queue manager manages several local queues. Message destined for a remote queue manager are put into a *remote queue*. A remote queue is not a real queue; it is the definition of a local queue in the remote machine. A remote queue is associated with a transmission (xmit) queue which is a local queue. Usually, there is one xmit queue for each remote queue manager.
A transmission queue is associated with a message channel. Message channels are uni directional, meaning that you have to define two channels for a conversational type of communication. Also, you have to define each channel twice, once in the system that sends the message (sender channel) and once in the system that receives the message (receiver channel). Each channel pair (sender and receiver) must have the same name.

3.6.1 How to Define a Connection between Two Systems

Figure 26. Communication between Two Queue Managers

Figure 26 shows the required MQSeries objects for connecting two queue managers. In each system we need:

- A remote queue definition that links to a transmission queue
- A transmission queue that holds all messages destined for the other system until the channel transmits them
A sender channel that gets messages from the xmit queue and transmit them to the other system
A receiver channel that receives messages and put them into a local queue
A local queue from which the program gets its messages.

In each system, you must define the appropriate queue manager objects.

The two .tst files are below:

| Table 1. MQSeries Objects for MQM-to-MQM Connection |
|---------------------------------|---------------------------------|
| **Sender (System A)** | **Receiver (System B)** |
| DEFINE QREMOTE('Queue1') REPLACE + | DEFINE QLOCAL('Queue1') REPLACE + |
| RNAME('Queue1') + | DESCR('Messages from system A') |
| RQMNAME(SYSTEMB) + | |
| XMITQ(A.TO.B) + | |
| DESCR('Queue 1 in system B') | |
| DEFINE QLOCAL(A.TO.B) REPLACE + | DEFINE QLOCAL(A.TO.A) REPLACE + |
| USAGE(xmitq) + | |
| DESCR('Xmit Queue') | |
| DEFINE CHANNEL(A.TO.B) + | DEFINE CHANNEL(A.TO.A) + |
| CHLTYPE(sdr) REPLACE + | |
| TRPTYPE(tcp) CONNAME(9.24.104.116) + | |
| XMITQ(A.TO.B) + | |
| DESCR('Sender channel from A to B') | |
| DEFINE QLOCAL('Queue2') REPLACE + | DEFINE QLOCAL('Queue2') REPLACE + |
| DESCR('Messages from system B') | |
| DEFINE QREMOTE('Queue2') REPLACE + | DEFINE QREMOTE('Queue2') REPLACE + |
| RNAME('Queue2') + | RNAME('Queue2') + |
| RQMNAME(SystemA) + | RQMNAME(SystemA) + |
| XMITQ(B.TO.A) + | XMITQ(B.TO.A) + |
| DESCR('Queue 2 in system A') | DESCR('Queue 2 in system A') |
| DEFINE QLOCAL(B.TO.A) REPLACE + | DEFINE QLOCAL(B.TO.A) REPLACE + |
| USAGE(xmitq) + | |
| DESCR('Xmit queue') | |
| DEFINE CHANNEL(B.TO.A) + | DEFINE CHANNEL(B.TO.A) + |
| CHLTYPE(rcvr) REPLACE + | |
| TRPTYPE(tcp) + | |
| DESCR('Receiver channel from B to A') | |
| File Name: sytema.tst | File Name: systemb.tst |
3.6.2 How to Start Communication

First, the objects have to be known to the queue managers. Use RUNMQSC to create the objects. Make sure that the queue manager is running.

Next, start the listeners and the channels. You need to start only the sender channel in each system. MQSeries starts the receiver.

Table 2 shows the commands to create the objects and to start communication.

<table>
<thead>
<tr>
<th>Table 2. Start MQM-to-MQM Connection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sender (System A)</td>
</tr>
<tr>
<td>-------------------</td>
</tr>
<tr>
<td>Do the following only once.</td>
</tr>
<tr>
<td>strmqm SYSTEMA</td>
</tr>
<tr>
<td>runmqsc &lt; systema.tst &gt; a.a</td>
</tr>
<tr>
<td>Do this every time when you establish connection.</td>
</tr>
<tr>
<td>start runmqlsr -t tcp</td>
</tr>
<tr>
<td>runmqsc</td>
</tr>
<tr>
<td>start channel(A.TO.B)</td>
</tr>
<tr>
<td>Ctrl+C</td>
</tr>
<tr>
<td>Now start the applications in both systems.</td>
</tr>
</tbody>
</table>

In this scenario, you have to start the channels manually, and the application, too. In the next section, we show how to start channels automatically.

3.6.3 How to Start Channels

You can use the channel initiator to start channels. Instead of the commands shown in Table 2 enter the following commands in both systems:

```
start runmqlsr -t tcp
start runmqchi
```

With the first command you start the listener and with the second the channel initiator program.

You have to define a process that tells the channel initiator what channel, that is, message channel agent to start. Figure 27 on page 33 shows that the queue manager can trigger the process that starts the channel program in three ways:
When the first message is put into the transmission queue
Every time a message is put into the xmit queue
When the queue contains n messages

Figure 27. Triggering Channels

In order to trigger the channels you have to modify the .tst files shown in Table 1 on page 31. The table below shows the changes in bold:

<table>
<thead>
<tr>
<th>Table 3. MQSeries Objects to Start Channels</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sender (System A)</strong></td>
</tr>
<tr>
<td>DEFINE QLOCAL(A.TO.B) REPLACE +</td>
</tr>
<tr>
<td>USAGE(xmitq) +</td>
</tr>
<tr>
<td>TRIGGER TRIGTYPE(every) +</td>
</tr>
<tr>
<td>INITQ(System.CHANNEL.INITQ) +</td>
</tr>
<tr>
<td>PROCESS(A.TO.B) +</td>
</tr>
<tr>
<td>DESCR(‘Xmit Queue’)</td>
</tr>
<tr>
<td>DEFINE PROCESS(A.TO.B) +</td>
</tr>
<tr>
<td>USERDATA(A.TO.B) +</td>
</tr>
<tr>
<td>DESCR(‘Process to start channel’)</td>
</tr>
</tbody>
</table>

Since messages are placed into the xmit queue sporadically, we trigger the channel every time. We use the default initiation queue for the trigger message. The parameter USERDATA in the process defines the name of the channel to start every time a message is put into the xmit queue.

MQSeries also provides a way to trigger application programs automatically. The next section describes how to do that.
3.7 How to Trigger Applications

This section describes how to trigger an application program that runs in the server machine. Both triggering and triggered applications can run in the same machine or in different machines. We base the following explanations on an MQSeries client/server connection, where the client program runs in the client’s machine but its MQSeries objects are in the server.

Since there are MQI channels of the type server connection between clients and server (Figure 22 on page 26), all clients use the queue manager in the server machine. When a client puts a message on a queue it has to be read and processed by some program. This program can be started when the server starts, or the queue manager starts it when it is needed, that is, using the MQSeries triggering mechanism.

Figure 28 on page 35 shows two clients connected to a server. Both clients request services from the same program (Appl S1). Since that application runs in the same system as the queue manager, we have only local queues. Some queues are specifically for a particular client, for example, QA1 is the reply queue for client A and QB1 is the reply queue for client B. Other queues are used by the client and by the server, for example, QS1 is used as output queue for the clients and as input queue for the server program.

The next sections describe the MQSeries objects and API sequences in both client and server.

3.7.1 How a Client Sends a Request and Receives a Reply

The client starts a program that puts a message on a queue. For this function, five MQSeries API calls are executed:

- MQCONN connects to the queue manager (VAQMGR) in the server.
- MQOPEN opens the output message queue (QS1).
- MQPUT puts the message on the queue.
- MQCLOSE closes the queue (QS1).
- MQDISC disconnects from the queue manager (VAQMGR).

Of course, the program can put many messages in the queue before it closes it and disconnects. Closing the queue(s) and disconnecting from the queue manager could be done when the application ends.

The MQSeries client code that runs in the client machine processes the API calls and routes them to the machine defined in the environment variable.
MQSeries uses an MQI channel for such a connection, and it is defined as in the example below:

```bash
set MQSERVER=VACH1/TCP/9.24.1104.206
```

The environment variable MQSERVER contains three pieces of information:

- The name of the MQI channel type server connection that is defined in the server machine (VACH1)
- The transmission protocol (TCP/IP)
- The IP address or the host name of the server
After the server program S1 has processed the request, it puts a message into the reply queue for the client. Since several clients use the same server application, it is advisable to tell the client the name of that queue. Use reply-to-queue in the message header.

### 3.7.2 How a Server Triggers an Application

In the server machine, the following queue manager objects are needed:

1. A channel, VACH1, of the type server connection
2. A local queue, QS1, into which the clients put their request messages
3. An initiation queue into which the queue manager puts a trigger message when a message is put into QS1
4. A process definition, process.appl1, that contains the name of the program (S1) to be started when the trigger event occurs
5. Local queues into which the program puts the reply messages for the clients

You define the objects as follows:

```plaintext
DEFINE CHANNEL('VACH1') CHLTYPE(SVRCONN) REPLACE + TRPTYPE(TCP) MCAUSER(' ')

DEFINE QLOCAL('QS1') REPLACE +
    DESCR('Request queue') + TRIGTYPE(EVERY) + TRIGGER INITQ (system.default.initiation.queue) + PROCESS (process.appl1)

DEFINE PROCESS(process.appl1) REPLACE +
    DESCR('Process for server program') + APPLTYPE(OS2) + APPLICID('c:\mqtest\S1.exe')

DEFINE QLOCAL('QA1') REPLACE +
    DESCR('Reply queue client A')

DEFINE QLOCAL('QB1') REPLACE +
    DESCR('Reply queue client B')
```

In the server machine, two programs have to be started: the trigger monitor and the listener. Use the following commands:

```
start runmqtrm
start runmqslsr /t tcp
```
Note: On AIX, process amqcrsta is started via inetd for the listener process.

The listener listens for messages on the channel and puts them on the queue QS1. By default, the MQM puts a trigger message on the trigger queue each time a message is put on QS1. When a message is placed on the trigger queue, the trigger monitor starts the program defined in the process.

3.8 Message Queuing Interface (MQI)

A program talks directly to its local queue manager. It resides in the same processor or domain (for clients) as the program itself. The program uses the Message Queuing Interface (MQI). The MQI is a set of API calls that request services from the queue manager.

There are eleven APIs. The most important ones are:

- MQPUT: Put a message on a queue.
- MQGET: Get a message from a queue.

The other calls are used less frequently:

- MQCONN: Establish connection with a queue manager.
- MQOPEN: Open or obtain access to a queue.
- MQCLOSE: Close a queue.
- MQDISC: Disconnect from the queue manager.
- MQPUT1: Open a queue, put a message on it and close the queue.
- MQINQ: Request information about one of the queue manager’s objects.
- MQSET: Change the attributes of a queue.
- MQCMIT: A syncpoint has been reached. Messages put as part of a unit of work are made available to other applications. Messages retrieved as part of a unit of work are deleted.
- MQBACK: The queue manager has to back out all message puts and gets that have occurred since the last syncpoint. Messages put as part of a unit of work are deleted. Messages retrieved as part of a unit of work are reinstated on the queue.
Example:

The code fragment in Figure 29 shows the APIs to put a message on a queue and get the reply from another queue.

```c
MQHCONN HCon;  // Connection handle
MQHOBJ HObj1;   // Object handle for queue 1
MQHOBJ HObj2;   // Object handle for queue 2
MQLONG CompCode, Reason; // Return codes
MQLONG options;

MQOD od1 = {MQOD_DEFAULT}; // Object descriptor for queue 1
MQOD od2 = {MQOD_DEFAULT}; // Object descriptor for queue 2
MQMD md = {MQMD_DEFAULT}; // Message descriptor
MQPMO pmo = {MQPMO_DEFAULT}; // Put message options
MQGMO gmo = {MQGMO_DEFAULT}; // Get message options

// 1 Connect application to a queue manager.
strcpy (QMName,"MYQMGR");
MQCONN (QMName, &HCon, &CompCode, &Reason);

// 2 Open a queue for output
strcpy (od1.ObjectName,"QUEUE1");
MQOPEN (HCon,&od1, MQOO_OUTPUT, &Hobj1, &CompCode, &Reason);

// 3 Put a message on the queue
MQPUT (HCon, Hobj1, &md, &pmo, 100, &buffer, &CompCode, &Reason);

// 4 Close the output queue
MQCLOSE (HCon, &Hobj1, MQCO_NONE, &CompCode, &Reason);

// 5 Open input queue
options = MQOO_INPUT_AS_Q_DEF;
strcpy (od2.ObjectName,"QUEUE2");
MQOPEN (HCon, &od2, options, &Hobj2, &CompCode, &Reason);

// 6 Get message
gmo.Options = MQGMO_NO_WAIT;
buflen = sizeof(buffer - 1);
memcpy (md.MsgId, MQMI_NONE, sizeof(md.MsgId);
memset (md.CorrelId, 0x00, sizeof(MQBYTE24));
MQGET (HCon, Hobj2, &md, &gmo, buflen, buffer, 100, &CompCode, &Reason);

// 7 Close the input queue
options = 0;
MQCLOSE (HCon, &Hobj2, options, &CompCode, &Reason);

// 8 Disconnect from queue manager
MQDISC (HCon, &CompCode, &Reason);
```

Figure 29. Fragments of an MQSeries Program
Note: The fields CompCode and Reason will contain completion codes for the APIs. You can find them in the Application Programming Reference.

1 This statement connects the application to the queue manager with the name MYQMGR. If the parameter QMName does not contain a name, then the default queue manager is used. HCon receives the handle to the queue manager. This handle must be used in all subsequent APIs.

2 To open a queue the queue name must be moved into the object descriptor that will be used for that queue. This statement opens QUEUE1 for output only (open option MQOO_OUTPUT). Returned are the handle to the queue and values in the object descriptor. The handle Hobj1 must be specified in the MQPUT.

3 MQPUT places the message assembled in a buffer on a queue. Parameters for MQPUT are:
   • The handle of the queue manager (from MQCONN)
   • The handle of the queue (from MQOPEN)
   • The message descriptor
   • A structure containing options for the put (refer to the Application Programming Reference)
   • The message length
   • The buffer containing the data

4 This statement closes the output queue. Since the queue is predefined no close processing takes place (MQOC_NONE).

5 This statement opens QUEUE2 for input only using the queue-defined defaults. You could also open a queue for browsing, meaning that the message will not be removed.

6 For the get the nowait option is used. The MQGET needs the length of the buffer as input parameter. Since there is no message ID and correlation ID specified, the first message from the queue is read.

7 This statement closes the input queue.

8 The application disconnects from the queue manager.

Communication between programs is time-independent. The sender can continue processing without waiting for a reply. The receiving program does not even have to run. MQSeries holds the messages until it is ready to process them.

MQSeries applications are message-driven. The arrival of a message triggers an event. Just like clicking on a push button in a GUI invokes some procedure, a message starts a program that processes the message data.
Chapter 4. How VisualAge and MQSeries Work Together

This chapter describes the MQSeries features build into VisualAge, the VisualAge MQSeries parts and discusses some important features you need to know when writing MQSeries applications.

4.1 The VisualAge MQSeries Parts

VisualAge for Smalltalk provides reusable parts that you can add to your application to interface with MQSeries. The code to invoke MQSeries APIs is written in Smalltalk and wrapped inside those reusable parts. The four parts are:

- MQSeries Procedure Dialog part
- MQSeries Connection part
- MQSeries Connection Specifications part
- MQSeries Message part

Their usage and functions are described below.

4.1.1 MQSeries Procedure Dialog Part

When to use:
Use the ProcDialog when you want one conversational message exchange with another MQI application, passing data in a structured format (records).

Function:
Performs the CONNECT to a queue manager and an OPEN of a pair of queues, the request queue and reply queue. Queue manager and queue names are specified in the Connection Spec described below. You provide a record structure for the data you want to pass between client and server applications. A one-time message exchange
between the client/server pair is executed, and then both queues are closed. The connection to the queue manager is also terminated.

An example on how to use this part is provided in Chapter 5, “The First VisualAge for Smalltalk MQ Application” on page 47.

4.1.2 MQSeries Connection Part

When to use:
Use the Connection part when you want to control MQSeries related actions, such as CONNECT and DISCONNECT, and when you want to pass message in other formats besides records, such as strings and objects.

Function:
You can invoke and control almost all MQSeries APIs provided by the MQI:
- CONNECT
- DISCONNECT
- GET
- PUT Reply and PUT Request
- COMMIT
- BACKOUT

4.1.3 MQSeries Connection Specifications Part

When to use:
You use the Connection Spec to provide information needed to establish connection to the queue manager, such as the names of the queue manager, request queue and reply queue.

Function:
In this part, connection specification is kept. Specify MQ resources and your message record format. You can specify your target code page for your structured message for data conversion.

4.1.4 MQSeries Message Part

When to use:
Use the message part when you want to construct your own message with your own descriptor options other than using the default message options setup by VisualAge for Smalltalk.
4.2 How VisualAge and MQSeries Interact

When you write VisualAge for Smalltalk applications that use messaging and queueing, you use the VisualAge MQSeries parts provided by the product. You don’t have to write any code on the MQI level. Simply incorporate the parts into your client or server programs. You can access the pre-written MQI actions, events and attributes through visual connections or through Smalltalk code.

The VisualAge MQSeries parts are operated under a scenario that is common in MQSeries client/server programming. If your application design is outside the scope of that scenario, you can still use those parts with minimum additional Smalltalk code to cover the variations.

The scenario is elucidated in Figure 30 on page 44 and described below:

- A client application always puts request messages into a designated request queue of the target queue manager. It expects to receive replies from a designated reply queue.

- A server application always gets request messages from a designated local request queue and puts reply messages into a designated reply queue.

- The client/server pair must coordinate their choice of request and reply queues.

- VisualAge starts a background job when an application connects to a queue manager. For each CONNECT action, one background job is started, one for the client and one for the server.

At the client side, the background job gets all messages from the designated reply queue and put them in an ordered collection for later client retrievals. On the other hand, the background job initiated by the server when it connected to a queue manager, retrieves all messages from the designated request queue.

- The life span of the background job is the same as the life span of the application that created it. It stays alive even when the connection to a queue manager has been terminated.

The background job temporarily stores messages in an ordered collection list. A message stays there until it is retrieved by a GET issued against the queue or until the order collection is garbage collected, that is, the background job terminates.
4.3 More about the Background Job

When you use the Connection part to connect to a queue manager, a background job is implicitly forked. The background job reads all messages from the queue and stores them in an ordered collection. The removal of messages from queue can be a problem if the queue is shared with other applications. Also note, when the background job is terminated, the remaining messages, if any, in the ordered collection will be lost. In VisualAge for Smalltalk Version 4.0, you can control the background job to retrieve only certain messages. Your selection criteria can be the message ID and/or the correlation ID.

We provide two examples that demonstrate how to do that:

1. Chapter 8, “Extended To-Do List - Using Message ID” on page 95
2. Chapter 9, “Getting a Particular Message” on page 103
4.4 About Reply and Request Queues

In VisualAge for Smalltalk, you can tear off attributes from a part. That allows you to further utilize the properties in the lower level subparts. For instance, you can tear off the reply queue from a Connection part so that you can connect all actions, events and attributes pertained to the queue.

Detailed examples are provided in Chapter 7, “To-Do List - A Client/Server Application” on page 79.

4.5 About Syncpoint Processing

You can specify syncpoint processing in the Connection Spec settings. In MQSeries, you can specify a syncpoint in the GET and PUT APIs. However, when you select syncpointing in VisualAge for Smalltalk, both GET and PUT are issued with the syncpoint option. Therefore, when a COMMIT or BACKOUT is issued, it affects both the reply queue and the request queue.

An example is provided in Chapter 10, “Syncpoint Processing” on page 109.

4.6 About Data Conversion and Record Parser

MQSeries and VisualAge MQSeries parts both support data conversion. However, the MQSeries data conversion is limited to string data only. On the other hand, VisualAge for Smalltalk supports only record format data conversion. You can use Connection Spec to specify your target code page. Data will be converted to the target code page before the formatted message is sent to the queue. You can provide your C header file or Cobol record definition (copy book) to VisualAge for Smalltalk in the Connection Spec.

Record parser is provided if you load the particular language support feature, such as C or COBOL.
Chapter 5. The First VisualAge for Smalltalk MQ Application

In this chapter, we explain how to create a small application using the MQSeries Procedure Dialog part. This part provides a simple message exchange with a server application. The message itself is defined in a C or COBOL header file. That is the reason you have to load the language feature when you set up your image (see 2.5, “Using VisualAge for Smalltalk for the First Time” on page 13).

The MQSeries Procedure Dialog part, Proc Dialog for short, follows a predefined sequence of steps:

1. Connect to the server and open request and reply queues.
2. Send a request message to the server.
3. Receive a reply message from the server.
4. Disconnect from the server and close the queues when the window is closed.

In the following sections we describe this mini application in great detail. It is intended for VisualAge novices who want to get a first practical look at the product.

Smalltalk

No Smalltalk code required for this example.

5.1 What the Application Does

The purpose of this exercise is to learn how to use the MQSeries Procedure Dialog part. Our first application uses the fixed message flow described above.

For this example, we do not create a server application. Instead, we use the MQSeries sample program, amqsput.exe, to put a message to a reply queue. Figure 31 on page 48 shows the programs and the message flow:

1. Use the AMQSPUT program to put a message into the reply queue.
2. The application puts a message into the request queue.
3. The application gets all messages from the reply queue.
4. You may use AMQSGET to look at the message in the request queue.

The GUI is shown in Figure 45 on page 59.
To develop this application we have to do the following:

- Define the MQSeries objects
- Set up VisualAge
- Create the user interface
- Test the application

5.2 How to Set Up MQSeries

Since this application does not connect to any application in a remote system, we need only three MQSeries objects:

- A queue manager with the name VAQMGR
- A request queue with the name REQUEST
- A reply queue with the name REPLY

Note: The names are case-sensitive.

```
DEFINE QLOCAL (REQUEST) REPLACE +
  DESCR('Request queue')
DEFINE QLOCAL (REPLY) REPLACE +
  DESCR('Reply queue')
```

The definitions for the two local queues are in the file vaqmgr.tst, shown in Figure 32. All other parameters are defaults.
If you do not have a queue manager in your machine, follow these steps:

Step 1. To create the queue manager VAQMGR as the default queue manager, type on a command prompt:
    `crtmqm /q VAQMGR`

Step 2. Start the default queue manager with the command:
    `strmqm`

Step 3. To create the default objects for the queue manager enter:
    `runmqsc < c:\mqm\mqsc\amqscoma.tst > a.a`

Step 4. Check the output file a.a for any errors.

Step 5. To create the objects for the application enter:
    `runmqsc < vaqmgr.tst > a.a`

Step 6. Check the output file a.a for any errors.

You may leave the queue manager running.

5.3 How to Define a Message

We define the message structure in a C header file as follows:

```c
struct RecordOne {
    char text[20];
};
```

*Figure 33. First Application: Message Structure*

The message contains only one field. For this exercise, do not add any more fields. You would get an error. The structure allows us to create a quick form later on.

Name the file first.h and place it in the directory \vafirst.

**Important**

This structure is used to parse record from the REPLY queue. Make sure that the records you put into that queue are 20 bytes long. Otherwise, you get an error.
5.4 How to Set Up VisualAge for Smalltalk

Before you can develop your first application, you have to set up the VisualAge for Smalltalk environment. This application sends a message to the request queue REQUEST and gets a message from the reply queue REPLY. Whether or not the queues reside in your machine or in a server depends on the MQSeries software you have installed in your system. If you installed the MQSeries client, queue manager and objects reside in the server; otherwise, they are in your machine. There are different libraries for MQSeries client and MQSeries server. You must specify which one VisualAge shall use. Choose one of the following:

- MQSeries Client (MQIC)
- MQSeries Client and Server (MQM)

To find out which library VisualAge is set up to use execute the following steps:

1. In the System Transcript window click on File.
2. Select New from the menu to bring up a Workspace window and type:

   Smalltalk
   PlatformLibrary logicalName: 'MQSERIESDLL'.

3. Swipe (highlight) this text, click the right mouse button and select Display from the menu.
4. One of the following will be displayed:
   - PlatformLibrary[MQSERIES - > 'MQM']
     if the setting is for the MQM interface
   - PlatformLibrary[MQSERIES - > 'MQIC']
     if the setting is for the MQSeries client interface

To change the settings, execute one of the following commands:

   Smalltalk
   PlatformLibrary mapLogicalName: 'MQSERIESDLL' toPhysicalName: 'MQM'
   or
   PlatformLibrary mapLogicalName: 'MQSERIESDLL' toPhysicalName: 'MQIC'

Swipe the text, click the right mouse button and select Execute from the menu.
Notes:

1. Systems used for application development usually have a queue manager installed (MQM).
2. VisualAge uses MQIC as default.
3. You may change from MQIC to MQM each time you start VisualAge.

Language feature: You need also the C or COBOL feature to parse the header file that contains the structure of the message. For this example, load the C feature. Section 2.5, “Using VisualAge for Smalltalk for the First Time” on page 13 describes how to do that.

5.5 How to Develop an Application

In the following sections we describe:

• How to create a VisualAge application
• How to place a push button in the window
• How to put a label and an input/output field in the window
• How to invoke the MQSeries Procedure Dialog part

5.5.1 How to Create a VisualAge Application

Before you can create your first GUI you have to create an application. To do that follow these steps:

1. In the VisualAge Organizer window click on Application and then select New from the menu.
2. In the subsequent window, shown in Figure 34, type the name of the application, ApplOne.
3. Click on OK.

![New Application](image_url)

Figure 34. First Application: New Application
4. This adds two entries to the VisualAge Organizer window (see Figure 35 on page 52).

![VisualAge Organizer Window](image1)

**Figure 35. First Application: VisualAge Organizer Window**

5.5.2 How to Create a Visual Part

Now create a visual part. Since this part will become a GUI we name it MQOneView.

1. From the Parts menu in the VisualAge Organizer window select **New**.

2. In the subsequent New Part window overwrite the part class name as shown in Figure 36 and click on **OK**.

The part will be added to the Organizer window. Then you will see a Composition Editor window.

![New Part Window](image2)

**Figure 36. First Application: Create New Part**
5.5.3 How to Create Controls

The GUI will contain three objects:

- A label
- An entry field
- A push button

Later on you will see how the text field and label are created automatically using the quick form feature. The text field serves two purposes:

1. The user types some text that will be moved into the request message.
2. The program displays the text received in the reply message.
The push button will activate the message transfer. To create it follow these steps:

1. Click on the buttons folder on the left and then on the push button icon to its right.
2. Move the cursor into the window and click the left mouse button drop the widget into it.
3. Double-click on the push button in the window and you see its settings in notebook form (Figure 38).
4. Change its label to Execute and click on OK.

Note: To display the settings this way you must install the feature VisualAge, Notebook Style Setting Views V4.0.
5.5.4 **How to Use the MQSeries Procedure Dialog**

In this section, we explain one way you can use the MQSeries Procedure Dialog part. The procedure dialog will be invoked when the push button is clicked. To connect this part with the push button follow these steps:

1. Click on ![MQSeries Parts](image) to display the MQSeries parts.

2. Click on ![Move Object](image), move the cursor into the free space (outside the window) and drop the object by clicking the left mouse button.

3. Double-click on **MQSeries ProcDialog1** to open the settings notebook shown in Figure 39. Enter here the names of the queue manager and queues.

![Figure 39. First Application: ProcDialog Settings - Destination](image)

![Figure 40. First Application: ProcDialog Settings - Records](image)
4. In the Records settings in Figure 40:
   - Enter the file name of the record structure. The structure is shown in Figure 33 on page 49. Enter the file name as shown or click on the Find button to search for it.
   - Click on the C radio button because we have a C language structure.
   - Click on the 32 bit radio button.
   - Click on Parse to convert the C structure to a Smalltalk structure. When this is done, recordone appears in the list box of available records.

5. Select recordone and click on OK.

6. With the right mouse button click on the Execute button.

7. Select Connect from the menu and then clicked as shown in Figure 41.

8. A line will appear. Use it to connect the push button with the Proc Dialog part outside the window.

9. After you clicked the left mouse button, a small pop-up menu appears. Select Execute.

10. The line turns green. Clicking on the push button will now invoke the execute event of the procedure dialog.
11. This step is optional. You may want to check if the clicked event triggers the execute action of the MQSeries Procedure Dialog part.

   • With the right mouse button, click on the green line.
   • Select Open Settings from the pop-up. This opens the Event-to-action Connection Settings window.
   • Ensure that both event and action are highlighted as shown in Figure 42.
   • Select Cancel.

12. With the right mouse button, click on the Proc Dialog part and select Quick Form from the menu. This is the same menu as shown in Figure 41 on page 56.

13. From the subsequent menu select recordone.

14. Move the cursor inside the window and click the left mouse button. In the Composition Editor’s window will appear what you see in Figure 43 on page 58.

Quick Form places label and text field in the GUI, tears off “recordone of MQSeries Proc Dialog1” from the Proc Dialog and connects the part with the text field. VisualAge obtains the word text from the record structure shown in Figure 33 on page 49.

Now the GUI is completed and you are ready to test it.
5.6 How to Test the Application

Since the GUI uses the MQSeries Procedure Dialog part, it puts a message into the request queue, waits for a reply message to arrive, and gets the reply message from the queue. Therefore, we have to put some messages into the reply queue first, so that the application can read them and doesn’t time out.

Use the MQSeries sample program amqsput to put some messages into the queue REPLY. Make sure that the queue manager is running and type the commands shown in bold in Figure 44 on page 59.

Then you can execute the program from the Composition Editor. Click on

When the window in Figure 45 on page 59 appears, enter some text in the input field and click on **Execute**. The text will be moved into a message, and the message is put into the reply queue.
The MQSeries sample program amqsget gets the messages from the REQUEST queue and displays them.

```
[C:\]cd mqm\tools\c\samples\bin
[C:\mqm\tools\c\samples\bin]amqsput REPLY VAQMGR
Sample AMQSPUTO start
target queue is REPLY
111112222333344444
222223333444444445555
33333444445555566666
<---- Press Enter twice to end amqsput
Sample AMQSPUTO end
[C:\mqm\tools\c\samples\bin]
```

Figure 44. Using AMQSPUT. Type the text shown in bold.

```
[C:\mqm\tools\c\samples\bin]amqsget REQUEST
Sample AMQSGET0 start
message <aabbccdd>
no more messages <--- Wait for time out
Sample AMQSGET0 end
[C:\mqm\tools\c\samples\bin]
```

Figure 46. Using AMQSGET. Type the text shown in bold.
Chapter 6. A Quick Tour of a VisualAge for Smalltalk MQ Application

In the following sections, we explain how to create a small application using three VisualAge parts:

- MQSeries Connection part
- MQSeries Connection Specifications part
- MQSeries Message part

In this chapter, we explain by means of another small application:

- How to create a user interface
- How to connect VisualAge MQSeries parts to a window
- How to connect to a queue manager
- How to put a request message on a queue
- How to get a reply message from a queue
- How to disconnect from the queue manager
- How to handle MQSeries error conditions in a VisualAge for Smalltalk application
- How to build a log
- How to write a Smalltalk method
- How to connect events to a method

This example program is a typical re-usable VisualAge for Smalltalk part.

Figure 47 on page 62 shows two instances of the program at run time. Both programs run in the same workstation. Therefore, we can use two local queues to exchange messages between the two instances. The program has two GUIs. One is used to send and receive messages; the second one is used as a log. The figure shows only one of the logs.

6.1 How to Run the Program

- Make sure that the queue manager is running (strmqm).
- Choose a request queue name from the first combo box, REQUEST for the first instance and REPLY for the second.
- Choose a reply queue name from the second combo box, REPLY for the first instance and REQUEST for the second.
- Click on Connect to connect to the queue manager.
• Type some text in the Put Message text field.

• Click on **Put Message**.

• Click on **Get Message** to display the message text the other instance sent.

• Click on **Disconnect** to disconnect from the queue manager.

**Note:** The queue names must be known to the queue manager; that is, they have to be defined before you start your application. Refer to section 5.2, “How to Set Up MQSeries” on page 48 for more information.

*Figure 47. Second Application: Two Instances at Run Time*
6.2 How to Create the User Interface

Figure 48 shows the user interface for the application. It contains:

- Five labels
- Two combo boxes from which input and output queues can be chosen
- A text field in which text for the outbound message can be typed
- A text field to display the text from the inbound message
- A text field to display messages
- Four push buttons to invoke MQSeries functions

To pre-define the queues specify their names in the settings for each of the combo boxes. Type one queue name in the field Initial contents and both names in Initial list items.
6.3 How to Connect and Disconnect

To connect to a queue manager, we use two MQSeries parts:

MQSeries Connection part

MQSeries Connection Specifications part. Move these parts into the Composition Editor, outside your GUI.

In the setting of the MQSeries Connection Specifications part, you specify the name of the queue manager (VAQMGR) and the names of the request and reply queues, as shown in Figure 49.

![MQ Series Connection Spec (MQ Series Connection Spec1) - !](image)

Message Queue Specifications

Queue manager name

VAQMGR

Request Queue name

REQUEST

Reply Queue name

REPLY

Alternate user ID

Sync point processing

Server

Figure 49. Second Application: MQ Series Connection Spec Settings
6.3.1 How to Connect to MQSeries Parts

First, connect the clicked event of the Connect push button with the connect action of the MQSeries Connection part:

- Click the right mouse button on the Connect push button.
- From the pop-up (shown in Figure 41 on page 56), select Connect and then clicked.
- Connect the line with the MQSeries Connection Spec1 part and a window with the following MQSeries actions appears:
  - backout
  - commit
  - connect
  - disconnect
  - get
  - putReply
  - putRequest
  - All Functions

Select connect.

You will see a broken line between the parts. This means that this connection needs a parameter, a connection spec. To pass a parameter,
connect the \textit{self} attribute of the MQSeries Connection Specifications part with the \textit{aConnectionSpec} attribute of the connection line.

- With the right mouse button, click on MQSeries Connection Spec1 in the Composition Editor’s window.
- Again, you see the Visual Parts Menu shown in Figure 41 on page 56. Select \texttt{connect} and then \texttt{self}.
- Connect the line with the line between the Connect button and the connection part.
- From the pop-up select \texttt{aConnectionSpec}.

Now the line is no longer broken. The parameters from the connection specification in Figure 49 on page 64 are available for the connection between the Connect button and the MQSeries Connection part.

Next connect the the \texttt{clicked} event of the Disconnect push button with the \texttt{disconnect} action of the MQSeries Connection part. The steps are as described for Connect above. However, you do not need a Connection Spec.

With the above steps we made three connections. The text below is displayed in the Composition Editor’s window when you click on one of the connection lines.

\begin{tabular}{ | l | }
\hline
\textbf{Connections} \\
\texttt{PushButton1,clicked} \rightarrow MQSeries Connection1,connect  \\
\texttt{((PushButton1,clicked} \\
\texttt{\quad } \rightarrow MQSeries Connection1,connect),aConnectionSpec  \\
\texttt{\quad } \rightarrow MQSeries Connection Spec1,self) \\
\texttt{PushButton4,clicked} \rightarrow MQSeries Connection1,disconnect  \\
\hline
\end{tabular}

\subsection{6.3.2 How to Display Error Messages}

We want to display an error message when the Connect push button is clicked but the queue manager is not running. To catch the error condition do the following:

- With the right mouse button, click on the connection part.
- From the pop-up window select \texttt{Tear-Off Attribute} and then \texttt{lastError}.
- Move the cursor into free space and drop the new part.
- With the right mouse button, click on that part and select \texttt{connect} and then \texttt{codeAsString}. 

66 Application Development with VisualAge for Smalltalk and MQSeries
6.3.3 The First Test

Now you can test the connect and disconnect functions of your application. Save your part first. If the queue manager is not running, the program displays a message in the Information text field.

Note: Make sure that you use the correct library (MQM or MQIC).
6.4 How to Put a Message Into a Queue

You type the text to be send in the message into the Put Message entry field. To put the text into the message and the message into the queue, follow these steps:

1. Connect the `clicked` event of the Put Message button to the `putRequest` action of the MQSeries Connection part.
   - With the right mouse button click on the Put Message button and select `connect` and `clicked` from the pop-ups.
   - Connect the line to the connection part and select `putRequest` from the menu.

2. Connect the `object` attribute of the Put Message entry field to `inputData` of the connection line.
   - With the right mouse button click on the Put Message entry field and select `connect` and `object` from the pop-ups.
   - Connect the line to the line between the push button and connection parts and select `inputData` from the menu.

   Connections

   • PushButton2,clicked --> MQSeries Connection1,putRequest
   • ((PushButton2,clicked --> MQSeries Connection1,putRequest),
     inputData --> Text2,object)

Save the parts and test. Since the get is not implemented yet, use the browse utility from MQSeries, amqsgbr.exe, to read the message in the queue.

And again, if an error occurs check what library you use. Clients use MQIC and systems that have a queue manager installed use MQM. Refer to page 50 for more information.

6.5 How to Get a Message from a Queue

Add the MQSeries Message part to your application. We need it to get a message from a queue and to display its contents.

1. Connect the `clicked` event of the Put push button to the `get` action of the MQSeries Connection part.
With the right mouse button click on the Get push button and select **connect** and then **clicked** from the pop-ups.

Connect the line to the connection part and select **get** from the menu.

2. Connect the **normalResult** attribute of the connection line to the **initializeWith** action of the MQSeries Message part.

   - With the right mouse button, click on the connection line.
   - From the pop-ups, select **connect** and then **normalResult**.
   - Drop the cursor into the MQSeries Message part part and select **initializeWith** from the subsequent menu.

3. Connect the **contentsAsString** attribute of the MQSeries Message part with the Get Message entry field.
• With the right mouse button, click on the MQSeries Message part and select `connect` and then `contentsAsString` from the pop-ups.

• Drop the cursor into the Get Message entry field and choose `object` from the menu.

---

**Connections**

• `PushButton3,clicked --> MQSeries Connection1,get`

• `((PushButton3,clicked --> MQSeries Connection1,get),normalResult --> MQSeries Message1,initializeWith) --> Text2,object)`

---

Finally, you can put a message and get a message from the VisualAge application. Figure 52 on page 69 shows the Composition Editor with the complete GUI. Save your parts and test it. You can put a message into your reply queue using the amqsput.exe provided by MQSeries.

The next sections describe options that make your application look nicer.

### 6.6 How to Add a Log Window

This sections describes how to create a log window for the application. For the log window, you have to create a method and add it to your visual part. The name of the method is `whereAmI:`. The method uses the instance variable `ws`. In VisualAge, variables must be declared before you write the method.

#### 6.6.1 How to Define Instance Variables

The Smalltalk code below defines the variable `ws`. Create it as follows:

• Bring up the Script Editor and click on `Instance`.

• Since we define an instance variable, click on `Instance`.

• The editor displays a definition. Add `ws` as shown below.

```smalltalk
AbtAppBldrView subclass: #ApplTwoView
instanceVariableNames: 'ws'
classVariableNames: '
poolDictionaries: ''
```

• To save it, click the right mouse button and select `save` from the menu.
6.6.2 How to Write a Method

- Bring up the Script Editor and click on **Methods**.
- From the pull-down, select **New Method Template**. In the Script Editor, the following template is displayed:

```
Smalltalk
messagePattern
"comment"
| temporaries |
statements
```

Type the method as shown in Figure 53 and save it. Click the right mouse button and select **save** from the menu.

6.6.3 How to Connect Events to a Method

We connect three events to the method:

1. The **connected** event of the MQSeries Connection part
2. The **disconnected** event of the MQSeries Connection part
3. The *put* event of the MQSeries Connection part

The following steps describe how to connect events to methods:

- With the right mouse button, click on the connection part and select **connect** and **All Features**.

- You will see the window shown in Figure 54. Select the event **connected** and click **OK**.

- Drop the cursor in the white space and select **Event to Script**.

- Now the window in Figure 55 on page 73 appears. Highlight the method name and click on **Set parameters**. This displays the window shown in Figure 56 on page 73.

- Enter the value **connected** and click **OK**.

- Click **OK** again when the previous window appears again.

- Now you see a line that connects the connection part with the border of the editor’s window.
Connect the other two events in the same fashion. Specify disconnected and put instead of connected. When completed, you will see three lines between the connection part and the border of the Composition Editor.

**Connections**

- MQSeries Connection1, connected --> ApplTwoView, whereAmI:
- MQSeries Connection1, disconnected --> ApplTwoView, whereAmI:
- MQSeries Connection1, put --> ApplTwoView, whereAmI:
6.7 How to Write a Combo Box Method

Here we describe how you can change queue names in the GUIs. The combo box allows you to select between several queues. Make sure that the queues are defined for the queue manager you use. In this example we have only two queues.

First, create a method that selects the highlighted queue name in the combo box. Which combo box to change is specified in the anOption parameter of the method shown below:

```smalltalk
setQueueName: anOption
anOption = 'request'
ifTrue: [ (self subpartNamed: 'MQ Series Connection Spec1')
  requestQueueName: ((self subpartNamed: 'Combo Box1')
    selectedItem)]
ifFalse: [ (self subpartNamed: 'MQ Series Connection Spec1')
  replyQueueName: ((self subpartNamed: 'Combo Box2') selectedItem)].
```

Connect the combo boxes to the method

To connect the `selectedItemChanged` event of combo boxes to the method `setQueueName` follow these steps:

- With the right mouse button, click on a combo box and select `connect` and then `All Features` from the menus.
- In the window shown in Figure 57 on page 75, highlight `selectedItemChanged` and click on `OK`.
- Drop the cursor in the white space and select `Event to Script` from the menu.
- The next window looks like the one shown in Figure 55 on page 73. Select the name of the method, `setQueueName`.
- Click on `Set Parameters`
- In the window shown in Figure 58 on page 75, type either `request` or `reply` and click `OK`.
- Click `OK` on the next window and you are done.
Connections

- Combo Box1, selectedItemChanged --> ApplTwoView, setQueueName:
- Combo Box2, selectedItemChanged --> ApplTwoView, setQueueName:

Figure 57. Second Application: Start Connection from Combo Box

Figure 58. Second Application: Parameters for Combo Box
6.8 How to Change the Window Title

To display the queue manager name in the window title create the following method:

```smalltalk
setNames

(self subpartNamed: 'Window')
  title: ('QManager',
    ((self subpartNamed: 'MQ Series Connection Spec1')
     queueManagerName)).
```

Then connect the `openedWidget` event to the method `setNames`.

- With the right mouse button, click somewhere in the window, but not on one of the objects.
- From the menu select **connect** and then **openedWidget**.
- Drop the cursor somewhere in the white space.
- Select **Event Script** from the menu.
- Highlight the method name **setNames** in the subsequent window (Figure 55 on page 73) and click on **OK**.

**Connection**

```
Window,openedWidget --> ApplTwoView,setNames
```

Figure 59 on page 77 shows the completed application in the Composition Editor.
Figure 59. Second Application: Completed in Composition Editor
Chapter 7. To-Do List - A Client/Server Application

In the following sections, we create an MQSeries version of a to-do list client and server application. The scenario is as follows:

- A to-do list client sends to-do items to a server.
- The server adds the date to each to-do item it receives and sends it back to the client.
- When the client gets the message back from the server it adds it to its to-do list.

Figure 60 shows the message flow for this application.

The application consists of two visual parts, one for the client and one for the server. In this example, both visual parts, or windows, run in the same machine. Therefore, both windows connect to the same queue manager and use the same request and reply queue.

Since each window is a separate program, each of them must connect to the queue manager individually. Communication between the client/server pair is through messages in the request queue and the reply queue.

Figure 61 on page 80 shows the two GUls and a System Transcript window that is used as a log for server activities. Clicking in the Connect button connects to the queue manager. When connected, the check box will be marked.
In the next sections, we describe:

- How to construct the client window
- How to write the server program
- How to run the application

### 7.1 The To-Do List Client Window

Figure 62 on page 81 shows the user interface of the to-do client. Get the visual parts from the pallets and drop them into the default windows as shown. Note that each part you drop to the window or free-form surface is given a default part name. For instance, when you drop the push button needed for the connect into the window, VisualAge gives it the part name Push Button1. When you drop the second push button, needed for the Add, VisualAge names it Push Button2.
The part name can be used to reference the part associated with it inside your Smalltalk script if you need to go down to the script level to manipulate attributes of that part.

We don’t need to go to that level in this example; however, it is always a good programming practice to name your part with a name that is more descriptive. In this case, it is better to change Push Button1 to something like ConnectButton. To name a part, simply point to the part and click the right mouse button to get a pop-up menu, then select Change Name.
No new parts are introduced in this example, except for the toggle button (check box) which we use to show whether the queue manager is connected or not. Select the button category from the parts palette, then select the Toggle Button (see Figure 62).

After the widgets (controls) are in your window, you need the following parts in the free-form surface:

Table 4. To-Do List: Parts

<table>
<thead>
<tr>
<th>Category</th>
<th>Part</th>
</tr>
</thead>
<tbody>
<tr>
<td>MQSeries</td>
<td>Connection, Connection Spec</td>
</tr>
<tr>
<td></td>
<td>Message</td>
</tr>
<tr>
<td>Models</td>
<td>Ordered Collection</td>
</tr>
</tbody>
</table>

We need the MQ Series parts to interface with MQSeries, whereas the Ordered Collection is used to keep all to-do items received from the to-do list server.

Change the names of the parts by selecting ChangeName from the pop-up that appears when you click the right mouse button. One immediate reward of doing so is that your part names are a lot shorter. We chose the following names:

Table 5. To-Do List: Part Names

<table>
<thead>
<tr>
<th>Part</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>New Item input field</td>
<td>NewItemText</td>
</tr>
<tr>
<td>ToDo List list box</td>
<td>ToDoListList</td>
</tr>
<tr>
<td>Error message field</td>
<td>ErrorText</td>
</tr>
<tr>
<td>Toggle button</td>
<td>isConnectedToggleButton</td>
</tr>
<tr>
<td>Connect push button</td>
<td>ConnectButton</td>
</tr>
<tr>
<td>Add push button</td>
<td>AddButton</td>
</tr>
</tbody>
</table>
7.1.1 Program Logic and Visual Parts

The logic of the to-do list client and the purpose of each part associated with it is explained below:

• When the Connect push button (ConnectButton) is clicked, the client is connected to a queue manager and both queues specified in the Connection Spec are opened.

  Note: An error condition may be reported in the lastError attribute inside the Connection part, for example, when the queue manager has not been started. You need to tear off lastError from the Connection part to get the error text displayed.

  To prevent clicking the Connect button twice, you can disable it after the connection has been established.

  Figure 63 on page 84 shows the connections.

• If no error has been detected, the client can put a request message into the REQUEST queue by clicking the Add button (AddButton). The message is drawn from the entry field NewltemText.

  After the work is done, the user closes the window. Closing the window disconnects the client from the queue manager.

  Figure 65 on page 88 shows the connections.

• When the to-do list server puts the reply message into the REPLY queue, our replyQueue notices this event. The replyQueue is contained in the Connection part. In order to work with the queue we need to tear that part off the Connection part.

  Next we build a connection that gets the message off the reply queue and puts it into the Reply Message part.

  We are only interested in the message contents. So, we get the message contents off the ReplyMessage part and put it into the Ordered Collection object.

  To display the message, we build a connection between the Ordered Collection part and the list box TodoListList.

  Figure 64 on page 87 shows the connections.

7.1.2 Building the Parts

You can implement all functions described in the previous section visually, that is, without writing one line of Smalltalk code. If that surprises you, then you are about to discover the strength of visual programming the VisualAge for Smalltalk product families provide. Follow the step-by-step instructions below to construct your program logic. You are about to build your own parts while having fun.
Figure 63. To-Do List: Connect to Queue Manager

Step 1. Enter your queue manager and queue names in the Connection Spec’s notebook page shown in Figure 49 on page 64. In this example, we type VAQMGR, REQUEST, and REPLY respectively.

Note: Those names are case-sensitive.

How to: Double-click on the Connection Spec. Enter the names in the notebook and click OK.

Step 2. Connect the Connect button’s clicked event with the Connection Part’s connect action.

How to: Click the right mouse button over button. Select Connect and then clicked. Drop line into Connection. Select connect.

Step 3. Connect the Connection Spec’s self attribute with the dotted-line created with the previous step as aConnectionSpec. The dotted-line will change to a solid line.

How to: Click right mouse button on Connection Spec. Select Connect and then self. Drop line into the line created from previous step. Select aConnectionSpec.

Step 4. Connect the Connection’s connected event with the action set of the isConnectedToggleButton.
How to: Click right mouse button on Connection.
Select Connect and then All Features.
Select event connected, click OK.
Drop line into toggle button.
Select set.

Step 5. Tear-off lastError from the Connection and drop it next to the Connection part.

How to: Click right mouse button on Connection.
Select Tear-Off Attribute.
Select lastError.
Drop line into the free-form space.

Step 6. Connect codesAsString of the lastError of the Connection part with object of the the field ErrorText.

How to: Click right mouse button on part.
Select Connect, then codesAsString.
Drop line into error field.
Select object.

Step 7. You may add a cosmetic feature; connect the Connection’s connected event to ConnectButton’s disable action. This is to prevent unnecessary clicks for connect to your queue manager once it is connected successful.

How to: Click right mouse button on Connection part.
Select Connect, then All Features.
Select event connected, click OK.
Drop line into button.
Select All Features.
Select action disable, click OK.

Step 8. Tear off the Connection’s replyQueue attribute and drop it to the free-form surface. This creates the replyQueue of Connection part.

How to: Click right mouse button on Connection.
Select Connect, then Tear-Off Attribute.
Select replyQueue.
Drop part to free-form surface.

Step 9. Connect the event messageReplyReceived of the replyQueue of Connection to the get action of the replyQueue of Connection.

Yes, you are connecting something to itself!
The event detected by a part (message arrived) can trigger an action provided by the part itself (get message).
How to: Click right mouse button on part.
Select Connect, then All Features.
Select event messageReplyReceived, click OK.
Drop line into same part.
Select get.

Note: The resulting connection line is totally submerged inside the
replyQueue of Connection part. However, this connection is not
different from any other connection we have created so far.

The only trick is to highlight the connection line so that you can
drag it outside of its parent part. You need to be able to do that
for the next step. Practice now and take a break afterward to rest
your tired mouse-controlled-fingers.

How to: Move the cursor so that the tip of it is just to the left of the
upper left corner of the black square on the left side of the
part.

Step 10. Connect the normalResult event of the resulting connection line
created from the step above with the initializeWith action of
ReplyMessage.

How to: Click right mouse button on line just created.
Select Connect and then normalResult.
Drop into ReplyMessage and select initializeWith.
Step 11. Connect ReplyMessage’s `changed` event with the `add:` action of the OrderedCollection part.

**How to:** Click right mouse button on Reply Message.
Select Connect, then changed.
Drop line into Ordered Connection.
Select add:.

Step 12. Connect ReplyMessage’s `contentAsString` to the resulting connection line from the previous step as `anObject`.

**How to:** Click right mouse button on Reply Message.
Select Connect, then `contentAsString`.
Drop line to line created in previous step.
Select anObject.

Step 13. Connect the `self` attribute of the Ordered Collection with `items` of the list box.
Figure 65. To-Do List: Add an Item

**How to:** Click right mouse button on Ordered Collection.
Select Connect, then self.
Drop line into list box.
Select items.

Step 14. At last, connect AddButton’s `clicked` event with `putRequest` of Connection, and then connect NewItemText’s `object` to the resulting connection `inputData`.

**How to:** Click right mouse button on Add button.
Select Connect, then `clicked`.
Drop line into Connection.
Select `putRequest`.

Click right mouse button on New Item field.
Select Connect, then `object`.
Drop cursor on line just created.
Select `inputData`.

Step 15. To disconnect from the queue manager when the window is closed, connect `aboutToCloseWidget` of the window with the `disconnect` action of the Connection part.
How to: Click right mouse button on the window.
Select Connect, then aboutToCloseWidget.
Drop line into Connection.
Select disconnect.

Now, you completed your client window. Save the TodoListClientView part.
The program is now ready to send a message to the TodoListServer.

Figure 66. To-Do List: Completed Client View
7.2 The To-Do List Server Program

Usually, server programs do not have a user interface. However, for illustrative purpose, we create a TodoListServerView to show:

- How to get a message
- How to manipulate a message
- How to send a reply to the requester

Figure 67 shows the user interface and the parts needed for the server program. Create the visual parts in the default window, with exception of the three rows on the top, queue manager and queue names. These labels and fields will be created automatically using the Quick Form feature.

After that, add a Connection, a Connection Spec and a Message from the MQSeries category.

![To-Do List Server View Diagram]

*Figure 67: To-Do List: Server Interface and Parts*
7.2.1 Program Logic

The logic of the server part is similar to that of the client, except that the server expects messages in the request queue and puts replies to the reply queue. After the server receives a request message, it performs the following functions:

- It writes a line of text to the system transcript to inform you that a message has been received.
- It puts a time stamp to the message and returns it to the client.
- It writes some text to the system transcript to indicate that message processing is completed.

Writing to the system transcript to log a server’s progress is often required by the design of a server program. You may choose another destination than the system transcript or write the log to a permanent storage such as a file.

7.2.2 Building the Parts

The following steps show how to construct the server program:

Step 1. Open the setting of the Connection Spec part and fill in the information as shown in Figure 49 on page 64. You have to enter the names of the queue managers and queues. In addition, tick the server check box in the Destination page of the setting notebook.

Step 2. Instead of dropping objects into the GUI yourself, you can let VisualAge for Smalltalk generate some preset quick forms for you.

Example: Queue names, queue manager name, and other attributes are preset in the Connection Spec part (see Figure 49 on page 64). To display them, you can select Quick Form from the ConnectionSpec pop-up. In the subsequent menu you will see some or all of the following items:

- bigEndian (Boolean)
- bufferLength (Integer)
- codePage (Integer)
- queueManagerName (String)
- replyQueueName (String)
- requestQueueName (String)
- server (Boolean)
- syncPoint (Boolean)
- userId (String)

Once you select the Quick Form you want, for instance, queueManagerName, the cursor will be loaded and you may drop
the item into the window (somewhere on the left side, if possible). Two objects will appear, the label queueManagerName and a text field next to it.

![Diagram of queued messages and buttons](image)

**Step 3.** Tear-off the requestQueue (not replyQueue) from Connection part.

**How to:**
1. Click the right mouse button on Connection.
2. Select Connect, then Tear-Off Attribute.
3. Select requestQueue.
4. Drop part to free-form surface.

**Step 4.** Connect the messageRequestReceived event of requestQueue of Connection to the get action of the same part. *Yes, you are connecting something to itself again!*

**How to:**
1. Click the right mouse button on the part.
2. Select Connect, then All Features.
3. Select event messageRequestReceived and click OK.
4. Drop the line into the same part and select get.

If you have difficulties read page 86 again.

**Step 5.** Connect the normalResult event to the initializeWith action of Message.

**How to:**
1. Click the right mouse button on line just created.
2. Select Connect, then normalResult.
3. Drop line into Reply Message.
4. Select initializeWith.

**Step 6.** Create an Instance public method called put with the Smalltalk script in Figure 68 on page 93.

**Step 7.** Connect the messageRequestReceived event of requestQueue of Connection to the free-form surface put script.

**How to:**
1. Click right mouse button on requestQueue.
2. Select Connect, then All Features.
3. Select messageRequestReceived, click OK.
4. Drop line to free-form surface and select Event to Script.
5. Choose ToDoServerView class and put method.
Figure 68. To-Do List: Server Code

Step 8. Connect the aboutToCloseWidget event of Window to the disconnect of the Connection.

How to: Click right mouse button in the window.
Select Connect, then aboutToCloseWidget.
Drop line into Connection.
Select disconnect.

Now, your TodoListServerView is also ready. Save it. The complete server view is in Figure 69 on page 94.

7.3 Executing the To-Do List Application

You can test the application from the Composition Editor.

1. Make sure that the queue manager is running.
2. Start MQTodoListClientView and MQTodoListServerView.
3. Click on the Connect button in both views.
   The connections to the queue manager are successful when both check boxes are marked.
   If one of the isConnectedToggle check boxes is not ticked, check the following:
   • Are you running in the right environment (MQM or MQIC)?
   • Is the queue manager running?
   • Is the name of the queue manager spelled correctly?
• Matching the queue manager name the one specified in the Connection Spec?

4. After the connections are established, type in something in the New Item field in the client's window and click the Add button.

Figure 61 on page 80 shows some test results when no error occurs. If an error occurred, you will see the following message in the system transcript:

--> return code: nnn
where nnn is the return code from MQI.
Chapter 8. Extended To-Do List - Using Message ID

An MQSeries message consists of user data and a message descriptor. The fields in the message descriptor are defined in the MQMD structure, and are available to you through the MQSeries Message part. You can assign values to and get values from those attributes either by a Smalltalk script or through your visual parts connections. In this chapter, we explain how to access fields in the message descriptor, using the message identifier as an example.

We expand the example from Chapter 7, “To-Do List - A Client/Server Application” on page 79. If you have already gone through that exercise, you may modify it by adding a few parts. If you haven’t done so, you may copy the ToDoClientView and ToDoServerView provided on the diskette accompanying this book.

Note: Refer to Appendix E, “Diskette Contents” on page 157 for guidance on how to import and load those views to your image.

You can copy view classes from the VisualAge Organizer this way:

- Select (highlight) a part.
- Click the right mouse button.
- Select copy from the pop-up menu.

If copy is not in the menu, click on Options in the organizer’s menu bar and select Full Menus from the pull-down.

The new example will have two views:

- MQTodoListMsgIDClientView
- MQTodoListMsgIDServerView

8.1 What the Application Does

The scenario is the same as in the to-do list developed in Chapter 7:

- The client view submits a new item to the server by putting the new item in a request queue.
- The server view receives the request, time stamps it, and puts the message into the reply queue.
- The client view get the message from the reply queue and displays it.
As you can see in Figure 70, we add for this exercise one new field, MsgIDText, to the client view. Into this field the user can type an ID, such as his or her serial number. The contents of this field becomes the message ID (MsgId in the MQMD structure). This allows the user to assign a message ID to each message sent to the server.

Be aware that we do not use the the default message provided by VisualAge for Smalltalk. Therefore, we need to add a MQSeries Message part and write a simple Smalltalk script to send the message.

We also make two modifications to the push buttons:

- The Add button is disabled as long as the program is not connected to a queue manager.
- The Connect button will be disabled as soon as the program is connected to the queue manager to avoid multiple connects.

8.2 Building the Example

If you are building this example from scratch, refer to Chapter 7, “To-Do List - A Client/Server Application” on page 79 for step-by-step instructions. The steps below show only how to implement the new functions.
8.2.1 Add-Ons to the Client

Step 1. Add the new field and a label for it as shown in Figure 71 on page 98. You have to re-arrange the other objects in the window. Name the field NewItemText.

Step 2. Drop an MQSeries Message part on the free-form surface and change the name from MQSeries Message1 to RequestMessage.

**How to:** Click right mouse button.
Select Change Name.

Step 3. Tear-off the descriptor from the MQSeries Message part. This creates the descriptor of RequestMessage part.

**How to:** Click right mouse button.
Select Tear-Off Attribute.
Select descriptor.
Drop the part to the free-form surface.

Step 4. Create a new method using the Smalltalk code below.

```
Smalltalk
putRequestMessage: aMessage
"put a request type message to request queue using a message part"

| replyToQ |

replyToQ := (self subpartNamed: 'Connection Spec')
replyQueueName. aMessage replyToQ: replyToQ.
aMessage msgType: MqmtRequest.
```

**How to:** Click on Methods in Script Editor.
Select New Method Template.
Type the method.
Click right mouse button and save.

Step 5. Delete the connection between the Add button and the Connection part. When you click on this line you see the following text in the editor’s window:

**Connection:** AddButton, clicked -> Connection, putRequest

**How to:** Click right mouse button.
Select Delete.
Step 6. Connect the New Item field to the RequestMessage part. This connection puts the text to the message part.

**Connection:** NewItemText,object --> RequestMessage,putString

**How to:** Click the right mouse button.
Select Connect, then object.
Drop line into part.
Select putString.

Step 7. Connect the new field for the message ID, MsgIDText, to the descriptor of RequestMessage.

**Connection:** MsgIDText,object --> descriptor of RequestMessage,msgid
How to: Click the right mouse button, and select connect, object, drop into.
Select Connect and then object, drop into.
Drop line into descriptor part.
Select attribute msgid.

Step 8. Connect the clicked event of the Add button to the method putRequestMessage:

Connection: AddButton,clicked --> MQTodoListMsgIDClientView,putRequestMessage:

How to: Click the right mouse button.
Select Connect and then clicked.
Drop line to free-form surface.
Select Event to Script.
Highlight putRequestMessage: and click on OK.

Step 9. Connect the self attribute of RequestMessage to parameter1 of the connection line from the previous step.

Connection: ((AddButton,clicked --> MQTodoListMsgIDClientView,putRequestMessage:),parameter1 --> RequestMessage,self)

How to: Click the right mouse button.
Select Connect and then self.
Drop cursor on line.
Select parameter1.

Step 10. To disable the Connect button after the program is connected to a queue manager, connect the event connected of the Connection with the action disable of the push button.

Connection: Connection,connected --> ConnectButton,disable

How to: Move cursor over Connection part.
Click right mouse button.
Select Connect and the All Features.
Select event connected and click OK.
Drop line on ConnectButton.
Select All Features, then action disable and click on OK.

Step 11. Now we disable the Add button. Open setting of the AddButton and remove the check mark from Part enabled.

How to: Double-click on the Add button.
Go to page 3 of the settings notebook.

Step 12. Enable the Add button after the program is connected to the queue manager.
Connection: Connection, connected --> AddButton, enable

How to: Move cursor over Connection part.
- Click right mouse button.
- Select Connect and the All Features.
- Select event connected and click OK.
- Drop line on Add button.
- Select All Features, then action enable and click on OK.

This completes the client part. Your editor window should look like Figure 71 on page 98.

8.2.2 Add-Ons to the Server

The visual portion of the server part is exactly the same as the one in Figure 69 on page 94. To handle the message ID, we replace the put method with the following Smalltalk code:

```
Smalltalk

put

"get data from message, than add time stamp to the message and return it to the client"

| message serverMessage rc msgObject msgid |

msgObject := (self subpartNamed: 'Message').
message := msgObject asString trimNull.
msgid := msgObject msgId trimNull.
serverMessage := Date today printString, ' - msgid - ',
msgid, ' - ', message.
msgObject reset.
msgObject putString: serverMessage.
msgObject msgType: MqmtReply.
(rc := (self subpartNamed: 'Connection')
putMQMessage: msgObject) isAbtError
ifTrue: [
    Transcript show: 'Todo List Server put message request failed';
    cr.]
Transcript show: '------> Return code: ', rc codesAsString;
    cr.]
ifFalse: [Transcript show:
    'Todo List Server Put message is ok'; cr].
∧RC
```
8.3 How to Run the Application

Step 1. Connect both the client and server application to the queue manager and open the request and reply queues. This happens when you click on the Connect buttons.

If no error occurred, both check boxes for the queue managers will be ticked.

Step 2. Type some text in the New Item field and enter an ID in the MsgIDText field. This is the message ID for your client message.

Step 3. Click on the Add button in the client’s window. You should see in the System Transcript window a text similar to the one below:

Todo List Server Put msgid-111- is ok
Get message- ccccc- from client.

This message is produced by the Smalltalk method running in the server.

Step 4. The time-stamped new item will also be displayed in the list box in the client’s window. It is preceded by the date and the message ID.

The user interfaces are shown in Figure 70 on page 96.
Chapter 9. Getting a Particular Message

MQSeries allows you to get a particular message from a queue. To control this function, two fields from the message descriptor (MQMD) are used:

- Message ID (MsgId)
- Correlation ID (CorrelId)

MQSeries gets only those messages from the queue whose message ID and/or correlation ID match the one(s) specified in the MQMD structure. VisualAge for Smalltalk supports this MQSeries feature through model message.

In this section, we show you an example of how to set up your model message for selecting a particular message in a queue. Let’s start our example with two views:

- MQTodoListModelMsgIDClientView
- MQTodoListModelMsgIDServerView

Scenario

The client view sends a to-do item to the server by putting a message to a request queue. The server view receives the request, time stamps it, and sends it back to the client via a reply queue. The client view removes the message from the queue and displays its contents.

This scenario is the same as described in Chapter 8, “Extended To-Do List - Using Message ID” on page 95. However, we make some changes with regard to the message ID:

- The client sends the request message without assigning a message ID.
- The server randomly assigns one of three possible message IDs to the reply message.
- The client selects one of the three preset message IDs and gets only reply messages that match that ID.

9.1 How to Build the Example

If you are building this example from scratch, refer to Chapter 8, “Extended To-Do List - Using Message ID” on page 95 for step-by-step instructions. The following sections outline only the delta from that example.
9.1.1 Add-Ons to the Client View

Step 1. Delete the following two parts:
- The MsgIdText field to the right of To-Do List for ID
- The descriptor of RequestMessage part

This deletes the connections to and from the parts, too. The Composition Editor's window now shows this:

Step 2. Drop a MQSeries Message part on the free form surface and change its name to ModelMessage.

Step 3. Tear-off descriptor from ModelMessage.

Step 4. Drop a combo box on the window to replace the MsgIdText we just deleted. Change its name to MsgIdComboBox.
Step 5. Open the setting of(MsgIdComboBox) and type the values shown in
the notebook page below.

Step 6. Select String as data type and click OK to save the settings.

![Combo Box Specifications](image)

Step 7. The Composition Editor displays(MsgIdComboBox) as you see
below:

**To-Do List For ID:** [111]

Step 8. Connect(MsgIdComboBox) entryObject to the(msgid) attribute of
descriptor of(ModelMessage).

Step 9. Open the setting of the resulting connection line from above, check
the Read only source check box, and click OK to save the settings.

Now the client part is completed. It should look like Figure 72.

---

9.1.2 Changes to the Server Program

The visual portion of the server part is exactly the same as described in Chapter 8, “Extended To-Do List - Using Message ID” on page 95. As the only change, we have to replace the `put` method with the `putRandomMessageID` method shown in Figure 73 on page 107.
Smalltalk

putMessageWithRandomMsgId

"get data from message, add time stamp to the message, randomly assign a message id between 111, 222, & 333 to the message, then return the message to the client"

| message serverMessage rc msgObject msgid x |

msgObject := (self subpartNamed: 'Message').
message := msgObject asString trimNull.
Transcript show:
'Get message- ', message, '- from client.'; cr.
"below generate a random number between 0-2"
x := Time now seconds rem: 3.
msgid := #('111' '222' '333') at: x + 1.
serverMessage := Date today printString, '- msgid -', msgid,
'-' message
msgObject reset.
msgObject putString: serverMessage.
msgObject msgType: MqmtReply.
msgObject resetMsgid.
msgid := msgid.
(rc := (self subpartNamed: 'Connection') putMQMessage: msgObject)
isAbtError ifTrue: [
  Transcript show: 'Todo List Server put message request failed';
  cr.
  Transcript show: '----- Return code: ', rc codesAsString;
  cr.]
ifFalse: [Transcript show: 'Todo List Server Put msgId is ',
msgid; cr].
^rc

Figure 73. Getting a Particular Message: Server Program

9.2 How to Run the Example

First, make sure that the queue manager is running. Then connect both client and server to the queue manager and open request and reply queues. To do this, click on the Connect buttons in the two GUIs.

If no error occurred, the check boxes Queue manager is connected are checked in both GUIs. If not, make sure that you run in the right environment, MQM or MQIC.

From the combo box, select one of the three message IDs. The client view will only get messages with that ID from the reply queue.
Then type some text in the New Item field and click on the Add button.

![Figure 74. Getting a Particular Message: Windows at Run Time](image)

In the example shown, the client selected the message ID 222 and sent messages with the contents one, two, three and four.

The system transcript documents the activities of the server. For each message pair two lines are displayed:

- The text of the request messages it receives from the client
- The message ID it puts in the reply message

You can see that the server generated the message ID 222 once. The client gets only this message off the queue and displayed it in the client’s window.
Chapter 10. Syncpoint Processing

When you put a message on a queue or get a message from a queue, you can specify if the message is part of a unit of work. A unit of work contains one or more messages. All messages in a unit of work will only be committed together when a syncpoint is reached, that is, an MQCMIT (commit changes) is issued.

To make a message part of a unit of work, you specify one of the following options:

- MQPMO_SYNCPOINT (for put)
- MQGMO_SYNCPOINT (for get)

All messages put with syncpoint option will only be made available to other applications when a MQCMIT is issued. Likewise, all messages gotten with syncpoint option will only be deleted from a queue when an MQCMIT is issued.

Users can issue MQBACK to back out all message gets and puts that have occurred since the last syncpoint. Messages put as part of a unit of work are deleted from queue; messages retrieved as part of a unit of work are reinstated on the queue.

VisualAge MQSeries parts supports syncpoint processing in both visual parts and Smalltalk script level. In this chapter, we show you how to do syncpoint processing using VisualAge for Smalltalk.

10.1 Application Description

We extend the example in Chapter 7, “To-Do List - A Client/Server Application” on page 79 to illustrate the syncpoint processing. If you haven’t already read that chapter yet, here is a brief summary of the application:

A client/server pair are both connected to a queue manager. The client sends to-do items to the server via a request queue. The server gets the to-do items, time stamps them, then sends them back, via a reply queue, to the client.

In this chapter, we add MQSeries’s syncpoint processing to the application. We invoke those functions through two push buttons, Commit and Backout.
• The client can send several new items and commit them all together, thus making them available to the server by one click on the Commit button.

• The client can also back out all messages it sent since its last commit by clicking the Backout button, so that those to-do items will not be seen by the server.

10.2 How to Build the Example

We are going to reuse the client/server parts we build in Chapter 7, “To-Do List - A Client/Server Application” on page 79. You may follow the instructions in that chapter to build this example from scratch, or simply copy the parts and modify them to fit this example. Let’s do the latter and name our new parts:

• ToDoListSyncPointClientView
• ToDoListSyncPointServerView

Once you have the To-Do List classes up, you can modify them as follows:

1. Change the window title of the ToDo List SyncPoint Client View.

2. Add two push buttons: CommitButton and BackoutButton.

3. Connect CommitButton **clicked** to Connection **commit**.

4. Connect BackoutButton **clicked** to Connection. **backout**

5. Double-click on **ConnectionSpec** to open its settings and mark the check box Syncpoint processing.

![Diagram showing queue manager and connection operations](image)

*Figure 75. Syncpoint Processing Example*
Syncpoint processing may seem complicated, but adding it via VisualAge for Smalltalk is just as simple as we just showed you. Figure 75 shows the push buttons and connections you have to add.

10.3 What about the Server?

The server view we use for this application is exactly the same as the one for the to-do list application shown in Figure 67 on page 90. You can use it without any change.

The reason why we don’t need to change the server part is that we activate the syncpoint processing only on the client side, not the server side. If you choose to add syncpoint processing to the server, you need to do the same simple things we have done to the client view.

10.4 How to Demonstrate Syncpointing

In this section, we describe how to run the to-do list client and server to demonstrate syncpointing with MQSeries.

1. Bring up both client and server views.

2. Connect both views to the queue manager and make sure that both are connected. In both windows, the check box Queue Manager is connected must be checked.

3. In the client window, type some text in the New Item field and then click the Add button.
   
   **Note:** Nothing should appear in the System Transcript, which is used by the server when it receives a to-do item message.

4. Add another to-do item and click on the Add button.
   
   **Note:** Again, check the system transcript to see if the server has written anything to it. You should not see any message.

5. Click the Commit button in the client’s window.
   
   **Note:** Now you see two messages in the system transcript, written to it by the server. After that, you will also see those to-do items displayed in the client’s to-do list.

6. You may repeat the same scenario for the backout.

Do you notice that when you back out your sent messages, the to-do item list keeps growing with some duplicated items? Do you want to report a program bug to VisualAge for Smalltalk or MQSeries? Before you decide to do so, please read on.
MQSeries commit and backout are done on queue manager level. So, when you do backout, it affects not only the get actions, but also the put actions. In VisualAge for Smalltalk, it affects both the REQUEST and the REPLY queues.

In this example, you control the put message to the request queue by clicking the Add button. However, the get message from the client’s reply queue happens automatically. So, when you back out of your add action, MQSeries, under the cover, also backs out all messages you have gotten before from the reply queue. Therefore, they are read in by VisualAge for Smalltalk again.

One last word of caution before you move on to another topic. If you are running this example in a Windows 95 environment, you may find discrepancies in the end result. We believe that is a limitation of MQSeries support on the Windows 95 platform.
Chapter 11. Hotel Reservation Application

In this chapter, we create another more complicated application using VisualAge MQSeries parts. It demonstrates how objects are passed from VisualAge for Smalltalk to MQSeries and vice versa.

Note: MQSeries sends messages, not objects. We need to convert objects to messages and messages to objects.

In the following sections, we explain how to:

• Pass objects using a user-written method
• Pass objects in its string format using a method provided by Smalltalk
• Pass objects using ObjectDumper and ObjectLoader

11.1 Application Overview

We create a hotel reservation application that consists of a client and a server. The client sends a hotel reservation request to a server. The server reserves a room and sends a confirmation number back to the client.

We have two business objects:

• HotelReq contains details of a hotel reservation.
• HotelRsv contains details of a hotel reservation confirmation.

Figure 76 on page 114 provides an overview of the application. It shows that we have three layers on top of the VisualAge MQSeries parts framework and base product: a view layer, a business object layer, and a service object layer.

In the service layer (MQ Broker), we map objects to messages.

Let’s start with the creation of the business objects.

Notice

The examples have been tested to the extent that they demonstrate the intended functions. To make the programs easier to understand error checking functions are kept to a minimum. Follow the steps closely to avoid the walkback window.
11.2 Defining Business Objects

Create two business objects, HotelReq and HotelRsv, inherited from the object class. Both have some attributes of their own, such as the client’s name and the number of nights to reserve. The following two tables list the attributes of the two business objects and their data type.

**Table 6. Hotel Reservation: HotelReq Attributes**

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Attribute Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>checkInDate</td>
<td>String</td>
</tr>
<tr>
<td>name</td>
<td>String</td>
</tr>
<tr>
<td>no</td>
<td>String</td>
</tr>
<tr>
<td>noOfNights</td>
<td>String</td>
</tr>
<tr>
<td>noOfRooms</td>
<td>String</td>
</tr>
<tr>
<td>place</td>
<td>String</td>
</tr>
<tr>
<td>type</td>
<td>String</td>
</tr>
</tbody>
</table>

---

![Diagram showing the relationship between HotelReqClient and HotelRsvServer objects with connectors for MQ/OS and MQBroker.

**Figure 76. Hotel Reservation: Objects**
You define attributes using the Public Interface Editor shown in Figure 77. Enter an attribute name, enter the attribute data types, in this case is String, and click on the **Add with defaults** push button. The remaining fields will be filled in. After defining all attributes, generate default scripts by selecting **Generate Default Scripts** from the File menu. VisualAge for Smalltalk generates a getter and a setter, which are used to automatically get and set the value of attributes.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Attribute Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>confirmationNo</td>
<td>String</td>
</tr>
<tr>
<td>hotel</td>
<td>String</td>
</tr>
<tr>
<td>name</td>
<td>String</td>
</tr>
<tr>
<td>no</td>
<td>String</td>
</tr>
<tr>
<td>noOfRooms</td>
<td>status</td>
</tr>
<tr>
<td>place</td>
<td>String</td>
</tr>
</tbody>
</table>

You define attributes using the Public Interface Editor shown in Figure 77. Enter an attribute name, enter the attribute data types, in this case is String, and click on the **Add with defaults** push button. The remaining fields will be filled in. After defining all attributes, generate default scripts by selecting **Generate Default Scripts** from the File menu. VisualAge for Smalltalk generates a getter and a setter, which are used to automatically get and set the value of attributes.

![Table 7. Hotel Reservation: HotelRsv Attributes](image)

![Figure 77. Hotel Reservation: Public Interface Editor](image)
These objects are passed using MQSeries. The client sends the object HotelReq to a server. The server sends the object HotelRsv to the client after the hotel reservation has been made.

As in the previous examples, the client uses the REQUEST queue to send a hotel request (object HotelReq) to the server. The server gets the request from that queue, processes it, and puts the confirmation (object HotelRsv) on the REPLY queue.

11.3 Client Application

The client application uses three forms, one to input the hotel request, one that displays the reply from the server, and the third to connect to and disconnect from MQSeries. Each form has three push buttons on it that allow you to switch from one form to another. Figure 78 shows the base window used for the client views.

![Figure 78. Hotel Reservation: Base Form for MQHotelReqView](image)

In the next sections we explain two of the the forms, HotelReqForm and HotelRsvForm. The MQClient for pertaining to MQSeries connect and
disconnect and connection activities is similar to that of the previous examples. Figure 93 on page 127 shows the window at run time.

The two forms shown in Figure 79 and Figure 80 on page 118 are examples of reusable visual parts. You can add these visual parts to any other visual part to build a more complex view.

**How to create a reusable part:** After opening a part, remove the default window from the composition editor. Select **Canvas** from the pallet and then select **Form**. Drop the form on the free-form surface. This form part is now a reusable part and you can add it to any window.

**Variable part:** A variable part can be used to hold an object, just like variables in other programming languages. These variables can be used in the methods defined to the class.

**Deferred part:** To minimize the object updating activities derive from GUI input, we use deferred parts. Each time the user types data into an entry field, an event occurs. Every event is sent to the HotelReq object to update the associated attributes. Using a deferred part, the event occurs only then when the changes are applied, thus updating all variables at once.

![Figure 79. Hotel Reservation: HotelReqForm](image-url)
How to create a form: To create a form, you can use quick form from the HotelReq or HotelRsv objects. Quick form automatically places entry fields and labels for them into the form and makes the required connections between entry fields and variable objects. Quick form allows you to make GUI prototypes quickly. After its creation, you may change entry fields to combo boxes or spin buttons, depending on your preferences.

The hotel reservation form in Figure 80 has two variables, HotelRsv1 and HotelReq1. These variables hold data input from the windows. The lines between entry fields and variables are attribute to attribute connections. Such a connection updates a variable when an entry field has been changed.

![HotelRsvForm - Composition Editor](image)

Figure 80. Hotel Reservation: HotelRsvForm

For HotelRsvForm, we do not need a deferred part since the form displays read-only fields, which cannot be changed by the user. Therefore, there is no update activities to minimize.

On the client side, we need to convert user input to MQ Message before we send it to the server. We show you how to do that right after our discussing of the server application.
11.4 Server Application

The server application consists of a non-visual part MQHotelRsvServer and a visual part MQHotelRserverView. The visual part contains the GUI only. Most of the server logic is in the non-visual part.

Figure 81. Hotel Reservation: Non-Visual Server Part

Figure 82. Hotel Reservation: Server - Public Interface Editor
The non-visual part MQHotelRsvServer in Figure 81 connects to and disconnects from a queue manager, gets requests and puts replies. This part is used by the MQHotelRsvServerView window in Figure 83 on page 120.

The #start and #end methods are public and are defined in the Public Interface Editor. The methods are used in MQHotelRsvServerView after the MQHotelRsvServer object has been added to the free-form surface. All MQSeries related actions are encapsulated in the MQHotelRsvServer part.

![Diagram of MQHotelRsvServerView](image)

*Figure 83. Hotel Reservation: MQHotelRsvServer*

### 11.5 Object to Message Conversion

In the following, we describe how to send objects using MQSeries. This can be done in three ways:

- Write a conversion method to convert each attribute to its string format and string them together using a field separator or a delimiter such as $ or & character.
- Use storeString provided by VisualAge to convert objects to their string formats.
- Use ObjectDumper to convert objects to streams and ObjectLoader to reconstruct them from their stream formats.
11.5.1 Contents Method

In our client application MQSeries Connection part putRequest: method is used. It performs the following functions:

- If a message is of the type string then it is passed as is.
- If a message is not a string type, then the contents method is called. The contents method returns a byte array.

For our application, we write our own contents method to return a ByteArray to our business object classes, HotelReq and HotelRsv. The $ character is used as separator between attributes. Figure 84 shows the contents and contentsAsString methods of the HotelReq class.

```
HotelReq publicMethods
contents
"return ByteArray of a flatted object."
| contents |
contents := self contentsAsString asByteArray.
^contents

contentsAsString
"return a String of a flatted object."
| contents |
contents := self name, ",", self place, ",", self type, ",", self checkInDate, ",", self noOfNights, ",", self noOfRooms, ",", self no, ".
^contents
```

Figure 84. Hotel Reservation: Contents Method

To put a message into the request queue, you need to call putRequest method. In the VisualAge MQSeries parts framework, the contents method of a class is called automatically before putting an object into a queue. Figure 85 on page 122 shows how to pass a HotelReq object using putRequest method.
To recreate an object from an MQSeries message, you will need to parse the message with a string separator. The following is a sample method that creates an object from an MQSeries message.

```
Smalltalk

HotelReqForm publicMethods

putObject

mqConnection putRequest: self hotelReq.
(self subpartNamed: 'messageLabel') object:
'Request has been submitted'.
```

*Figure 85. Hotel Reservation: putRequest Method*

```
Smalltalk

HotelReq class publicMethods

newFromString: aString

"new from a flatted object."

| req ans |

req := HotelReq new.

ans := aString subStrings: $,.
req name: (ans at: 1).
req place: (ans at: 2).
req type: (ans at: 3).
req checkInDate: (ans at: 4).
req noOfNights: (ans at: 5).
req noOfRooms: (ans at: 6).
req no: (ans at: 7).
∧req
```

*Figure 86. Hotel Reservation: Re-create an Object*

In the server application, the method in Figure 87 on page 123 is called to recreate HotelReq object from a message.
Smalltalk

MQHotelRsvServer publicMethods

reserveHotel: aReq

| req reply |

    req := HotelReq newFromString: aReq.
    reply := HotelRsv reserveHotel: req.
    ∧reply

Figure 87. Hotel Reservation: Re-create an Object

11.5.2 storeString Method

To convert an object to its string format, the object class in IBM Smalltalk provides storeString method. To restore an object from its string format, use the evaluate: method from the IBM Smalltalk Compiler class. Figure 88 shows how to use these methods.

Smalltalk

HotelReq class publicMethods

newFromString: aString
    "new from a flatted object."
    | req |

    req := Compiler evaluate: aString.
    ∧req

HotelReq publicMethods

contentsAsString
    "return a String of a flatted object."
    | contents |

    contents := self storeString.
    ∧contents

Figure 88. Hotel Reservation: Re-create an Object
11.5.3 ObjectDumper and ObjectLoader

Smalltalk

HotelReq publicMethods
dumpObject

| dumper estimatedSize byteArrays |

dumper := ObjectDumper new.
estimatedSize := dumper totalSizeBeforeUnload: self.
byteArrays := LibraryObjects byteArraysForObjectSize:
estimatedSize.
dumper
unload: self
  intoByteObjects: byteArrays
  offsetsIntoByteObjects: 0
  maximumLimit: estimatedSize
  errorStream: nil.
∧byteArrays

Figure 89. Hotel Reservation: ObjectDumper

Smalltalk

HotelReq class publicMethods

loadObject: aByteArray

| loader |

loader := ObjectLoader new.
∧loader
  loadFromByteObjects: aByteArray
  offsetsIntoByteObjects: 0
  errorStream: nil.

newFromString: aString
  "new from a flatted object."
| req |

req := self loadObject: aString.
∧req

Figure 90. Hotel Reservation: ObjectLoader

ObjectDumper is used to save your object to a file. Some methods are provided to return an array of the type ByteArray instead of a file. In this example, we use the following to convert an object to ByteArray format:
To restore an object, we use the method:
loadFromByteObjects:offsetsIntoObjects:errorStream: ObjectLoader

We recommend you use the ObjectDumper and ObjectLoader way to convert between an object and its message format. It is more complicated then using string separator, but it is more flexible. The storeString and evaluate: methods require a lot of system resource at run time because you must include a compiler in your run-time image.

11.6 Executing the Application

This section explains how to run the Hotel reservation application. You need to follow the step-by-step instructions closely, or you will get either walkbacks (debug window) or unpredictable results.

1. Test the MQHotelRsvServerView.
2. Click the Start button and you see a work space with connection status for the server.
3. Test the MQHotelReqView.
4. Click the MQ Setting button. That is the third one in the tool box on top of the window.
5. In the MQ Info window, click the Start button to connect the client.
6. Click the Hotel Request button which is the left one on the HotelReqView and you see a Hotel Request with data entry fields.
7. Fill in all fields and click the Reserve button.
8. You should see a message saying Request has been submitted on the client side, and Get message... in the workspace of the server.
9. Click the Hotel Confirmation button in the client view to see status and confirmation information.
10. If everything is OK, you may click the Next button in the client’s window and make another reservation.
11. If you decide to quit, you will need to end the server’s and then end the client’s connection to the queue manager.

Figure 91 on page 126 shows the window used to submit a request.
Figure 91. Hotel Reservation: Input Request

Figure 92 shows the confirmation the client receives.

Figure 92. Hotel Reservation: Get Confirmation

The workspace window shows activity of a server program. A server program gets a request and puts a message to REPLY queue.
Using the MQ Setting window, a client program can connect to queues and disconnect.

To see the complete application, import the following applications to your VisualAge for Smalltalk image.

<table>
<thead>
<tr>
<th>Application Name</th>
<th>Contents</th>
</tr>
</thead>
</table>
| MQHotelRsvApp 1.0 | Hotel Reservation Application using user defined string method.  
The code is in Appendix B on page 135. |
| MQHotelRsvApp 2.0 | Hotel Reservation Application using storeString method.  
The code is in Appendix C on page 151. |
| MQHotelRsvApp 3.0 | Hotel Reservation Application using ObjectDumper and ObjectLoader.  
The code is in Appendix D on page 153. |
Appendix A. Setup for Examples

All examples use the same MQSeries objects. What objects you use depends on your environment, that is, if client and server programs run in the same machine or in different ones. In all cases you need:

- **VAQMGR** Name of the queue manager
- **REQUEST** Name of the request queue
- **REPLY** Name of the reply queue

Notes:
1. These names are specified in the ProcDialog settings (for example 1) and in the Connection Spec settings (for all other examples). Don’t change them unless you change the programs.
2. If you run client and server in different machines you need:
   - Another queue manager with the name YURIKO
   - To know the host names or IP addresses of both client and server

To set up the queue manager and create its default objects execute the following commands:
```
crtmqm VAQMGR
strmqm VAQMGR
runmqsc VAQMGR < d:\mqm\mqsc\amqscexample.tst > a.a
```

Note: On the server, replace VAQMGR with YURIKO.

What objects you need for your application depends on the environment.
1. Both VisualAge client and server run in the same machine. That’s the usual test environment.
2. VisualAge client and server run in different machines, and both machines have a queue manager installed. That may be two systems used for program development.
3. The VisualAge client runs in a machine that does not have a queue manager installed. All MQSeries objects are in a server. You use the MQSeries client software to connect to the server.
A.1 Using One System

Execute the following command to create the MQSeries objects:

runmqsc VAQMGR < d:\test\localdef.tst > a.a

Contents of the file LOCALDEF.TST:

```
DELETE QREMOTE(REQUEST)
DEFINE QLOCAL (REQUEST) REPLACE +
  DESCR('Request queue')
DEFINE QLOCAL (REPLY) REPLACE +
  DESCR('Reply queue')
```

Note: Check the file a.a for any errors.

Before you run your applications, make sure that the queue manager is running. Execute this command:

strmqm VAQMGR

Note: If VAQMGR is the default queue manager, you don’t have to specify the name in any command.

You may want to check some MQSeries objects, such as finding out how many messages are in the queues. Execute this command:

runmqsc VAQMGR < commands.in > a.a

Contents of the file COMMANDS.IN:

```
dis qmgr ALL
  dis q (*) CURDEPTH
  dis process(*) ALL
```
A.2 Using MQSeries Client and Server

In the server, execute the following command to create the MQSeries objects:

```sh
runmqsc VAQMGR < d:\test\localdef.tst > a.a
```

Contents of the file LOCALDEF.TST:

```
DELETE QREMOTE(REQUEST)
DEFINE QLOCAL (REQUEST) REPLACE +
    DESCR('Request queue')
DEFINE QLOCAL (REPLY) REPLACE +
    DESCR('Reply queue')
```

Notes:

1. Check the file a.a for any errors.
2. Queue manager and queues used by the client run in the server.
3. The client application runs in the client’s machine, the server application runs in the server. Therefore, the MQSeries set up is the same as for using one system.
4. The delete command is only effective when you used two machines and switch back to run client and server application in one system.

In the client, you must set an environment variable to connect to the queue manager in the server.

```sh
set MQSERVER=VACH1/TCP/9.24.123.1234(1414)
```

Note: You must know either the IP address or machine name of the server.

Before you run your applications, make sure that the queue manager is running in the server machine. Execute this command:

```sh
start runmqtsr -t tcp
```
A.3 Using Two MQSeries Servers (MQM-to-MQM)

In this environment, the VisualAge client program and the VisualAge server program run in different systems. Both systems are MQSeries servers. Each of them has its own set of objects defined.

In the client, execute this command:

```bash
runmqsc VAQMGR < d:\test\client.tst > a.a
```

In the server, execute this command:

```bash
runmqsc YURIKO < d:\test\server.tst > a.a
```

**Note:** Make sure that both queue managers are running.

Before you start your programs execute the following commands:

- On the client:

  ```bash
  strmqm VAQMGR
  start runmqlsr /t tcp
  start runmqchi
  ```

- On the server:

  ```bash
  strmqm YURIKO
  start runmqlsr /t tcp
  start runmqchi
  ```

**Note:** You may put the commands in a command file.

Next, start the programs in both systems.

Make sure you click on the **Connect** button to connect to the queue managers before you send any messages.
Contents of the file CLIENT.TST:

```
ALTER QMGR DEADQ(SYSTEM.DEAD.LETTER.QUEUE)

DELETE QLOCAL REQUEST

DEFINE QREMOTE REQUEST REPLACE +
  RNAME('REQUEST') +
  RMQNAME(YURIKO) +
  XMITQ(CLIENT.TO.SERVER) +
  DESCR('Request queue for server program')

DEFINE QLOCAL CLIENT.TO.SERVER REPLACE +
  USAGE(xmitq) +
  TRIGGER TRIGTYPE(every) +
  INITQ(SYSTEM.CHANNEL.INITQ) +
  PROCESS(CHANNEL.TO.YURIKO) +
  DESCR('Xmit queue for messages to server')

DEFINE PROCESS(CHANNEL.TO.YURIKO) REPLACE +
  USERDATA(VAQMGR.TO.YURIKO) +
  DESCR('Process for starting channel')

DEFINE CHANNEL(VAQMGR.TO.YURIKO) +
  CHLTYPE(sdr) REPLACE +
  TRPTYPE(tcp) CONNAME(9.24.123.123) +
  XMITQ(CLIENT.TO.SERVER) +
  DESCR('Sender channel from VAQMGR to YURIKO')

DEFINE QLOCAL('REPLY') REPLACE +
  DESCR('Reply queue for client program')

DEFINE CHANNEL(YURIKO.TO.VAQMGR) +
  CHLTYPE(rcvr) REPLACE +
  TRPTYPE(tcp) +
  DESCR('Receiver channel from YURIKO to VAQMGR')
```
Contents of the file SERVER.TST:

```
ALTER QMGR DEADQ(SYSTEM.DEAD.LETTER.QUEUE)

DEFINE QLOCAL('REQUEST') REPLACE +
    DESCR('Request queue for server program')

DEFINE CHANNEL(VAQMGR.TO.YURIKO) +
    CHLTYPE(rcvr) REPLACE +
    TRPTYPE(tcp) +
    DESCR('Receiver channel from VAQMGR to YURIKO')

DEFINE QREMOTE('REPLY') REPLACE +
    RNAME('REPLY') +
    RQMNAME(VAQMGR) +
    XMITQ(SERVER.TO.CLIENT) +
    DESCR('Request queue for client program')

DEFINE QLOCAL(SERVER.TO.CLIENT) REPLACE +
    USAGE(xmitq) +
    TRIGGER TRIGTYPE(every) +
    INITQ(SYSTEM.CHANNEL.INITQ) +
    PROCESS(CHANNEL.TO.VAQMGR) +
    DESCR('Xmit queue')

DEFINE PROCESS(CHANNEL.TO.VAQMGR) REPLACE +
    USERDATA(YURIKO.TO.VAQMGR) +
    DESCR('Process for starting channel')

DEFINE CHANNEL(YURIKO.TO.VAQMGR) +
    CHLTYPE(sdr) REPLACE +
    TRPTYPE(tcp) CONNAME(9.24.123.123) +
    XMITQ(SERVER.TO.CLIENT) +
    DESCR('Sender channel from YURIKO to VAQMGR')
```
Appendix B. Hotel Reservation Application Smalltalk Code 1.0

This appendix contains the code for Version 1, using user-defined methods to convert objects to message and vice versa. There are three sections:

- Business objects
- Client application classes
- Server application classes

B.1 Business Objects

This application has two business object classes:

- HotelReq class (reservation request)
- HotelRsv class (reservation confirmation)

B.1.1 HotelReq Class

Object subclass: #HotelReq

instanceVariableNames: 'checkInDate place noOfNights noOfRooms no name type'
classVariableNames: 'HotelReqs'
poolDictionaries: ''

HotelReq class publicMethods

addHotelReq

    self hotelReqs add: (HotelReq new no: self hotelReqs size printString).

^HotelReqs last

addHotelReq: aHotelReq

    self hotelReqs add: aHotelReq.

^HotelReqs

checkHotelReqByNo: aNumber

    ^self hotelReqs detect: [ :e | e no = aNumber].
hotelReqs
  HotelReqs isNil
    ifTrue: [HotelReqs := OrderedCollection new].
  \^HotelReqs
initialize
  HotelReqs := OrderedCollection new.
newFromString: aString
  "new from a flatted object."
  | req ans |
  req := HotelReq new.
  ans := aString subStrings: $,.
  req name: (ans at: 1).
  req place: (ans at: 2).
  req type: (ans at: 3).
  req checkInDate: (ans at: 4).
  req noOfNights: (ans at: 5).
  req noOfRooms: (ans at: 6).
  req no: (ans at: 7).
  \^req
HotelReq publicMethods
checkInDate
  "Return the value of checkInDate."
  \^checkInDate
checkInDate: aString
  "Save the value of checkInDate."
  checkInDate := aString.
  self signalEvent: #checkInDate
    with: aString.
contents
  "return ByteArray of a flatted object."
  | contents |
  contents := self contentsAsString asByteArray.
  \^contents
contentsAsString
"return a String of a flatted object."
| contents |

contents := self name, ".", self place, ".", self type, ".", self checkInDate, ".", self noOfNights, ".", self noOfRooms, ".", self no, ".".
∧ contents

name
"Return the value of name."
∧ name

name: aString
"Save the value of name."
name := aString.
self signalEvent: #name
with: aString.

no
"Return the value of no."
∧ no

no: aString
"Save the value of no."
no := aString.
self signalEvent: #no
with: aString.

noOfNights
"Return the value of noOfNights."
∧ noOfNights

noOfNights: aString
"Save the value of noOfNights."
noOfNights := aString.
self signalEvent: #noOfNights
with: aString.
noOfRooms
    "Return the value of noOfRooms."
    ^noOfRooms

noOfRooms: aString
    "Save the value of noOfRooms."
    noOfRooms := aString.
    self signalEvent: #noOfRooms
        with: aString.

place
    "Return the value of place."
    ^place

place: aString
    "Save the value of place."
    place := aString.
    self signalEvent: #place
        with: aString.

type
    "Return the value of type."
    ^type

type: aString
    "Save the value of type."
    type := aString.
    self signalEvent: #type
        with: aString.
B.1.2 HotelRsv Class

Object subclass: #HotelRsv
  instanceVariableNames: 'status name hotel confirmationNo no'
  classVariableNames: 'Rand'
  poolDictionaries: ''

HotelRsv class publicMethods

newFromString: aString
  "new from a flatted object."
  | rsv ans |
  rsv := HotelRsv new.
  ans := aString subStrings: $,
  rsv status: (ans at: 1).
  rsv name: (ans at: 2).
  rsv hotel: (ans at: 3).
  rsv confirmationNo: (ans at: 4).
  rsv no: (ans at: 5).
  ^rsv

rand

  Rand isNil
    ifTrue: [Rand := EsRandom new].

  ^Rand

rand: aNo

  Rand := aNo.

reserveHotel: aReq
  | reply |
  reply := HotelRsv new
    name: aReq name;
    hotel: 'Residency Inn';
    status: 'success';
    confirmationNo: (10000 * self rand next) floor asInteger printString;
    no: aReq no.
  ^reply
HotelRsv publicMethods

confirmationNo

"Return the value of confirmationNo."

∧confirmationNo

confirmationNo: aString

"Save the value of confirmationNo."

confirmationNo := aString.
self signalEvent: #confirmationNo
with: aString.

contents

"return ByteArray of a flattened object."

| contents |

contents := (self status, ',', self name, ',', self hotel, ',', self confirmationNo, ',', self no, ',') asByteArray.

∧contents

hotel

"Return the value of hotel."

∧hotel

hotel: aString

"Save the value of hotel."

hotel := aString.
self signalEvent: #hotel
with: aString.

name

"Return the value of name."

∧name

name: aString

"Save the value of name."

name := aString.
self signalEvent: #name
with: aString.
no
   "Return the value of no."

^no

no: aString
   "Save the value of no."

   no := aString.
   self signalEvent: #no
       with: aString.

status
   "Return the value of status."

^status

status: aString
   "Save the value of status."

   status := aString.
   self signalEvent: #status
       with: aString.
B.2 Client Application Classes

The client application has four classes:

- HotelReqForm
- HotelRsvForm
- MQClientForm
- MQHotelReqView

B.2.1 HotelReqForm Class

AbtAppBldrView subclass: #HotelReqForm

    instanceVariableNames: 'mqConnection hotelReq'
    classVariableNames: ''
    poolDictionaries: ''

HotelReqForm publicMethods

    clearText

        (self subpartNamed: 'nameField') object: ' '.
        (self subpartNamed: 'checkinDateField') object: ' '.
        (self subpartNamed: 'placeField') object: ' '.

    hotelReq

        "Return the value of hotelReq."

        ^hotelReq

    hotelReq: aHotelReq

        "Save the value of hotelReq."

        hotelReq := aHotelReq.
        self signalEvent: #hotelReq
        with: aHotelReq.

    mqConnection: anObject

        "Save the value of mqConnection."

        mqConnection := anObject.
        self signalEvent: #mqConnection
        with: anObject.
next

    self hotelReq: HotelReq addHotelReq.
    self clearText.
    self setObjects.

put

    mqConnection putRequest: ((self subpartNamed: 'nameField') object).

putObject

    mqConnection putRequest: self hotelReq.
    (self subpartNamed: 'messageLabel') object: 'Request has been submitted'.

setObjects

    self hotelReq isNil
        ifTrue: [self hotelReq: HotelReq addHotelReq].

    (self subpartNamed: 'Hotel Req1') value: self hotelReq.
    (self subpartNamed: 'messageLabel') object: 'New Request'.

B.2.2 HotelRsvForm Class

AbtAppBldrView subclass: #HotelRsvForm
    instanceVariableNames: 'hotelRsv hotelReq'
    classVariableNames: ''
    poolDictionaries: ''

HotelRsvForm publicMethods

clear

    (self subpartNamed: 'Hotel Rsv1') value: nil.
    (self subpartNamed: 'Hotel Req1') value: nil.

    (self subpartNamed: 'checkInDateField') object: ''. 
    (self subpartNamed: 'confirmationNoField') object: ''. 
    (self subpartNamed: 'hotelField') object: ''. 
    (self subpartNamed: 'nameField') object: ''. 
    (self subpartNamed: 'noOfNightsField') object: ''. 
    (self subpartNamed: 'noOfRoomsField') object: ''. 
    (self subpartNamed: 'statusField') object: ''. 

Appendix B. Hotel Reservation Application Smalltalk Code 1.0 143
hotelRsv
   "Return the value of hotelRsv."

hotelRsv isNil
   ifTrue: [hotelRsv := HotelRsv new].

∧hotelRsv

hotelRsv: aHotelRsv
   "Save the value of hotelRsv."

hotelRsv := aHotelRsv.
self signalEvent: #hotelRsv
   with: aHotelRsv.

setResult
   (self subpartNamed: 'Hotel Rsv1') value: (self hotelRsv).

hotelReq := HotelReq checkHotelReqByNo: hotelRsv no.
   (self subpartNamed: 'Hotel Req1') value: hotelReq.

B.2.3 MQClientForm Class

AbtAppBlrdView subclass: #MQClientForm
   instanceVariableNames: 'message'
   classVariableNames: ''
   poolDictionaries: ''

MQClientForm publicMethods

getMessage
   | hotelRsv hotelRsvForm |


   hotelRsv := HotelRsv newFromString: message.

   hotelRsvForm := self parentPart components at: 'Hotel Rsv Form'.
   hotelRsvForm hotelRsv: hotelRsv;
   setResult.
message
"Return the value of message."
∧message

message: aHotelRsv
"Save the value of message."

message := aHotelRsv.
self signalEvent: #message
    with: aHotelRsv.

setNames

(self subpartNamed: 'QMGRLabel') object: 'Queue Manager: ',
((self subpartNamed: 'MQ Series Connection Spec1') queueManagerName).
(self subpartNamed: 'reqQLabel') object: 'Request Queue: ',
((self subpartNamed: 'MQ Series Connection Spec1') requestQueueName).
(self subpartNamed: 'repQLabel') object: 'Reply Queue: ',
((self subpartNamed: 'MQ Series Connection Spec1') replyQueueName).

B.2.4 MQHotelReqView Class

AbtAppBldrView subclass: #MQHotelReqView
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''

MQHotelReqView publicMethods

displayHotelReq

    (self subpartNamed: 'MQ Form') hide.
    (self subpartNamed: 'Hotel Rsv Form') hide.
    (self subpartNamed: 'Hotel Req Form') show.

displayHotelRsv

    (self subpartNamed: 'MQ Form') hide.
    (self subpartNamed: 'Hotel Req Form') hide.
    (self subpartNamed: 'Hotel Rsv Form') show.

displayMQ

    (self subpartNamed: 'Hotel Req Form') hide.
    (self subpartNamed: 'Hotel Rsv Form') hide.
    (self subpartNamed: 'MQ Form') show.
setForms

| hotelReqForm mqForm hotelRsvForm |

hotelReqForm := HotelReqForm new.
self setViewFramingSpec: hotelReqForm.

mqForm := MQClientForm new.
self setViewFramingSpec: mqForm.

hotelRsvForm := HotelRsvForm new.
self setViewFramingSpec: hotelRsvForm.

hotelReqForm mqConnection:
    (mqForm subpartNamed: 'MQ Series Connection1') yourself.

(self subpartNamed: 'Form1')
subpartNamed: 'Hotel Req Form' put: hotelReqForm;
subpartNamed: 'Hotel Rsv Form' put: hotelRsvForm;
subpartNamed: 'MQ Form' put: mqForm.

hotelReqForm
hide;
openWidget.

mqForm
hide;
openWidget.

hotelRsvForm
hide;
openWidget.

self displayHotelReq.

setViewFramingSpec: aForm
"set address view framing spec"

aForm framingSpec
leftEdge: (AbtRunEdgeAttachmentConstraint new attachment:
         XmATTACHFORM; offset: 0);
rightEdge: (AbtRunEdgeAttachmentConstraint new attachment:
          XmATTACHFORM; offset: 0);
topEdge: (AbtRunEdgeAttachmentConstraint new attachment:
          XmATTACHFORM; offset: 0);
bottomEdge: (AbtRunEdgeAttachmentConstraint new attachment:
           XmATTACHFORM; offset: 0).
B.3 Server Application Classes

The server application consists of two classes:

- MQHotelRsvServer
- MQHotelRsvServerView

B.3.1 MQHotelRsvServer Class

AbtAppBldrPart subclass: #MQHotelRsvServer
    instanceVariableNames: 'ws queueManager'
    classVariableNames: ''
    poolDictionaries: ''

MQHotelRsvServer publicMethods

connect

    | rc |
    (rc := queueManager connectUsing:
        ((self subpartNamed: 'MQ Series Connection Spec1') yourself ) ) isAbtError
        ifTrue: [ 
            ws show: 'Connect request failed'; cr.
            ^ws show: '----> Return code: ', rc codesAsString; cr.]
        ifFalse: [ws show: 'Connect is ok'; cr].
        ^rc

disconnect

    | rc |
    (rc := queueManager disconnect) isAbtError
        ifTrue: [ 
            ws show: 'Disconnect request failed'; cr.
            ^ws show: '----> Return code: ', rc codesAsString; cr.]
        ifFalse: [ws show: 'Disconnect is ok'; cr].
        ^rc

end

self disconnect.
put
| message rc reply |
message := (self subpartNamed: 'MQ Series Message1') asString.
ws show: 'Get message "". message,"" from client.'; cr.

reply := self reserveHotel: message.

(rc := (self subpartNamed: 'MQ Series Connection1') putReply: reply) isAbtError
  ifTrue: [
    ws show: 'Put message request failed'; cr.
    ws show: '-----> Return code: ', rc codesAsString; cr.]
  ifFalse: [ws show: 'Put message is ok'; cr].

reserveHotel: aReq
| req reply |
req := HotelReq newFromString: aReq.
reply := HotelRsv reserveHotel: req.

setUpWorkspace
"Set up workspace."

ws isNil
  ifTrue: [
    ws := EtWorkspace new
    label: 'Where am I? MQ HotelRsvServer';
    open].

start
| rc |
self setUpWorkspace.
ws show: 'Now start a server program.'; cr.
queueManager := (self subpartNamed: 'MQ Series Connection1') yourself.

self connect isAbtError
  ifTrue: [
    self disconnect].
B.3.2 MQHotelRsvServerView Class

AbtAppBldrView subclass: #MQHotelRsvServerView
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''

MQHotelRsvServerView publicMethods

setNames

(self subpartNamed: 'qmgrLabel') object: ('Queue Manager: ').
((self subpartNamed: 'M Q Hotel Rsv Server1') valueOfAttributeNamed: #mQSeriesConnectionSpec1QueueManagerName).
(self subpartNamed: 'reqQLabel') object: ('Request Queue: ').
((self subpartNamed: 'M Q Hotel Rsv Server1') valueOfAttributeNamed: #mQSeriesConnectionSpec1RequestQueueName).
(self subpartNamed: 'repQLabel') object: ('Reply Queue: ').
((self subpartNamed: 'M Q Hotel Rsv Server1') valueOfAttributeNamed: #mQSeriesConnectionSpec1ReplyQueueName).
Appendix C. Hotel Reservation Application Smalltalk Code 2.0

This appendix contains modifications to the Smalltalk code in the previous appendix when the storeString method is used. The classes are:

- HotelReq
- HotelRsv

C.1 Business Object HotelReq

Object subclass: #HotelReq
  instanceVariableNames: 'checkInDate place noOfNights noOfRooms no name type'
  classVariableNames: 'HotelReqs '
  poolDictionaries: '

HotelReq class publicMethods

newFromString: aString
  "new from a flattened object."
  | req |
  req := Compiler evaluate: aString.
  \req

HotelReq publicMethods

contentsAsString
  "return a String of a flattened object."
  | contents |
  contents := self storeString.
  \contents
C.2 Business Object HotelRsv

Object subclass: #HotelRsv

instanceVariableNames: 'status name hotel confirmationNo no'
classVariableNames: 'Rand'
poolDictionaries: ''

HotelRsv class publicMethods

newFromString: aString

"new from a flattened object."
|rsv|

rsv := Compiler evaluate: aString.
∧rsv

HotelRsv publicMethods

contents

"return ByteArray of a flattened object."
| contents |

contents := self storeString asByteArray.
∧contents
Appendix D. Hotel Reservation Application Smalltalk Code 3.0

Version 3 of the application uses ObjectLoader and ObjectDumper. Four classes are different from Version 1:

- Business object HotelReq class
- Business object HotelRsv class
- Client Application class HotelReqForm
- Server Application class MQHotelRsvServer

D.1 Business Object HotelReq

Object subclass: #HotelReq

instanceVariableNames: 'checkInDate place noOfNights noOfRooms no name type'
classVariableNames: 'HotelReqs'
poolDictionaries: ''

HotelReq class publicMethods

loadObject: aByteArray

| loader |

loader := ObjectLoader new.

\loader

loadFromByteObjects: aByteArray
offsetsIntoByteObjects: 0
errorStream: nil.

newFromString: aString

"new from a flatted object."
| req |

req := self loadObject: aString.
\req

HotelReq publicMethods

dumpObject

| dumper estimatedSize byteArrays |

dumper := ObjectDumper new.
estimatedSize := dumper totalSizeBeforeUnload: self.
D.2 Business Object HotelRsv

Object subclass: #HotelRsv
    instanceVariableNames: 'status name hotel confirmationNo no'
    classVariableNames: 'Rand'
    poolDictionaries: ''

HotelRsv class publicMethods

loadObject: aByteArray

    | loader |
    loader := ObjectLoader new.

    ^loader
        loadFromByteObjects: aByteArray
        offsetsIntoByteObjects: 0
        errorStream: nil.

HotelRsv publicMethods

dumpObject

    | dumper estimatedSize byteArrays |
    dumper := ObjectDumper new.
    estimatedSize := dumper totalSizeBeforeUnload: self.
    byteArrays := LibraryObjects byteArraysForObjectSize: estimatedSize.
    dumper
        unload: self
        intoByteObjects: byteArrays
        offsetsIntoByteObjects: 0
        maximumLimit: estimatedSize
        errorStream: nil.

    ^byteArrays
D.3 Client Application Class HotelReqForm

AbtAppBldrView subclass: #HotelReqForm
    instanceVariableNames: 'mqConnection hotelReq'
    classVariableNames: ''
    poolDictionaries: ''

HotelReqForm publicMethods

putObject
   | req |

    req := self hotelReq dumpObject.
    mqConnection putRequest: req size printString.
    req do: [ :each | mqConnection putRequest: each].

    (self subpartNamed: 'messageLabel') object: 'Request has been submitted'.
D.4 Server Application Class MQHotelRsvServer

AbtAppBldrPart subclass: #MQHotelRsvServer
    instanceVariableNames: 'ws queueManager no request '
    classVariableNames: ''
    poolDictionaries: ''

MQHotelRsvServer publicMethods

put
    | message rc reply |
    message := (self subpartNamed: 'MQ Series Message1').
    ws show: 'Get message "$ message asString," from client.'; cr.
    no = 0
        ifTrue: [no := message asString asNumber.
            request := Array new: no]
        ifFalse: [request at: (request size - no + 1) put: message contents.
            (no := no - 1) = 0
                ifTrue: [reply := self reserveHotel: request.
                    (rc := (self subpartNamed: 'MQ Series Connection1')
                        putReply: reply) isAbtError
                        ifTrue: [
                            ws show: 'Put message request failed'; cr.
                            ^ws show: '------> Return code: ',
                            rc codesAsString; cr.
                        ]
                        ifFalse: [ws show: 'Put message is ok'; cr].
                        ^rc]].

start
    | rc |
    self setUpWorkspace.
    ws show: 'Now start a server program.'; cr.
    queueManager := (self subpartNamed: 'MQ Series Connection1') yourself.
    no := 0.

    self connect isAbtError
        ifTrue: [
            self disconnect].
Appendix E. Diskette Contents

The diskette contains the examples developed in this book. It contains the following files:

<table>
<thead>
<tr>
<th>File Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>book.dat</td>
<td>All VisualAge for Smalltalk applications.</td>
</tr>
<tr>
<td>first.h</td>
<td>A C header file used in the example described in Chapter 5, &quot;The First VisualAge for Smalltalk MQ Application&quot; on page 47.</td>
</tr>
<tr>
<td>localdef.tst</td>
<td>Definitions of MQSeries objects needed when client and server use the same queue manager.</td>
</tr>
<tr>
<td>client.tst</td>
<td>Definitions of MQSeries objects needed for the VisualAge client.</td>
</tr>
<tr>
<td>server.tst</td>
<td>Definitions of MQSeries objects needed for the VisualAge server.</td>
</tr>
</tbody>
</table>

**Note:** Refer to Appendix A, “Setup for Examples” on page 129 for set up guidelines.

The file book.dat contains the following applications and classes:

<table>
<thead>
<tr>
<th>Applications</th>
<th>Classes</th>
<th>Chapter</th>
</tr>
</thead>
<tbody>
<tr>
<td>ApplOne</td>
<td>MQOneView</td>
<td></td>
</tr>
<tr>
<td>ApplTwo</td>
<td>ApplTwoView</td>
<td></td>
</tr>
<tr>
<td>ApplToDoList</td>
<td>ToDoClientView</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ToDoServerView</td>
<td></td>
</tr>
<tr>
<td>ApplExtToDo</td>
<td>ApplToDoListMsgIDClientView</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ApplToDoListMsgIDServerView</td>
<td></td>
</tr>
<tr>
<td>ApplModelMsg</td>
<td>MQTodoListModelMsgClientView</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MQTodoListModelMsgServerView</td>
<td></td>
</tr>
<tr>
<td>ApplSyncPoint</td>
<td>MQTodoListSyncPointClientView</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MQTodoListSyncPointServerView</td>
<td></td>
</tr>
</tbody>
</table>

© Copyright IBM Corp. 1997
Table 10 (Page 2 of 2). Applications and Classes

<table>
<thead>
<tr>
<th>Applications</th>
<th>Classes</th>
<th>Chapter</th>
</tr>
</thead>
<tbody>
<tr>
<td>MQHotelRsvApp</td>
<td>HotelReq</td>
<td>Chapter 11</td>
</tr>
<tr>
<td></td>
<td>HotelReqForm</td>
<td></td>
</tr>
<tr>
<td></td>
<td>HotelRsv</td>
<td></td>
</tr>
<tr>
<td></td>
<td>HotelRsvForm</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MQClientForm</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MQHotelReqView</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MQHotelRsvApp</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MQHotelRsvServerView</td>
<td></td>
</tr>
<tr>
<td>MQHotelRsvServer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MQHotelRsvServer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MQHotelRsvServer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MQHotelRsvServer</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Notes:

1. The version numbers of the hotel reservation applications are:
   - Redbook1
   - Redbook2
   - Redbook3

2. The version number of all other applications is
   - Redbook
Appendix F. Special Notices

This publication is intended to help application designers and programmers to develop applications using VisualAge and MQSeries. The information in this publication is not intended as the specification of any programming interfaces that are provided by Visual Age and MQSeries. See the PUBLICATIONS section of the IBM Programming Announcement for VisualAge and MQSeries for more information about what publications are considered to be product documentation.

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 IBM’s product, program, or service may be used. Any functionally equivalent program that does not infringe any of IBM’s intellectual property rights may be used instead of the IBM product, program or service.

Information in this book was developed in conjunction with use of the equipment specified, and is limited in application to those specific hardware and software products and levels.

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 the IBM Director of Licensing, IBM Corporation, 500 Columbus Avenue, Thornwood, NY 10594 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 the information which has been exchanged, should contact IBM Corporation, Dept. 600A, Mail Drop 1329, Somers, NY 10589 USA.

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

The information contained in this document has not been submitted to any formal IBM test and is distributed AS IS. The information about non-IBM ("vendor") products in this manual has been supplied by the vendor and IBM assumes no responsibility for its accuracy or completeness. The use of this information or the implementation of any of these techniques is a customer responsibility and depends on the customer’s ability to evaluate and integrate them into the customer’s operational environment. While each
item may have been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the same or similar results will be obtained elsewhere. Customers attempting to adapt these techniques to their own environments do so at their own risk.

The following terms are trademarks of the International Business Machines Corporation in the United States and/or other countries:

- AIX
- APPN
- FlowMark
- IBM
- MQSeries
- Open Blueprint
- Operating System/2
- OS/2
- RISC System/6000
- RS/6000
- ThinkPad
- VisualAge

The following terms are trademarks of other companies:

- C-bus is a trademark of Corollary, Inc.
- PC Direct is a trademark of Ziff Communications Company and is used by IBM Corporation under license.
- UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Company Limited.
- Pentium, MMX, ProShare, LANDesk and ActionMedia are trademarks or registered trademarks of Intel Corporation in the United States and other countries.
- Microsoft, Windows, and the Windows 95 logo are trademarks or registered trademarks of Microsoft Corporation.
- Java and HotJava are trademarks of Sun Microsystems, Inc.

Other company, product and service names may be trademarks or service marks of others.
Appendix G. Related Publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

G.1 International Technical Support Organization Publications

For information on ordering these ITSO publications see “How to Get ITSO Redbooks” on page 163.

- *VisualAge: Concepts and Features*, GG24-3946
- *VisualAge and Transaction Processing in a Client/Server Environment*, GG24-4487

G.2 Redbooks on CD-ROMs

Redbooks are also available on CD-ROMs. **Order a subscription** and receive updates 2-4 times a year at significant savings.

<table>
<thead>
<tr>
<th>CD-ROM Title</th>
<th>Subscription Number</th>
<th>Collection Kit Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>System/390 Redbooks Collection</td>
<td>SBOF-7201</td>
<td>SK2T-2177</td>
</tr>
<tr>
<td>Networking and Systems Management Redbooks Collection</td>
<td>SBOF-7370</td>
<td>SK2T-6022</td>
</tr>
<tr>
<td>Transaction Processing and Data Management Redbook</td>
<td>SBOF-7240</td>
<td>SK2T-8038</td>
</tr>
<tr>
<td>AS/400 Redbooks Collection</td>
<td>SBOF-7270</td>
<td>SK2T-2849</td>
</tr>
<tr>
<td>RS/6000 Redbooks Collection (HTML, BkMgr)</td>
<td>SBOF-7230</td>
<td>SK2T-8040</td>
</tr>
<tr>
<td>RS/6000 Redbooks Collection (PostScript)</td>
<td>SBOF-7205</td>
<td>SK2T-8041</td>
</tr>
<tr>
<td>Application Development Redbooks Collection</td>
<td>SBOF-7290</td>
<td>SK2T-8037</td>
</tr>
<tr>
<td>Personal Systems Redbooks Collection</td>
<td>SBOF-7250</td>
<td>SK2T-8042</td>
</tr>
</tbody>
</table>

G.3 Other Publications

These publications are also relevant as further information sources:

- *VisualAge for Smalltalk Getting Started*, SC34-4535
- *MQSeries Clients*, GC33-1632
- *MQSeries Distributed Queuing Guide*, SC33-1139
- *MQSeries Installation and System Management Guide*, SC33-1371
How to Get ITSO Redbooks

This section explains how both customers and IBM employees can find out about ITSO redbooks, CD-ROMs, workshops, and residencies. A form for ordering books and CD-ROMs is also provided.

This information was current at the time of publication, but is continually subject to change. The latest information may be found at URL http://www.redbooks.ibm.com.

How IBM Employees Can Get ITSO Redbooks

Employees may request ITSO deliverables (redbooks, BookManager BOOKs, and CD-ROMs) and information about redbooks, workshops, and residencies in the following ways:

- **PUBORDER** — to order hardcopies in United States
- **GOPHER link to the Internet** - type GOPHER.WTSCPOK.ITSO.IBM.COM
- **Tools disks**
  - To get LIST3820s of redbooks, type one of the following commands:
    - TOOLS SENDTO EHONE4 TOOLS2 REDPRINT GET SG24xxxx PACKAGE
    - TOOLS SENDTO CANVM2 TOOLS REDPRINT GET SG24xxxx PACKAGE (Canadian users only)
  - To get BookManager BOOKs of redbooks, type the following command:
    - TOOLCAT REDBOOKS
  - To get lists of redbooks:
    - TOOLS SENDTO USDIST MKTTOOLS MKTTOOLS GET ITSOCAT TXT
    - TOOLS SENDTO USDIST MKTTOOLS MKTTOOLS GET LISTSERV PACKAGE
  - To register for information on workshops, residencies, and redbooks:
    - TOOLS SENDTO WTSCPOK TOOLS ZDISK GET ITSOREGI 1996
  - For a list of product area specialists in the ITSO:
    - TOOLS SENDTO WTSCPOK TOOLS ZDISK GET ORGCARD PACKAGE
- **Redbooks Home Page on the World Wide Web**
  - http://w3.itso.ibm.com/redbooks
- **IBM Direct Publications Catalog on the World Wide Web**
  - IBM employees may obtain LIST3820s of redbooks from this page.
- **REDBOOKS category on INEWS**
- **Online** — send orders to: USIB6FPL at IBMMAIL or DKIBMBSH at IBMMAIL
- **Internet Listserver**
  - With an Internet e-mail address, anyone can subscribe to an IBM Announcement Listserver. To initiate the service, send an e-mail note to announce@webster.ibmlink.ibm.com with the keyword subscribe in the body of the note (leave the subject line blank). A category form and detailed instructions will be sent to you.
How Customers Can Get ITSO Redbooks

Customers may request ITSO deliverables (redbooks, BookManager BOOKs, and CD-ROMs) and information about redbooks, workshops, and residencies in the following ways:

- **Online Orders** (Do not send credit card information over the Internet) — send orders to:

  - IBMMAIL
    - In United States: usib6fpl at ibmmail
    - In Canada: caibmbkz at ibmmail
    - Outside North America: dkibmbsh at ibmmail
  - Internet
    - usib6fpl@ibmmail.com
    - imannix@vnet.ibm.com
    - bookshop@dk.ibm.com

- **Telephone orders**
  - United States (toll free) 1-800-879-2755
  - Canada (toll free) 1-800-IBM-4YOU
  - Outside North America
    - (+45) 4810-1320 - Danish
    - (+45) 4810-1420 - Dutch
    - (+45) 4810-1540 - English
    - (+45) 4810-1670 - Finnish
    - (+45) 4810-1220 - French
    - (long distance charges apply)
    - (+45) 4810-1020 - German
    - (+45) 4810-1620 - Italian
    - (+45) 4810-1270 - Norwegian
    - (+45) 4810-1120 - Spanish
    - (+45) 4810-1170 - Swedish

- **Mail Orders** — send orders to:

  - IBM Publications
    - Publications Customer Support
    - P.O. Box 29570
    - Raleigh, NC 27626-0570
    - USA
  - IBM Publications
    - 144-4th Avenue, S.W.
    - Calgary, Alberta T2P 3N5
    - Canada
  - IBM Direct Services
    - Sortemosevej 21
    - DK-3450 Allerød
    - Denmark

- **Fax** — send orders to:

  - United States (toll free) 1-800-445-9269
  - Canada 1-403-267-4455
  - Outside North America (+45) 48 14 2207 (long distance charge)

- **1-800-IBM-4FAX (United States)** or (**+1**)001-408-256-5422 (Outside USA) — ask for:
  - Index # 4421 Abstracts of new redbooks
  - Index # 4422 IBM redbooks
  - Index # 4420 Redbooks for last six months

- **Direct Services** - send note to software@vnet.ibm.com

- **On the World Wide Web**

  - Redbooks Home Page http://www.redbooks.ibm.com

- **Internet Listserv**

  With an Internet e-mail address, anyone can subscribe to an IBM Announcement Listserv. To initiate the service, send an e-mail note to announce@webster.ibmlink.ibm.com with the keyword subscribe in the body of the note (leave the subject line blank).
# IBM Redbook Order Form

Please send me the following:

<table>
<thead>
<tr>
<th>Title</th>
<th>Order Number</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

First name ____________________________  Last name ____________________________

Company ____________________________

Address __________________________________________

City ____________________________ Postal code __________ Country __________

Telephone number ____________________________ Telefax number ____________________________ VAT number ____________________________

- Invoice to customer number ____________________________________________
- Credit card number ____________________________________________

Credit card expiration date ____________________________ Card issued to ____________________________ Signature ____________________________

*We accept American Express, Diners, Eurocard, Master Card, and Visa. Payment by credit card not available in all countries. Signature mandatory for credit card payment.*

**DO NOT SEND CREDIT CARD INFORMATION OVER THE INTERNET.**
List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Application Program Interface</td>
</tr>
<tr>
<td>DDE</td>
<td>dynamic data exchange</td>
</tr>
<tr>
<td>DLL</td>
<td>dynamic link library</td>
</tr>
<tr>
<td>DSOM</td>
<td>distributed system object model</td>
</tr>
<tr>
<td>EMSRV</td>
<td>envy manager server</td>
</tr>
<tr>
<td>GUI</td>
<td>graphical user interface</td>
</tr>
<tr>
<td>IBM</td>
<td>International Business Machines Corporation</td>
</tr>
<tr>
<td>ITSO</td>
<td>International Technical Support Organization</td>
</tr>
<tr>
<td>LAN</td>
<td>local area network</td>
</tr>
<tr>
<td>MCA</td>
<td>message channel agent</td>
</tr>
<tr>
<td>MQ</td>
<td>Message Queuing</td>
</tr>
<tr>
<td>MQI</td>
<td>message queuing interface</td>
</tr>
<tr>
<td>MQM</td>
<td>MQ queue manager</td>
</tr>
<tr>
<td>MVC</td>
<td>model view controller</td>
</tr>
<tr>
<td>OO</td>
<td>object-oriented</td>
</tr>
<tr>
<td>SOM</td>
<td>system object model</td>
</tr>
<tr>
<td>VA</td>
<td>VisualAge</td>
</tr>
<tr>
<td>VMT</td>
<td>visual modeling technique</td>
</tr>
</tbody>
</table>
## Index

### Numerics

| 32 bit | 56 |

### A

- abbreviations 167
- acronyms 167
- action 5
  - connect 65
  - disconnect 66
  - get 69
  - initializeWith 69
  - putRequest 68
- alias queue 24
- amqcrsta 37
- amqscoma.tst 20
- AMQSGBR 68
- AMQSGET 59
- AMQSGETC 28
- AMQSPUT 59
- AMQSPUTC 28
- API 17
- APIs 37
- application
  - client/server 79
  - create new 51
  - execute 58
  - organizer 51
- application development 1
- asynchronous messaging 18
- attribute 5
  - aConnectionSpec 66, 84
  - contentsAsString 69
  - define 115
  - get and set 115
  - normalResult 69
  - object 69
  - self 66

### B

| back out | 110 |
| background job | 43, 44 |
| BACKOUT | 45 |
| bibliography | 161 |
| book.dat | 157 |
| browse queue | 68 |
| business object layer | 113 |

### C

- C language 51
- canvas 117
- change name 81, 82
- change user 16
- change window title 76
- channel 21, 25
  - message channel 21
  - MQI channel 21
- check box 82
- client 8, 25
- client connection
- client/server
  - connection 27
  - start connection 28
  - test connection 28
- COBOL 51
- combo box 63, 74
- COMMIT 45, 109
- communications subsystem 1
- communications/transactions 8
- composition editor 1, 53
- CONFIG.SYS 27
- connect to queue manager 44, 65, 83
  - possible errors 93
- connection
  - attribute-to-attribute 118
  - event-to-action 57, 65
  - event-to-method 71
  - move line 86
  - to itself 85
instance 61
instance variable 70

L
label 63
language feature 51
library
  MQI 50
  MQM 50
library of parts 1
library supervisor 14
listener 28, 37
load features 13
load/unload features 15
local queue 23
log 61, 70, 91

M
manager library 8
directory 11
manipulate message 90
map object/message 113
message 18
descriptor 18
types 18
message channel 21, 26
message descriptor 95
message driven 39
message ID 95, 103
message part
described 42
message queuing 18
messaging 18
messaging and queuing 18
method
  change combo box 74
  change window title 76
  connect events 71
  get message data 100
  how to write one 71
  message ID 107
  put reply data 100
  put reply message 107
  put request message 97
  random number 107
  method (continued)
    timestamp 100
    variable 70
    write to log 71
middleware 17
model message 103
move connection line 86
MQBACK 37, 109
MQCLOSE 37
MQCMIT 37
MQCONN 37
MQDISC 37
MQGET 37, 68
MQGMO_SYNCPOINT 109
MQI 17, 19
MQI calls 37
elements 38
MQI channel 21, 26
MQIC library 50
MQINQ 37
MQM 17, 19
MQM library 50
MQMD structure 95
MQOPEN 37
MQPMO_SYNCPOINT 109
MQPUT 37, 68
MQPUT1 37
MQSeries 17
  actions (VisualAge) 65
  API 17, 41
  at run time 17
  channels 25
definitions 129
  overview 17
  principle 18
  set up for execution 129
  set up for VisualAge examples 48
MQSeries Connection part 61
MQSeries Connection Specifications part 61
MQSeries Message part 61
MQSERIESDLL 50
MQSERVER 27
MQSET 37
S
sample program 76
change window title 76
collection part 61
collection spec 61
get particular message 103
hotel reservation 113
message part 61
MQSeries APIs 38
MQSeries definitions 129
procedure dialog 47
syncpoint 109
syncpoint processing 109
to-do list 79
use message ID 95
save image 13
script editor 71
scripting language 4
sequence of ProcDialog 47
server 25
server connection 27
how to test 28
triggering 34
service object layer 113
set parameters 72
set up MQSeries 129
setter 115
settings
combo box 105
destination 55
event-to-action connection 57
MQSeries Connection Spec 64
proc dialog 55
push button 54
records 55
Smalltalk objects 2
stand alone 8
string 105
STRMQM 20
SVRCONN 27
synchronous messaging 18
syncpoint 37, 45, 91, 109
example 109
system transcript 13, 14, 80, 91

T
team environment 13
team programming 3
tear-off 45, 66
lastError 66, 85
message descriptor 104
replyQueue 85
requestQueue 92
test 67, 93
set up 129
text field 63
time independent 39
to-do list 79
client view 80
extended 95
server view 90
toggle button 82, 85
transaction management 2
transmission queue 24
trigger message 37
trigger monitor 37
triggering 34
tst files 157

U
unit of work 37, 109
unload features 13
user interface 63

V
variable definition 70
view layer 113
visual modeling technique 2
visual part 52
visual parts 4
visual parts menu 56
visual programming 1
VisualAge
action 5
attribute 5
communications subsystem 1
composition editor 1
development environment 7
event 5
VisualAge (continued)
features 15
installation 7
language support 3
library of parts 1
MQ features 41
non-visual parts 4
overview 1
packaging 6
public interface 4
scripting language 4
set up MQSeries 48
syncpoint processing 109
team environment 13
team programming 3
transaction management 2
using for the first time 13
visual parts 4
VisualAge for Smalltalk
products 7
VisualAge organizer 16, 52

W
window title 76
ITSO Redbook Evaluation

Application Development with VisualAge for Smalltalk and MQSeries
SG24-2117-00

Your feedback is very important to help us maintain the quality of ITSO redbooks. Please complete this questionnaire and return it using one of the following methods:

• Use the online evaluation form found at http://www.redbooks.com
• Fax this form to: USA International Access Code + 1 914 432 8264
• Send your comments in an Internet note to redeval@vnet.ibm.com

Please rate your overall satisfaction with this book using the scale:
(1 = very good, 2 = good, 3 = average, 4 = poor, 5 = very poor)

Overall Satisfaction ____________

Please answer the following questions:

Was this redbook published in time for your needs? Yes____ No____
If no, please explain:
_____________________________________________________________________________________________________
_____________________________________________________________________________________________________
_____________________________________________________________________________________________________
_____________________________________________________________________________________________________

What other redbooks would you like to see published?
_____________________________________________________________________________________________________
_____________________________________________________________________________________________________
_____________________________________________________________________________________________________
_____________________________________________________________________________________________________

Comments/Suggestions: ( THANK YOU FOR YOUR FEEDBACK! )
_____________________________________________________________________________________________________
_____________________________________________________________________________________________________
_____________________________________________________________________________________________________
_____________________________________________________________________________________________________

© Copyright IBM Corp. 1997 175