VisualAge: Building GUIs for Existing Applications

Document Number GG24-4244-00

July 1994

International Technical Support Organization
San Jose Center

Take Note!

Before using this information and the products it supports, be sure to read the general information under “Special Notices” on page xvii.

First Edition (July 1994)

This edition applies to VisualAge for OS/2, Version 1.0, program number 5621-387, and VisualAge Team for OS/2, Version 1.0, program number 5621-388.

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address given below.

An ITSO Technical Bulletin Evaluation Form for reader′s feedback appears facing Chapter 1. If the form has been removed, comments may be addressed to:

IBM Corporation, International Technical Support Organization
Dept. 471 Building 070B
5600 Cottle Road
San Jose, California 95193-0001

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.

Abstract

This document describes the process of building a graphical user interface (GUI) for an existing host application. The document shows how a developer can use a GUI front end to provide new application functions on the workstation and access to workstation tools.

A simple application with one input and one output map is used to explain the principles of host and workstation control for VisualAge EHLLAPI applications. Different implementation approaches for GUI front-end applications are investigated, evaluated, and documented. Conclusions and recommendations are provided.

This document is written for application developers assigned the task of writing GUI front ends for existing host applications using VisualAge's EHLLAPI support. Some VisualAge knowledge is assumed.

AD LS (244 pages)

Contents

Figures

Tables

Special Notices

This publication is intended to help application developers write GUI front ends for existing host applications using VisualAge′s EHLLAPI support. The information in this publication is not intended as the specification of any programming interfaces that are provided by VisualAge. See the PUBLICATIONS section of the IBM Programming Announcement for VisualAge 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, 208 Harbor Drive, Stamford, CT 06904 USA.

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 document contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples contain the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

The following terms, which are denoted by an asterisk (*) in this publication, are trademarks of the International Business Machines Corporation in the United States and/or other countries:

CICS	CICS OS/2
CICS/ESA	CICS/MVS
CICS/VSE	CICS/400
CICS/6000	CUA
DB2	DB2/2
DB2/400	DB2/6000
DRDA	IBM
OS/2	Presentation Manager

SQL/DS	VisualAge

The following terms, which are denoted by a double asterisk (**) in this publication, are trademarks of other companies:

Lotus	Lotus Development Corporation
OSF/Motif	Open Software Foundation, Inc.
Wordperfect	Novell, Inc.

Other trademarks are trademarks of their respective companies.

Preface

This document explains VisualAge EHLLAPI support with a practical example. It describes the process of building a graphical user interface (GUI) for an existing host application. The document shows how a GUI front end can be used to add value to an existing application by implementing new application functions on the workstation and providing access to workstation tools.

A simple application with one input and one output map is used to explain the principles of host and workstation control for VisualAge EHLLAPI applications. Different implementation approaches for GUI front-end applications are investigated, evaluated, and documented. Conclusions and recommendations are provided.

This document is intended for application developers assigned the task writing GUI front ends for existing host applications using VisualAge′s EHLLAPI support. Some VisualAge knowledge is assumed.

How This Document is Organized

The document is organized as follows:

Part 1, " VisualAge EHLLAPI Concepts"

- Chapter 1, "Client/Server Models"

  This chapter introduces various client/server models based on a model popularized by the Gartner Group. A short description of each of the five client/server models is given.

- Chapter 2, "EHLLAPI"

  This chapter introduces the EHLLAPI communication protocol and explains the EHLLAPI support provided by VisualAge. Advantages and disadvantages of the EHLLAPI communication protocol are described.

- Chapter 3, "Applying the VisualAge EHLLAPI Parts with a Simple Example"

  This chapter introduces the concept of host and workstation control for an EHLLAPI-based GUI application. The chapter shows how the VisualAge EHLLAPI parts can be applied based on a simple host application with one input and one output map.

Part 2, " Implementing the Sample Application"

- Chapter 4, "Design and Implementation Considerations"

  This chapter provides general design and implementation considerations for GUI applications built with VisualAge′s EHLLAPI support. It addresses naming conventions for parts and applications, design models for GUI applications, and the concept of model-view separation.

- Chapter 5, "Sample GUI Application Design Overview"

  This chapter explains the objectives of our project and the design and implementation steps we performed when building our sample application.

- Chapter 6, "Designing the GUI"

This chapter introduces the text based user interface of the existing host application and a first-cut GUI design for our sample application.

- Chapter 7, “Running the Sample Application”

  This chapter explains the functions of the sample application based on windows captured during application execution.

- Chapter 8, “Implementation Walkthrough”

  This chapter provides a detailed description of all of the parts that were developed to implement the sample application. Hints and tips based on our experience are provided.

- Appendix A, “Screen Field Monitor Tool”

  This appendix provides information about a tool we developed during the project. This tool makes it easier to find the input and output fields on the host maps and relate them to the fields in the GUI windows.

- Appendix B, “Changing Host Sessions Dynamically without Using Scripts”

  This appendix shows an approach to dynamically selecting the host session at application execution time.

- Appendix C, “Jumping to the 3270 Screen”

  This appendix explains a method of jumping to the host 3270 emulator session from a VisualAge EHLLAPI application.

Related Publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this document.

- *Construction from Parts Architecture: Building Parts for Fun and Profit*, SC34-4488-00
- *VisualAge User's Guide and Reference*, SC34-4490-00
- *Introduction to Object-Oriented Programming with IBM Smalltalk*, SC34-4491-00
- *VisualAge: Guide to the IBM Smalltalk Development Environment*, SC34-4492-00
- *VisualAge: IBM Smalltalk Programmer's Reference*, SC34-4493-00
- *VisualAge Communications Guide and Reference*
- *VisualAge Development Guide*
- *VisualAge Team Development Guide*
- *IBM Smalltalk Catalog of Classes*
- *Object-Oriented Interface Design: IBM Common User Access Gudelines*, SC34-4399, ISBN 1-56529-170-0
- *Client/Server Computing with AD/Cycle Application Generators*, GG24-3760
- *OS/2 EE 1.3 EHLLAPI Programming Reference*, S01F-0297

International Technical Support Organization Publications

A complete list of International Technical Support Organization publications, with a brief description of each, may be found in:

Bibliography of International Technical Support Organization Technical Bulletins, GG24-3070.

How to Order ITSO Technical Bulletins (Redbooks)

IBM employees in the USA may order ITSO books and CD-ROMs using PUBORDER. Customers in the USA may order by calling 1-800-879-2755 or by faxing 1-800-284-4721. Visa and Master Cards are accepted. Outside the USA, customers should contact their IBM branch office.

Customers may order hardcopy redbooks individually or in customized sets, called GBOFs, which relate to specific functions of interest. You may also order redbooks in online format on CDROM collections, which contain the redbooks for multiple products.

Acknowledgments

The advisor for this project was:

Christian Peterhans
International Technical Support Organization, San Jose Center

The authors of this document are:

Reginaldo W. Barosa
IBM Brazil

Urs Halter
IBM Switzerland

This publication is the result of a residency conducted at the International Technical Support Organization, San Jose Center.

Thanks to the reviewers who contributed to improve this document.

International Technical Support Organization, San Jose Center

Part 1. VisualAge EHLLAPI Concepts

Chapter 1. Client/Server Models

VisualAge* is a new object-oriented application development environment in which to build the client pieces of a client/server application. Client/server is an exciting and rapidly changing area of computing requiring some form of systematization. In this chapter we present our systematization of client/server computing. We position the VisualAge application development environment and some complementary tools that can be used to productively deliver client/server function.

A client/server system consists of three logical components: clients, servers, and a connecting network. In a client/server system it is essential that we have separate client applications that request services from separate server applications. The logical network that connects clients and servers, however, need not be a physical network. Client and server applications may reside on the same system.

So, in a client/server system we must have clients and servers. We can divide the data processing between clients and servers in numerous ways. One model, popularized by the Gartner Group, for systematizing the functions of clients and servers has gained acceptance within the computing industry. This model is two-dimensional. The two dimensions are application component and platform distribution. At this level of abstraction there are three logical application components—presentation, function, and data—and five levels of platform distribution—distributed presentation, remote presentation, distributed function, remote data management, and distributed data management (see Figure 1).

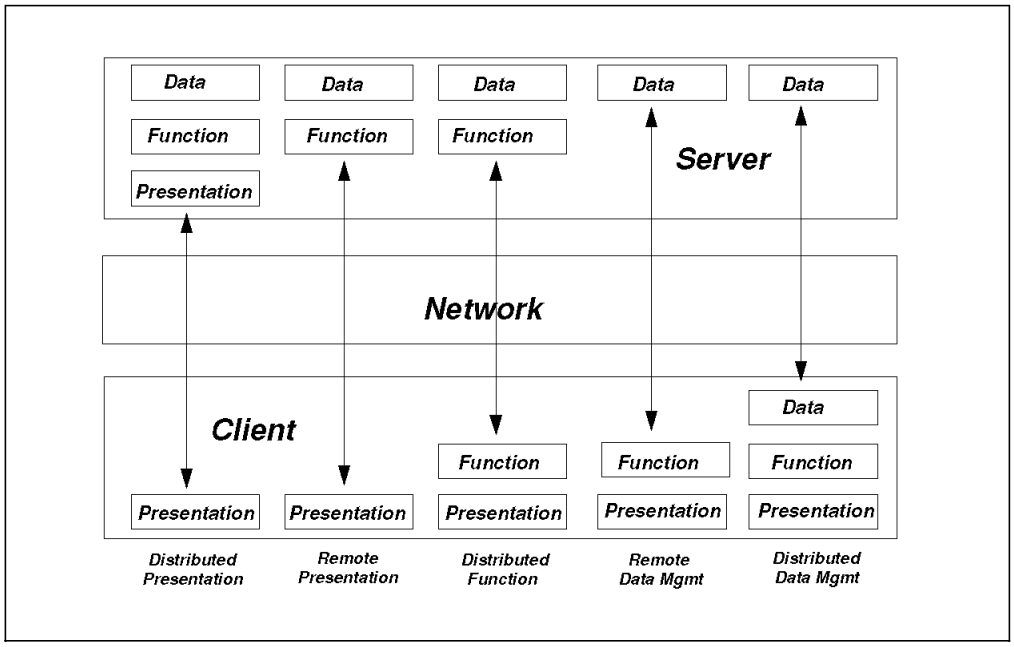

Figure 1. Client/Server Models

The key client/server strength of VisualAge lies in the distributed presentation, remote presentation, and distributed function levels. We describe these three models in 1.1, “Distributed Presentation” on page 4, 1.2, “Remote Presentation” on page 4, and 1.3, “Distributed Function” on page 4. VisualAge, however, also fully supports the remote data management and the distributed data

management functions delivered by the database and transaction manager products. See 1.4, “Remote Data Management” on page 5 and 1.5, “Distributed Data Management” on page 5.

1.1 Distributed Presentation

Distributed presentation occurs when the presentation, function, and data components reside on the server platform and parts of the presentation component reside on the client platform.

An example of distributed presentation is “screen scraping”—that is, an application on the client platform on a workstation intercepts the 3270 data stream from a program executing on a host computer. The emulator high-level language application programming interface (EHLLAPI) makes it easier for the client application to translate the 3270 data stream into a graphical user interface (GUI), for example, OS/2* Presentation Manager*.

In this book we explore a specific client/server implementation in the distributed presentation model illustrated in Figure 1 on page 3. To do this, we implement a GUI for an application developed in CSP/AD and documented in *Client/Server Computing with AD/Cycle Application Generators*, GG24-3760, using the EHLLAPI support in VisualAge.

1.2 Remote Presentation

Remote presentation occurs when the function and data components reside on the server platform and the presentation component resides on the client platform. Remote presentation and distributed function are logically different models, but they are combined for the purposes of presentation in this book in 1.3, “Distributed Function.”

1.3 Distributed Function

Distributed function occurs when the presentation component and parts of the function component reside on the client platform and parts of the function component and the data component reside on the server platform. Of the five client/server models we discuss in this chapter, the distributed function model offers the most flexibility and efficiency. With the distributed function model you can expand and change the applications as your business grows and changes. You also have the option of placing the function and data component on the most cost-efficient platform. The cost of this flexibility and efficiency is the complexity of the underlying infrastructure.

The distributed function model is also *mechanism transparent*, allowing access to any kind of data, whereas with the remote data management and distributed data management models, data must be stored in SQL format in a relational database.

1.4 Remote Data Management

Remote data management occurs when the presentation and function components reside on the client platform and the data component resides on the server platform.

Remote unit of work (RUW) in IBM's Distributed Relational Database Architecture (DRDA*) is an example of remote data management. The RUW level of DRDA is a database feature and as such is available in the following IBM products:[1]

- DB2*
- DB2/2*
- DB2/6000*
- DB2/400*
- SQL/DS*.

VisualAge supports the RUW level of DRDA during both development and execution.

Function shipping in the CICS* product family is another example of remote data management. Function shipping is the ability of one program executing in one CICS system, for example, CICS OS/2*, to transparently access data owned by another, connected CICS system, for example, CICS/ESA*. Function shipping is a transaction manager function and as such is available in the following CICS products:

- CICS/ESA
- CICS/MVS*
- CICS OS/2
- CICS/6000*
- CICS/400*
- CICS/VSE*.

VisualAge supports CICS function shipping during both development and execution.

1.5 Distributed Data Management

Distributed data management occurs when the presentation and function components and parts of the data component reside on the client platform and parts of the data component reside on the server platform.

Distributed unit of work (DUW) in DRDA is an example of distributed data management. The DUW level of DRDA is a database feature and as such is available only in DB2 V3.1. All products participating in DRDA will over time support the DUW level.

VisualAge supports the DUW level of DRDA during both development and execution.

[1] DRDA is an open architecture. Several vendors have announced or made available products participating in DRDA.

Chapter 2. EHLLAPI

EHLLAPI is an interface provided by a terminal emulator that allows a program to behave as a programmed operator. The program types keystrokes and reads text from a host screen. The programmed operator can automate repetitive tasks and provide better interfaces to existing (legacy) applications. Existing applications do not need to be changed; they continue to operate as though a user is providing input to them. They do not need to be aware that they are now interacting with programs rather than real users.

EHLLAPI allows you to build programs that use pieces of existing applications. For example, you can use EHLLAPI to capture an entire screen from the host. Your new programs can use technologies such as VisualAge and Smalltalk while maximizing the use of existing applications. Programs using EHLLAPI can also enable new functions to be delivered in stages. New programs can coexist with existing applications, and users of existing applications can migrate to the new programs at their discretion.

EHLLAPI was designed to use the same communication protocol as existing emulator programs; therefore all existing wiring and control units can continue to be used.

Note: Because the communication protocol was designed to handle human response times and speeds, EHLLAPI is best for low-volume transactions. Massive transfers of data can be accomplished more quickly through Advanced Program-to-Program Communications (APPC) or CICS-to-CICS communications.

2.1 VisualAge′s EHLLAPI Support

You can use VisualAge to build a workstation-based GUI that can access host applications designed for 3270 terminals. A VisualAge application can interact directly with the host application, and users can interact with the VisualAge application rather than entering data in a 3270 terminal emulator session.

From the host application′s perspective, the VisualAge application functions like a user at a 3270 terminal. There is no need to modify the host application.

To communicate with host applications, VisualAge uses EHLLAPI, a feature provided by 3270 terminal emulators, such as OS/2 Communications Manager/2 (see Figure 2 on page 8).

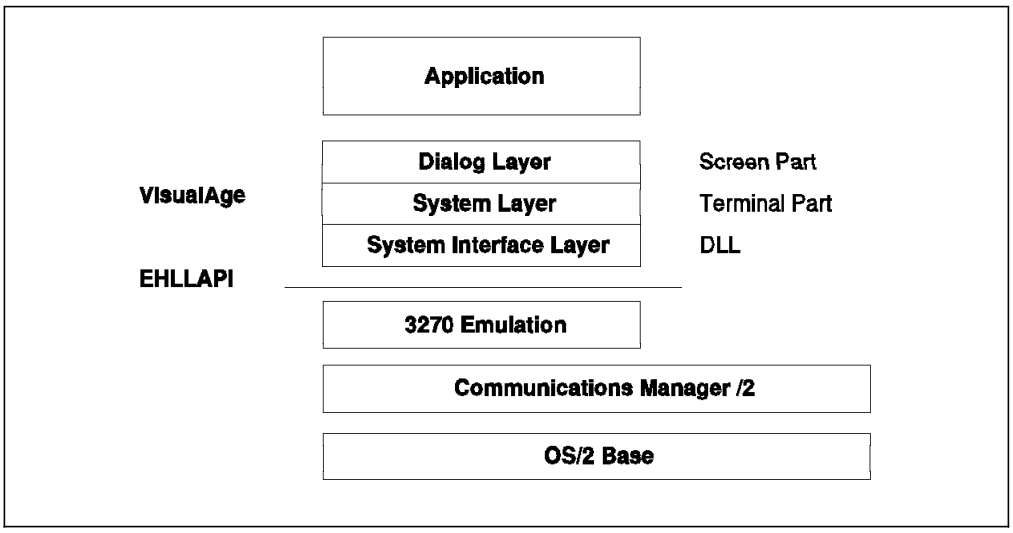

Figure 2. System Structure for VisualAge EHLLAPI Support

As shown in Figure 2, VisualAge provides a layered interface to the EHLLAPI functions. The dialog layer provides a simple get and put interface that is independent of the underlying EHLLAPI protocol. The system layer provides a lower level interface that still hides certain peculiarities of the underlying EHLLAPI protocol, such as the command sequences to connect to and disconnect from a host session. The system interface layer provides an object-oriented interface to the EHLLAPI dynamic link library (DLL).

For details refer to the *VisualAge Communications Guide and Reference* online manual.

2.1.1 Understanding VisualAge EHLLAPI Parts

Figure 3 on page 9 shows the nonvisual parts that VisualAge provides to implement the three layers of VisualAge EHLLAPI support.

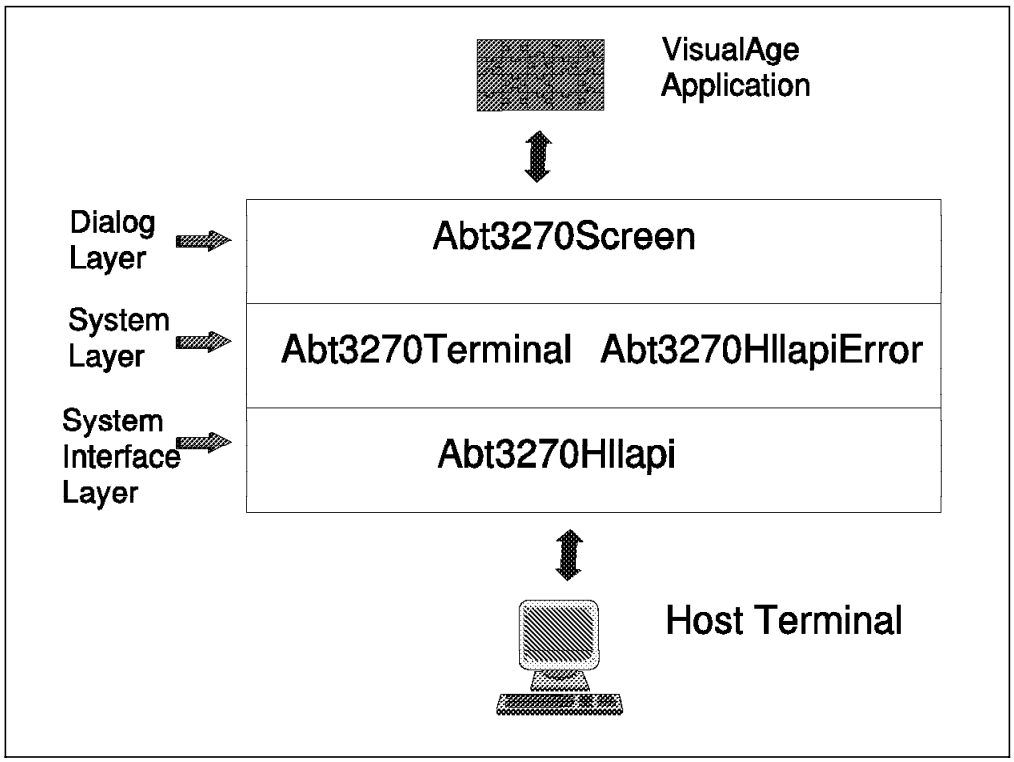

Figure 3. VisualAge EHLLAPI Support

2.1.1.1 The Abt3270Screen Part

The Abt3270 Screen part represents a single formatted panel on the 3270 host screen as two records (input and output). By treating the screen as a record this part simplifies the coding needed to have an application work with the host screen.

One of the instance variables kept in the Abt3270Screen part is the *keyString*, which must match a unique character string on the 3270 host screen before any action can complete. If this variable is defined as an empty string, it will match all screens.

The Abt3270Screen part provides a simple get and put interface to the host application.

2.1.1.2 The Abt3270Terminal Part

The Abt3270Terminal part is a subpart of the Abt3270Hllapi part and provides all of the functions of its parent part. In addition, it hides the command sequences required for a workstation to connect to and disconnect from a host session and provides an easier protocol for interacting with a host application.

This terminal-level protocol can be expanded as users discover more generally useful functions. If an appropriate high-level protocol is not available, an application can still access any basic EHLLAPI functions directly or extend this part with new actions.

Basic EHLLAPI functions need to be processed inside a *doOperation:* or *doWindowOperation:* block. This reduces the number of EHLLAPI connect and disconnect sequences.

The actions generally return either a useful object or a return code if successful, or an *Abt3270HllapiError* object if an unanticipated error occurs. This can be checked by using the *isError* protocol.

2.1.1.3 The Abt3270HllapiError Part

An instance of an Abt3270HllapiError part is returned by an *Abt3270Terminal* when it encounters an unexpected error. Develpers cannot easily access the Abt3270HllapiError object because it is not part of the visual programming interface.

2.1.1.4 The Abt3270Hllapi Part

The Abt3270Hllapi part provides the basic EHLLAPI functions as defined by reference manuals such as the *OS/2 EE 1.3 EHLLAPI Programming Reference*, S01F-0297. It provides a basic interface that encapsulates the functions in the EHLLAPI DLL.

Users should place enhancements built on top of the primitive EHLLAPI DLL operations in a subpart, such as the Abt3270Terminal, rather than extending the protocols available in the Abt3270Hllapi part.

When possible, the same action names and terms as those used in the EHLLAPI reference manuals are used. Refer to those manuals for complete details about return codes and interdependencies.

Many of the EHLLAPI functions behave differently depending on the value set for the EHLLAPI session parameters. Those parameters are global for the entire Smalltalk image and are set to reasonable defaults so that several programs running in one Smalltalk image can depend on consistent settings.

The EHLLAPI term *host presentation space (host PS)* refers to the text and attributes that are displayed in the terminal window. Most functions against a host PS take an offset into that PS and require that a *connectHostPS* operation be performed first.

The *isHllapiActive* message can be sent to find out whether EHLLAPI is installed and available. It will also open (or reopen) the EHLLAPI DLL and set the default *sessionParameters*. The default *sessionParameters* are also automatically set the first time the Abt3270Hllapi function is requested (because this part controls the initial opening of the DLL).

2.1.2 Abt3270Screen Actions and Events

Table 1 shows the events triggered by the Abt3270Screen part.

Table 1 (Page 1 of 2). Actions and Events Triggered by the Abt3270Screen Part

Action	Event Raised	Comments
clearInput	keySent	
getFieldData	dataRefreshed	
pressXXXX	keySent	Can be: pressClear, pressEnter, pressPA:, pressPF:

Table 1 (Page 2 of 2). Actions and Events Triggered by the Abt3270Screen Part		
Action	**Event Raised**	**Comments**
putData	dataSent	Update the screen with the data in the *inputFields* record. A field is updated only if the contents of the *inputFields* record is different from the current contents of the host screen.
refreshFieldDefs	fieldDefsRefreshed	Learn the field definitions for both input and output from the current host screen.

2.1.2.1 Understanding the screenChanged Event

The *screenChanged* event must be used when you need host control for your GUI application. It is important to understand when and how often the *screenChanged* event is raised, as this event triggers the whole application in an implementation with host control. See 3.3, “Implementing a GUI with Host Control” on page 21 for details.

Table 2 shows the outside events that trigger actions of the Abt3270Screen part.

Table 2. Outside Actions and Events that Trigger the Abt3270Screen Part		
Action	**Raised Event**	**Comments**
3270 screen is modified	screenChanged	This event is raised when a host session is updated by the user or the system

Note that the *screenChanged* event can be raised multiple times for a host screen change because the host updates the screen in several short blocks, and each block raises one *screenChanged* event.

Sometimes a *screenChanged* event is raised, but the data for the host map field has not been sent. To prevent such problems, the Abt3270Screen part provides the *Screen Settle Time*.

The *Screen Settle Time* represents a window, defined in seconds, during which *screenChanged* events are collected and reported as a single event. The *Screen Settle Time* should be set to the response time for the network. The *Screen Settle Time* can be specified in multiples of 1 second; 1 second is the default.

In Figure 4 on page 12 the *Screen Settle Time* is set once to 0 seconds and once to 5 seconds, and the host sends the data for one host 3270 screen as five blocks. With the *Screen Settle Time* set to 0 seconds each host block sent results in one *screenChanged* event raised; that is, for the five host blocks sent you will see five *screenChanged* events. With the *Screen Settle Time* set to 5 seconds all *screenChanged* events received during the 5-second window are collected and reported as one *screenChanged* event at the end of the 5-second window; that is, for the five host blocks sent you will see one *screenChanged* event.

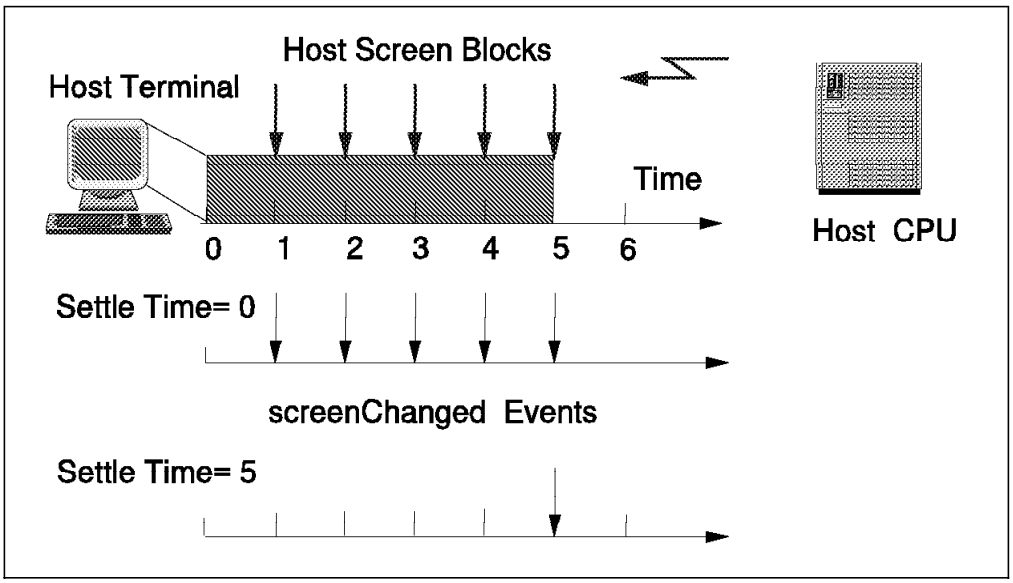

Figure 4. screenChanged Events and Settle Time

Conclusion

The *Screen Settle Time* has an impact on response time if the *screenChanged* event triggers the application.

In the example illustrated in Figure 4, with the *Screen Settle Time* set to 5 seconds, application response time is always more than 5 seconds.

It would be nice to have the *Screen Settle Time* set to zero whenever possible. However, if the *screenChanged* event triggers your GUI application, and the application logic is sensitive to the number of times the event is raised, you need to specify a value for the *Screen Settle Time*.

Sometimes it may be hard to find the best value for the *Screen Settle Time* especially if your application executes on local and network-attached programmable workstations (PWSs) with very different network response time characteristics or your network has large fluctuations in network response time.

You may want to consider using the Abt3270Terminal part instead of the Abt3270Screen part and having application control at the PWS rather than the host (see 3.3.1.3, “Advantages and Disadvantages of This Approach” on page 31).

2.1.2.2 Visualizing Events in the Transcript Window

Because the VisualAge environment is an event and action environment, it is sometimes important to know when and how often a specific event is raised. For example, to find out how often a *screenChanged* event is raised you can write a simple VisualAge script that writes a text to the transcript window (see Figure 5).

```
screenWasChanged
 "when screen is changed, write in transcript"
 Transcript show: 'Screen Changed Raised'; cr.
```

Figure 5. Script That Writes to the Transcript Window

You can now create an event-to-script connection from the *screenChanged* event to your script (see Figure 6).

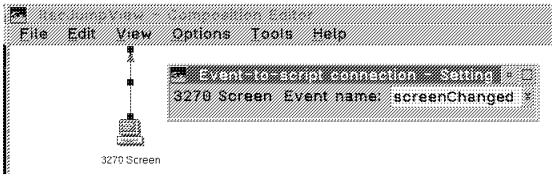

Figure 6. Event-to-Script Connection

Each time the *screenChanged* event is raised the text from the script is written to the transcript window. Figure 7 shows the transcript window after several host *screenChanged* events were raised during application testing.

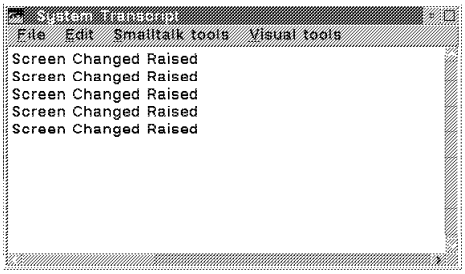

Figure 7. Transcript Window after Several screenChanged Events

2.1.3 Abt3270Terminal Actions and Events

Table 3 shows the events triggered by the Abt3270Terminal part.

Table 3. Actions and Events Triggered by the Abt3270Terminal Part

Action	Raised Event	Comments
findString:	searchSuccessful or searchFailed	Perform the findString: action
Various	errorOccurred	For example, *type:* has been attempted with the cursor in a write-protected location.

Note: In the beta code we used for our project the *errorOcurred* event was never raised, which was a bug in the code.

The example in the online documentation is also wrong. The *errorOcurrred* event is not raised when you attempt the *type:* action with the cursor positioned in a write-protected location. What happens in this case is that the terminal is reset and the cursor is back-tabbed to an unprotected field. If the host map has no unprotected fields, nothing happens.

2.2 Why Use EHLLAPI?

You can use the EHLLAPI functions to interface with legacy applications, automate repetitive tasks, and perform low-volume transactions. These functions are also useful for prototyping and implementing new applications quickly, without waiting for software or hardware upgrades to support better communication protocols.

Using the EHLLAPI objects, you can isolate a program from the details of the origin of the data; the same application can continue to be used even if the source of the data changes.

You can reuse VisualAge EHLLAPI programs by subclassing them and overriding only the methods you need to make those programs work in a particular environment.

Because VisualAge's EHLLAPI support can dynamically capture the format of the host screen, minor changes that do not affect the order of the fields will not affect your application.

2.3 Advantages and Disadvantages of Using EHLLAPI

Some of the **advantages** of using EHLLAPI and VisualAge are as follows:

- You do not have to change host applications to enable them to communicate with the workstation.
- You do not have to understand communications protocols.
- There is no need for complex hardware and software; EHLLAPI uses OS/2 Communications Manager/2.
- Communication between applications running in different environments is enabled. For example, you can write a VisualAge application that transfers data from IMS/DC to CMS, TSO, or CICS using the 3270 screen as the interface (see Figure 8 on page 15).
- It is easy to pass data from host applications to workstation products, such as Lotus** and Wordperfect**.

  For example, you can write a VisualAge application that executes an existing host application that displays the address of a customer in the host screen. The VisualAge application takes the data and starts a Wordperfect task that writes a letter to the customer.
- Changes to the host application that do not modify the screen do not affect the VisualAge application.
- Minor changes to host screens that do not affect the order of the fields do not affect your application.

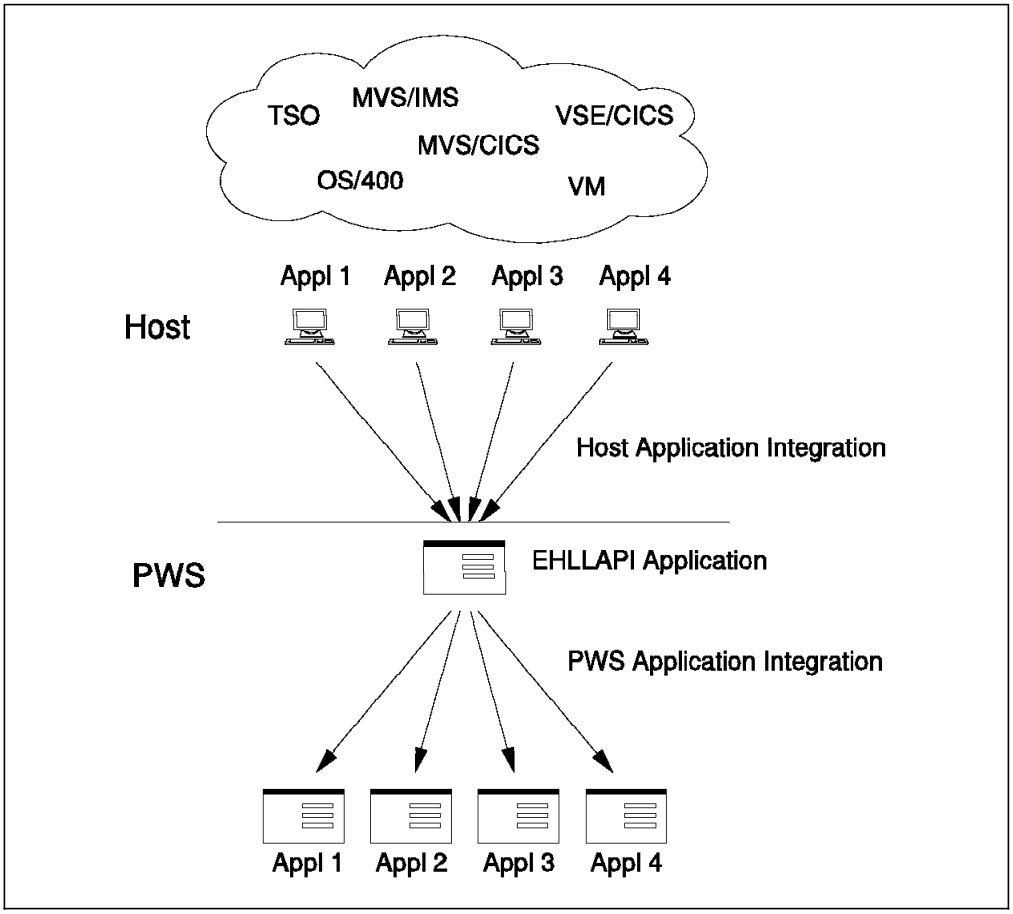

Figure 8. Application Integration through EHLLAPI Application

Some of the **disadvantages** of using EHLLAPI and VisualAge are as follows:

- Error handling is not as sophisticated as in other protocols, such as APPC or CICS-to-CICS communications.
- Performance is not as good as in other protocols.
- A LUW cannot span environments.
- EHLLAPI programs are very sensitive to changes in their environment, which makes them hard to reuse. This sensitivity is due to the number of parameters that must be adjusted to get EHLLAPI programs to work in a particular environment. The VisualAge EHLLAPI support makes EHLLAPI programs more protocol independent and therefore easier to reuse.
- Some applications have complex host maps that require good Smalltalk skills.

Chapter 3. Applying the VisualAge EHLLAPI Parts with a Simple Example

Before you start coding a real production GUI application that uses VisualAge′s EHLLAPI support, we suggest that you use a simple application to learn the concepts. This chapter describes a simple application that we created and used during our project and explains the lessons we learned and our conclusions.

3.1 Sample Application

To understand the VisualAge EHLLAPI parts we coded a sample application with one input and one output map. We used CSP/AD to create our sample application, but you can use any other tool, such as REXX, to create a similar example. Figure 9 and Figure 10 show the host maps of our simple host application. The protected map fields are marked with asterisks (*), and the unprotected map fields with dashes(-).

```
VA1OMO1                HOST APPLICATION - First Map

Enter ID...... ----

 ********************************************************************************
 PF3=END
```

Figure 9. Host Application Map 1. This map has one unprotected (-) and one protected (*) field.

```
VA1OMO2               HOST APPLICATION - Second map

Name .......  ****************

 ********************************************************************************
 ENTER=RETURN
```

Figure 10. Host Application Map 2. This map has two protected (*) fields.

Figure 11 through Figure 13 on page 18 show a sample execution sequence of the 3270 host screens:

- The user enters a valid ID (20) and presses the Enter key.

```
VA1OMO1                HOST APPLICATION - First Map

Enter ID...... 20


 PF3=END
```

Figure 11. Host Application Input Map

- The second map is shown, and the user presses the Enter key to return to the first map.

```
VA1OMO2                HOST APPLICATION - Second map

Name .......  REGI ANDORINHA

 PRESS ENTER TO RETURN
 ENTER=RETURN
```

Figure 12. Host Application Output Map

- If an invalid ID is entered on the first map, a message appears in the message line of the first map as shown in Figure 13.

```
VA1OMO1                 HOST APPLICATION - First Map

Enter ID......   11

 ENTER ID (10 OR 20)
 PF3=END
```

Figure 13. Host Application Input Map with an Invalid ID

Note that we can distinguish the following two situations in this simple application:

- The user enters an invalid ID; the first map is reshown with a message.
- The user enters a valid ID; the second map is shown with the requested data and a message.

3.2 Possible GUI Implementations

When you create a GUI application using VisualAge′s EHLLAPI support you have several implementation alternatives. You can use the Abt3270Screen part, the Abt3270Terminal part, or a combination of the two. You also may have to write VisualAge scripts in certain situations. Which implementation alternative you choose depends on the existing host application and the functional requirements of your new GUI application.

An important design decision for your new GUI application is whether the application should be controlled from the host or the PWS.

Having control at the host means that the host session triggers actions and events of the GUI application on the PWS. Each time the host 3270 screen changes, the *screenChanged* event is raised and triggers the actions to be taken by the GUI application.

Host control for a VisualAge EHLLAPI application must be implemented with the Abt3270Screen part.

Host control for a VisualAge EHLLAPI application can be compared to the *asynchronous* communication that is required by certain applications. For a simple example of a host-controlled VisualAge EHLLAPI application, see 3.3, “Implementing a GUI with Host Control” on page 21.

Having control at the PWS means that the VisualAge EHLLAPI application triggers the events and actions to execute functions at the host side. For example, the *findString* action searches for a text string in the 3270 host map and, depending on whether the text string is found or not, raises either the *searchSucessful* or the *searchFailed* event. The VisualAge EHLLAPI application on the PWS decides what to do next depending on the raised event.

PWS control for a VisualAge EHLLAPI application must be implemented with the Abt3270Terminal part.

PWS control for a VisualAge EHLLAPI application can be compared to *synchronous* communication. The PWS triggers an action and waits for an answer in the form of an event. For a simple example of a PWS-controlled VisualAge EHLLAPI application, see 3.4, “Implementing a GUI with PWS Control” on page 44.

We investigated the following alternatives to implement host or PWS control for the VisualAge EHLLAPI application that provides a GUI for our simple host application:

- Control at the host
  - Use the *Abt3270Screen part* only
  - Use the *Abt3270Screen part* and the *Abt3270Terminal part*.
- Control at the PWS
  - Use the *Abt3270Terminal part*, but no VisualAge scripts, only visual programming
  - Use the *Abt3270Terminal part*, and VisualAge scripts.

Figure 14 on page 20 shows the two approaches we used to implement a GUI front end with host control for our simple host application.

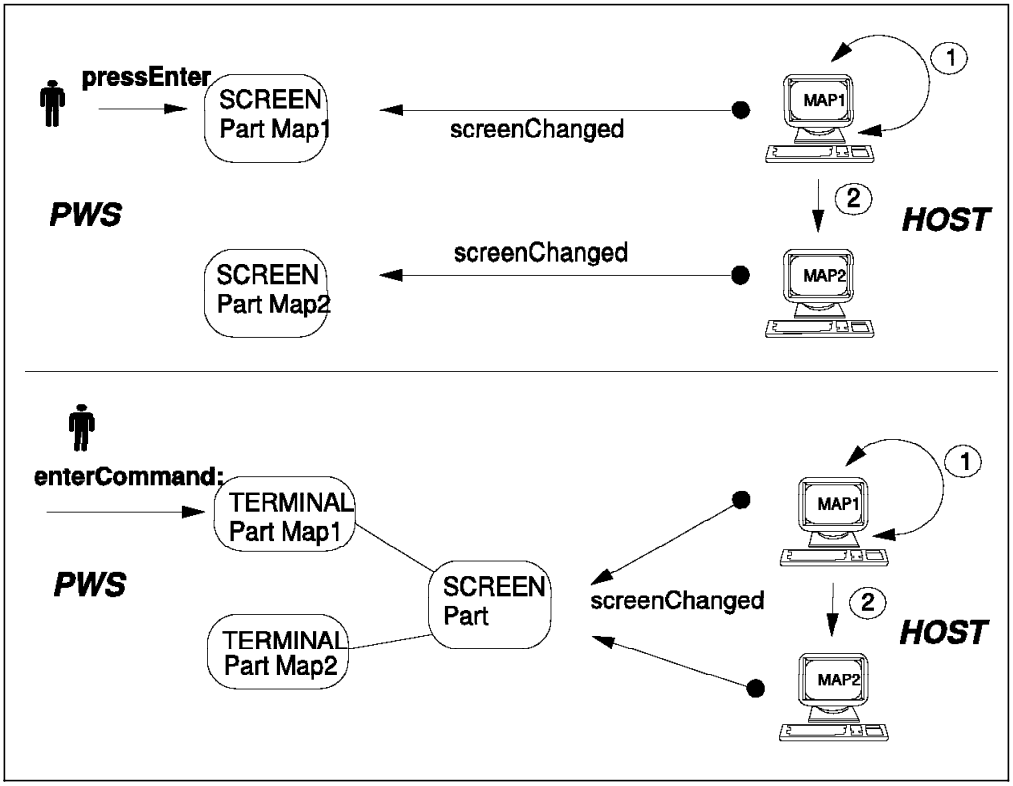

Figure 14. Host Control Implementations

Figure 15 shows the two approaches we used to implement a GUI front end with PWS control for our simple host application.

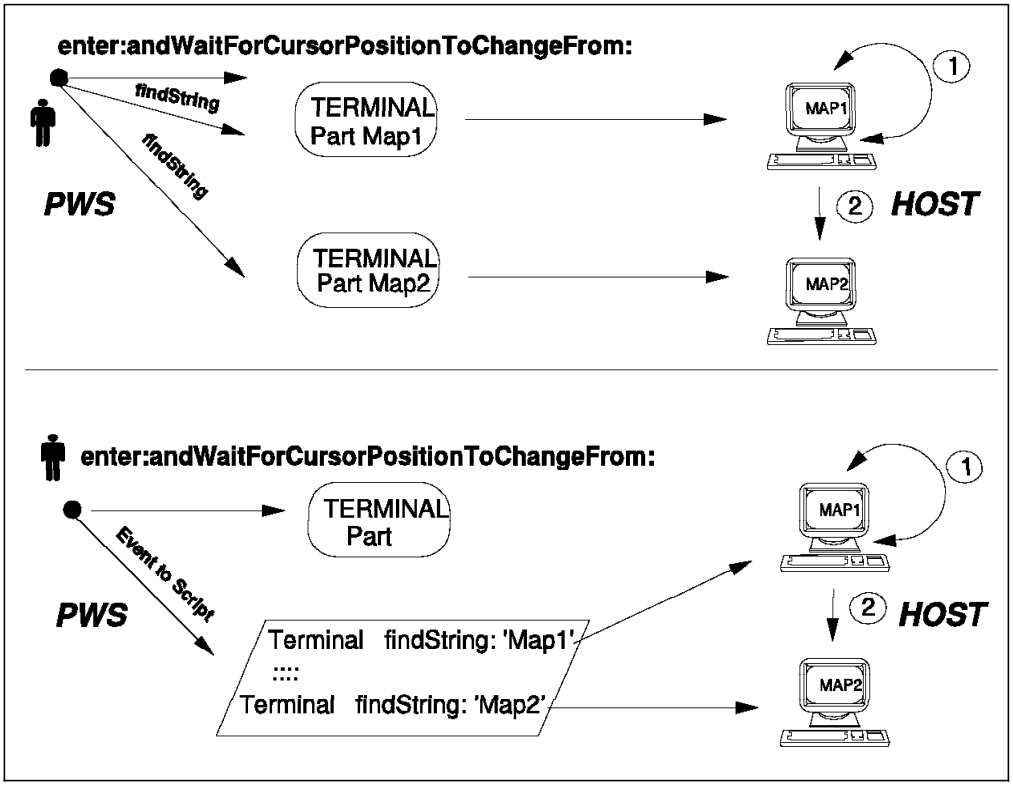

Figure 15. PWS Control Implementations

3.3 Implementing a GUI with Host Control

As mentioned before, you must use the Abt3270Screen part to implement host control for your VisualAge EHLLAPI application. We implemented two alternative approaches without writing VisualAge scripts as follows:

- Using the Abt3270Screen part only
- Using the Abt3270Screen part and the Abt3270Terminal part.

Details of our implementations are provided below.

3.3.1 Abt3270Screen Part Only

Figure 16 shows our implementation using the Abt3270Screen part only.

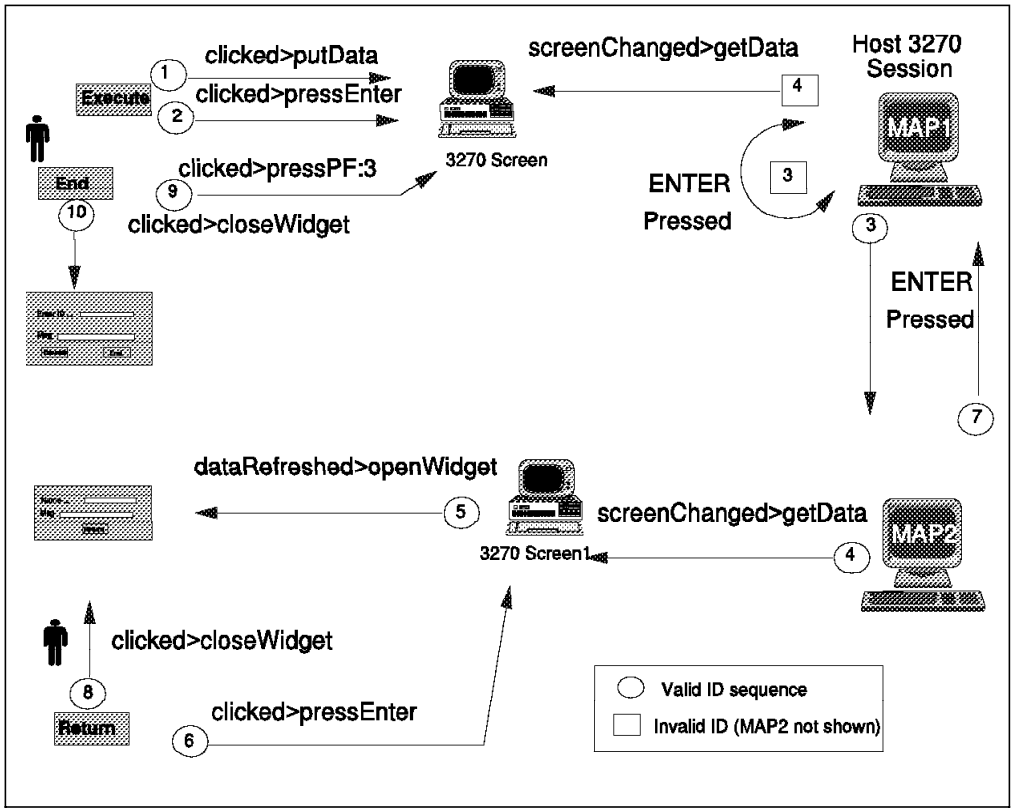

Figure 16. Sequence of Events and Actions Using the Abt3270Screen Part

The sequence of events and actions is as follows:

1. The user clicks on the *Execute* push button, which triggers the *putData* action to send the input data to the host.
2. After sending the input data, the *pressEnter* action is sent to the host.
3. The host application executes the Enter key based on the *pressEnter* action.
4. The *screenChanged* event is raised when the host screen changes and either MAP1 or MAP2 is shown. The *screenChanged* event is raised in each Abt3270Screen part in the VisualAge EHLLAPI application. Only the Abt3270Screen part for which the key string matches (MAP 1 or MAP 2) executes the *getData* action, as follows:
   - If the user entered an invalid ID, a message is shown in MAP1, and this message appears in the first GUI window.

- If the user entered a valid ID, MAP2 is shown at the host.

5. When MAP2 is shown and *getData* is executed, the *dataRefreshed* event is raised and triggers the *openWidget* action to open the second GUI window. The data from the host map is shown in the second GUI window.
6. The user clicks on the *Return* push button, which triggers the *pressEnter* action for MAP2.
7. The host application executes the Enter key.
8. Clicking on the *Return* push button also executes the *closeWidget* action, and the second GUI window is closed.
9. The user clicks on the *End* push button, and the *pressPF:3* action is sent to the host.
10. Clicking on the *End* push button also triggers the *closeWidget* action to close the first GUI window.

3.3.1.1 Coding the Application

Perform the following steps to code a VisualAge EHLLAPI front end using the Abt3270Screen part only:

1. Start the host session using Communications Manager/2 and the host application for which you create the GUI windows. In our example the host session was defined as **G**.

   Note: It is important to have the session active before starting the Composition Editor. If you try to start the Composition Editor for a VisualAge application that includes Abt3270Screen parts and the session defined in those Abt3270Screen parts is not started, you will receive a Smalltalk walkback window, as shown in Figure 17.

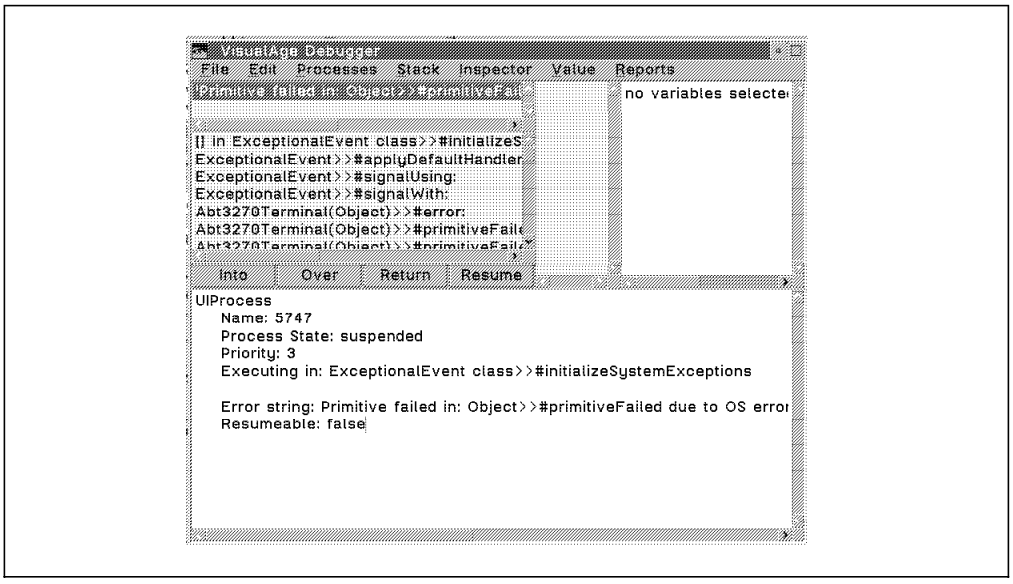

Figure 17. Debugger Window When Session Not Available

2. Start VisualAge and create an application and a visual part. Be sure to follow a naming convention for the part names. Refer to 4.1.2, “Naming Convention for Parts” on page 81 for a proposed naming convention.
3. Using the Composition Editor, drag the Abt3270Screen part from the parts palette and drop it on the free form surface.

4. Select the 3270 Screen icon that you dropped and double-click mouse button 1. Figure 18 on page 23 shows the screen settings for the first host screen. Do not forget to click on the *Build Screen Records* push button to initiate VisualAge's parsing of the host screen to capture the protected and unprotected fields.

   Be sure to select a unique text string on the host screen to use for the *Key String* field. The *Key String* can be up to 132 characters long. The Abt3270Screen part uses the *Key String* together with the specified session ID to decide whether a raised event is relevant for this particular Abt3270Screen part. A blank *Key String* matches every Abt3270Screen part in the application.

   If your host screens do not have unique text strings that can be used as key strings, you may want to use the Abt3270Terminal part or VisualAge scripts.

   Note: To ensure that the specification for the *Key String* exactly matches what is defined on the host screen you may want to use the OS/2 copy/paste functions to copy the text string from the host screen and paste it on the 3270 screen settings.

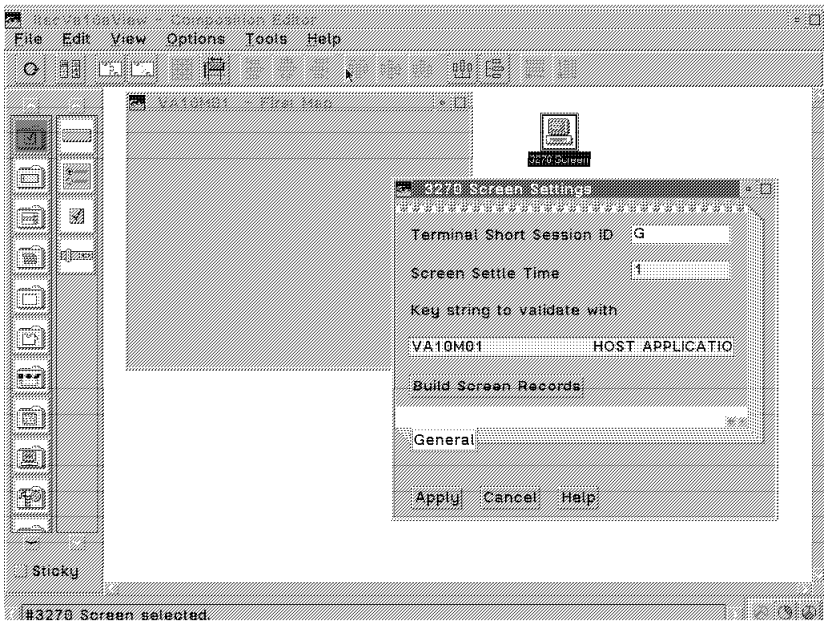

Figure 18. 3270 Screen Settings: Abt3270Screen Part Only

5. With the 3270 Screen icon selected, click mouse button 2 and select Tear-Off Attributes from the pop-up menu to tear off inputFields and outputFields as shown in Figure 19 on page 24.

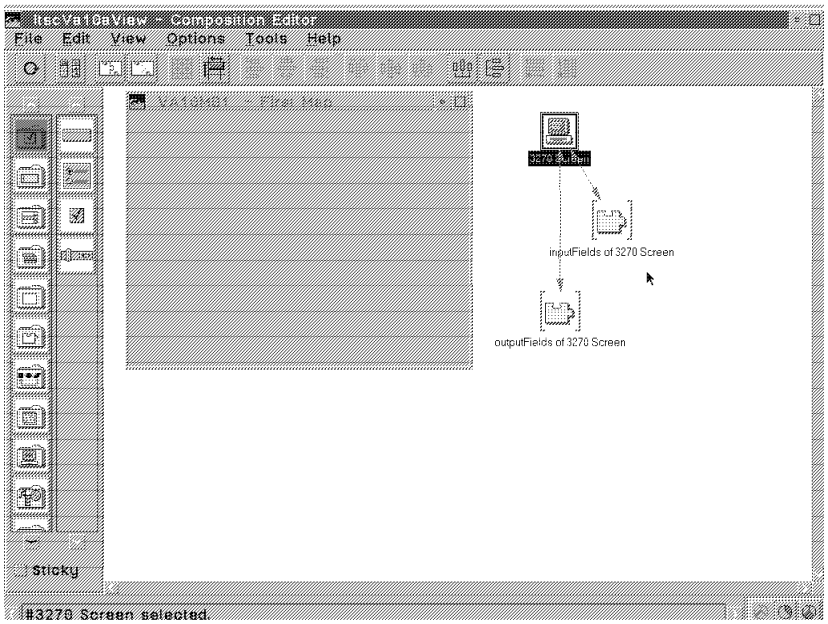

Figure 19. Tear-Off Attributes

6. With the inputFields of the 3270 Screen icon selected, click mouse button 2 and select Quick Form from the pop-up menu to create a quick form for self (or FieldNumber1). Move the mouse to the desired position in the window part and click mouse button 1. Figure 20 shows the result of the operation.

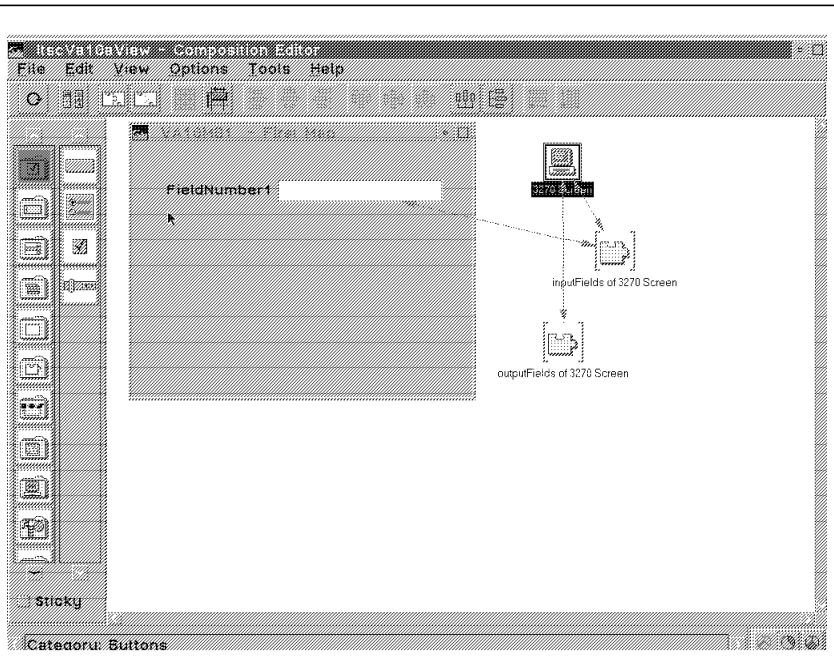

Figure 20. Create the Input Field

7. Identify which output field represents the message field and must be connected to the window.

VisualAge collects all protected fields as output fields when it parses the host screen. Not all protected fields are actually output fields for the application. Some of the protected fields contain constants, such as titles and field names. Therefore VisualAge collects more output fields than you might expect. For example, for our simple application VisualAge collected five output fields, and the only one we needed to connect (*Msg*) was *FieldNumber2*.

It is hard to map the input and output fields collected by VisualAge to the actual fields on the host screen, especially when the host screen contains a considerable number of unprotected and protected fields. Because VisualAge parses the host screen and not the code of the host map, the field names assigned by VisualAge do not correspond to the logical names used in the host application. All fields are named FieldNumberx (where x is a sequence number), which does not make the mapping any easier.

You can use the Quick Form option to map all input and output fields to a window and then execute the application to get the values of all fields. This approach can be useful with host screens with only a few fields.

Figure 21 shows the output fields for our simple application.

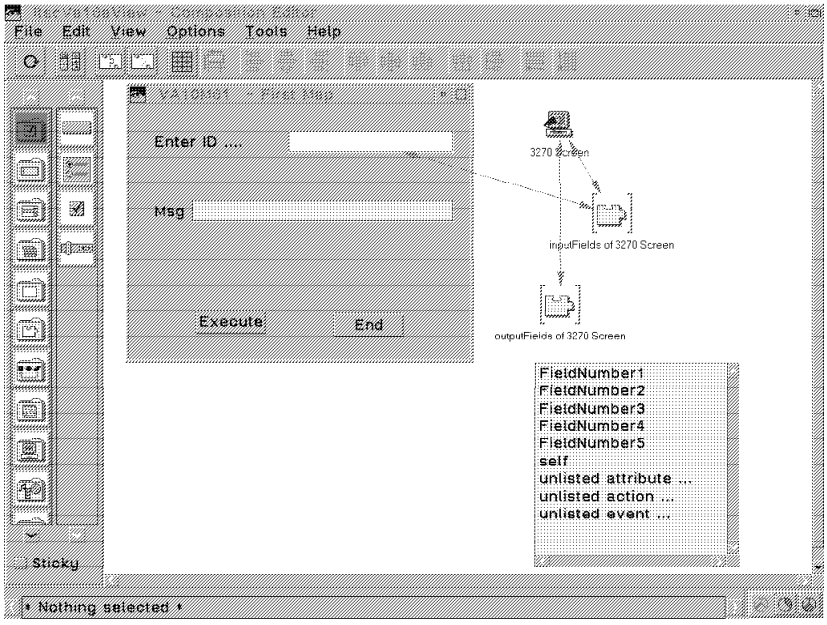

Figure 21. 3270 Screen Output Fields

We created a tool during our project to make the input and output field mapping easier. This tool, the Screen Field Monitor, is an independent application that collects all input and output fields from the host screen and shows their contents, position, and size in two list boxes. Appendix A, “Screen Field Monitor Tool” on page 213 provides more details about this tool. Figure 22 on page 26 shows the input and output fields for Map1 using our tool.

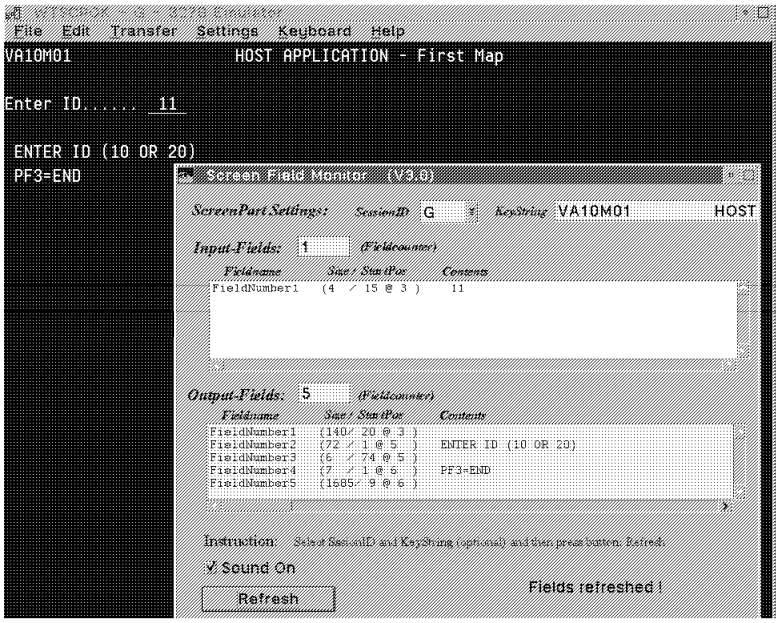

Figure 22. Input and Output Fields for Map1 Using the Screen Field Monitor

8. Connect the output field that contains the host message (FieldNumber2) to the first window. Figure 23 shows the result of the operation.

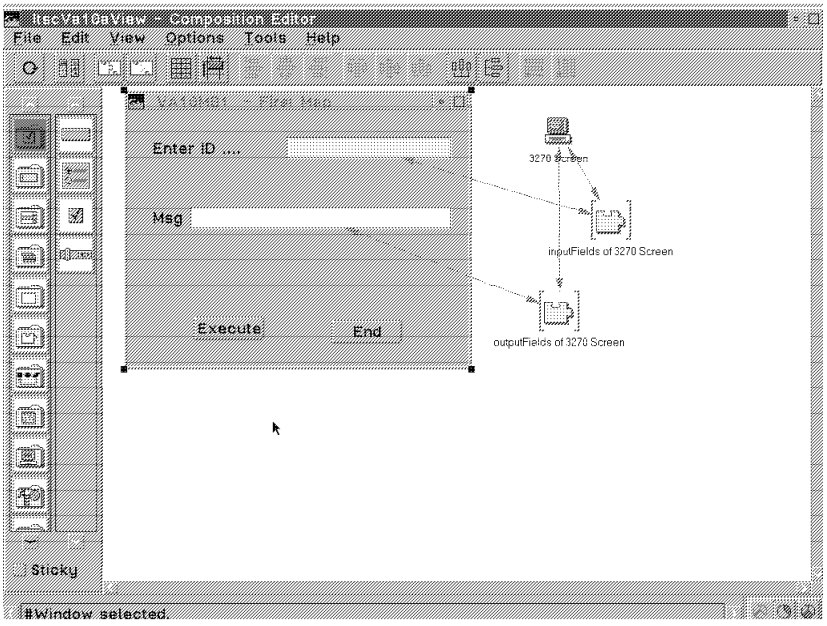

Figure 23. Connecting Map1 Output Field

9. Drag two push button parts from the parts palette and drop them on the window. Align the push buttons and label them *Execute* and *End*, respectively.

10. Make two connections from the *Execute* push button to the 3270 Screen:

    - Connect the *Execute* push button (#clicked) to 3270 Screen (#putData).

- Connect the *Execute* push button (#clicked) to 3270 Screen (#pressEnter).

The sequence in which you define the connections corresponds to the execution sequence of the application. Be sure to make the connections in the correct sequence.

11. Connect the 3270 Screen to itself:

    - Connect 3270 Screen (#screenChanged) to 3270 Screen (#getData).

    Figure 24 shows the settings for the connections.

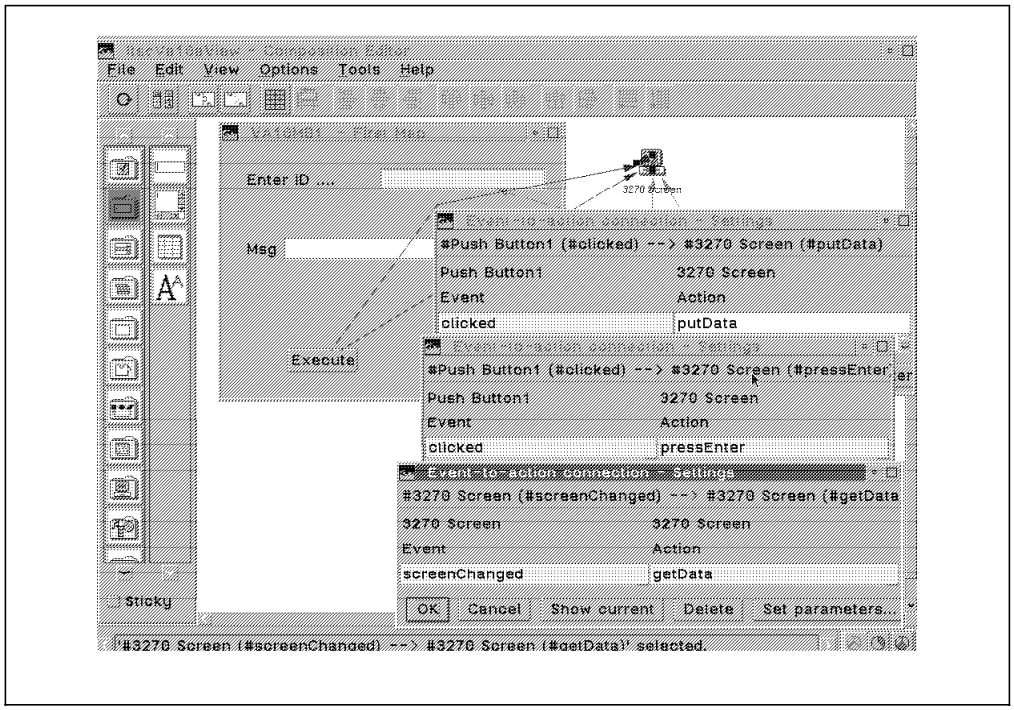

Figure 24. Connections for First GUI Window

12. Create the GUI window for the second host screen as you did for the first host screen.

    We used the Screen Field Monitor to identify the output fields for host Map2. Figure 25 on page 28 shows the input and output fields of host Map2 using our tool.

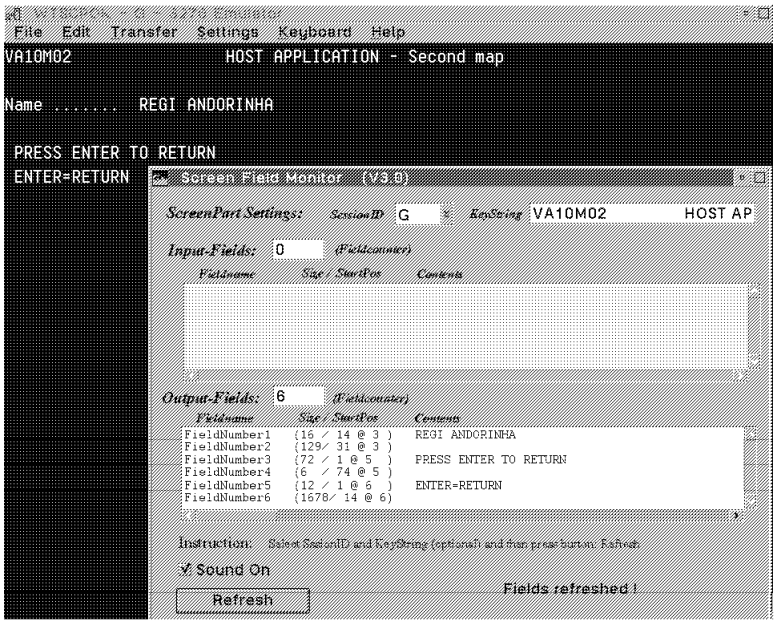

Figure 25. Input and Output Fields for Map2 Using the Screen Field Monitor

The output fields to be connected are *FieldNumber1* (Name) and *FieldNumber3* (Msg). Figure 26 shows the second GUI window and the output field settings.

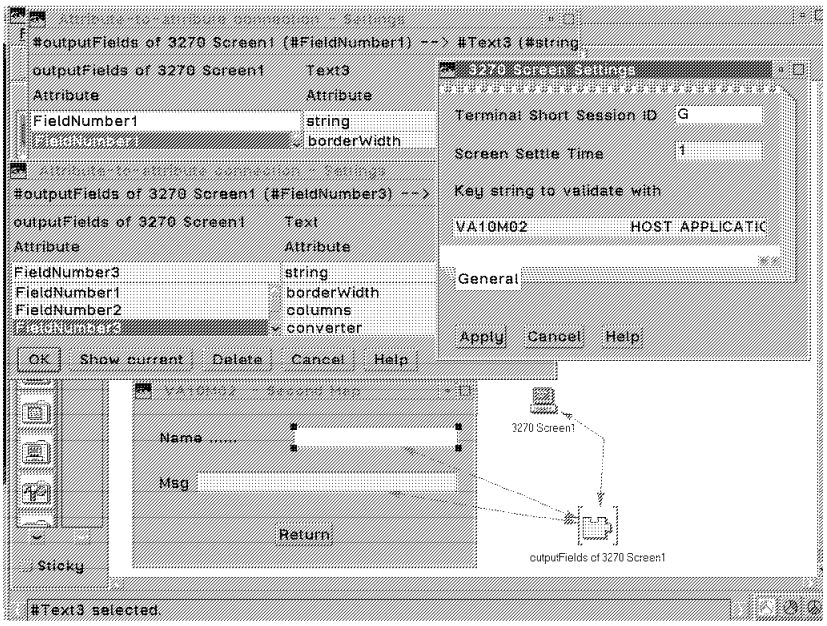

Figure 26. Output Fields for Map2

- The second host screen has no input fields and it is not necessary to tear off attributes for input fields.

- The *Key String* for 3270 Screen1 must be set to a unique text string on the second host screen (Map2).

13. Connect 3270 Screen1 to itself:
    - Connect 3270 Screen1 (#screenChanged) to 3270 Screen1 (#getData).
14. Connect 3270 Screen1 to the second window:
    - Connect 3270 Screen1 (#dataRefreshed) to window1 (#openWidget).

    The second GUI window will be opened when the second map is shown at the host. Figure 27 shows the second GUI window and its connection settings.

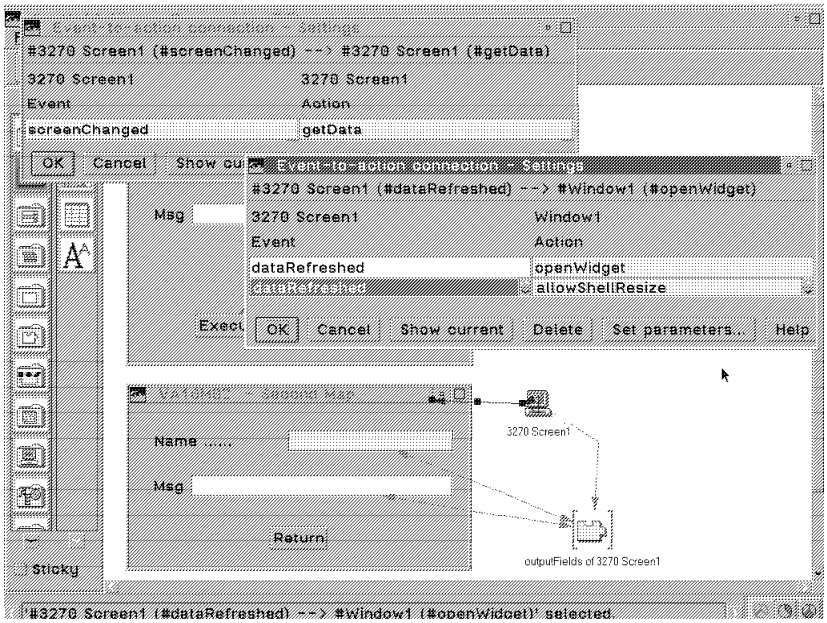

Figure 27. Connections for Second GUI Window

15. Final connections:
    - Connect the *Return* push button (#clicked) to 3270 Screen1 (#pressEnter).
    - Connect the *Return* push button (#clicked) to window1 (#closeWidget).
    - Connect the *End* push button (#clicked) to the 3270 Screen (#pressPF:).
    - Double-click on the dashed (---->) line, select Set Parameter, and enter 3 (PF3 ends the application).
    - Connect the *End* push button (#clicked) to the first window (#closeWidget).

    Figure 28 on page 30 shows all connections.

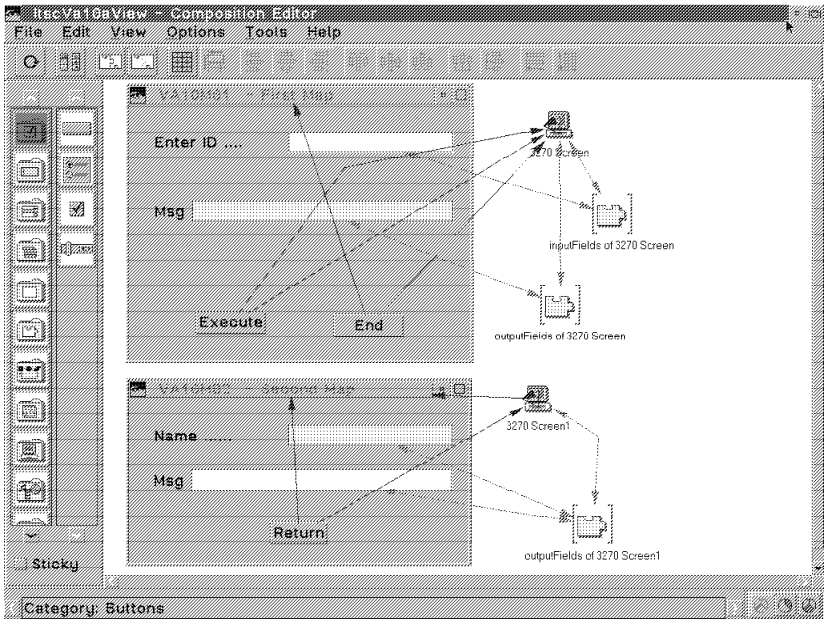

Figure 28. All Connections: Abt3270Screen Part Only

3.3.1.2 Executing the Application

Figure 29 through Figure 31 on page 31 show what the user sees when the applicaton is executing. In the figures, the host screens are black, and the GUI windows are shown as the active windows.

The steps to execute the application are as follows:

1. The user enters a valid ID (20) and clicks on the *Execute* push button.

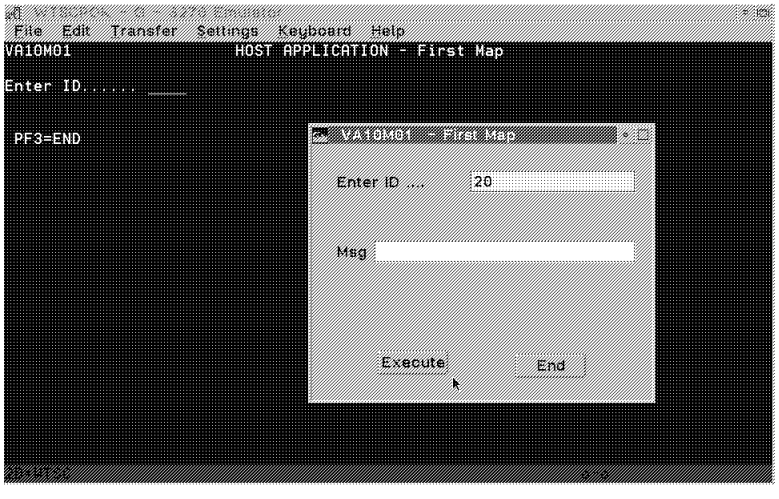

Figure 29. User Enters a Valid ID

2. The second window is displayed.

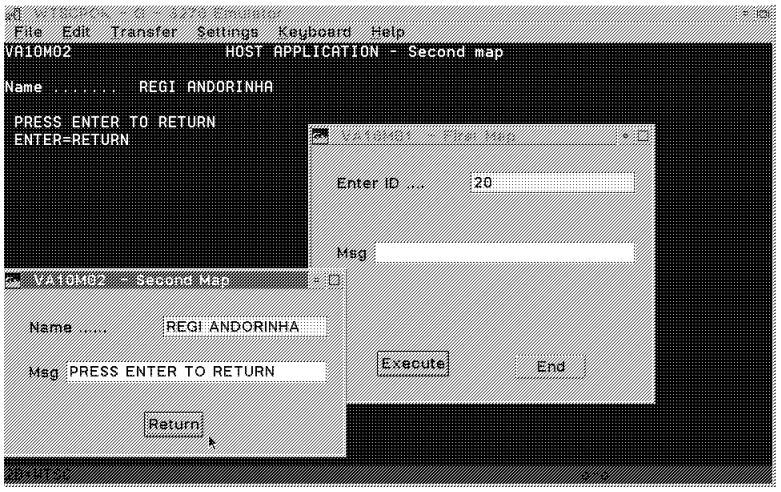

Figure 30. Second Window is Displayed

3. The user clicks on the *Return* push button on the second window, enters an invalid ID (22) in the first window, and clicks on the *Execute* push button.
4. A message is displayed in the first window.

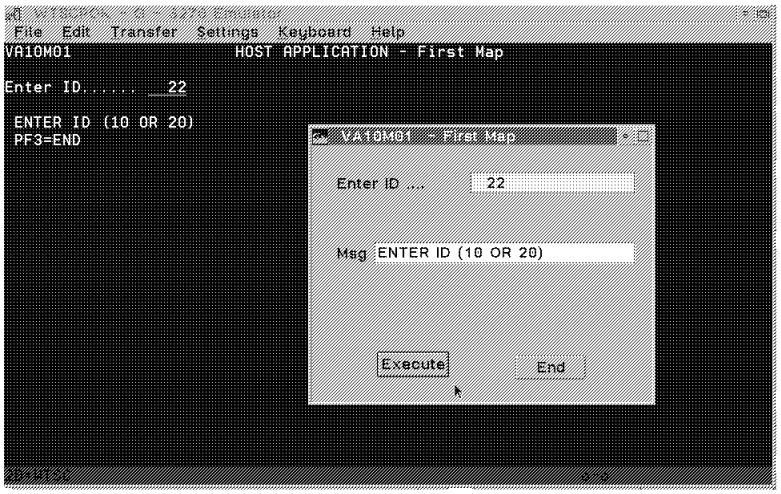

Figure 31. User Enters Invalid ID

3.3.1.3 Advantages and Disadvantages of This Approach

Advantages

- Easy to code
- Does not require Smalltalk skills; all coding is visual.

Disadvantages

- Hard to identify the input and output fields during development. For example, one host screen can have 50 or 60 output fields, and it is difficult to identify the fields.

  This disadvantage disappears when using the tool we created during our residency project. See Appendix A, “Screen Field Monitor Tool” on page 213 for details of this tool.

- Poor response time compared to other implementations during execution (testing and run time) because of the *Screen Settle Time* delay. The *Screen Settle Time* specified in the Abt3270Screen part settings adds to the normal response time for the application.

- The host session must be active at development and run time when using the Abt3270Screen part. For example, if a developer specifies session G in the settings for the Abt3270Screen part, the G session always must be started when the Composition Editor for the part containing the Abt3270Screen part is started and when the packaged application is executing.

3.3.2 Abt3270Screen Part and Abt3270Terminal Parts

In this section we describe the implementation where we mixed the Abt3270Screen part and Abt3270Terminal part with VisualAge scripts (see Figure 32 on page 33). We decided not to use the *getData* action of the Abt3270Screen part because we wanted to avoid the *Screen Settle Time* delay. We also wanted to demonstrate that the Abt3270Screen part and the Abt3270Terminal part can be combined **but application control can still be at the host**. Note that we used two Abt3270Terminal parts and one Abt3270Screen part for this implementation.

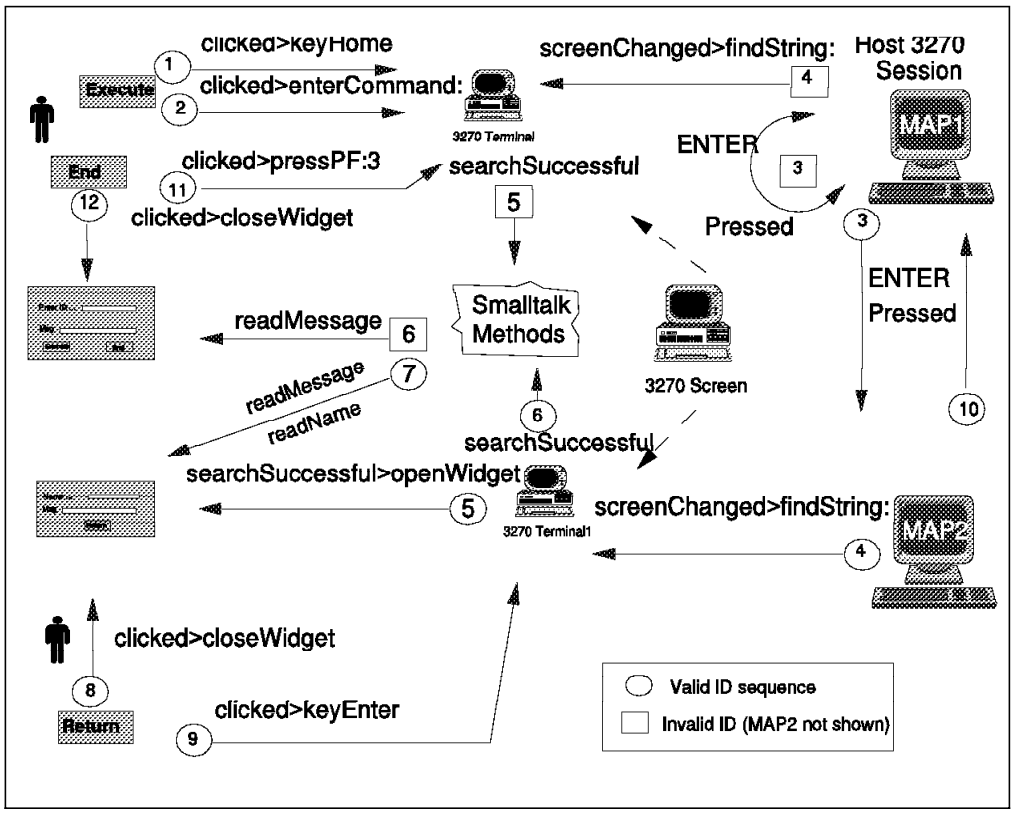

Figure 32. Sequence of Events and Actions When Using Abt3270Terminal Parts and an Abt3270Screen Part

The sequence of events and actions is as follows:

1. The user clicks on the *Execute* push button, and the *keyHome* action moves the cursor to the first input field on the host screen (this ensures that data is entered in the correct position).
2. The *enterCommand* action sends the input data to the host and activates Enter.
3. The host application executes the Enter key.
4. The *screenChanged* event is raised when the host screen changes and either MAP1 or MAP2 is shown. The *findString* action is executed for each Abt3270Terminal part, as follows:
   - If the user entered an invalid ID, a message is shown in MAP1, and this message appears in the first GUI window.
   - If the user entered a valid ID, MAP2 is shown at the host.
5. This step depends on which host screen is shown and therefore which *findString* action yields the *searchSuccessful* event.

   If the first host screen (Map1) is shown and the *findString* action raises the *searchSuccessful* event, the *readMessage* action (script) is executed.

   If the second host screen (Map2) is shown and the *findString* action raises the *searchSuccessful* event, the *openWidget* action is executed to open the second GUI window.
6. If the first host screen (Map1) is shown at the host, the *readMessage* action moves the message information to the first GUI window.

If the second host screen (Map2) is shown, the *searchSuccessful* event as a result of the successful execution of the *findString* action triggers the execution of the *readMessage* and *readName* actions.

7. The *readMessage* and *readName* actions move the message information (or blanks) and the contents of the name fields to the second GUI window.
8. The user clicks on the *Return* push button, which executes the *closeWidget* action, and the second GUI window is closed.
9. Clicking on the *Return* push button also triggers the *pressEnter* action for MAP2.
10. The host application executes the Enter key.
11. The user clicks on the *End* push button, and the *pressPF:3* action is sent to the host.
12. Clicking on the *End* push button also triggers the *closeWidget* action to close the first GUI window.

3.3.2.1 Coding the Application

Perform the following steps to code a VisualAge EHLLAPI front end using a combination of an Abt3270Screen part and two Abt3270Terminal parts:

1. Start the host session using Communications Manager/2 and the host application for which you create the GUI windows. In our example the host session was defined as **A**.

   Note: It is important to have the session active before starting the Composition Editor. If you try to start the Composition Editor for a VisualAge application that includes Abt3270Screen parts and the session defined in those Abt3270Screen parts is not started, you will receive a Smalltalk walkback window.

2. Start VisualAge and create an application and a visual part. Be sure to follow a naming convention for the part names. Refer to 4.1.2, “Naming Convention for Parts” on page 81 for a proposed naming convention.
3. Using the Composition Editor, drag the Abt3270Screen part from the parts palette and drop it on the free form surface. Even though we used the Abt3270Terminal part, we needed an Abt3270Screen part for the *screenChanged* event for host control.
4. Select the 3270 Screen icon that you dropped and double-click mouse button 1. Figure 33 on page 35 shows the screen settings for the first host screen.

   For this example we specified the A session in the *Terminal Short Session ID* field; specification of a *Key String* is not required. You do not need to click on the *Build Screen Records* push button, as host screen parsing is not required for this example.

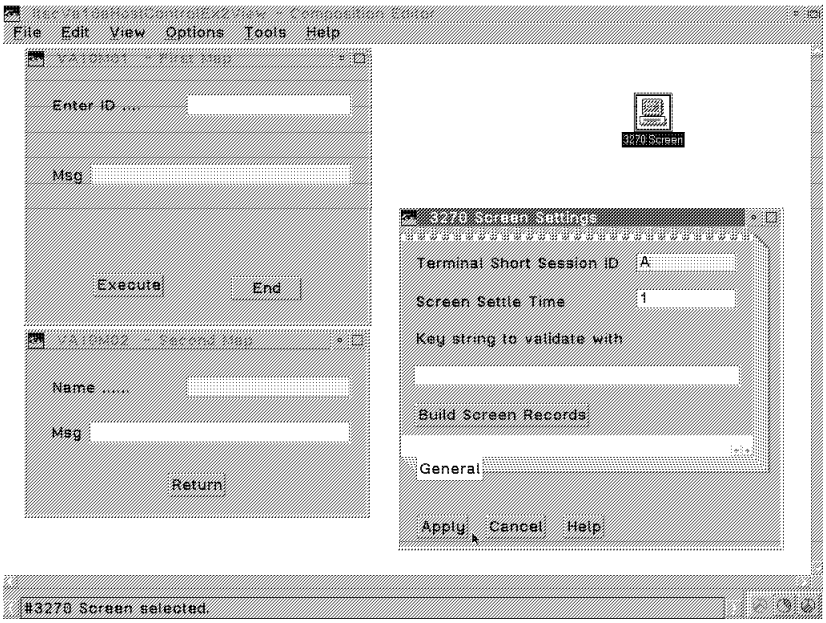

Figure 33. 3270 Screen Settings: Abt3270Screen and Abt3270 Terminal Parts

5. Using the Composition Editor, drag two Abt3270Terminal parts from the parts palette and drop them on the free form surface (one for each host screen). Add two data entry fields to the free form surface. These two data entry fields are used to specify a text string that uniquely identifies each of the host screens. Double-click on the data entry fields to open their settings view, which allows you to define a static text string for each of the data entry fields. In our example, we specified the host map Ids *VA10M01* and *VA10M02* (see Figure 34).

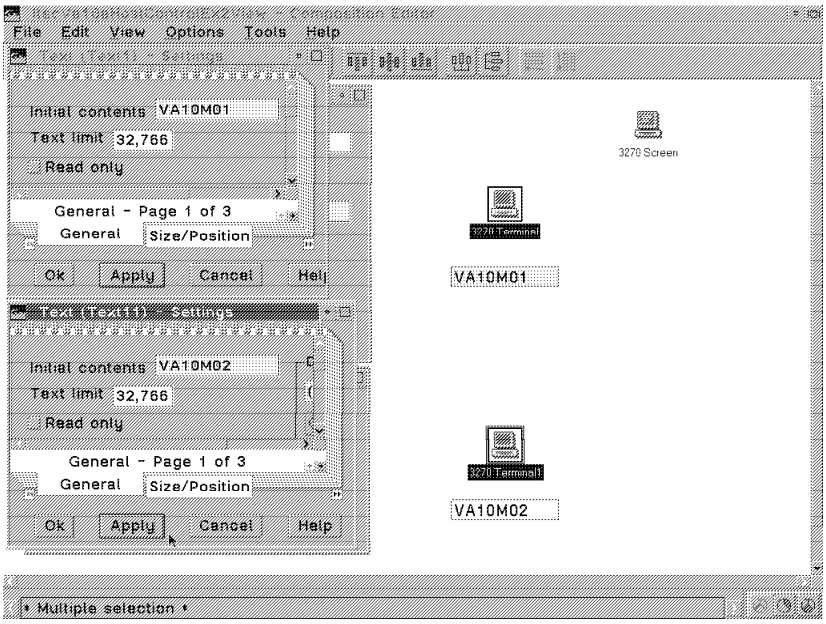

Figure 34. Adding the Abt3270Terminal Parts

We used two Abt3270Terminal parts in this example because the *findString* action is used twice (once for each host screen), and we would not have been able to distinguish which execution of the action was successful when using a single Abt3270Terminal part.

We could have used a script instead of two Abt3270Terminal parts.

6. Connect the *3270 Screen*, the *3270 Terminal*, and the data entry field as follows:

   - Connect 3270 Screen (#screenChanged) to 3270 Terminal (#findString:). This is an incomplete connection (dashed); the next connection will provide the value of the data entry field.
   - Connect Text1 (#string) to the connection (#aString).

   Figure 35 shows the connections for the first Abt3270Terminal part.

   Repeat step 6 to create the connections for the second Abt3270Terminal part.

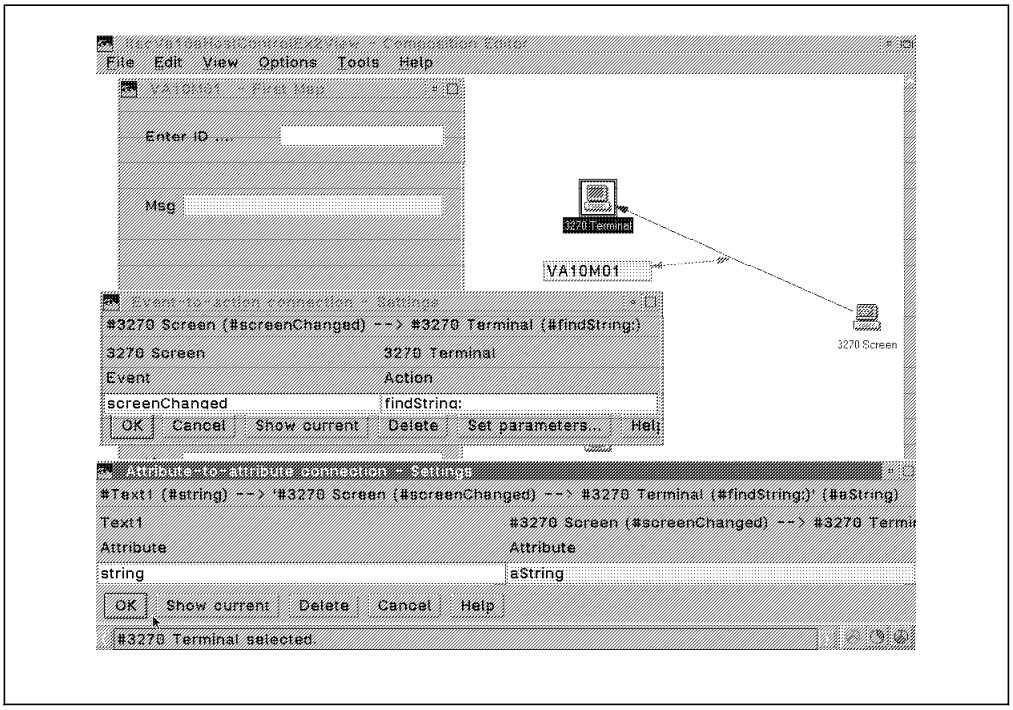

Figure 35. Connecting the Abt3270Screen Part and the Abt3270Terminal Part

7. Connect the *Execute* push button to the first Abt3270Terminal part:

   - Connect the *Execute* push button (#clicked) to 3270 Terminal(#keyHome).
   - Connect the *Execute* push button (#clicked) to 3270 Terminal (#enterCommand:).
     This is an incomplete connection (dashed); the next connection will provide the value of the Enter ID field.

     Connect the *Text3* field (#string) to the connection (#aString).

   The sequence in which you define the connections corresponds to the execution sequence of the application. Be sure to make the connections in the correct sequence. Figure 36 on page 37 shows the settings for the connections.

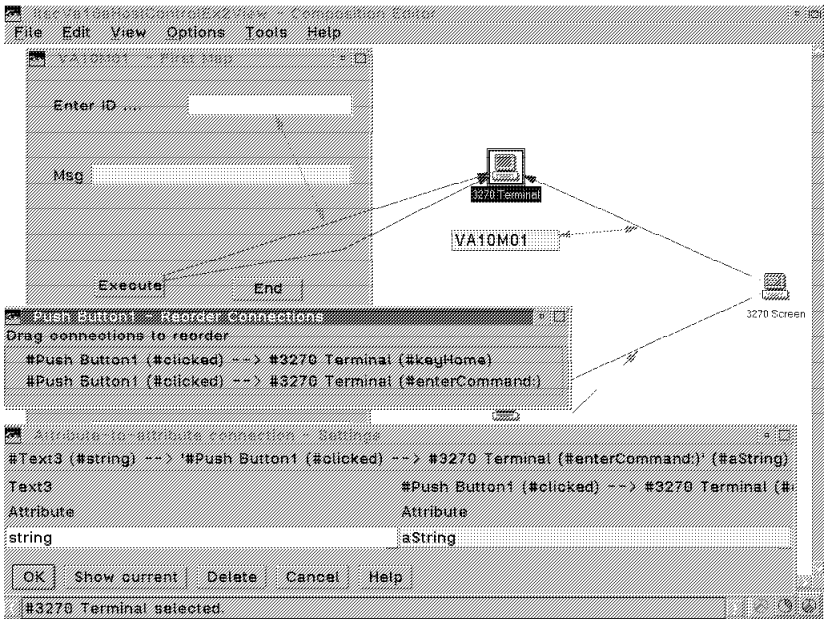

Figure 36. Connecting the Execute Button

8. The connections for the second GUI window are as follows:
   - Connect 3270 Terminal1 (#searchSucessful) to Window1 (#openWidget)
   - Connect the *Return* push button (#clicked) to 3270 Terminal1 (#keyEnter)
   - Connect the *Return* push button (#clicked) to Window1 (#closeWidget).

   Figure 37 shows the settings for the connections.

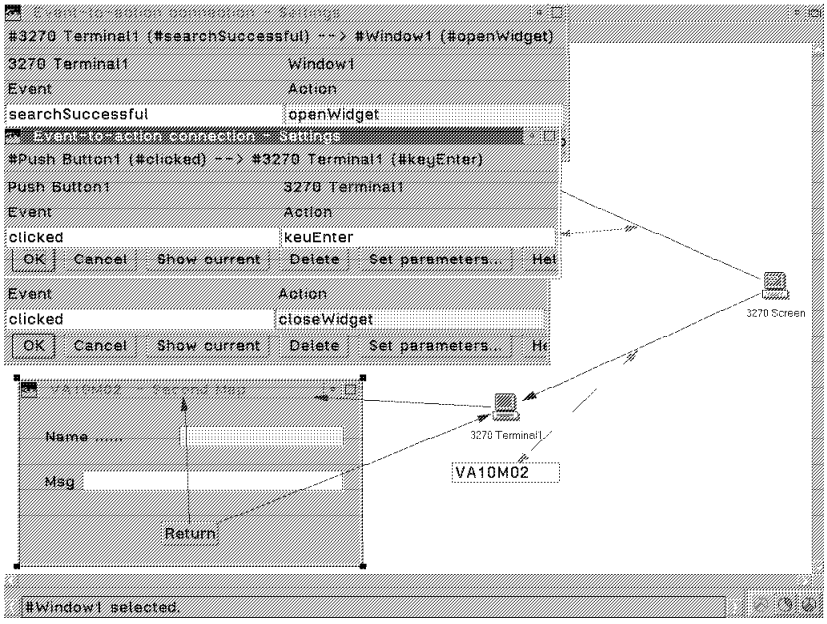

Figure 37. Connections for Second GUI Window

At this point we had some connections, but we needed to get the data from the host screens into our windows. We created two VisualAge scripts for that purpose.

3.3.2.2 Using the VisualAge Script Editor to Create Instance Methods

We did not use the *getData* action to get the data from the host screens, we used the *stringAt:for:* action. This action returns the contents of the screen starting from a specific row and column on the host screen and in a specified length. The action takes two parameters. The first parameter is the location on the screen (*point*), and the second parameter is an *integer* specifying the length.

We decided to provide the required parameters for the *stringAt:for:* action through two VisualAge scripts. Alternatively we could have specified the required parameters through the settings view of the event-to-action connection from *searchSuccessful* to *stringAt:for:*.

We used the VisualAge Script Editor to create the Smalltalk instance methods (scripts in VisualAge). We created two methods, *readMessage* and *readName*, to get the data from the host screens and put it into the GUI windows. Perform the following steps to create the methods:

1. Select View from the action bar and then Script Editor to switch to the Script Editor. Select Methods from the action bar of the Script Editor and then New Method Template (see Figure 38).

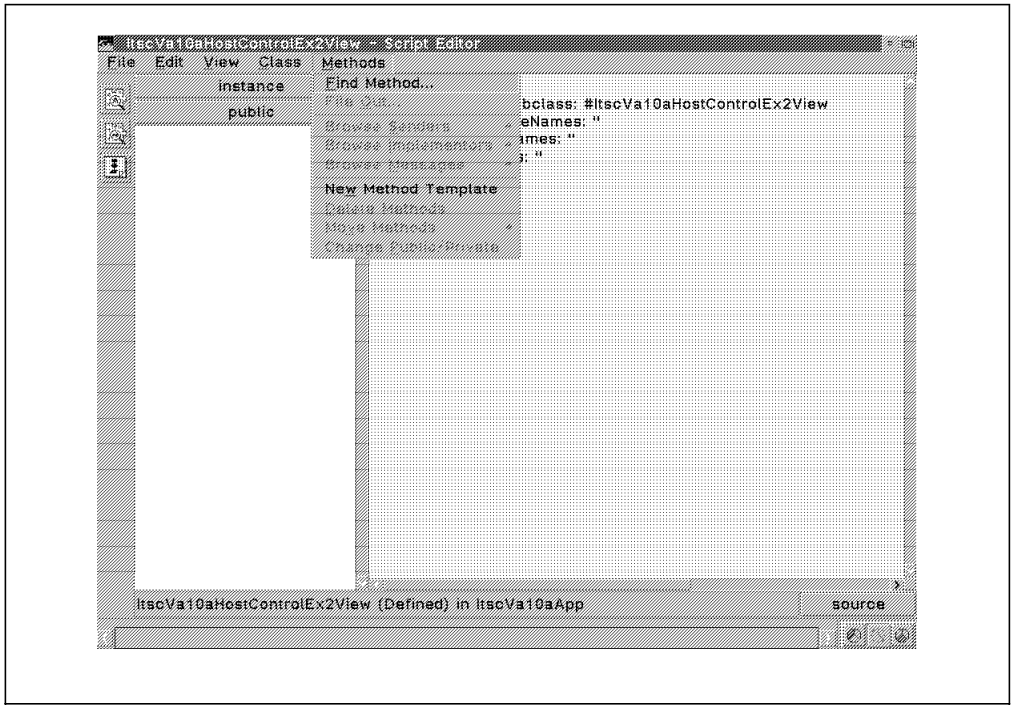

Figure 38. Creating a New Method

2. To create the method, type the method name (*readMessage*) in the template (replace *messagePattern*), provide a description for your new method (between quotes), and delete the other lines (temporaries, statements).
3. Select the Action icon from the tools palette.

4. Form the subparts list, select *3270 Terminal*, and from the actions list select *stringAt:for:* (see Figure 39).
5. Click on the *Paste* push button to paste the selected method to your Script Editor. You need to modify the pasted method to suit your needs.

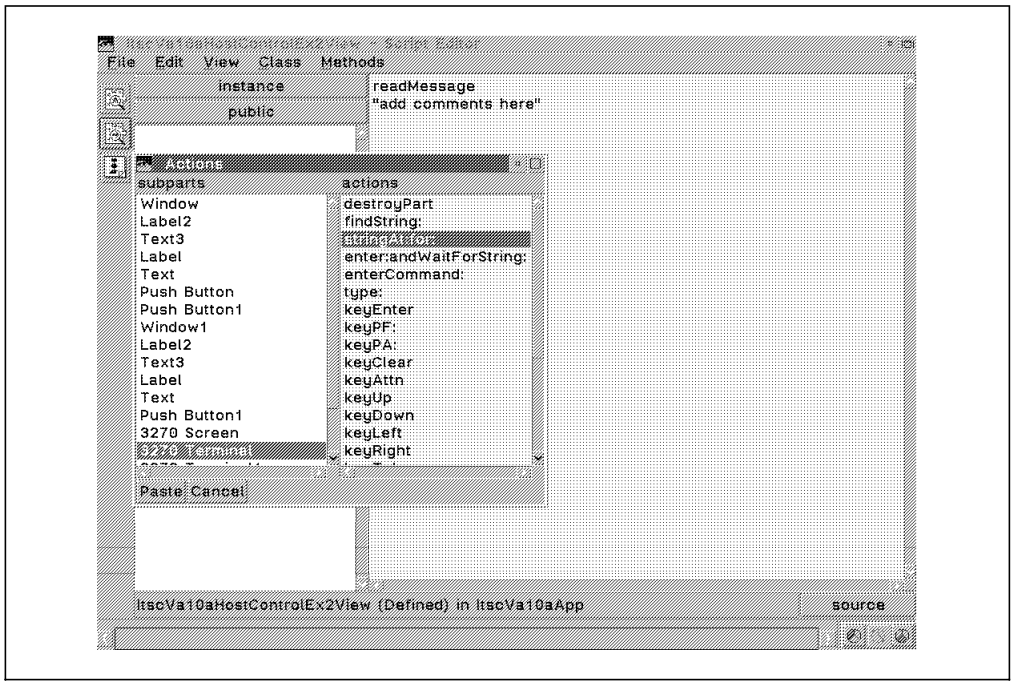

Figure 39. Action Selection for Method Creation

6. Change the arguments of the pasted method to the position of the host field (message field) and provide the field length. Change "<aPoint (Point)> with: <aLength (Integer)>" to "2 @ 5 with: 72."
7. Add a *caret* **^** before *self* to get the string value.
8. Select File from the action bar and then Save Script to save the method. The finished *readMessage* method is shown in Figure 40.

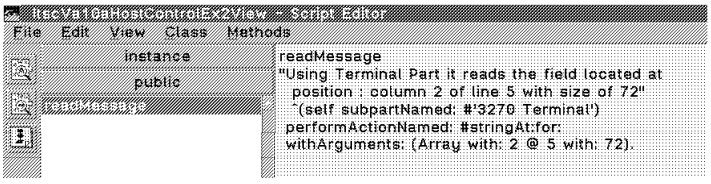

Figure 40. Method: readMessage

Repeat the previous steps to create the *readName* method. This method will get the field *Name* located at row 3, column 15, and length 14 from the host screen. The finished *readName* method is shown in Figure 41 on page 40.

```
readName
 "Using Terminal Part it reads the field located at
  position : column 15 of line 3 with size of 14"
 ^(self subpartNamed: #'3270 Terminal1')
   performActionNamed: #stringAt:for:
   withArguments: (Array with: 15 @ 3 with: 14).
```

Figure 41. Method: readName

3.3.2.3 Event-to-Script Connections

At this point you have most of the connections and the methods to get the data from the host, but you need to tell VisualAge when to execute the methods. You need to make the following connections:

1. Switch back to the Composition Editor.
2. Click mouse button 2 on the *3270 Terminal* and select *Event-to-script connection....*
3. Select the *searchSucessful* event and the *readMessage* script in the settings view (see Figure 42).

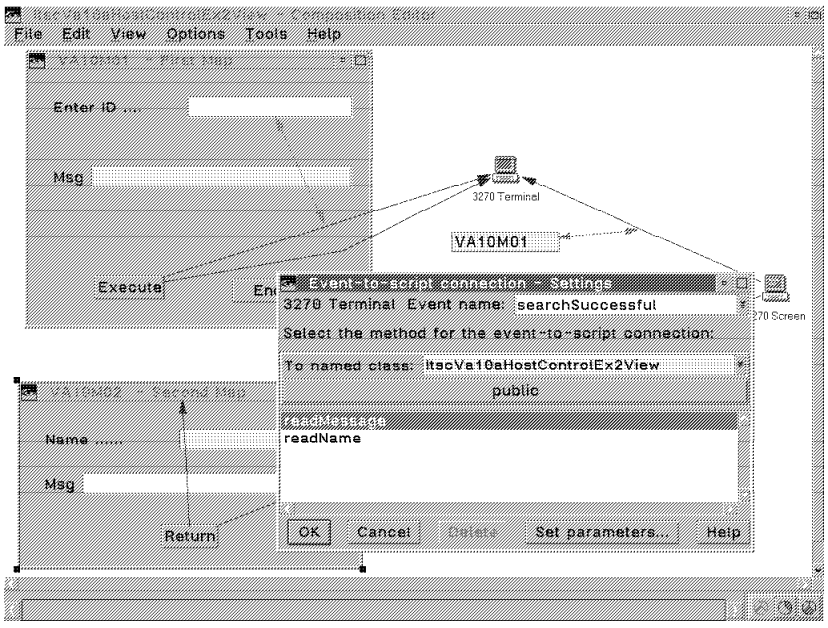

Figure 42. Event-to-Script Connections for One Terminal

4. Repeat Steps 1 through 3 for the *3270 Terminal1*. 3270 Terminal1 has two event-to-script connections, one for *readMessage* and one for *readName*. Figure 43 on page 41 shows the connections for both 3270 terminals.

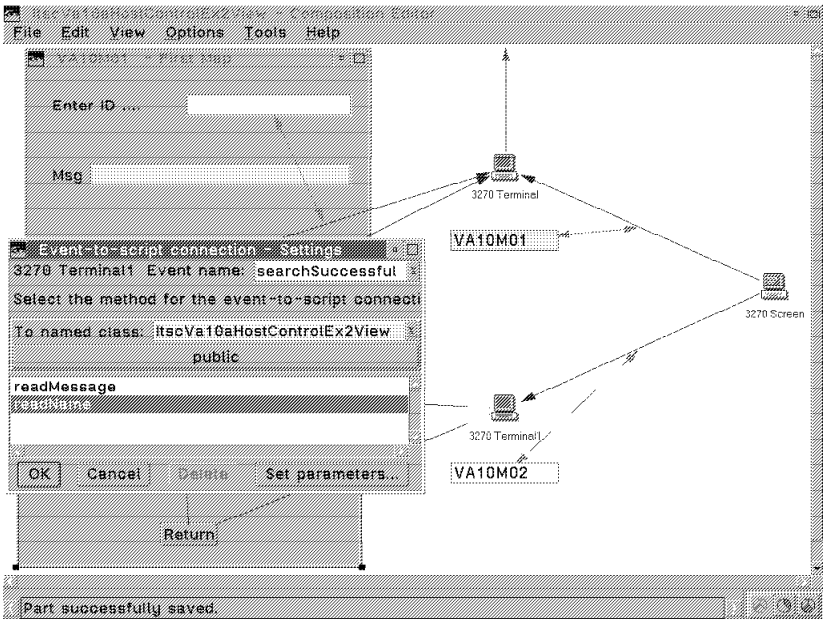

Figure 43. Event-to-Script Connections for Both Terminals

5. Now, you must move the data retrieved by the methods to the GUI windows. To connect the result of the event-to-script connections to the GUI window fields, click mouse button 2 on the Event-to-script connection and connect the *result* to the Msg field (see Figure 44). The connection is from the *result* of the method execution to the *string* of the Msg field.

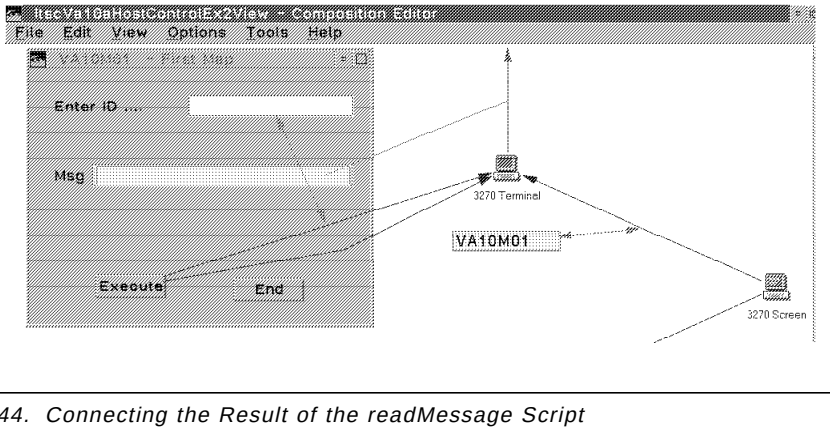

Figure 44. Connecting the Result of the readMessage Script

6. Repeat step 5 to connect:
   - The result of *readName* to the name field on the second GUI window
   - The result of *readMessage* to the msg field on the second GUI window (see Figure 45 on page 42).

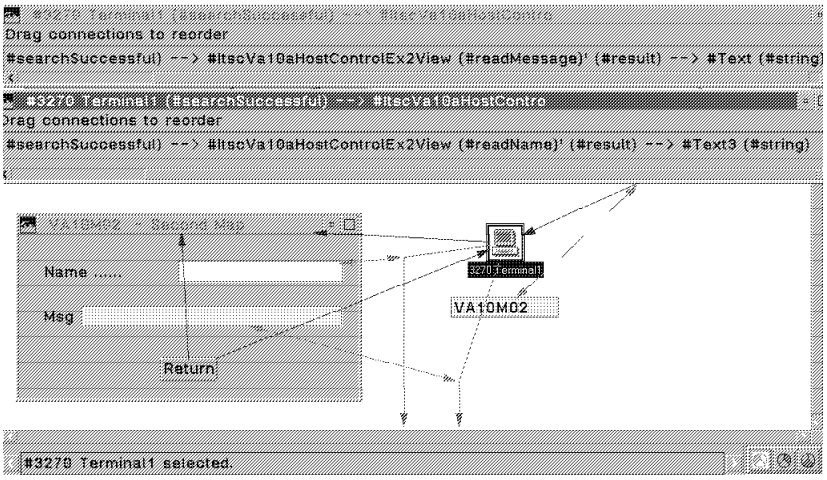

Figure 45. Connecting the Result of the readName and readMessage Methods

7. Before testing the application you need to connect the *shortSessionId* of the 3270 Screen to the *shortSessionId* of the 3270 Terminal.

   You will receive a message like that shown in Figure 46. You can ignore the message.

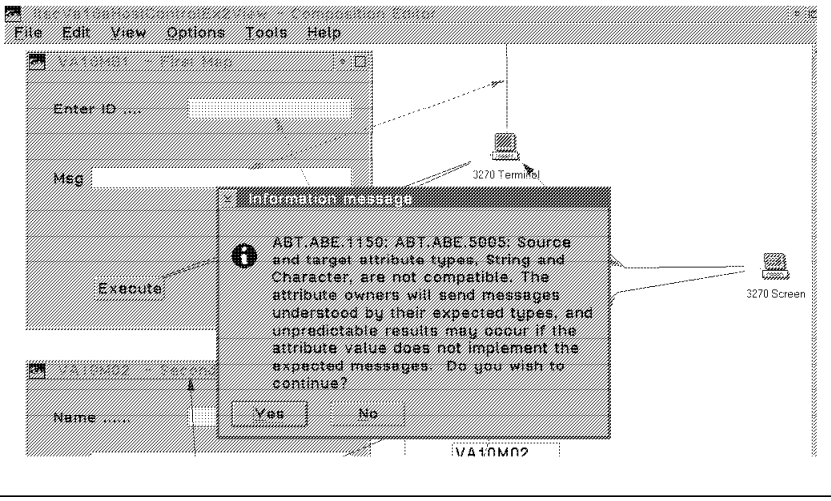

Figure 46. Message When Connecting shortSessionIds

The direction of this connection is important. The connection must be from 3270 Screen to 3270 Terminal or you will receive an error when testing the application.

Figure 47 on page 43 shows the connection for the first 3270 Terminal.

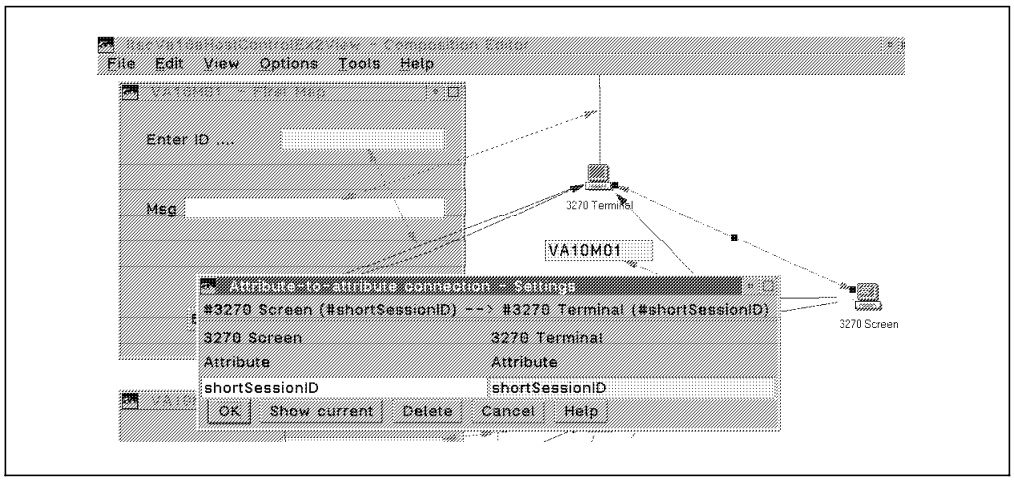

Figure 47. Connecting shortSessionId of 3270 Screen to 3270 Terminal

8. Repeat step 7 for 3270 Terminal1. Connect the *shortSessionId* of the 3270 Screen Part to the *shortSessionId* of the 3270 Terminal1.
9. Finally:
   - Connect the *End* push button (#clicked) to the 3270 Screen (#pressPF:).
   - Double-click on the dashed (---->) line, select Set Parameter, and enter 3. You must press PF3 to end the application.
   - Connect the *End* push button (#clicked) to the first window (#closeWidget).

   Figure 48 shows all connections and the application ready to be executed.

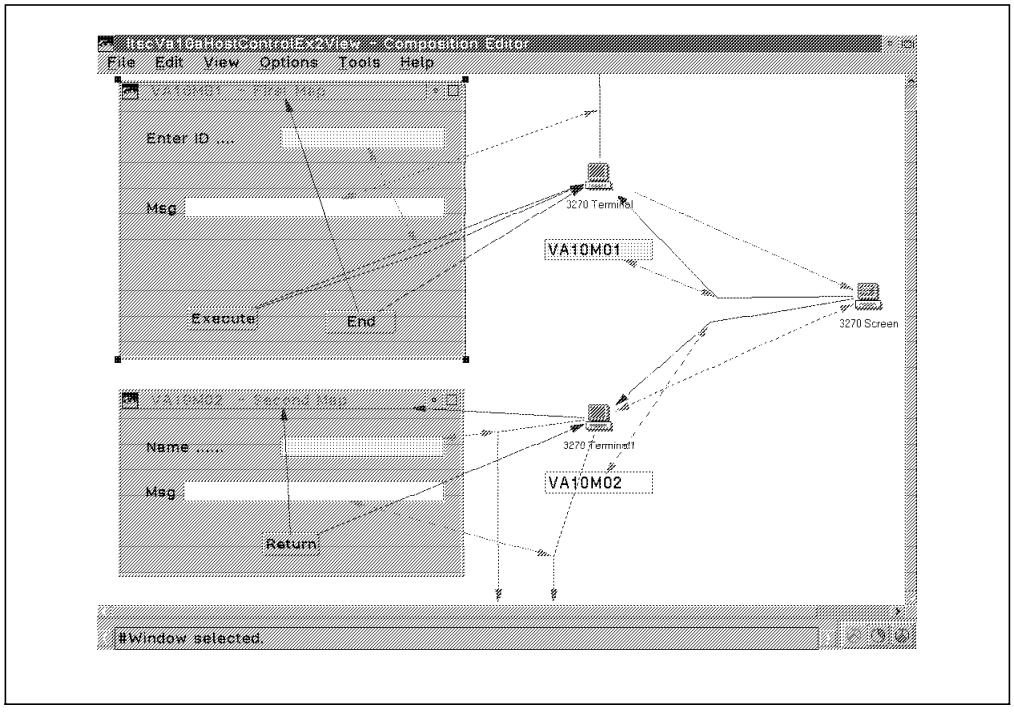

Figure 48. All Connections: Abt3270 Screen and Abt3270 Terminal Parts

3.3.2.4 Advantages and Disadvantages of This Approach

Advantages

- Good response time. We could specify a *Screen Settle Time* of 0 seconds and execute the *findString* action each time a *screenChanged* event was raised. This approach can create additional processing overhead, however (see Disadvantages).
- Easy identification for input and output fields on the host screens. We created Smalltalk methods that use the physical field location on the host screen (row and column) rather than field names.

  This advantage disappears when using the tool we created during our residency project. See Appendix A, “Screen Field Monitor Tool” on page 213 for details of this tool.

Disadvantages

- The host session must be active at development and run time when using the Abt3270Screen part. For example, if a developer specifies session G in the settings for the Abt3270Screen part, the G session always must be started when the Composition Editor for the part containing the Abt3270Screen part is started and when the packaged application is executing.
- Poor execution performance (test and run time) compared to other implementations because the *screenChanged* event is raised multiple times, and each time it is raised, the *findString* action is executed.

3.4 Implementing a GUI with PWS Control

Implementing a GUI with PWS control can be achieved in several ways. We investigated and implemented the following two approaches for our simple application:

- Use the Abt3270Terminal part and no VisualAge scripts.
- Use the Abt3270Terminal part and VisualAge scripts.

In the sections that follow we explain how we implemented the GUI front end for our simple application having control at the PWS. To simplify the examples we assume that the host 3270 screen already shows Map1.

3.4.1 Entering Host Commands with the Abt3270Terminal Part

You have a number of methods for entering a host command with the Abt3270Terminal part. We investigated the available methods and found some interesting situations that you may encounter.

The Abt3270Terminal part provides the following methods for entering host commands:

- enterCommand:
- enterCommandLine:
- enter: andWaitForCursorPositionToChangeFrom:
- enter: andWaitForCursorPositionToChangeTo:.

Please note that the last three methods in the list are not part of the public interface of the Abt3270Terminal part. We do not discuss the last method because we did not need to use it.

3.4.1.1 enterCommand:

First we tried to use the *enterCommand:* method to enter host commands through our simple application, but this method **did not work for the first host screen (Map1)**.

The *enterCommand:* method sends the input data from the GUI window to the host and waits for a change in the 3270 host screen. Figure 49 shows the code for the method.

```
enterCommand: aString
 "Type the specified string, press Home, then press Enter and wait for
 the cursor to be moved from the Home position."

  | cursorPos |
  self
    type: aString;
    keyHome.
  cursorPos := self cursorPosition.
  ^ self
      keyEnter;
      waitFor: [ self cursorPosition ~= cursorPos ].
```

Figure 49. Method: enterCommand:

Because our host Map1 had only one unprotected field, the cursor was always positioned at the unprotected field (home position) and never moved away from it; therefore, our application waited indefinitely, and, when we tested our application, we received the error message shown in Figure 50.

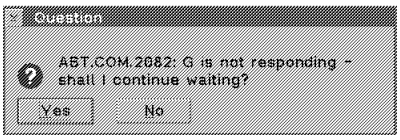

Figure 50. VisualAge Error Message

We easily could have “fixed” that problem by eliminating the *keyHome* command from the *enterCommand:* method, but we could not determine all of the implications that such a change would have in the other methods and therefore decided not to use this method.

Conclusion:

The *enterCommand:* method can be used only for host maps that contain more than one unprotected field.

3.4.1.2 enterCommandLine:

The *enterCommandLine:* method **did not work for the second host screen (Map2)**. Please note that this method is not part of the Abt3270Terminal part's public interface, but it easily can be added.

The *enterCommandLine:* method sends the input data from the GUI window to the host and waits for a change in the 3270 host screen. Figure 51 shows the code for the method.

```
enterCommandLine: aString
 "Type the specified string, then press Enter and wait for
 the cursor to be moved from the upper left-hand corner."

  ^ self enter: aString andWaitForCursorPositionToChangeFrom: (1 @ 1).
```

Figure 51. Method: enterCommandLine:

When we tested our application we received the data from host Map1 in the first GUI window, but not in the second GUI window.

Because our host Map2 had no unprotected fields, the cursor was always positioned at the upper left-hand corner (position 1 @ 1) and never moved away from that position; that is, our application waited forever, and we received the error message shown in Figure 50 on page 45.

Conclusion:

The *enterCommandLine:* method can be used only for host maps that have at least one unprotected field.

3.4.1.3 enter: andWaitForCursorPositionToChangeFrom:

The *enter: aStringandWaitForCursorPositionToChangeFrom: aPoint* method proved to be the best choice for us and we used it to implement a GUI with PWS control and for the sample application we implemented later. Please note that this method is not part of the Abt3270Terminal part's public interface, but it easily can be added. Figure 52 shows the code for the method.

```
enter: aString andWaitForCursorPositionToChangeFrom: aPoint
 "Type the specified string,  position the cursor at aPoint,
 then press Enter and wait for the cursor to be repositioned."
  ^ self
      type: aString;
      cursorPosition: aPoint;
      keyEnter;
      waitFor: [ self cursorPosition ~= aPoint ].
```

Figure 52. Method: enter: andWaitForCursorPositionToChangeFrom:

The only problem with this method was that we needed to pass the cursor position (*aPoint*) as a parameter and this added more parts to the free form surface for situations where we did not want to use VisualAge scripts.

This method worked in all situations we encountered during our project because we could move the cursor to column 80 row 24, and we did not encounter an

application where the cursor was still at that position after the application activated the Enter key.

3.4.2 Abt3270Terminal Parts Only and No Scripts

In this section we describe how we used Abt3270Terminal parts only and no VisualAge scripts to implement PWS control for a VisualAge EHLLAPI application. Figure 53 shows the model we implemented. Note that we used two Abt3270Terminal parts.

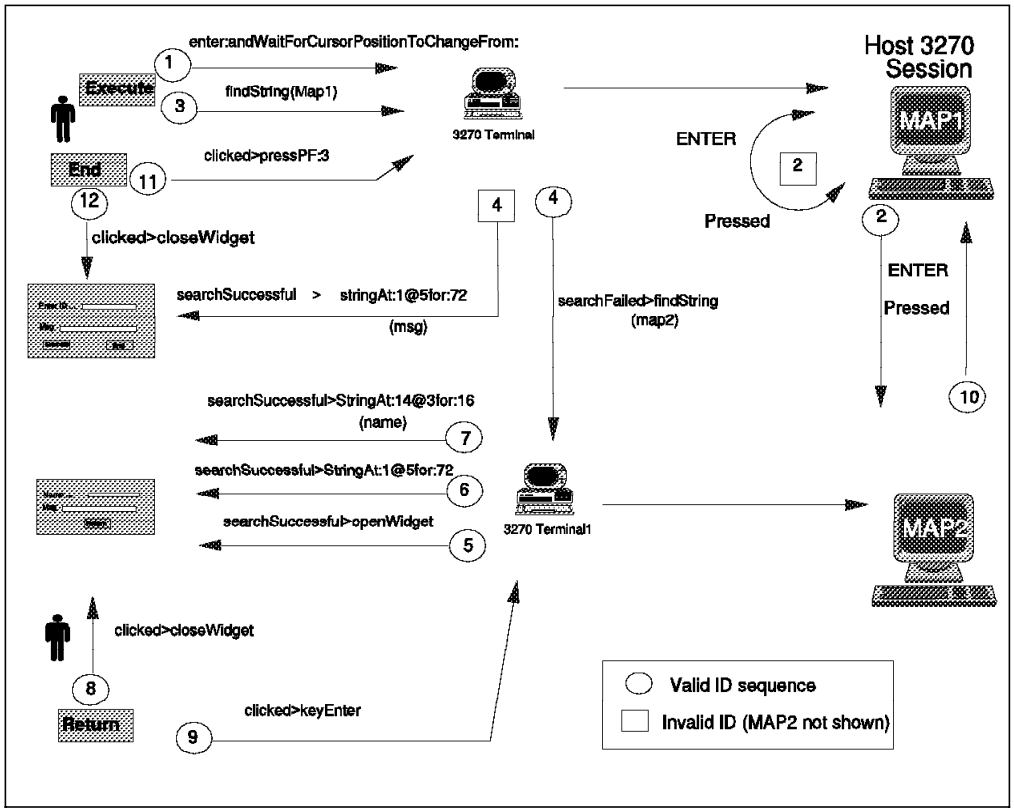

Figure 53. Sequence of Events and Actions with PWS Control

The sequence of events and actions is as follows:

1. The user clicks on the *Execute* push button to have the *enter:andWaitForCursorPositionToChangeFrom:* method send the input data to the host, move the cursor to position (80,24), and wait for the cursor to move from that position.
2. The host application executes the Enter key, and either MAP1 or MAP2 is shown:
   - If the user entered an invalid ID, a message is shown in MAP1.
   - If the user entered a valid ID, MAP2 is shown at the host.
3. The *findString* action is executed after the host screen is refreshed (cursor moved from position 80 x 24 to home) and looks for Map1. As a result of this action either the *searchSuccessful* or *searchFailed* event is raised.
4. This step depends on which host screen is shown:
   - If the first host screen (Map1) is shown, the *searchSuccessful* event triggers the *stringAt:1@5for:72* method, which moves the message data to the first GUI window.

- If the second host screen (Map2) is shown, the *searchFailed* event triggers the *findString* action that looks for Map2.

5. The *searchSuccessful* event for Map2 triggers the *openWidget* action to open the second GUI window.
6. The *searchSuccessful* event for Map2 also triggers the *stringAt:1@5for:72* action to move the host message data to the second GUI window.
7. The *searchSuccessful* event for Map2 also triggers the *stringAt:14@3for:16* action to move the contents of the host name field to the second GUI window.
8. The user clicks on the *Return* push button, which executes the *closeWidget* action, and the second GUI window is closed.
9. Clicking on the *Return* push button also triggers the *pressEnter* action for MAP2.
10. The host application executes the Enter key.
11. The user clicks on the *End* push button, and the *pressPF:3* action is sent to the host.
12. The *End* push button triggers the *closeWidget* action to close the first GUI window.

3.4.2.1 Coding the Application

As already mentioned, the *enter:andWaitForCursorPositionToChangeFrom:* method is not in the public interface of the Abt3270Terminal part. We suggest that this method be available for all developers through the public interface if you plan to use the VisualAge EHLLAPI support.

Perform the following steps to add a method to the public interface of a part:

1. Go to the Composition Editor, add an Abt3270Terminal part to the free form surface, select it, and click mouse button 2. In Open Settings select *Edit Part* as shown in Figure 54 on page 49.

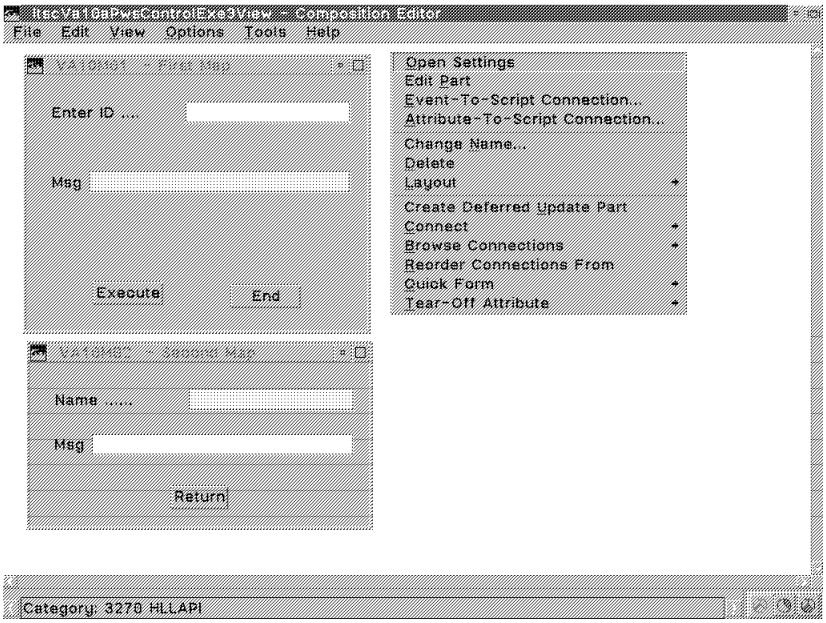

Figure 54. Editing the Abt3270Terminal Part

2. The Script Editor for the Abt3270Terminal part is shown (see Figure 55). Select the icon to open the public interface editor.

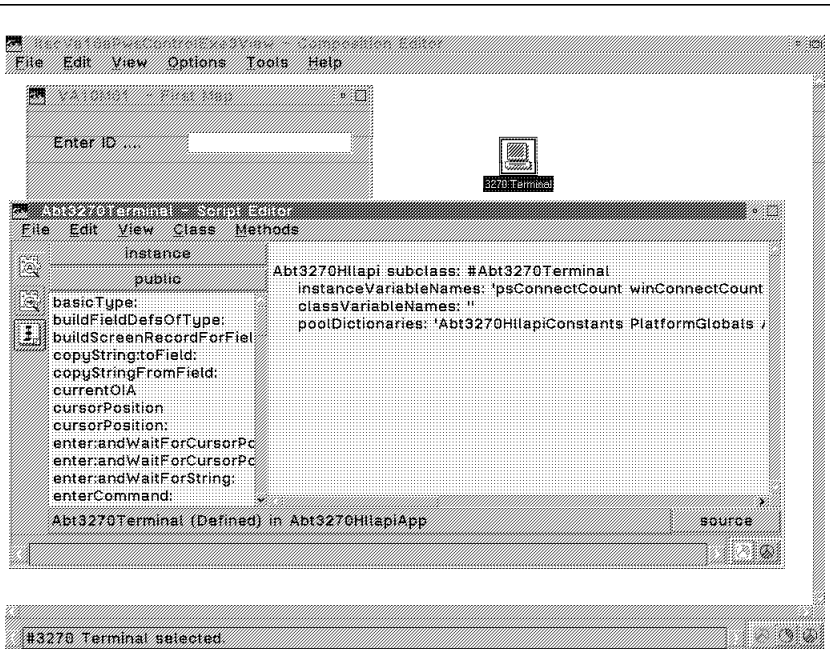

Figure 55. Abt3270Terminal Part Script Editor

3. To add an action to the public interface follow these steps:
   a. Select the *Action* tab of the public interface editor notebook.

b. Select the method you want to add to the public interface, in this case, *enter:andWaitForCursorPositionToChangeFrom:*, from the *Action selector* drop-down list.

c. Type the action name in the *Action name* field. We suggest you use the method name. Use the *Ctrl+Insert* keys to copy the method name to the OS/2 clipboard and the *Shift+Insert* keys to paste it to the *Action name* field.

d. Click on the *Add with defaults* button (see Figure 56).

After saving the part, the newly added action is available for visual connections.

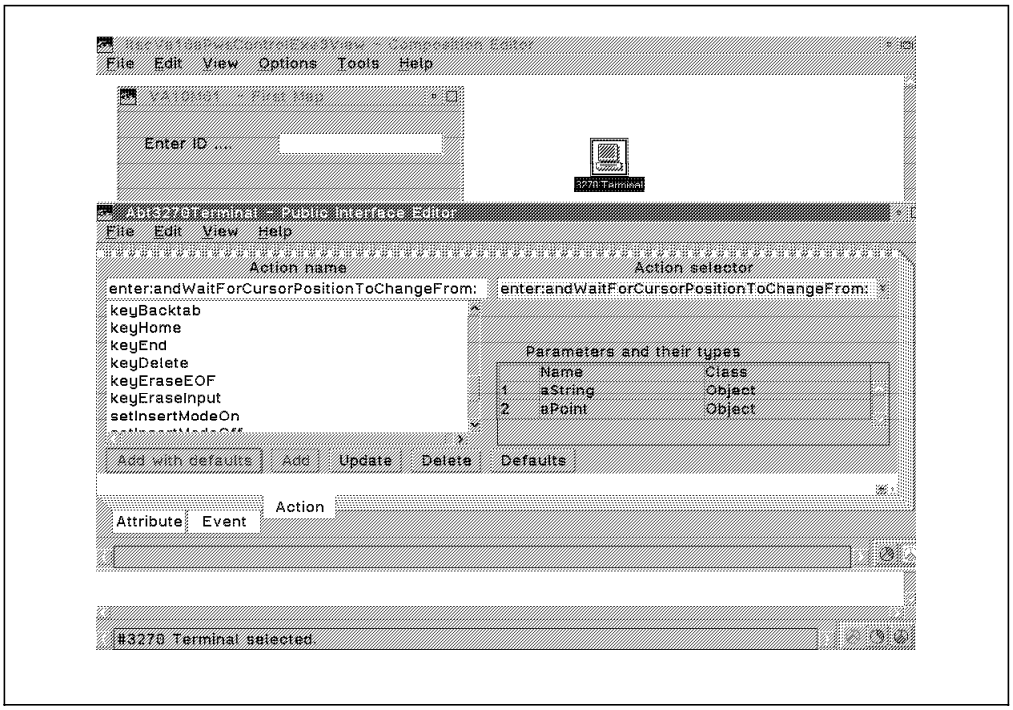

Figure 56. Adding an Action to the Public Interface

4. Add another Abt3270Terminal part to the free form surface as this example requires two Abt3270Terminal parts. You can use the Copy/Paste function of the Composition Editor.

Because we wanted to implement this example without Smalltalk scripts, we had to add parts (objects) to the free form surface that we could use as parameters for the actions we used.

The *enter:andWaitForCursorPositionToChangeFrom:* method required two parameters, one *string* and one *point* object. Therefore, we had to add a *Point* part to the free form surface by selecting Options from the action bar and then Add Part and entering *Point* for the *Class name*. We could then provide values for the x and y coordinates of the *Point* object.

When we used methods that required a *Point* object as a parameter, we connected the self attribute of the *Point* object to the incomplete action connection.

We had to create the *Point* on the free form surface as a part and then connect the x and y values. We could not create the *Point* as a variable, because the

Point class is complex (it has x and y as instance variables) and must be initialized (*new*) before it can be used. This initialization is done automatically for added parts only. If we added the *Point* as a variable, it would not be initialized (*new*), and we would not be able to connect the values (80,24) to x and y. Figure 57 shows how to add a *Point* part.

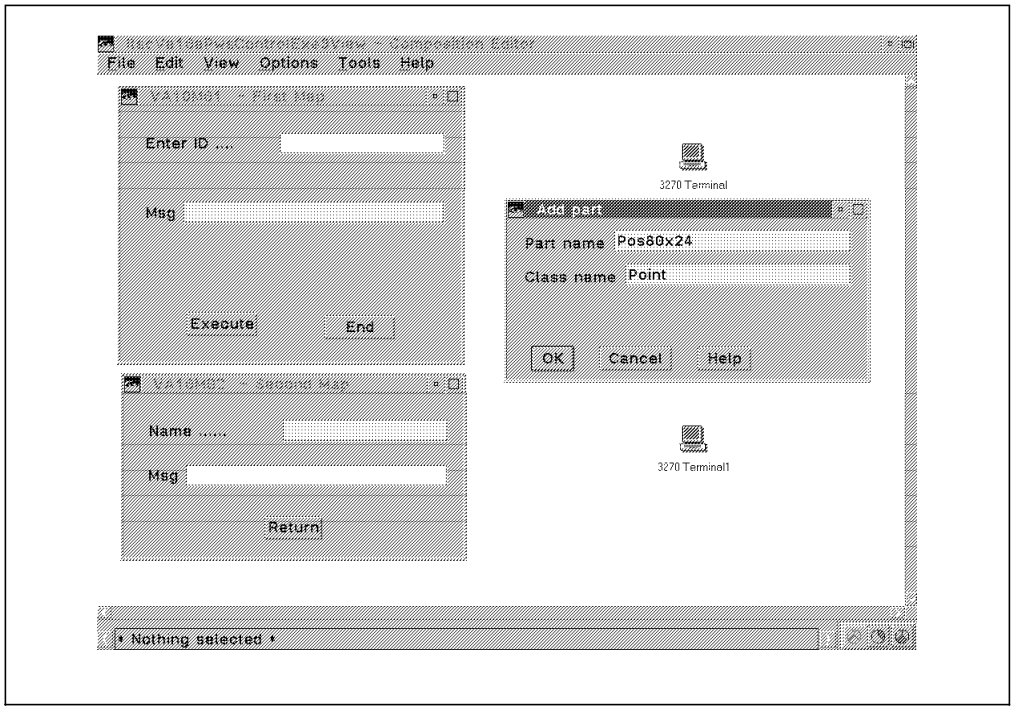

Figure 57. Adding a Point Part

Figure 58 on page 52 shows the *Point* part on the free form surface with the connections to the x and y values. Note that the values must be *integer* objects, and the connection must connect the *object* to *x* or *y*.

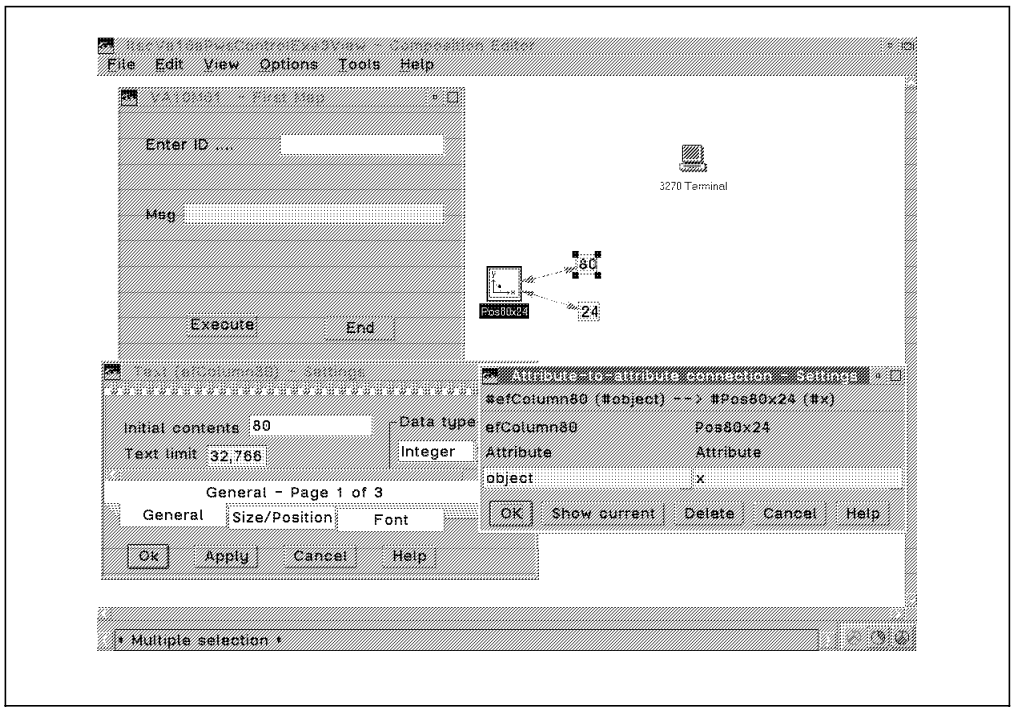

Figure 58. Initializing a Point with Values

3.4.2.2 Connections for the First Abt3270Terminal Part

The sequence in which you define the connections corresponds to the execution sequence of the application. Be sure to make the connections in the correct sequence. Make the following visual connections:

1. Connect the *Execute* push button (#clicked) to the 3270 Terminal (#enter: aString andWaitForCursorPositionToChangeFrom:).
   This connection is incomplete, and you need to provide the ID to be sent to the host (aString) and the cursor position (aPoint) as parameters.
2. Connect Text (#string) to the incomplete connection (#aString).
3. Connect *Pos24x80* (#self) to the incomplete connection (#aPoint).
   The connection is now complete and changes from a dashed line to a solid line. Figure 59 shows the connections.

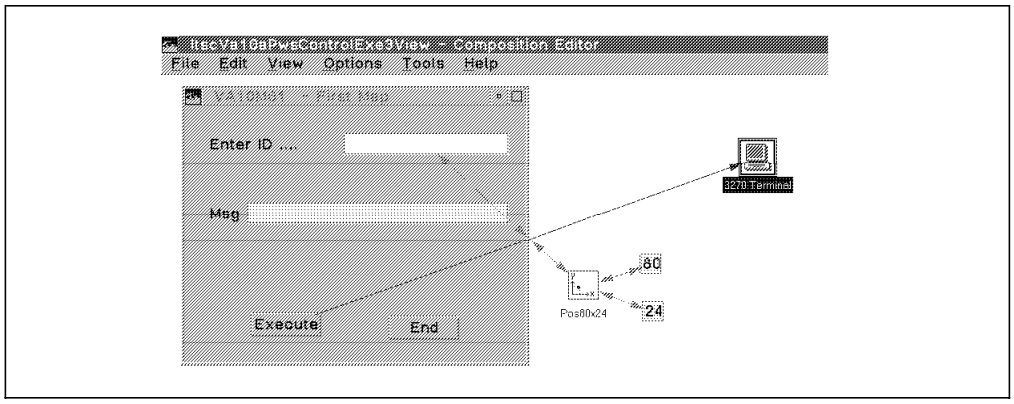

Figure 59. Connecting aString and aPoint

4. When you click on the *Execute* push button, the host shows either Map1 or Map2, depending on the input you provided. To find out which host screen is shown, connect the push button to the 3270 Terminal.

   Connect the *Execute* push button (#clicked) to the 3270 Terminal (#findString:).

   Pass as a parameter the string to be found (VA10M01) by connecting the *labelString* to the connection (#aString) (see Figure 60).

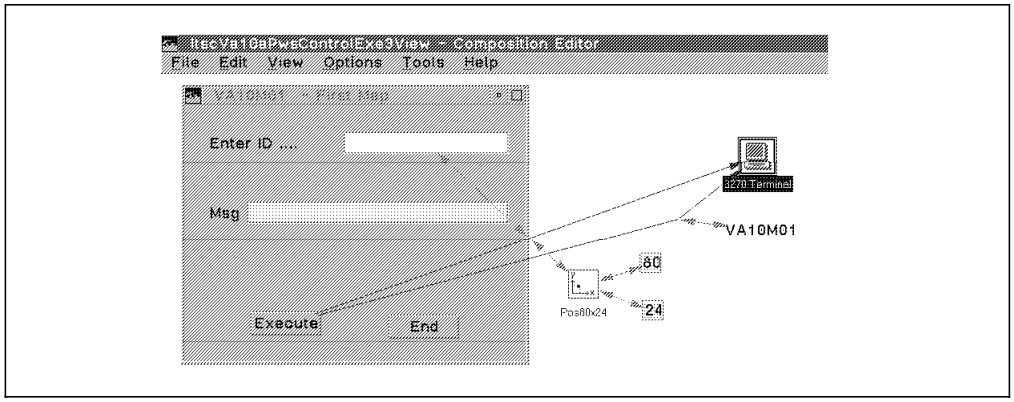

Figure 60. Connecting findString

5. If Map1 is found, you need to get the contents of the message field on the host screen.

   Connect the 3270 Terminal (#searchSuccessful) to the 3270 Terminal (#stringAt:for:).

   Pass as parameters the position of the host screen message field (1 @ 5) and its length (72).

   You need to add a *Point* part. You can use the Copy/Paste function of the Composition Editor for that purpose. Figure 61 shows the connection.

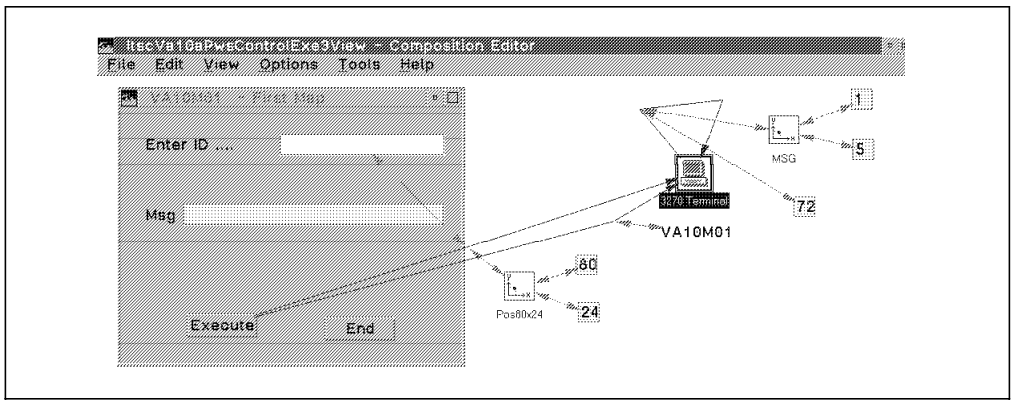

Figure 61. Connecting the 3270 Terminal to Itself

6. You need to connect the data read from the host screen to the message field of the first GUI window.

   Connect the *result* of the previously added connection (3270 Terminal to 3270 Terminal) to the Msg field (#string).

7. Now add a label with the session ID (B) and connect it (*#labelString*) to the *shortSessionID* of the 3270 Terminal. This connection results in a warning message, which you can ignore.

Figure 62 shows all connections for the first Abt3270Terminal part. You are now ready to test the application with an invalid ID as input.

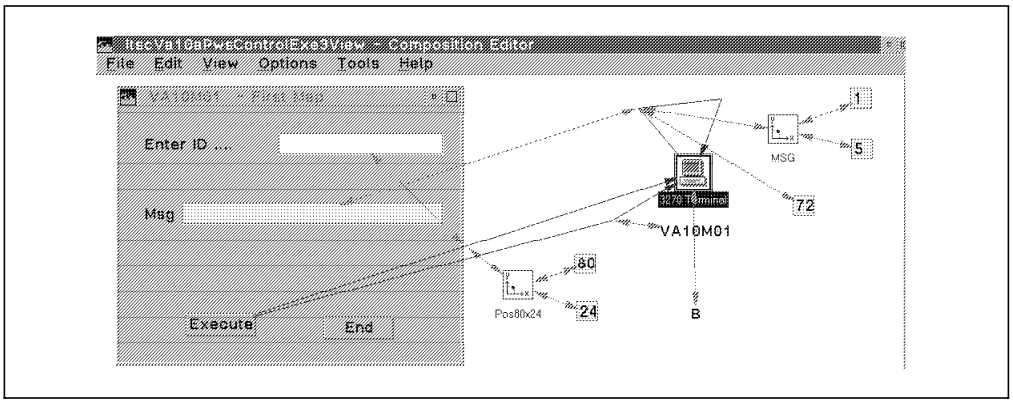

Figure 62. All Connections for the First Abt3270Terminal Part

3.4.2.3 Connections for the Second Abt3270Terminal Part

To connect the second Abt3270Terminal part, make the following connections:

1. Connect 3270 Terminal (#searchFailed) to 3270 Terminal1 (#findString).
2. Connect 3270 Terminal1 (#searchSuccessful) to Window1 (#openWidget).
3. Connect 3270 Terminal1 (#searchSuccessful) to 3270 Terminal1 (#stringAt: for:).

   This connection will retrieve the name field from Map2. Pass as parameters the position of the host screen name field (14 @ 3) and its length (16).
4. Connect the result of the previous connection to the *Name* field in the second GUI window.
5. Connect 3270 Terminal1 (#searchSuccessful) to 3270 Terminal1 (#stringAt: for:).

   This connection will retrieve the message field from Map2. Pass as parameters the position of the host screen message field (1 @ 5) and its length (72).

   Note: You could use the same *Point* object that you used for the first 3270 Terminal, but we suggest you create a separate *Point* object to make the visual coding easier to understand.
6. Connect the result of the previous connection to the *Msg* field in the second GUI window.
7. Connect the *Return* push button (#clicked) to 3270 Terminal1 (#keyEnter).
8. Connect the *Return* push button (#clicked) to Window1 (#closeWidget).
9. Connect the *End* push button (#clicked) to 3270 Terminal (#keyPF:3).
10. Connect the *End* push button (#clicked) to Window (#closeWidget).

Figure 63 on page 55 shows all connections.

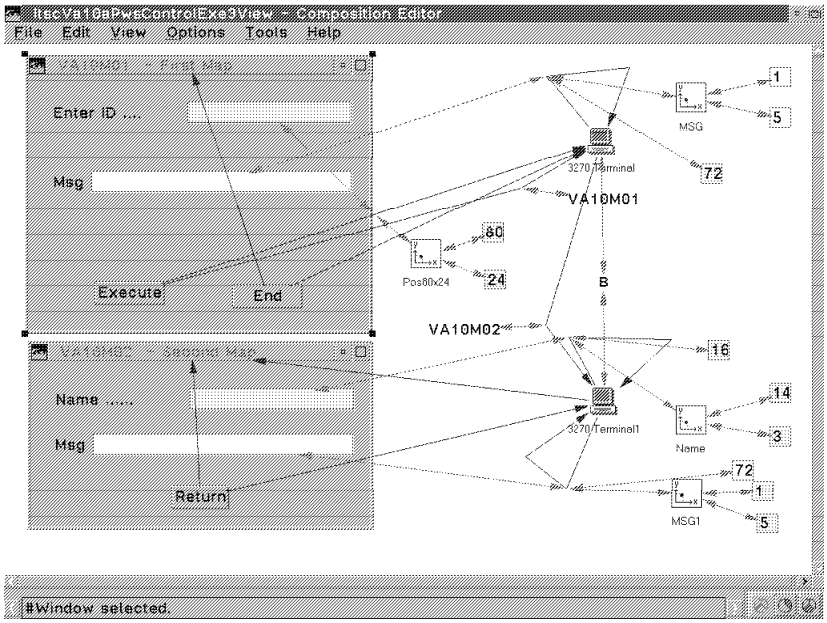

Figure 63. All Connections for Both Abt3270 Terminal Parts

3.4.2.4 Advantages and Disadvantages of This Approach

Advantages

- Does not require Smalltalk skills; all coding is visual.
- Good response time. Because this approach does not use the Abt3270Screen part, we avoided the *Screen Settle Time* delay.
- Easy identification of input and output fields on the host screens. We created Smalltalk methods that use the physical field location on the host screen (row and column) rather than field names.

  This advantage disappears when using the tool we created during our residency project. See Appendix A, “Screen Field Monitor Tool” on page 213 for details of this tool.
- Automatic timeout warning when the host session does not respond.
- No need to have the host session active during development.

Disadvantages

- Requires checking to find out which host screen is displayed.

  When using the Abt3270Screen part, the key string defined for the Abt3270Screen part is used to validate the active host screen and prevents the sending or receiving of data from an incorrect host screen.
- All coding is visual, which usually means that there are many connections, and understanding the code can become difficult.

3.4.3 Abt3270Terminal Part and Scripts

In this section we describe how we used an Abt3270Terminal part and VisualAge scripts to implement PWS control for the GUI application. We used VisualAge facilities to generate the scripts whenever possible.

Note that we used one Abt3270Terminal part instead of two because we used VisualAge scripts instead of the *searchSuccessful* event. Figure 64 shows the model we implemented.

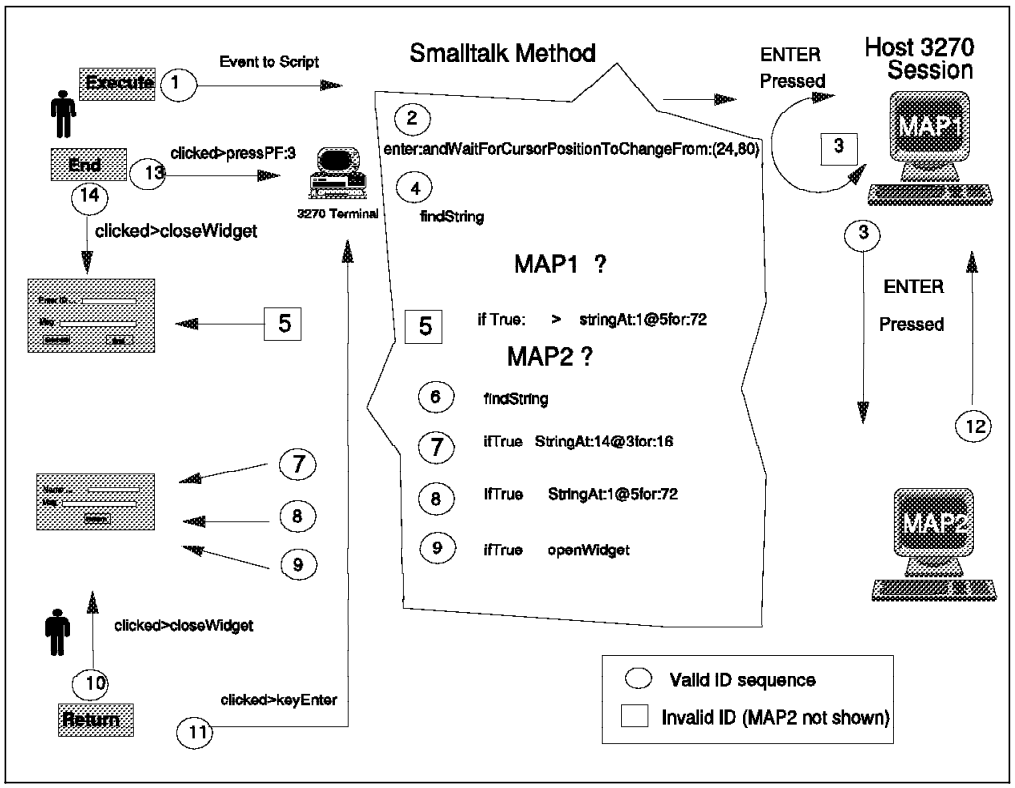

Figure 64. Sequence of Events and Actions with PWS Control and Scripts

The sequence of events and actions is as follows:

1. The user clicks on the *Execute* button to execute a VisualAge script.
2. The script executes the *enter:andWaitForCursorPositionToChangeFrom:* method to send the input data to the host, move the cursor to position (80,24), and wait for the cursor to move from that position.
3. The host application executes the Enter key, and either MAP1 or MAP2 is shown.
   - If the user entered an invalid ID, a message is shown in MAP1.
   - If the user entered a valid ID, MAP2 is shown at the host.
4. The *findString* action is executed and searches for the string MAP1.
5. If the string is found (*true* result), the *stringAt:1@5for:72* method moves the message data to the first GUI window.
6. If the string is not found (*false* result), the *findString* method searches tor the string MAP2.
7. If the string is found (*true* result), the *stringAt:14@3for:16* method moves the name data to the second GUI window.

8. The *stringAt:1@5for:72* method moves the message data to the second GUI window.
9. The *true* result also triggers the *openWidget* action, which opens the second GUI window.
10. The user clicks on the *Return* button, which executes the *closeWidget* action, and the second GUI window is closed.
11. Clicking on the *Return* push button also triggers the *pressEnter* action for MAP2.
12. The host application executes the enter key.
13. The user clicks on the *End* push button, and the *pressPF:3* action is sent to the host.
14. The *End* push button triggers the *closeWidget* action to close the first GUI window.

3.4.3.1 Coding the Application

Using the Composition Editor, drag an Abt3270Terminal part and two variable parts from the parts palette and drop them on the free form surface as shown in Figure 65.

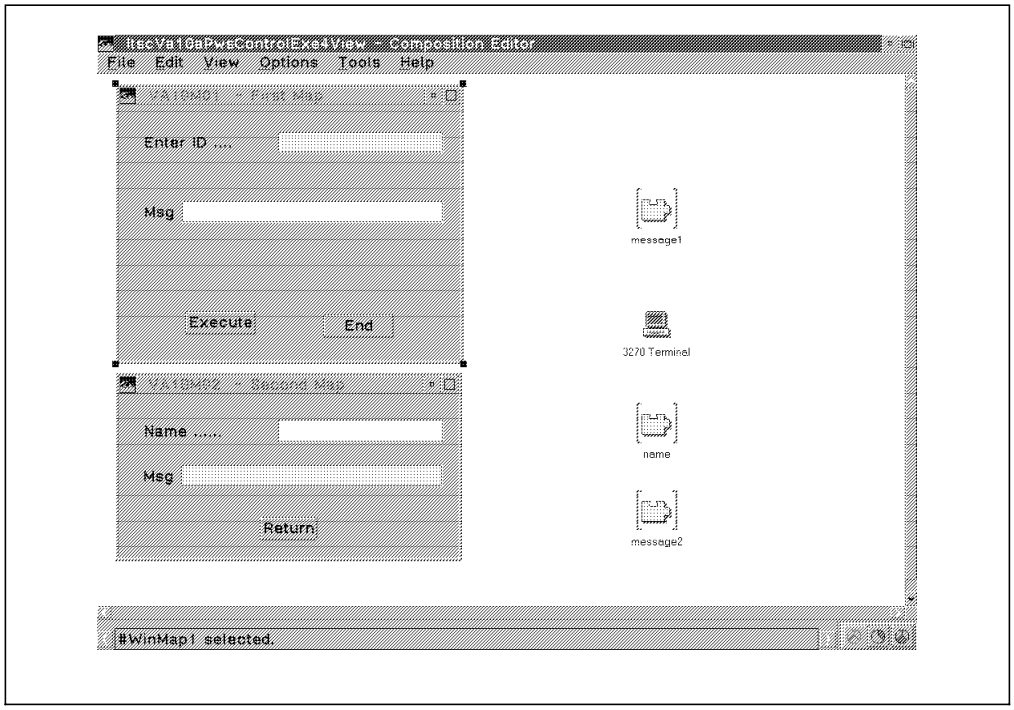

Figure 65. Abt3270Terminal Part and Variable Parts

When you access VisualAge parts from Smalltalk code or use the VisualAge Script Editor to generate Smalltalk code, you could find it difficult to distinguish different parts in the same category. The part names VisualAge generates differ only in sequence number, and you should provide your own part names for easy identification. We changed the part names as suggested in 4.1.2, “Naming Convention for Parts” on page 81.

To change the names, select the part you want to rename and click on it with mouse button 2. Select *Change Name...* and type the desired name in the dialog box that shows the default name.

Table 4 shows the naming convention we used.

Table 4. Naming Convention Used for VisualAge Parts

Default Name	Changed Name
Canvas Category	
Window	winMap1
Window1	winMap2
Buttons Category	
Push Button (Window)	pbExecute
Push Button1 (Window)	pbEnd
Push Button2 (Window1)	pbReturn
Data Entry Category	
Text (ID - Window)	efID
Text1 (Msg - Window)	efMsg1
Text2 (Name - Window1)	efName
Text2 (Msg - Window1)	efMsg2
Models Variable Category	
Object	message1
Object1	name
Object2	message2

3.4.3.2 Writing the Script

The Smalltalk method (script) that provides the functions behind the *Execute* push button should do the following:

- Assign the host session ID to the Abt3270Terminal part.
- Send the input data to the host and wait for the host screen to change.
- Find out which host screen is displayed.
- Move the host data to the appropriate GUI window fields.

We used the VisualAge Script Editor to write the method and named it *enterID:*. The *enterID:* method expects the input data for the host as a parameter.

Using the Script Editor, create a new method by selecting Methods and New Method Templates from the action bar. Define two temporary variables where data sent to and received from the host will be stored (*aMessageString* and *aNameString*). Figure 66 shows the Script Editor.

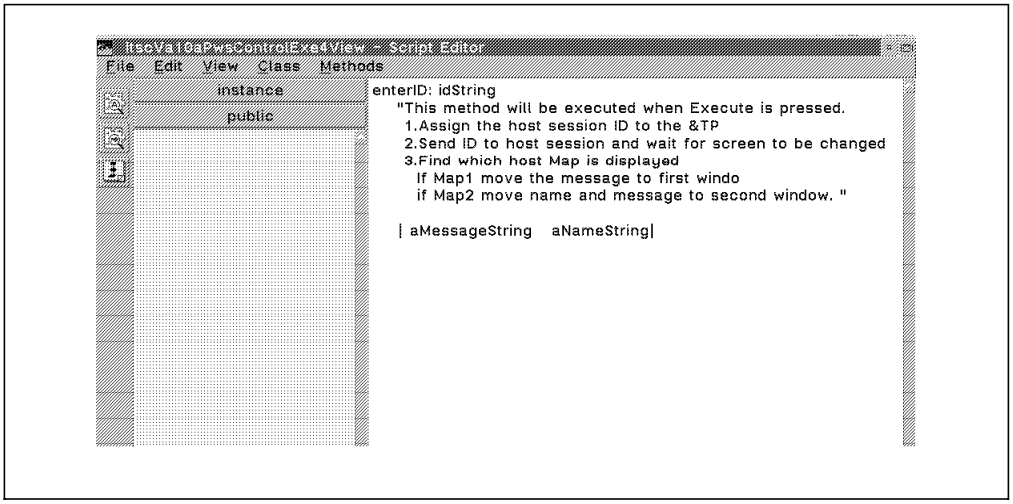

Figure 66. Creating the enterID: Method

To assign the host session ID to the Abt3270Terminal part, perform the following operations using the Script Editor:

1. Click on the Attributes icon, select *3270 Terminal* from the subparts list and *shortSessionId* from the attributes list, and click on the *Paste 'set'* push button (see Figure 67).

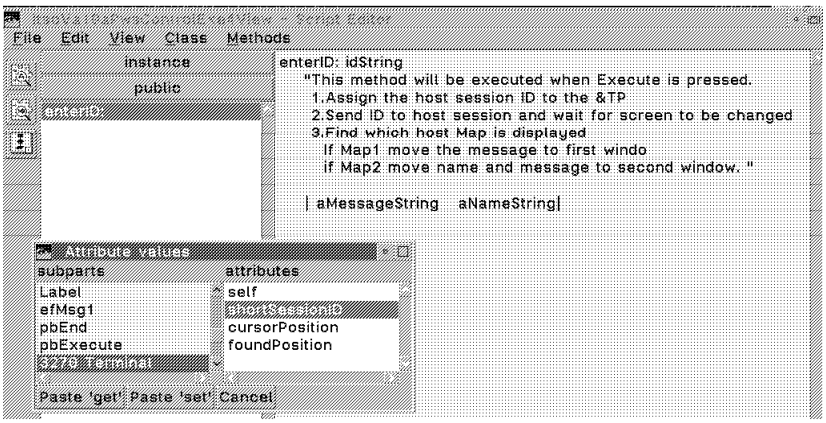

Figure 67. Using Script Editor to Generate Code

Be sure the cursor in the edit pane is positioned where you want the code to be pasted. Figure 68 shows the result of the *Paste 'set'* operation.

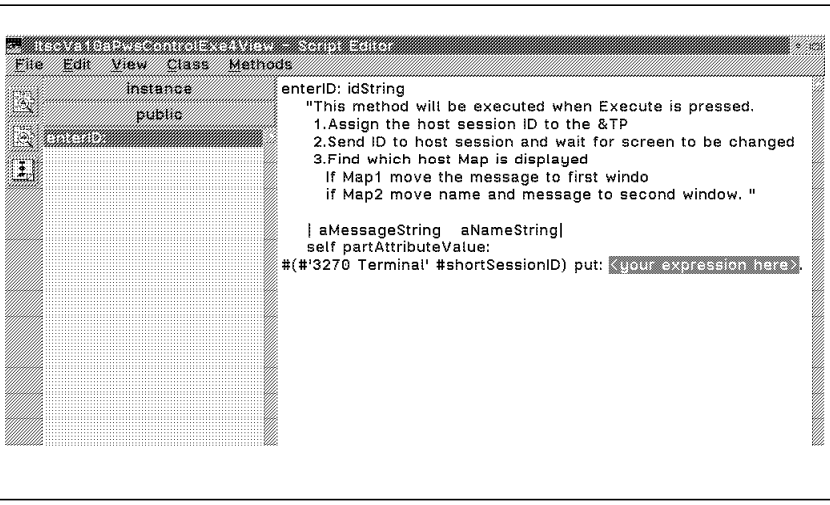

Figure 68. Code Generated by the Script Editor: Attribute

2. Replace *<your expression here>* with the host session ID. We used the G session in our example (*$G*).

We used the *enter:aString andWaitForCursorPositionToChangeFrom:* method, which we added to the &TP's public interface, to send the input data to the host.

To generate the Smalltalk code perform the following operations using the Script Editor:

1. Click on the Action icon, select *3270 Terminal* from the subparts list and *enter:aString andWaitForCursorPositionToChangeFrom:* from the actions list, and press the *Paste* push button (see Figure 69 on page 60).

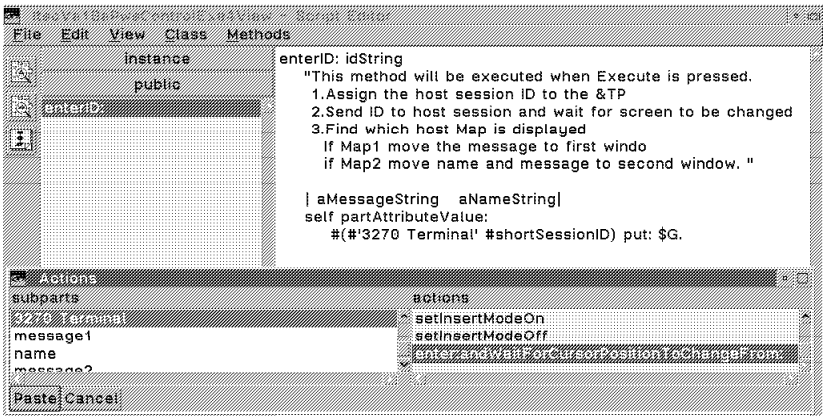

Figure 69. Using Script Editor to Generate Action Code

Be sure the cursor in the edit pane is positioned where you want the code to be pasted. Figure 70 shows the result of the *Paste* operation.

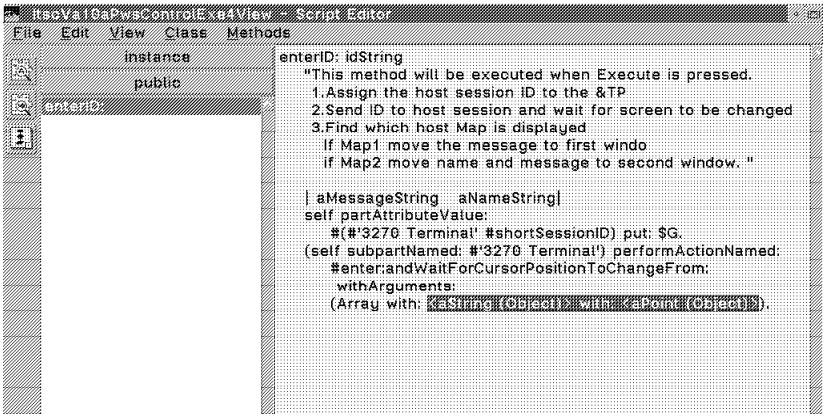

Figure 70. Code Generated by the Script Editor: Action

2. Replace *<aString(Object)>* with: *<aPoint(Object)>* with *idString with: 15@3*. This represents column 15 row 3 on the host screen, which is where the input data must be entered before the enter key is pressed.

After the input data is sent to the host you need to find out which host screen is displayed. We used the *findString:* action to identify the host screen.

Generate the Smalltalk code as described in the previous step. Figure 71 on page 61 shows the generated Smalltalk code.

Replace the *<aString(String)>* with
'VA10M01 HOST APPLICATION - First Map'
to uniquely identify the host screen.

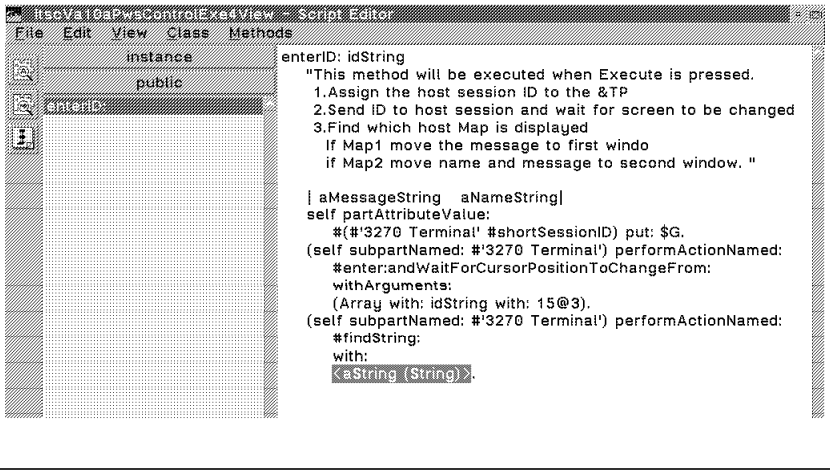

Figure 71. Code Generated by the Script Editor: Action findString:

Now you must generate the code that implements the logic. That logic depends on whether the user entered a valid or invalid ID and therefore on which host screen is displayed. If you are not familiar with Smalltalk, you can use the Script Editor language elements dialog (third icon from the top on the left).

For example, to generate the *ifTrue* statement select *Control Statements* from the Categories list and *ifTrue* from the Language elements list as shown in Figure 72.

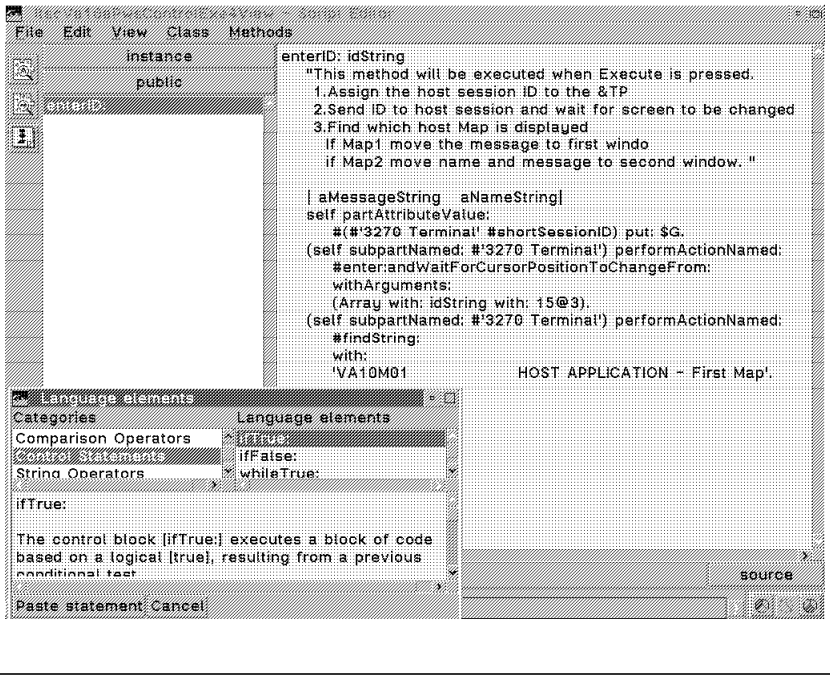

Figure 72. Generating ifTrue Statement

Use the *Paste statement* push button to paste the language element to the edit pane (see Figure 73 on page 62).

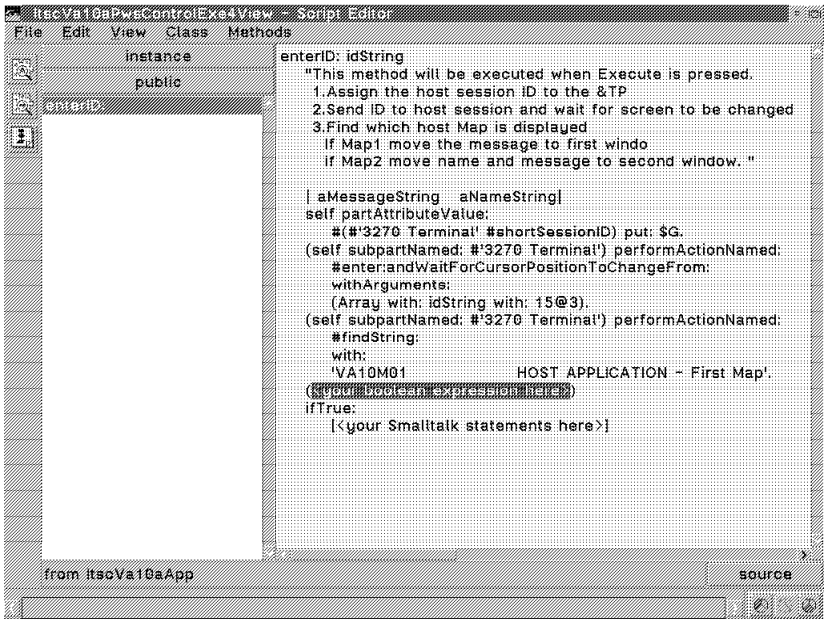

Figure 73. Generated ifTrue Statement

Replace the *<your Boolean expression here>* text with the code block that tests for the host screen. Figure 74 shows the modified code.

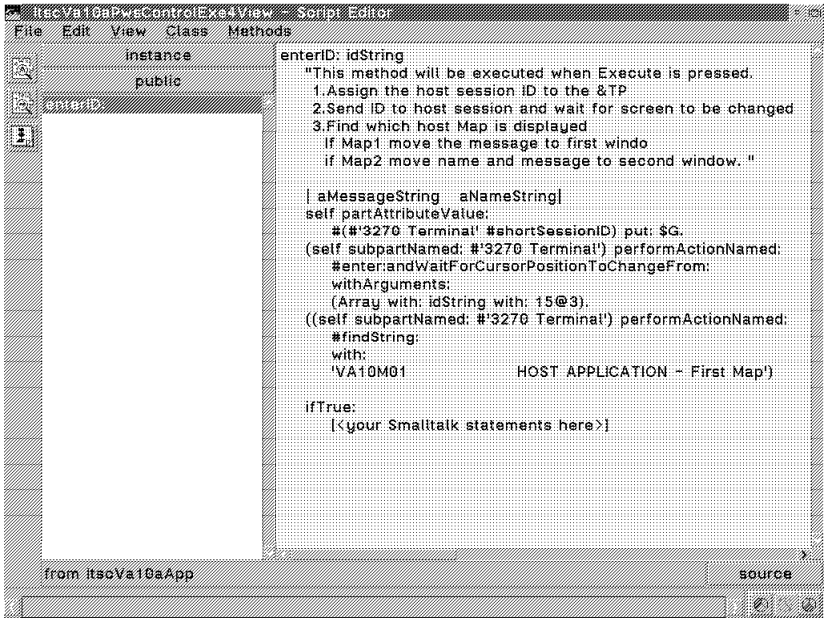

Figure 74. Modified ifTrue Statement

Now you must generate the code that moves the data from the host screen to the first GUI window. Perform the following steps to generate the Smalltalk code:

1. Use the *aMessageString* Smalltalk variable to store the message data. Type *aMessageString :=* between the brackets as shown in Figure 75.

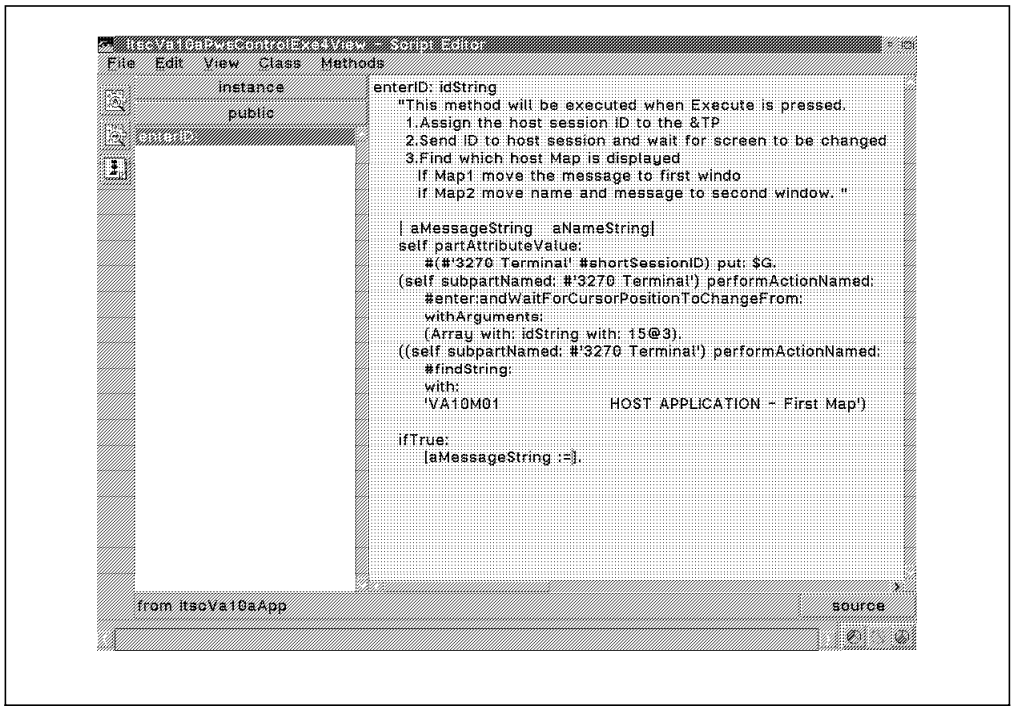

Figure 75. Using the Script Editor to Generate Code: Assigning a Value to Variable

2. Paste the *stringAt:for:* action as you did for the other actions. Figure 76 shows the result of the *Paste* operation.

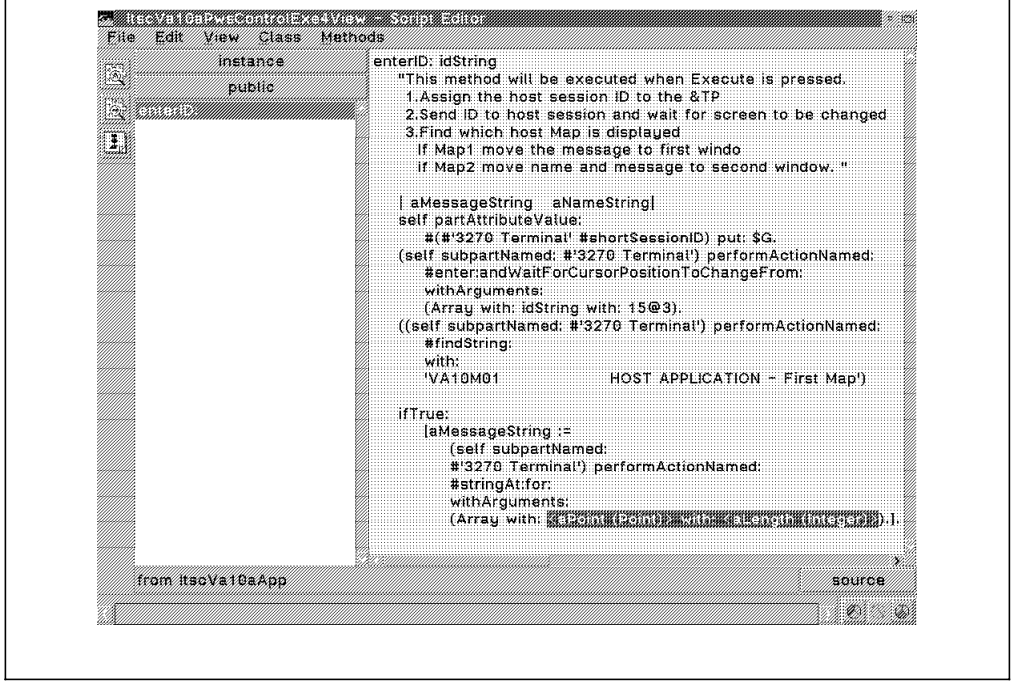

Figure 76. Using the Script Editor to Generate Code: Pasting Action

Replace *<aPoint (Point)> with: <aLength (Integer)>*

with *1@5 with: 72*, the location and size of the message field on the host screen.

3. As part of the same if block ([ ]) you should set the *message1* variable on the free form surface to the contents of the *aMessageString* Smalltalk variable.

   You can use the Script Editor to generate the required code as shown in Figure 77.

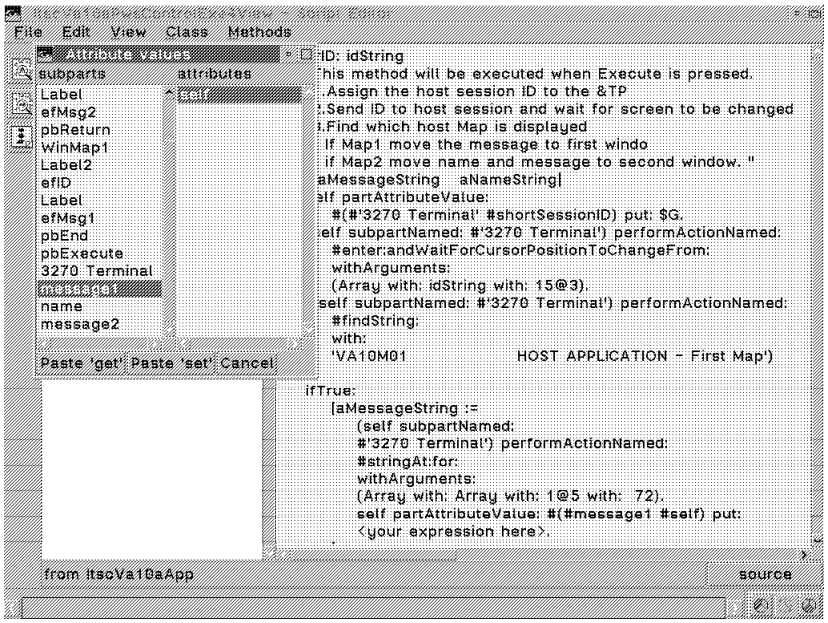

Figure 77. Using the Script Editor to Generate Code: Pasting Set message1

4. Replace the *<your expression here>* text with *aMessageString*.

   Figure 78 shows the complete ifTrue block.

```
ifTrue:
  [aMessageString :=
   (self subpartNamed:
   #'3270 Terminal') performActionNamed:
   #stringAt:for:
    withArguments:
   (Array with: 1@5 with:  72).
   self partAttributeValue: #(#message1 #self) put:
   aMessageString.
  ].
```

Figure 78. Generated ifTrue Block

5. You can generate an *ifFalse* block to handle the situation when the second host screen (Map2) is displayed. The steps to generate that code are similar to the steps to handle the first host screen (Map1) and are not repeated here. The difference is that you need to open the second GUI window when you find Map2. The required code can be generated using the Script Editor. Figure 79 on page 65 shows the complete Smalltalk method.

```
enterID: idString
 "This method will be executed when Execute is pressed.
     1.Assign the host session ID to the Abt3270Terminal part
     2.Send ID to host session and wait for screen to be changed
     3.Find which host Map is displayed
       If Map1 move the message to first window
       if Map2 move name and message to second window. "
    | aMessageString   aNameString|
 self partAttributeValue:
  #(#'3270 Terminal' #shortSessionID) put: $G.
(self subpartNamed: #'3270 Terminal') performActionNamed:
  #enter:andWaitForCursorPositionToChangeFrom:
  withArguments:
  (Array with: idString with: 15@3).
 ((self subpartNamed: #'3270 Terminal') performActionNamed:
  #findString:
   with:
  'VA1OMO1                 HOST APPLICATION - First Map')

ifTrue:
 [
  aMessageString := (self subpartNamed: #'3270 Terminal')
   performActionNamed: #stringAt:for:
        withArguments:  ( Array with: 1@5 with:  72).
  self partAttributeValue: #(#message1 #self) put: aMessageString.
 ]
 ifFalse:
 [
  ((self subpartNamed: #'3270 Terminal') performActionNamed:
   #findString:
  with: 'VA1OMO2                HOST APPLICATION - Second map')
   ifTrue:
   [
   (self subpartNamed: #WinMap2) performActionNamed: #openOwnedWidget.
   aNameString := (self subpartNamed: #'3270 Terminal')
    performActionNamed: #stringAt:for:
     withArguments: (Array with: 14@3 with: 16).
   self partAttributeValue: #(#name #self) put: aNameString.

   aMessageString := (self subpartNamed: #'3270 Terminal')
    performActionNamed: #stringAt:for:
      withArguments: (Array with: 1@5 with:  72).
   self partAttributeValue: #(#message2 #self) put:  aMessageString.
   ]
 ].
```

Figure 79. Generated Smalltalk Method

3.4.3.3 Making the Connections

You have to make the following visual connections to finish the application:

1. Use an event-to-script connection to hook the *enterID:* method to the *Execute* push button and then pass the input data from the Enter ID field to that event-to-script connection as a parameter.

2. After the *enterID:* method has executed, the data from the host screen is stored in the variables defined on the free form surface. Connect the variables to the appropriate fields in the GUI windows.
   Figure 80 shows the first group of visual connections.

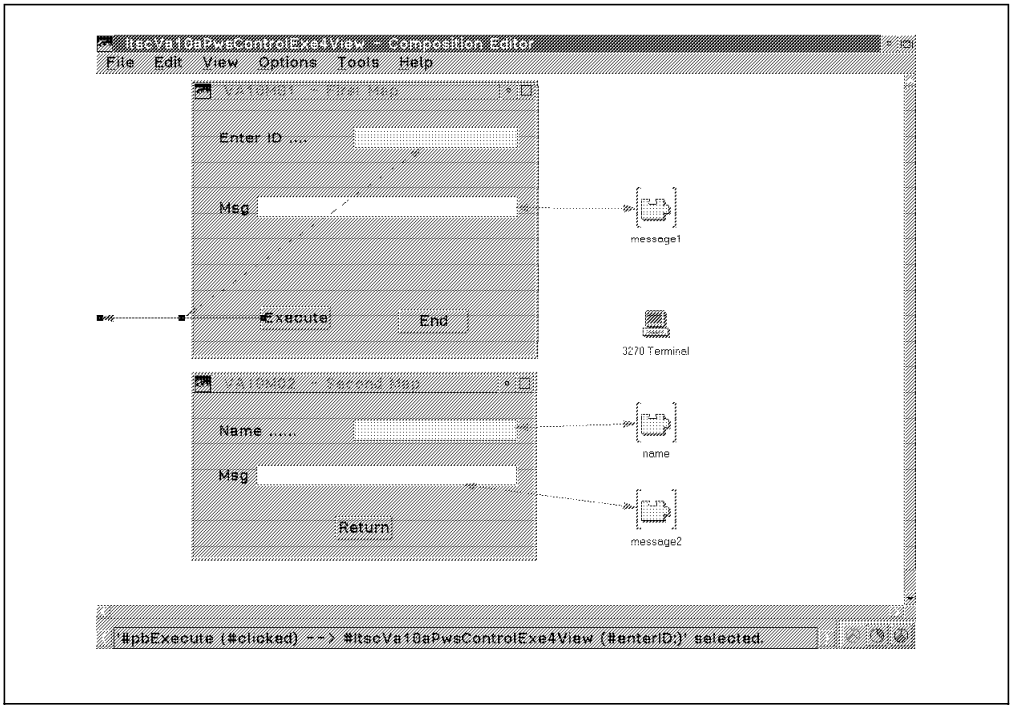

Figure 80. Making the Visual Connections

3. Connect the *Return* push button (#clicked) to 3270 Terminal1 (#keyEnter).
4. Connect the *Return* push button (#clicked) to winMap2 (#closeWidget).
5. Connect the *End* push button (#clicked) to 3270 Terminal(#keyPF:3).
6. Connect the *End* push button (#clicked) to winMap1 (#closeWidget).

Figure 81 on page 67 shows all visual connections.

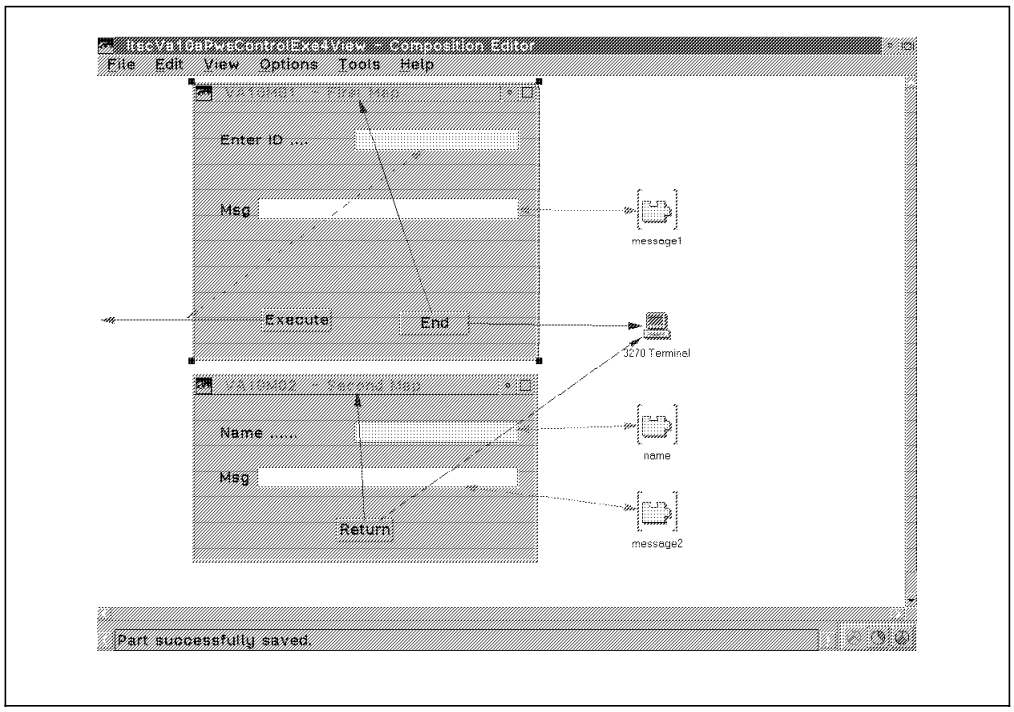

Figure 81. All Connections for this Approach

3.4.3.4 Optimizing the Generated Code

As you would expect, the Smalltalk code that the VisualAge Script Editor generates is not as effective and optimized as code that an experienced Smalltalk programmer would write. You may want to optimize the generated code for better performance and/or readability.

We modified the generated code for the *enterID:* method to make it easier to read and understand. Figure 82 on page 68 shows the two Smalltalk methods that provide the same result. The left-hand part of the figure shows the Smalltalk code that the Script Editor generates, and the right-hand part shows our modified (optimized) statements.

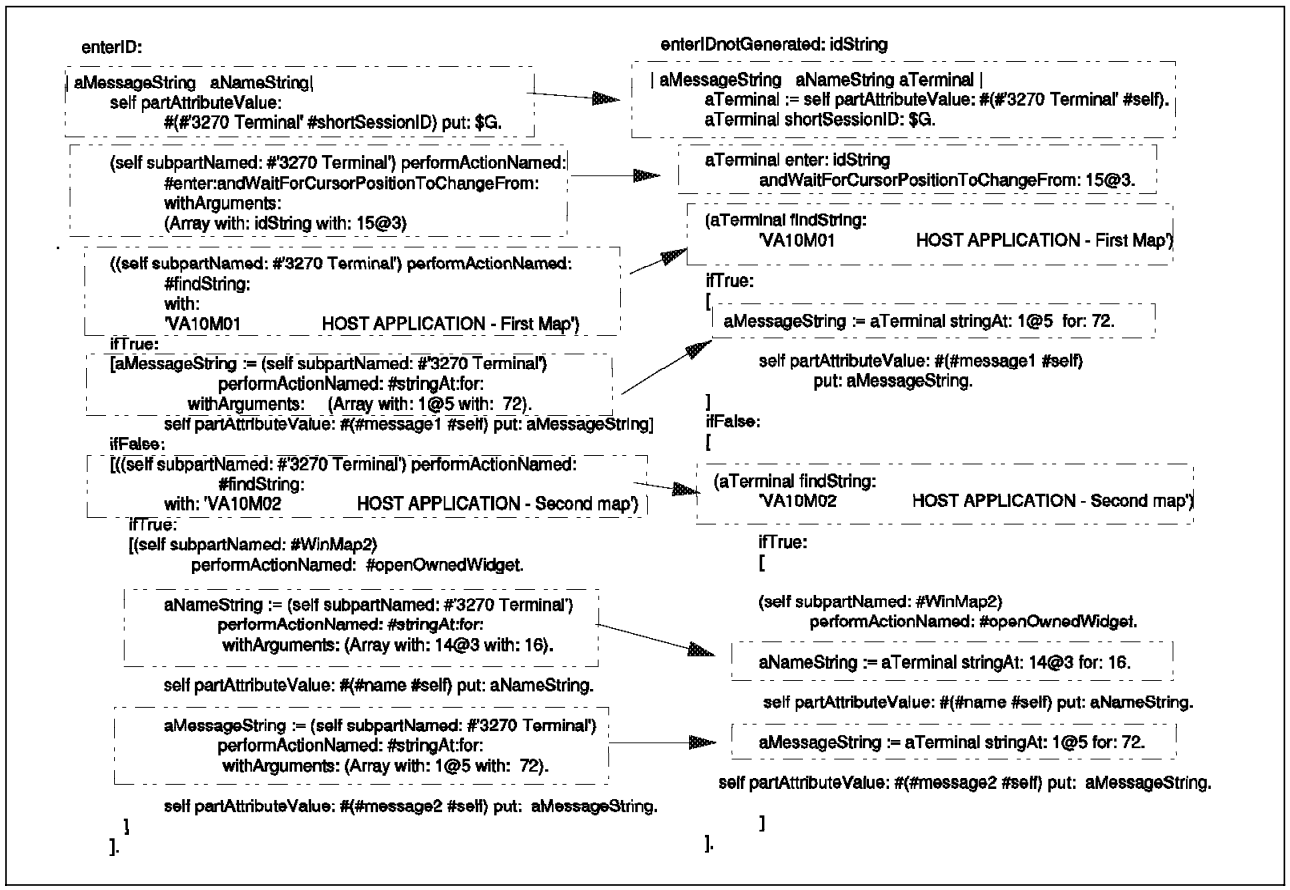

Figure 82. Smalltalk Methods: Generated and Optimized

3.4.3.5 Advantages and Disadvantages of This Approach

Advantages

- Good response time. Because this approach does not use the Abt3270Screen part, we avoided the *Screen Settle Time* delay.
- Better performance than the previous approaches because most of the code is implemented with Smalltalk methods without using the visual connections.
- Easy identification of input and output fields on the host screens. We created Smalltalk methods that use the physical field location on the host screen (row and column) rather than field names.

  This advantage disappears when using the tool we created during our residency project. See Appendix A, “Screen Field Monitor Tool” on page 213 for details of this tool.
- Automatic timeout warning when the host session does not respond.
- No need to have the host session active during development.

Disadvantages

- Requires more Smalltalk skills than the approach with no scripts (see 3.4.2.), although we used the VisualAge generation function that creates the Smalltalk code.
- Requires checking to find out which host screen is displayed.

  When using the Abt3270Screen part, the key string defined for that part is used to validate the active host screen and prevents the sending or receiving of data from an incorrect host screen.

3.5 Isolating the Communication Services

In the approaches described so far we assumed that the communication protocol would always be EHLLAPI and that it would not change over the lifetime of the GUI application. If we had decided to implement another VisualAge communication service, such as APPC or CICS OS/2, for our application, we would have had to change our application dramatically.

In this section we describe an implementation where the communication services are encapsulated in an independent part of the application. Should the communication services ever change, only that part of the application would be affected.

We modified the approach explained in 3.4.3, “Abt3270Terminal Part and Scripts” on page 56 for the GUI application described in this section. Figure 83 shows the model we implemented.

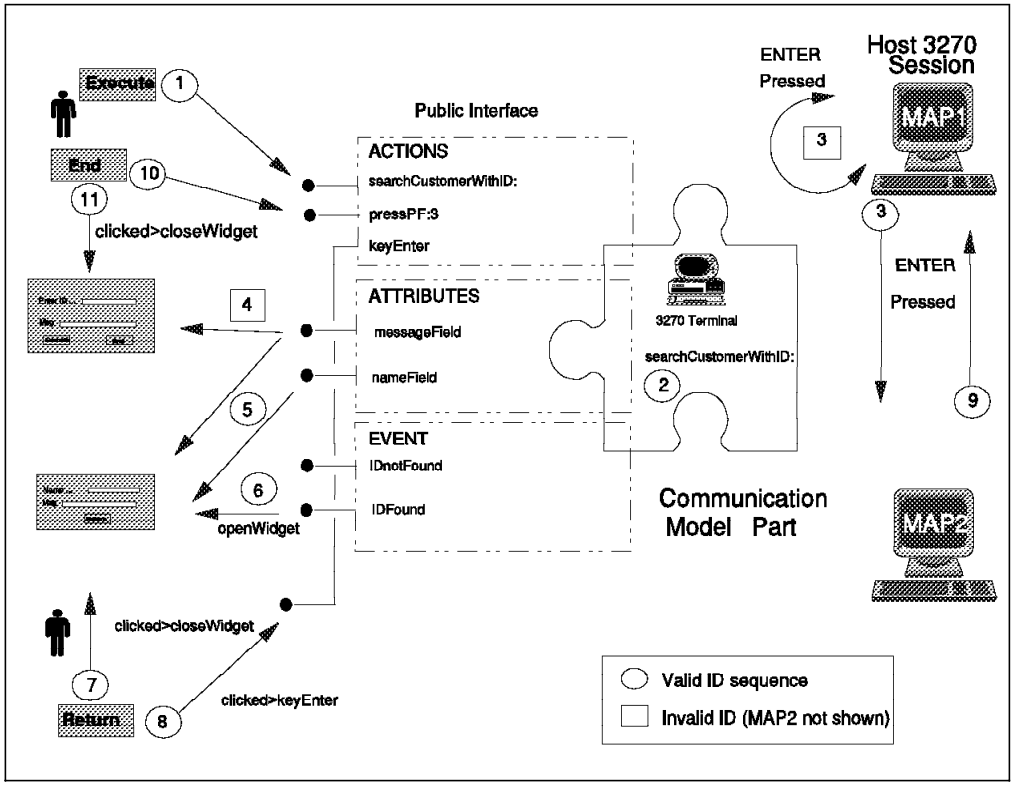

Figure 83. Isolating the Communication Services

The sequence of events is as follows:

1. The user clicks on the *Execute* push button to execute the *searchCustomerWithID* method. The field ID is the parameter for this method. Note that this method belongs to a nonvisual part and is available through its public interface.
2. The *searchCustomerWithID* method uses the *enter:andWaitForCursorPositionToChangeFrom:* method to send the input data to the host, move the cursor to position (80,24), and wait for the cursor to move from that position.
3. The host application executes the Enter key, and either MAP1 or MAP2 is shown.
   - If the user entered an invalid ID, a message is shown in MAP1.
   - If the user entered a valid ID, MAP2 is shown at the host.
4. The *findString* action is executed and searches for the string MAP1.
5. If Map1 is shown, the *messageField* attribute from the Communication Model Part contains the message data to be sent to first GUI window.
6. If Map2 is shown, the *messageField* attribute and *nameField* from the Communication Model Part contain the message and name data to be sent to second GUI window.
7. In addition, if Map2 is found, the *IDFound* event is signaled, and the *openWidget* action opens the second GUI window.
8. The user clicks on the *Return* push button to execute the *closeWidget* action, and the second GUI window is closed.
9. Clicking on *Return* also triggers the *pressEnter* action for MAP2 through the Communication Model Part.
10. The host application executes the Enter key.
11. The user clicks on the *End* push button to send the *pressPF:3* action to the host through the Communication Model Part.
12. The *End* push button triggers the *closeWidget* action to close the first GUI window.

3.5.1 Coding the Application

We created a visual and a nonvisual part to implement the GUI application.

The visual part provided the two GUI windows as described in the previous approaches, and we do not repeat the detailed steps to create that part in this section.

The nonvisual part was used to encapsulate the VisualAge communication services; EHLLAPI in this case.

3.5.1.1 Creating the Nonvisual Abt3270Terminal Part

Create a nonvisual part and add the Abt3270Terminal part to the free form surface as shown in Figure 84 on page 71. You may want to change the Abt3270Terminal part's name to something like *Communication Services*.

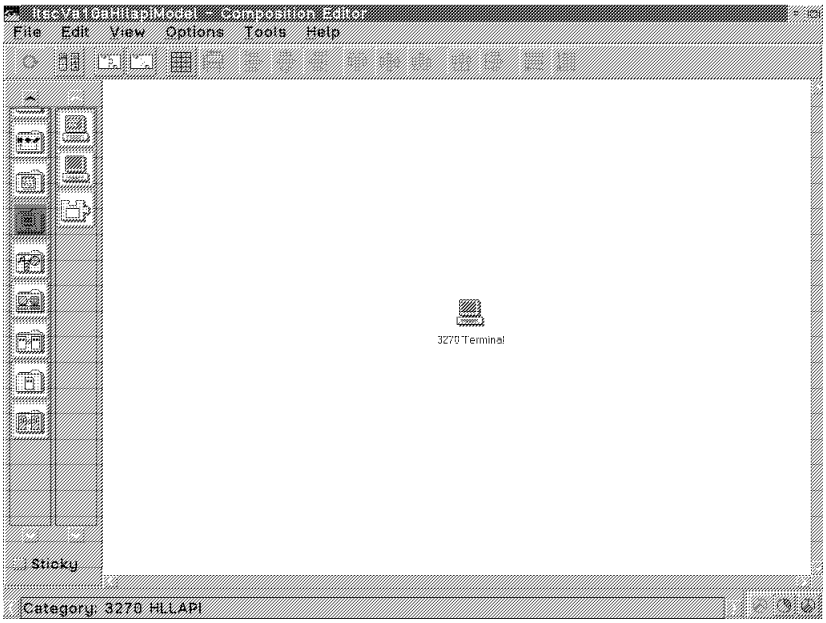

Figure 84. Composition Editor with Abt3270Terminal Part

3.5.1.2 Generating Scripts for the Attributes

Because we needed access to the attributes (instance variables) of the nonvisual part from the methods we wrote, we had to generate, using the public interface editor, the default scripts to access the attributes.

Note: We could have accessed the part attributes directly from within our methods, but using the generated default scripts ensures that all required events are properly defined and raised.

Our nonvisual part had the following attributes:

- messageField
- nameField.

To add the messageField attribute to the public interface and generate the default scripts, perform these steps:

1. Switch to the public interface editor.
2. Select the *Attribute* tab.
3. Type *messageField* in the *Attribute name* field.
4. Click on the *Add with defaults* push button.
5. Select File and Generate Default Scripts... from the action bar.
6. Select *messageField* from the Instance variables list (see Figure 85 on page 72).
7. Click on the *Generate all* push button to perform the generation.
8. Repeat steps 1 through 7 for the *nameField* attribute.

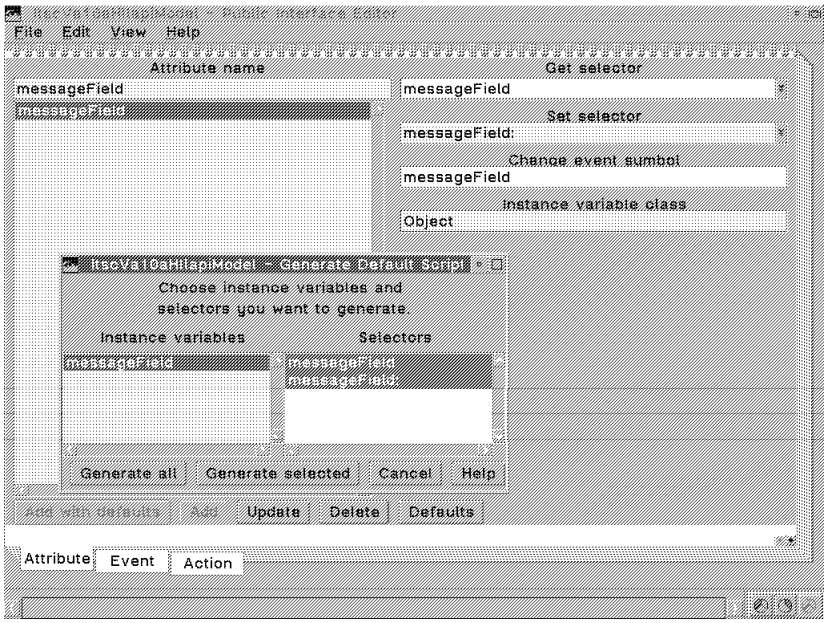

Figure 85. Generating the Default Scripts

The previous steps added the attributes and default scripts as instance variables and methods to the Smalltalk class that implemented our nonvisual part. Figure 86 shows the instance variables and generated methods for the *ItscVa10aHllapiModel* class.

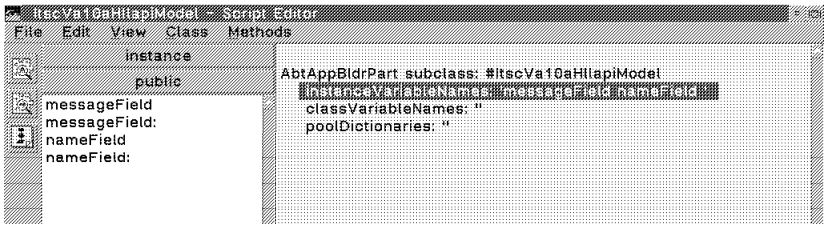

Figure 86. Instance Variables and Generated Methods

The two instance variables in the generated class had to be initialized to avoid problems at run time. Smalltalk initializes instance variables with *nil* by default, which usually results in a *nil does not understand <message>* debugger message at run time.

We used the generated get selector methods to provide a “lazy initialization” for the two instance variables. Figure 87 shows the generated get selector method for the *messageField*.

```
messageField
 "Return the value of messageField."
 ^messageField
```

Figure 87. Generated Get Selector Method for messageField

Figure 88 shows the get selector method for the *messageField* with the “lazy initialization” code added.

```
messageField
 ”Return the value of messageField.”
 (messageField isNil) ifTrue:
 [self messageField: ’’].
 ^messageField
```

Figure 88. Modified Get Selector Method for messageField

Note that the initialization technique used the generated get selector method to set the instance variable to an empty string.

3.5.1.3 Writing the searchCustomerWithID Method

The Smalltalk method we wrote was very similar to the method described in 3.4.3.4, “Optimizing the Generated Code” on page 67, and you should consult the referenced section for details. Figure 89 shows the *searchCustomerWithID:* method.

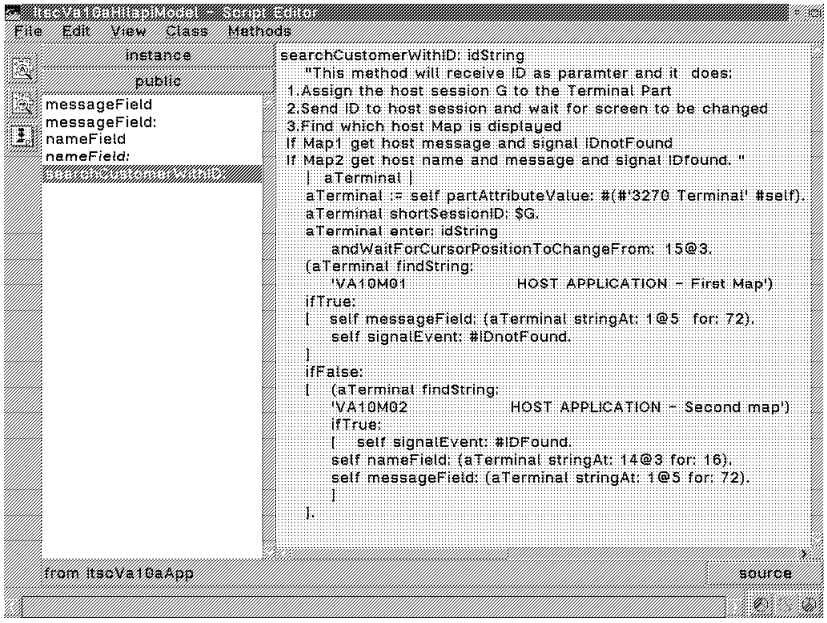

Figure 89. Method: searchCustomerWithID:

3.5.1.4 Creating the Public Interface

To finish our nonvisual part and make it a real VisualAge part as opposed to a Smalltalk class, we had to create a public interface for it. We used the public interface to plug our nonvisual part into the other part of the application, the visual (windows) part.

Figure 90 on page 74 illustrates the relationship between the public interface of our nonvisual part and the Smalltalk class interface.

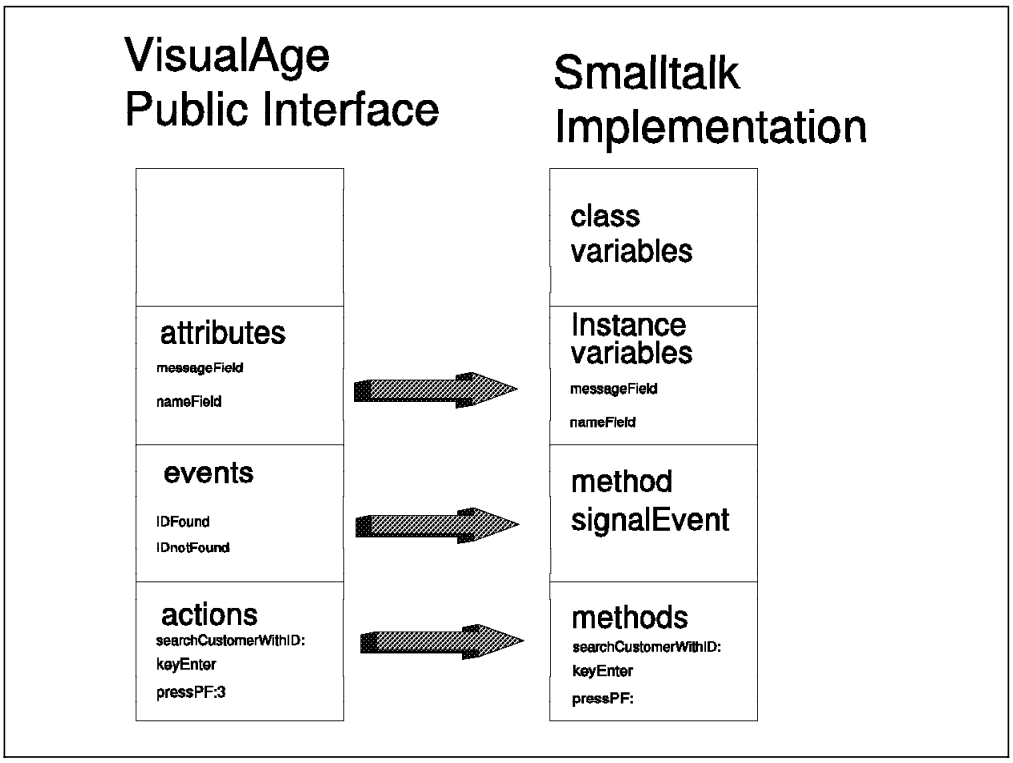

Figure 90. VisualAge Public Interface and Smalltalk Class Interface

The *messageField* and *nameField* attributes already had been added to the public interface in an earlier step (see 3.5.1.2, “Generating Scripts for the Attributes” on page 71). So, what we had to add here were the actions and events.

To add the required actions and events to the public interface, perform these steps:

1. Switch to the public interface editor.
2. Select the *Event* tab.
3. Type *IDnotFound* in the *Event name* field.
4. Click on the *Add with defaults* push button (see Figure 91 on page 75).

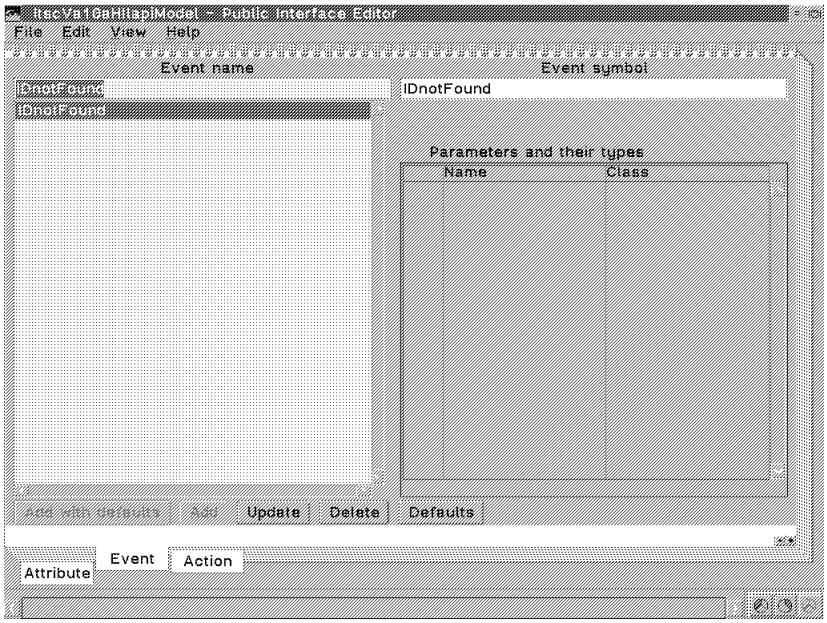

Figure 91. Adding the IDnotFound Event to the Public Interface

5. Repeat steps 1 through 4 for the *IDFound* event.
6. Select the *Action* tab of the notebook.
7. Select the action to be added from the *Action selector* drop-down list.
8. Type the desired action name in the *Action name* field.
9. Click on the *Add with defaults* push button (see Figure 92).

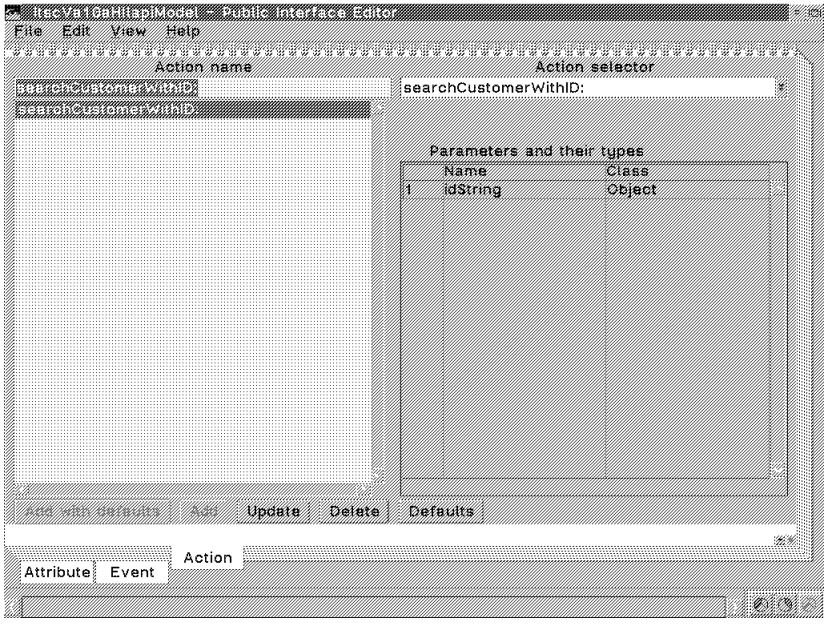

Figure 92. Adding the searchCustomerWithID Action to the Public Interface

At this point, the Communication Services part is ready for use.

3.5.1.5 Making the Connections

We now had to connect the GUI windows to the Communication Services part. Figure 93 shows how to add the Communication Services part to the free form surface.

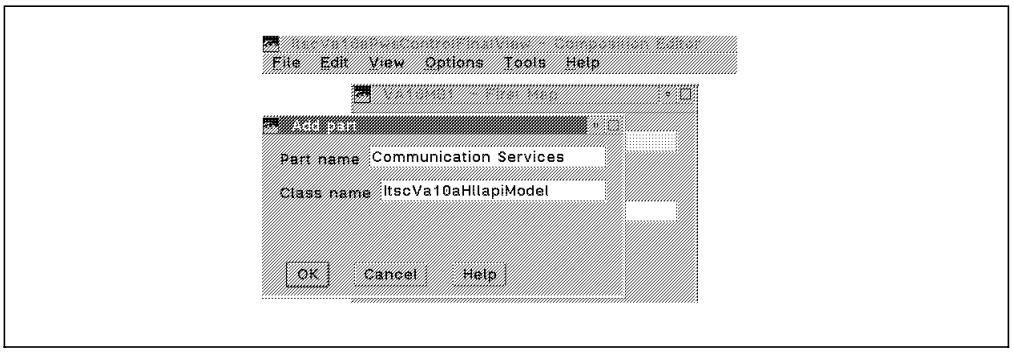

Figure 93. Adding the Nonvisual Part to the Free Form Surface

Make the visual connections as follows:

1. Connect the *Execute* push button (#clicked) to Communication Services (#searchCustomerWithID:).
   This connection is incomplete, and you need to provide the ID to be sent to the host as a parameter.
2. Connect the *entry field ID* (#string) to the incomplete connection (#idString).
3. Connect *Communication Services* (#messageField) to efMsg1(#string).
4. Connect *Communication Services* (#IDFound) to WinMap2(#openWidget).
5. Connect *Communication Services* (#messageField) to efMsg2(#string).
6. Connect *Communication Services* (#nameField) to efName(#string).
7. Connect the *Return* push button (#clicked) to WinMap2(#closeWidget).
8. Connect the *End* push button (#clicked) to Communication Services(#keyPF:3).
9. Connect the *End* push button (#clicked) to winMap1(#closeWidget).

Figure 94 on page 77 shows all of the connections.

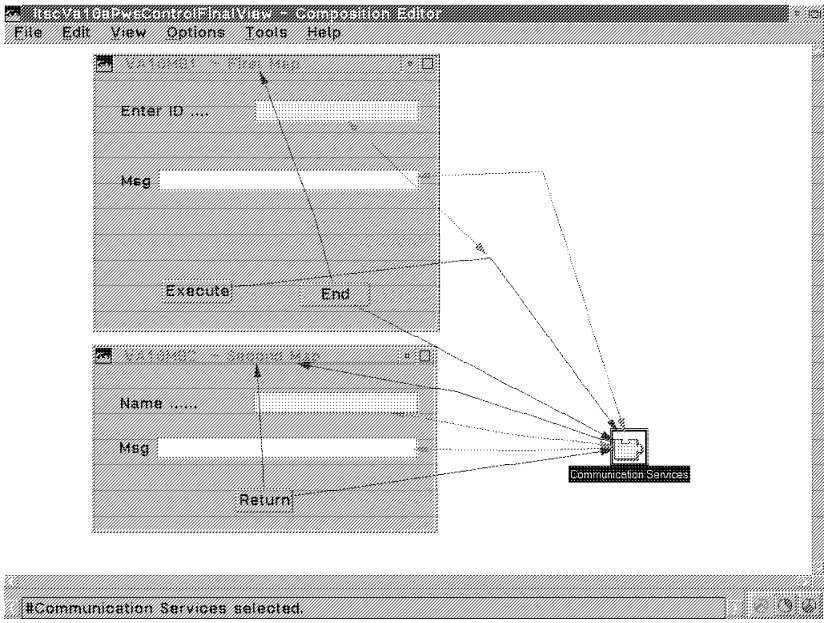

Figure 94. All Connections for this Approach

Conclusion

This is the best approach for our simple application, and it is the recommended implementation approach for real customer applications.

The parts are reuseable and can be used and shared across the installation. The communication services are encapsulated to make maintenance easier. In our example, if the communication services change from EHLLAPI to APPC, CICS OS/2, or any other interface, only the Communication Services part is affected.

3.6 Analyzing the Size of Each Implementation

We analyzed the five implementations for the same GUI application to get an idea of the size of the generated Smalltalk code, which affects the performance of the application. Table 5 shows the results of our analysis.

Table 5. Application Sizes

Description	**Size in Bytes of File-Out**
ItscVa10aView - Host Control - Example 1	24,152
ItscVa10aHostControlEx2View - Host Control - Example 2	27,068
ItscVa10aPwsControlExe3View - PWS Control - Example 3	45,033
ItscVa10aPwsControlExe4View - PWS Control - Example 4	21,593
ItscVa10aHllapiModel - HLLAPI part + ItscVa10aPwsControlFinalView - PWS Control - Example 5	3,939 + 19,088

Conclusions

Example 4, explained in 3.4.3, “Abt3270Terminal Part and Scripts” on page 56, generates the least amount of Smalltalk code and will probably yield the best performance.

Example 3, explained in 3.4.2, “Abt3270Terminal Parts Only and No Scripts” on page 47, generates the largest amount of Smalltalk code because the whole example is coded visually. Example 3 will probably use the most resources (memory and CPU).

We recommend that you use Smalltalk code if the issue is performance.

Part 2. Implementing the Sample Application

Chapter 4. Design and Implementation Considerations

In this chapter we introduce some general issues and approaches to consider before you implement a GUI application.

We address the issue of naming conventions, we talk about different approaches to implementing a GUI for an existing host application, and we discuss the important model-view separation approach for the GUI application.

The issues and approaches discussed in this chapter are valid for the design and implementation of any VisualAge project that has to provide a GUI for an existing application.

4.1 Naming Conventions

When you develop applications with VisualAge you typically create new parts, and it is important that you agree on a naming convention before you start. Such agreement will avoid naming conflicts and misunderstandings when you develop applications in a team.

The naming conventions we used for our project are described in the sections that follow.

4.1.1 Naming Convention for Applications

The suggested naming convention for applications that is used by the product itself is described in the VisualAge manual as follows:

- Choose a three-character **package prefix** to minimize the possibility that component names in one package or application conflict with component names in a different application. An application can be equivalent to a package, or it can be implemented in several packages. The prefix can be an abbreviation of the project, the developer, or the development team. VisualAge itself provides parts with the prefix Abt. We used the prefix Its for our application.
- Use a descriptive name for the application and avoid abbreviations unless their meaning is clear. For example, we used CspSample for our application.
- Use the string **App** as an application suffix.

The complete name for our sample application according to the previously described naming convention is ItsCspSampleApp.

4.1.2 Naming Convention for Parts

The naming convention for parts should underscore the fact that reuse is possible only if it is easy to find the things to be reused. With a meaningful name for a part, the chances of finding the right part as quickly as possible are improved. Here is the suggested naming convention:

- Choose a three-character **package prefix**, for example, Its.
- Use the name of the application for parts that are only reusable within the application, for example, CspSample. Do not use the application name for application-independent parts.

- Use a meaningful name to describe the part, for example, CustomerList.
- Use a **category suffix** for visual parts. The suffix View is not meaningful enough to recognize a candidate for reuse; use such suffixes as Window, Form, Groupbox. Nonvisual parts have no suffix at all.

Sample part names from our application are ItsCspSampleMainWindow and ItsCspSampleCustomerNumberForm.

4.1.3 Naming Convention for Category Parts

Whenever you add parts from the palette (so-called category parts) to the Composition Editor, VisualAge generates a name with a sequence number for the part. For example, the primary window in the Composition Editor is named *Window*. When adding a second window part, this new part is named *Window1*.

These generated names can be difficult to use, especially when writing scripts that need access to the parts in the Composition Editor. You may have a considerable number of parts from the same category, such as push buttons or data entry fields, in your Composition Editor, and it is not easy to distinguish those parts using the generated names.

We suggest that you use names like *pbXxxxx*, where Xxxxx provides a description of the part's use. For example, the *Help* push button is named *pbHelp*. In the case of push buttons it is a good idea to use the text written on the button for Xxxxx. Table 6 shows the suggested names.

Table 6 (Page 1 of 2). Suggested Naming Convention for Category Parts

Category Part	VisualAge Default	Suggested Name
Button Categories		
Push button	Push Buttonx	pbXxxxx
Radio button	Radio Button Setx	rbXxxxx
Toggle button	Toggle Buttonx	tbXxxxx
Scale	Scalex	scXxxxx
Data Entry Categories		
Text	Textx	efXxxxx
Multiline edit	Multi-line Editx	mleXxxxx
Table	Tablex	taXxxxx
Label	Labelx	laXxxxx
List Categories		
List	Listx	liXxxxx
Multiple select list	Multiple Select Listx	mslXxxxx
Drop-down list	Drop-down Listx	ddlXxxxx
Combo box	Combo Boxn	cbXxxxx
Menu Categories		
Push button	Push Buttonx	pbXxxxx
Toggle button	Toggle Buttonx	tbXxxxx
Cascade button	Cascade Buttonx	caXxxxx
Separator	Separatorx	seXxxxx
Menu	Menux	meXxxxx
Canvas Categories		
Window	Windowx	winXxxxx

Table 6 (Page 2 of 2). Suggested Naming Convention for Category Parts		
Category Part	**VisualAge Default**	**Suggested Name**
Form	Formx	foXxxxx
Group box	Group Boxn	gbXxxxx
Notebook	Notebookx	noXxxxx
Container	Containerx	conXxxxx

4.2 Design Models to Map Host Screens to GUI

As described in Part 1, different models exist to implement a GUI for an existing host application with EHLLAPI. We distinguished four models, each with specific characteristics for the GUI application. The characteristics that describe those four models are:

Control Control to trigger the GUI application can be on the host or the PWS side. Control here means which application triggers the next step, for example, to open a window or to change the screen.

Flow control The application flow can be driven by the user or by the application itself. Application driven means that the actual application function provides a restricted number of choices to the user to go further in the application. User driven means that the user can select any application function and jump between different functions without the need to follow a strict sequence of application functions.

Mapping Mapping addresses the relationship of the GUI windows to the host screens. Either each host screen is transformed (1:1) to a separate GUI window or a GUI window represents many (1:m) host screens.

Instances Instances reflect the style of the GUI application in terms of window instantiation. Single instance means that each type of window can be opened once in the application. Multiple instance means that it is possible to have more than one window of the same type opened concurrently.

Figure 95 on page 84 depicts the four identified models.

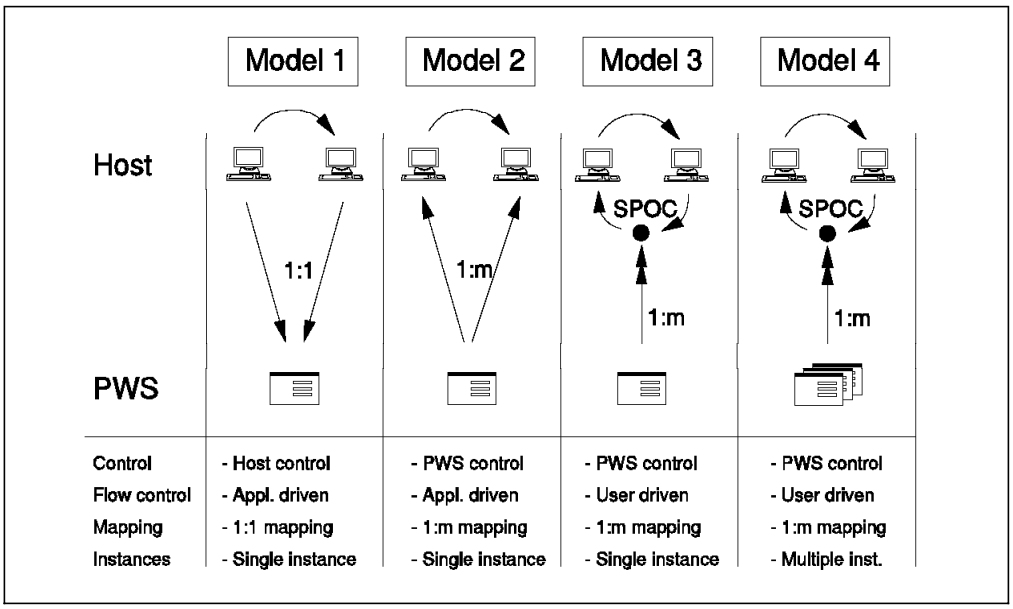

Figure 95. Four Design Models for GUI to Host Mapping

Implementation complexity and flexibility in user interaction grow from model 1 to model 4. That is, model 1 is the easiest model to implement but also the most limited model in terms of user interaction. Model 4 is the most complex to implement but also the most flexible and powerful model for the user.

4.2.1 Design Model 1

In model 1 the master is the host application and the GUI application is the slave that reacts to the actions of the host application. The Abt3270Screen part is used to implement this approach. The process is triggered by the *screenChanged* event.

This design model assumes that the GUI window and the host map are synchronized at any given time. The host map that is transformed 1:1 into a GUI window is always present on the host session. This approach is limited because only one GUI window can be opened at any given time. Whenever the application flow moves on, the actual window must be closed before the next window can be opened.

Model 1 provides a modal GUI application where the user has no flexibility and derives little benefit from the window interface. The user cannot see more information than that provided by a single host map in a single GUI window. This model does not allow multiple window instances.

A general rule for the implementation of GUI applications is to avoid modality in window behavior as much as possible. Therefore, we do not recommend model 1.

4.2.2 Design Model 2

Model 2 is host application driven as is model 1. The difference is that the GUI application triggers the next step in the host application. Also, a window can trigger a sequence of host screens to be run and the GUI application knows about the sequence. The restriction is that after a sequence of host screens, the window and the host screen are synchronized again. It is only possible to start a sequence of host screens that the host application flow allows.

Model 2 provides a modal GUI application where a window has to be closed before another window can be opened because the map behind the window remains open and waits for the next input. With this approach it is not possible to have multiple instances of a window.

Because of its modality, the GUI application still provides limited flexibility to the user, and we do not recommend model 2 if it is possible to select model 3 or model 4.

4.2.3 Design Model 3

The difference between models 1 and 2 and this model (and model 4) is the synchronization between the host screen and the window. In this model, it is possible to start every host function from the same point, the point we call single point of control (SPOC). The host screen does not have to remain open as long as a window is open. Instead it is possible to have multiple different windows open concurrently, for example, a customer detail window of one customer and a customer order window of another customer.

We recommend model 3 (or model 4) whenever possible as the implementation model for an EHLLAPI application because it provides a nonmodal PWS application where the user has control over the application flow.

4.2.4 Design Model 4

Model 4 differs from model 3 in that it allows you to create more than one window of the same type as multiple instances. For example, it is possible to have more than one customer detail window with different customers open concurrently.

This model is just a little bit more difficult to implement than model 3.

4.3 Design of Model-View Separation

A common application segmentation model in the client/server environment divides an application into three components (see Figure 1 on page 3):

- Presentation
- Function
- Data.

The **presentation** component defines how the user and the system interact. This segment is responsible for presenting information to the user and accepting input from the user on behalf of the business logic. Because it does not implement any of the business logic, it can be updated or completely replaced without affecting the business logic. We refer to this user interface function as a **view**.

The **function** component implements the real-world objects of the application, such as a customer, a calendar, or a contract. It defines the behaviors of these objects and their interrelationships without consideration for how they are presented to users or how users interact with them. The design of the objects with their behavior and interrelationships is called the object model. We refer to the implementation of the real-world objects as a **model**.

We do not discuss the **data** component because, from the application builder's perspective, it can be thought of simply as an extension of the model.

With the model-view separation approach, we distinguish between model objects and view objects. Segmenting an application this way provides several benefits:

- It enables prototyping and parallel development. Prototyping of the views can be done by a user interface specialist working with end users. This activity can take place in parallel and somewhat independent of the development of the underlying business model objects.
- It supports multiple views of the same model object. Users can have several concurrent views of the business model objects.

  To use of this approach, it is necessary to implement the general concept of Model-View-Controller (MVC). In this concept, there are two important points to keep in mind:

  – Views can directly update models, but models cannot directly update views. Models inform the controller about a change, and the controller tells all of the concurrent views to refresh their contents by reading the model object again. So the controller knows all of the view objects, and which model object's data they present.

  – Views contain only presentation and user manipulation logic. Business logic should exist only in the model objects.
- It facilitates the distribution of the application in a client/server environment. According to the different client/server models of application distribution, the business logic can be distributed between a client and one or more servers. The view objects do not depend on the location of the model objects; they are also independent of any communications protocol. The model objects encapsulate the communication between client and server and provide methods to access distributed business logic. In this way, the user interface can change to support different media, and the business logic and the protocols of the client/server communication can change without changing the view.

VisualAge supports the implementation of the model-view separation with two different parts:

- **Visual parts** are elements of the application that the user can see at run time. They are components of a presentation surface, such as a window, an entry field, or a push button. These parts are edited in their visual run-time form on the free form surface of the Composition Editor.
- **Nonvisual parts** are elements of the application that the user does not see at run time. These parts are represented at development time by icons on the free form surface. Examples of nonvisual parts are business logic, arrays of data, communication access parts, and database queries.

The VisualAge mechanism to signal events supports the MVC concept. With this mechanism, an object, or in this context, a part, informs the other parts that are connected to it with attribute-to-attribute connections about a change to its attribute values. We have to implement the controller object, which keeps the information about which view object presents the data of which model object.

A simpler approach combines the view and the controller in a view component. This simplification is also known as the MV/C concept. An implementation with VisualAge parts needs a composite view part that contains all of the views of a

model object that can be opened concurrently and are linked to the same model object with connections. Of course, these views themselves can be implemented as separate parts to provide a high degree of modularization.

We did not implement a separate controller part for our application. However, we implemented separate view and model parts (refer to Figure 97 on page 93). The details of our model-view separation design are described in 5.4, “Model-View Separation” on page 93.

Chapter 5. Sample GUI Application Design Overview

In this chapter we present the objectives and assumptions of our project and illustrate the design and implementation steps we went through when developing our GUI application. We explain our design decisions, the model-view separation implementation, and the restrictions of our design.

5.1 Project Objectives

The overall objective of our project was to build a GUI front end for an existing, CICS-based host application using VisualAge′s EHLLAPI support. We wanted to understand the EHLLAPI support provided by VisualAge and investigate different implementation approaches for building GUI front ends using VisualAge. Our experiences were to be documented and, if possible, recommendations made for the implementation of real customer applications.

Before we started the project we established a number of ground rules to be followed throughout the project:

- Provide added value to the existing application.
- Do not change the existing host application.
- Follow CUA* 91 guidelines for the GUI front end.
- Write as little Smalltalk code as possible.

We agreed that customers would not expend the effort and cost to build a GUI front end for an existing application simply to provide the same function and information as the existing application. To present information in a GUI window rather than in a text screen would certainly not be enough. It was our highest priority for our GUI application to provide added value to the existing application.

We assumed that, in real life, customers would use an EHLLAPI application in situations where they could not or did not want to change an existing application. We therefore decided that not changing the existing application was a high priority. We wanted to verify and prove that an EHLLAPI application could be developed using VisualAge′s EHLLAPI support without changing the existing application.

As more and more applications provide a GUI as their user interface, the need to standardize GUIs throughout an enterprise becomes more and more important. Several GUI standards, such as CUA and OSF/Motif**, exist today, and it is important for an enterprise to decide on a standard and apply it to all new GUI development. We decided to follow CUA 91 guidelines for our project.

One aspect of our project was to get a feeling for what could be done visually in VisualAge and what required Smalltalk code. We decided to try to use visual programming whenever possible and write Smalltalk code only when absolutely required. As it turned out, we did not quite follow through on that guideline, and we used Smalltalk code in situations where we could have used visual programming to make the coding easier and more understandable.

5.2 Design and Implementation Steps

We followed the steps illustrated in Figure 96 to design and implement our sample application.

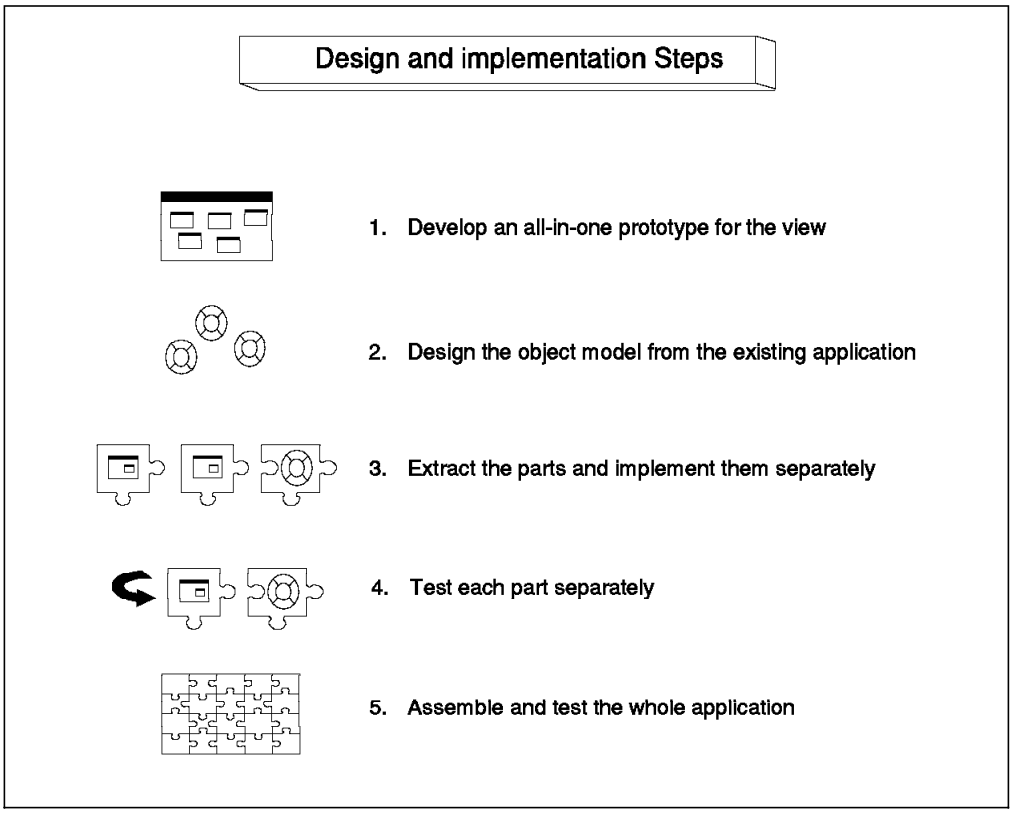

Figure 96. Design and Implementation Steps

5.2.1 Step 1: Develop an All-in-One Prototype for the View

The first step consisted of developing a running prototype of the user interface component of the application. The prototype consisted of a number of windows with hardcoded test data. The windows were connected to simulate the flow of the GUI application. This prototype can be shown to the end user to verify the look and feel as well as the function of the GUI application.

The best way to develop the prototype is to work directly with the end user. VisualAge supports prototyping with its ability to change the running application easily. The Smalltalk environment allows you to switch directly from development in the Composition Editor to test mode without compiling or linking code.

When you start prototyping the GUI for an existing host application, you must analyze and understand the host application with its flow and sequence of host maps. You also need to understand the details of the application functions and the host map constellations that can occur. The best way to document this analysis is a flow diagram of host maps with detailed descriptions of each map.

After analyzing the host application, we developed a first idea of the GUI windows and a simple sequence of the windows on paper. An experienced VisualAge developer could do this first-cut GUI design with VisualAge directly.

For beginners it may be better to develop the first-cut GUI windows and their relationship without any tool to keep the focus on developing the look and feel.

With these first-cut GUI windows, we used the Composition Editor to implement the windows and their sequence. At this point it is not important to think about separate parts. It is easier to develop a prototype with all windows in one Composition Editor view than to start with separate visual parts. The idea is to cut and paste the windows from this view in a later step to develop them as separate parts.

If a single window of the prototype is isolated within the application and has no connections to attributes of other windows, it can be prototyped as a separate part from the beginning and added as a subpart to the main prototype view. This is also true for groups of windows that are connected but isolated from other groups of windows.

The advantage of the all-in-one view is that it is not necessary to provide public interface definitions for the separate parts from the beginning. Controls on the windows can be linked directly with attribute-to-attribute connections, for example, two entry fields or two list boxes in different windows. The disadvantage of the all-in-one view is that the windows in the same Composition Editor view may have to overlap in order to have them in the right position during test.

To provide test data for the prototype, we put controls (labels, entry fields, and list boxes) directly onto the free form surface, specified initial values, and connected those controls to the controls in the windows. Another way would be to define initial values for the controls in the windows.

5.2.2 Step 2: Design the Object Model from the Existing Application

To design the object model from the existing application we looked at the host application with its screens and the functions that could be executed on those screens.

From a detail screen we derived one or multiple real-world (model) objects, for example, a customer or a customer order. From the browse screens we derived the list model objects, for example, a customer list or a customer order list. You cannot usually derive model objects from screens that show menu structures or input fields to search for information.

The object model should show all model objects of the application with attributes and actions. If possible, it would be nice at this point to define some events that the model objects should provide—at least those events that signal whether an action has executed successfully or not.

The object model provides an almost complete definition of the public interface of the nonvisual or model parts. You may want to document your object model with pictures, as in Figure 199 on page 176.

5.2.3 Step 3: Extract the Parts and Implement Them Separately

Based on our prototype we decided that each window should be a separate part and that each element of a window that could be potentially reused should be implemented as a separate part. The more granularity you have the greater the degree of reusability.

There is no fixed rule about which elements of a window should be implemented as separate parts. The guideline should be the potential for part reuse, for example, for a specific entry field or a group of entry fields. You should also consider common logic in similar parts (classes) that can be implemented in abstract parts and provided through inheritance to the similar parts.

To document this step, you can provide a picture that shows the assembly structure of a visual part with its subparts (refer to Figure 125 on page 122).

5.2.4 Step 4: Test Each Part Separately

After developing each part or a group of parts, we tested these basic building blocks for the application. The test can be done separately. In some situations it is useful to test a nonvisual part with a test driver window instead of a real visual part.

5.2.5 Step 5: Assemble and Test the Whole Application

Following the idea of construction from parts, the different parts, which we tested separately, can be combined to form composite parts and, at the end, an application. The number of connections between the different parts can be used as an indicator of the quality of an application constructed from parts. You should have only a few connections between the different parts in the Composition Editor view of the application.

Once the entire application is assembled it must be tested to ensure the proper functioning of the application.

5.3 Design Decisions

We made the following design decisions for our GUI application:

- Use design **model 4** to map the host screens to a GUI.

  With this model we could provide a sophisticated GUI application, where the user has control of the application, and the PWS triggers communication to the host session. Behind a single GUI window, a sequence of host screens is processed, and the start and end point of the sequence is always the same host screen. This is the SPOC in the existing host application. Windows of the same class are allowed to be open concurrently as multiple instances within the application.

- Implement **model-view separation** without a separate controller component.

  With this approach, the visual parts of the application are independent of the underlying communication protocol. Changing the protocol between the client and the server affects only the model side of the application.

- Construct the application from parts.

  Our intention was to design and implement reusable parts for the view, model, and communication aspects of the application. Some of the parts are

abstract parts, which pass on their behavior and their interface to specialized part subclasses.

5.4 Model-View Separation

Figure 97 shows the basic design model of our GUI application with the view components and the model components.

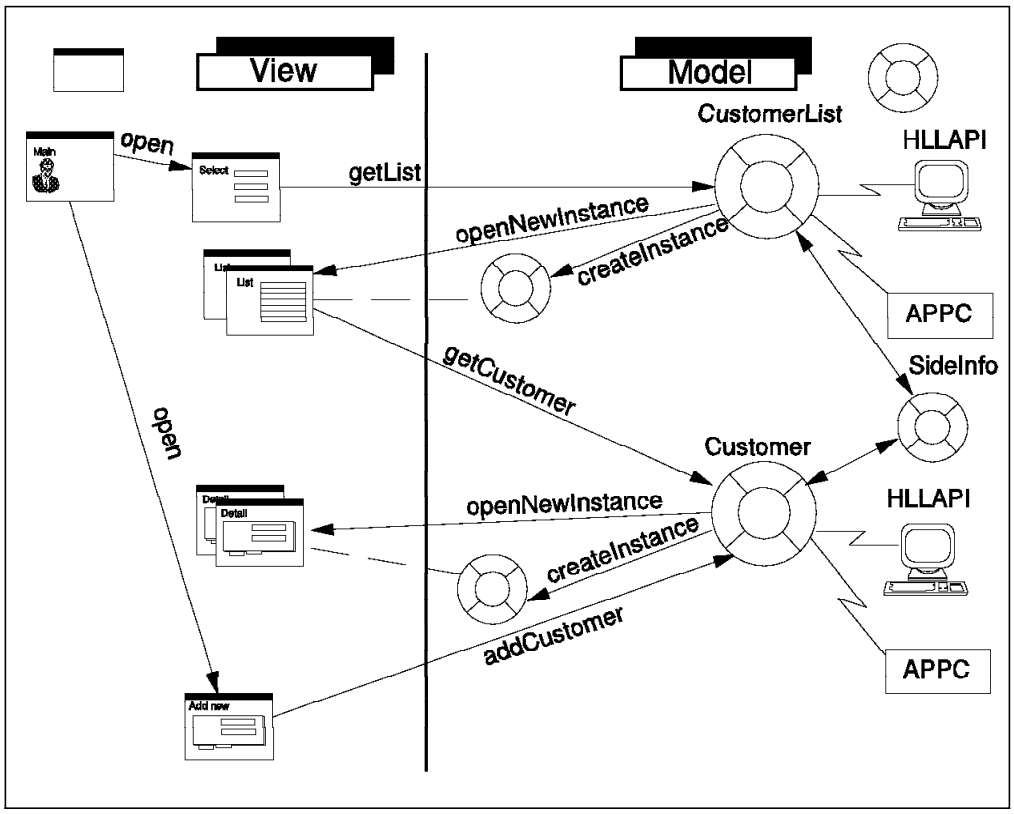

Figure 97. Model-View Separation for the GUI Application

The static aspect of our design model consists of view and model objects. We have several window objects on the view side (left):

- The main window of the application as a single instance window
- The select customer list dialog window as a single instance window
- Multiple instances of customer list windows
- Multiple instances of customer detail windows
- A single instance of a window to add a new customer.

We have three basic model objects on the model side (right):

- A CustomerList object that creates customer list collections
- A Customer object that creates a new instance for each customer to be shown in a view
- A technical communication object (SideInfo) that provides communication-specific information.

The dynamic aspects of our design model are described by the following steps:

1. The window to enter the selection criteria for a customer list is opened from the main window.
2. After the selection criteria are entered in the view part, a getList request is sent to the model part CustomerList.
3. The CustomerList object exists only once in the whole application. It has a method to read the list from the existing host screens using the SideInfo part, and it creates a customer list items collection instance. At the same time, the CustomerList object creates and opens an instance of a CustomerList window, where the customer list items collection instance is used as a variable connected to a list box.
4. The selection of an item in the list box with the customer name and number triggers a getCustomer request, which is sent to the Customer model object.
5. The Customer model object executes the getCustomer method with the customer number passed as a parameter, which triggers the host application to read the details of the customer.
6. The result of this method is the creation of a Customer instance and a customer view object instance, which is opened from the Customer object. The new instance of the Customer object is passed as a variable to the customer view. The messages to update, refresh, or delete this customer are sent from the window to the Customer object passed as a variable.
7. A single instance of a window to add a new customer can be opened from the main window.
8. The addCustomer request from this window is also sent to a Customer object, which adds the customer using the existing host application.

We used the following mechanisms to connect the view with the model objects in our application:

- The main window has the CustomerList model object added as a subpart, and the selection dialog window sends its requests to this subpart.
- The CustomerList model object passes a pointer to itself (#self) to a variable in the instantiated customer list view object.
- The customer list view object has a Customer model object added as a subpart to send a getCustomer request.
- The Customer model object creates another instance of a Customer model object and passes this instance as the value of a variable to the new customer window instance.
- The add customer view object contains a Customer model object as a subpart.

5.5 Restrictions of the Design

Because we did not implement a pure MVC architecture, we could not control whether multiple instances of a customer list window or a customer window presented the same model object. It is possible to open more than one window for the same customer and update the customer data in one window. The other windows are not updated simultaneously.

A solution would be to control the opened customer windows in a way that only one window for a customer with the same customer number can be opened.

We do not address the queueing of communication requests from several model objects. Because the host session is the bottleneck in an EHLLAPI application and the number of parallel host sessions is limited, you would need to lock the host session while a single request is being processed on the session. To avoid the situation where the functions that can start a host communication are disabled in all windows while the host session is occupied, you would have to queue the requests for the host.

Chapter 6. Designing the GUI

The first step in building our VisualAge EHLLAPI application was to design the GUI for the host application. We used the host application described in *Client/Server Computing with AD/Cycle Application Generators* for our project. The source code for this application is provided on a diskette with the Redbook. The application was coded with CSP/AD and runs under CICS.

Note that the language used to write the host application or the environment in which the host application executes made no difference for our VisualAge EHLLAPI application. For example, the host application could run under CMS, TSO, CICS, or IMS/DC, and the VisualAge EHLLAPI application would be the same. This is one advantage of using the VisualAge EHLLAPI support.

In this chapter we explain the steps we went through to design the GUI for our host application.

6.1 Understanding the Host Application

Before we started the design of our GUI interface for the existing host application we had to understand the existing, text-based application interface. There was no need to understand the internals of the application, such as how it accessed the data or whether it used DB2 or VSAM, but we needed to understand the 3270 screens and their relationships to each other.

We executed the host application and performed all available functions to display all 3270 screens. We also looked at the source code of the host application to find all host screen maps to be sure our testing was complete.

You might have to make some decisions at this point—for example, should the help screens from the existing application be used for the GUI application or should help be implemented at the PWS using VisualAge′s help facility. VisualAge′s help facility is nice, but sometimes host applications have a very sophisticated help facility using help texts stored in host databases. It is your decision as to whether you want to migrate the host help facility to the VisualAge help facility.

Remember that you must provide added value to the existing application through the GUI interface. It is not enough to provide the same functions as the existing user interface or present the same information in a GUI window.

6.1.1 Sequence of Host 3270 Screens

Figure 98 on page 98 through Figure 102 on page 100 demonstrate a sequence of 3270 host screens resulting from a user updating a partner. Here are the activities:

- The user entered *B** in the Partner Name field of map TCL0M01 (see Figure 98 on page 98).

```
TCLOM01                          Customer Inquiry




Partner Number  : _______
Partner Name    : B*_____________________________________________
Search to       : _____________________________________________

Command  ==>_____________________________________________________________

Enter F1=Help  F3=End    F5=Refresh
```

Figure 98. Customer Inquiry Screen. The fields highlighted with underscoring (_) are not protected.

- The result of entering this input was a list of partners as shown in Figure 99.

```
TCLOM01                          Customer Inquiry
   Partner
   Number  Partner Name
__ 0000501 Baylis, John G
__ 0000551 Birsfelder, Markus
__ 0000552 Bonde, Anders
__ 0015001 Baylis, B.
__ 0015051 Birsfelder, A.
__ 0015052 Bonde, A.
__ 0015201 Baylis, B.
__ 0015251 Birsfelder, B.
__ 0015252 Bonde, B.
__ 0015301 Baylis, C.
__ 0015351 Birsfelder, C.
__ 0015352 Bonde, C.
__ 0015401 Baylis, G.
                                                                More: +
Partner Number  : _______
Partner Name    : B*_____________________________________________
Search to       : _____________________________________________

Command  ==>_____________________________________________________________

Enter F1=Help  F3=End    F5=Refresh             F8=Forward
```

Figure 99. List of Partners Starting with B. The fields highlighted with underscoring (_) are not protected.

- The user pressed *PF8* until the name to be updated (*Barosa*) was on the screen and entered *U* for update (see Figure 100 on page 99).

```
TCLOMO1                          Customer Inquiry
   Partner
   Number  Partner Name
__ 0015451 Birsfelder, G.
__ 0015501 Baylis, D.E.
__ 0015551 Beelder, D.E.
__ 0015552 Bonde, D.E.
__ 0015601 Baylis, H.
__ 0015651 Birsfelder, H.
__ 0015652 Bonde, H.
__ 0015656 Beavis
__ 0015657 Butt-head
U_ 0015658 Barosa, R. W.



                                                              More:   -
Partner Number  : _______
Partner Name    : B*_____________________________________________
Search to       : _____________________________________________

Command  ==>_____________________________________________________

Enter F1=Help  F3=End    F5=Refresh F7=Previous
```

Figure 100. Partner Selected for Update. The fields highlighted with underscoring (_) are not protected.

- On the Customer Update screen, the user changed the fields as desired and pressed enter (see Figure 101).

```
TCLOMO3                          Customer Update

  Update details and press Enter

    Name....... Mr.    Barosa, R. W._________________

    Address.... Rua das Andorinhas 10_____   Phone... 4084289900____

                apt 104_____________________   Fax..... 40869____

                Campinas____________________   Telex... _____________

    State...... SP_______  Zip.... 04735____

    Industry... Diving_____________

    Contact.......... Ms. Dude_____________________

    Customer since... 05/22/1990

Modify details and press Enter
Command==> ___________________
 F1=Help  F3=End
```

Figure 101. Update Screen. The fields highlighted with underscoring (_) are not protected.

- The customer list was displayed again as shown in Figure 102 on page 100, and the user could use PF5 to refresh the list.

```
TCLOMO1                          Customer Inquiry
   Partner
   Number  Partner Name
__ 0015451 Birsfelder, G.
__ 0015501 Baylis, D.E.
__ 0015551 Beelder, D.E.
__ 0015552 Bonde, D.E.
__ 0015601 Baylis, H.
__ 0015651 Birsfelder, H.
__ 0015652 Bonde, H.
__ 0015656 Beavis
__ 0015657 Butt-head
__ ** Record updated - press F5 to refresh list **



                                                                  More:   -
Partner Number  : _______
Partner Name    : B*_____________________________________________________
Search to       : _____________________________________________________

Command  ==>_____________________________________________________________

Enter F1=Help  F3=End    F5=Refresh F7=Previous
```

Figure 102. Successful Update Screen. The fields highlighted with underscoring (_) are not protected.

6.2 Sequence of GUI Windows

We did the first-cut design for our GUI windows on paper as a team exercise. We followed CUA 91 recommendations for the GUI design. A good source of information for GUI design is *Object-Oriented Interface Design: IBM Common User Access Guidelines*. Figure 103 on page 101 shows our first-cut design.

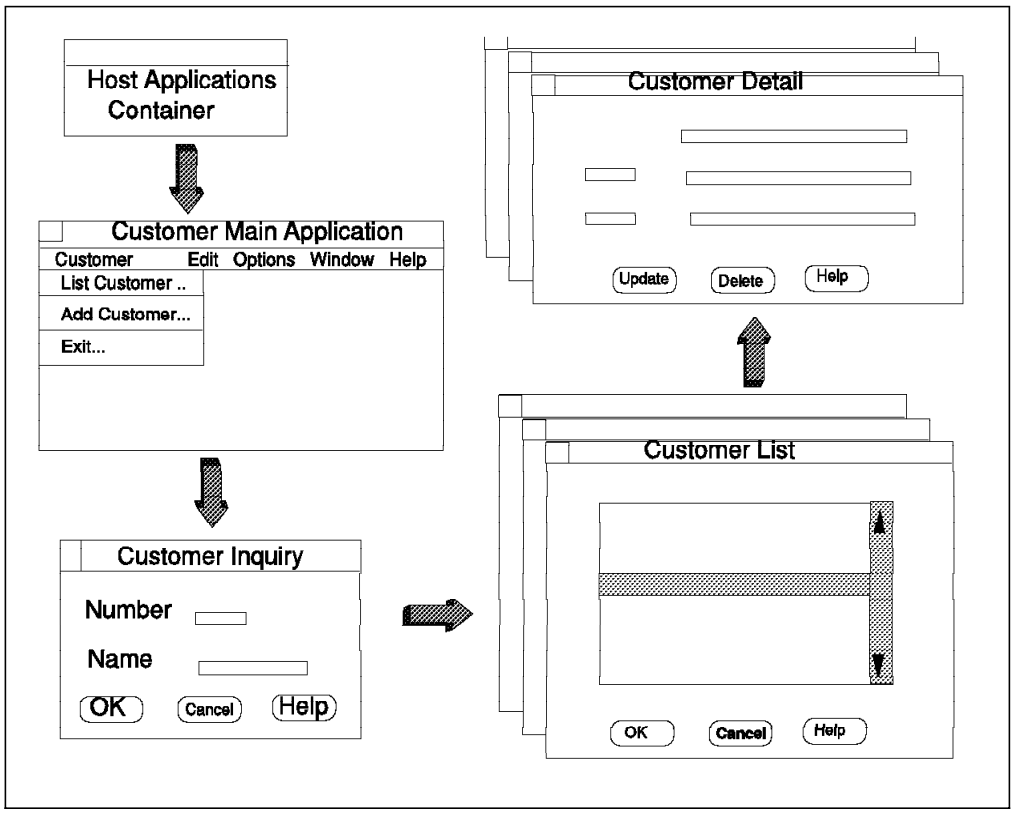

Figure 103. Sequence of GUI Windows at the PWS

6.2.1 What Are the Advantages of the GUI Interface?

We decided to implement the GUI application such that multiple customer list windows and customer detail windows could be open at the same time. This was a feature not available in the existing application unless multiple host sessions were used.

The existing host application displayed the customer list in partner number sequence only. We decided to implement a feature in our GUI application that allowed users to sort the customer list by partner name without changing the existing host application. If we decided to implement that feature in the host application, depending on how the host application was coded, the changes required could be severe because another set of I/Os to the database could be required to sort the results by name.

We also decided to add a feature to the GUI application that would allow users to invoke a PWS editor to write a letter to a customer by just clicking on a push button on the customer detail window. The letterhead would be created automatically from the available customer information.

The GUI application we developed can be used in an environment where nonprogrammable terminals (NPTs) and PWSs coexist in the same network. Users sitting at a PWS can benefit from the added value provided by the GUI application, and users sitting at an NPT can still use the existing, text-based interface of the host application. This allows for a smooth transition from text-based user interfaces to GUIs.

Other features that could easily be added to our GUI application include multimedia, automatically initiated phone calls, and automatically sending FAXes to a selected partner.

6.3 Prototyping the GUI Windows

After we finished our first-cut GUI design we implemented a prototype of our GUI interface with VisualAge. Implementing a prototype is an important step, because it allows you to show the sequence of GUI windows to the end users.

VisualAge was an excellent tool for GUI prototyping. We could easily create prototypes, and, if required, easily change them. Basically there are two approaches to building a GUI prototype:

- All-in-one
- Layered.

Both approaches had advantages and disadvantages.

An all-in-one prototype was easy to create and kept everything on one Composition Editor free form surface. The drawbacks were that the Composition Editor got cluttered, and reusing parts from the prototype for the final GUI application was difficult.

A layered prototype allowed the reuse of components from the prototype for the final GUI application, and the Composition Editor was not cluttered. However, a layered prototype was a little more difficult to create, and the all-in-one view effect in the Composition Editor was lost.

We decided to create an all-in-one prototype to give us an idea of the GUI window sequence. Figure 104 on page 103 shows our prototype in the Composition Editor.

It is not good design practice to have the entire application in one part. For our final implementation we used the construction from parts technique. *Construction from Parts Architecture: Building Parts for Fun and Profit* is a good reference regarding the technique.

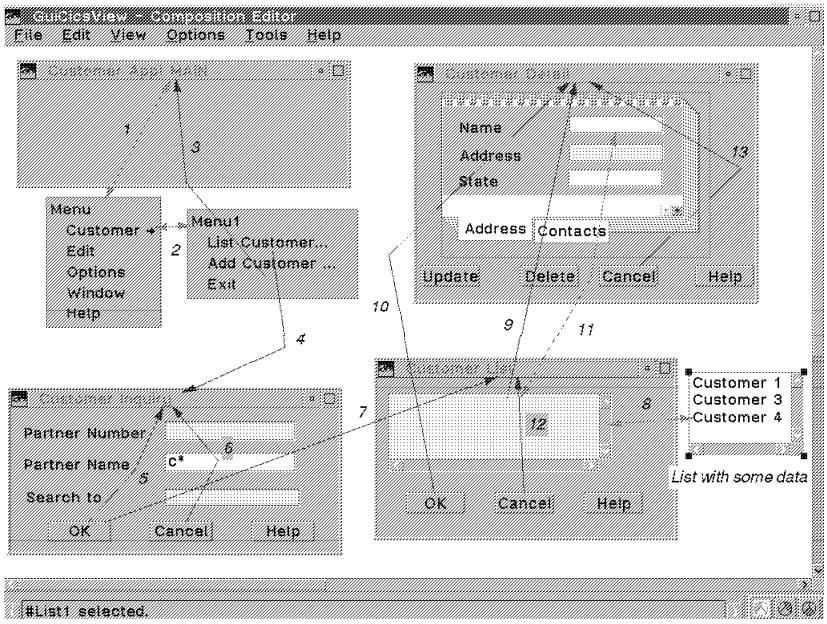

Figure 104. GUI Prototype in the Composition Editor

Note that we also added a list box with some test data to the free form surface to allow us to select a specific customer from the customer list window.

To explain the connections, we added numbers and text (in *italics*) to the Composition Editor. Those numbers are related to the numbers in Figure 105 where the visual connections are explained.

```
1.  Shell1(#menu)                  --> Menu (#self)
2.  Customer cascade Button(#menu)--> Menu1(#self)
3.  Exit Push Button (#clicked)    --> Main Window(#closeWidget)
4.  List Customer Button(#clicked)--> Customer Inquiry(#openWidget)
5.  OK Push Button (#clicked)      --> Customer Inquiry(#closeWidget)
6.  Cancel Push Button (#clicked) --> Customer Inquiry(#closeWidget)
7.  OK Push Button (#clicked)      --> Customer List(#openWidget)
8.  List(#items)                   --> List with some work data(#items)
9.  List(#defaultActionRequested) --> Customer Detail Window(#openwidget)
10. OK Push Button (#clicked)      --> Customer Detail Window(#openwidget)
11. List(#selectedItem)            --> Name Entry Field (#string)
12  Cancel Push Button (#clicked) --> Customer List Window (#closeWidget)
13  Cancel Push Button (#clicked) --> Customer Detail(#closeWidget)
```

Figure 105. Connections for the GUI Prototype

Chapter 7. Running the Sample Application

In this chapter, we show the flow of the running application. We explain the function of each window and discuss the sequence executed by the host application.

7.1 Host Applications Window

After starting the application, the Host Applications window (Figure 106) is displayed. This first window is a container view of icons, each representing a host application that the user can select with a double-click. Our implementation provides icons for two applications, the DB2 sample application and the CSP sample application.

No EHLLAPI call is needed for this window. This is a stand-alone GUI window that does not talk to the 3270 session. Therefore, our host session in the background shows the VAMP screen.

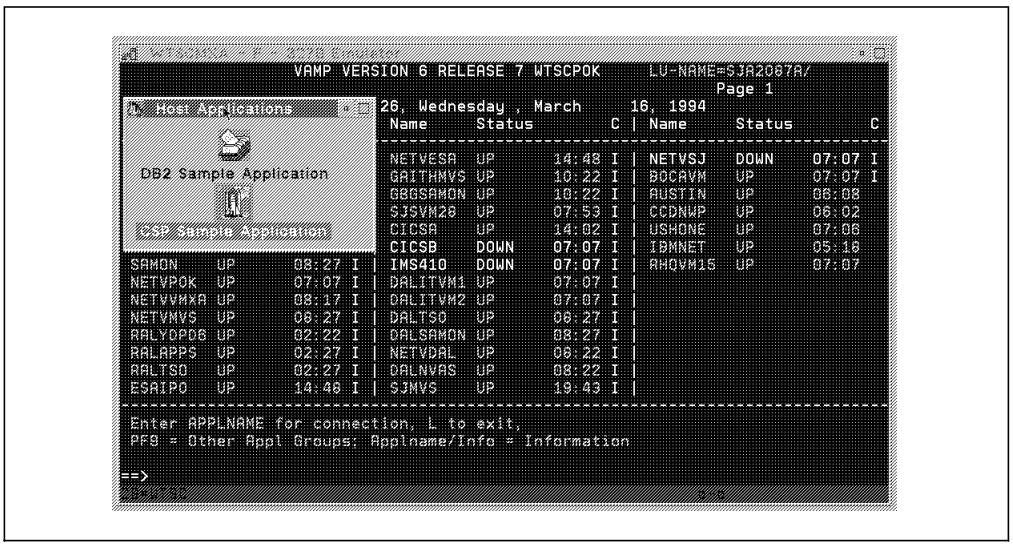

Figure 106. Host Applications Window

7.2 Logon to the CSP Sample Application

After selecting the CSP sample application icon in the Host Applications window, the Logon to CSP Sample Application window is displayed (Figure 107 on page 106). The window provides a list box with all active 3270 sessions. We can select a session in the list box, in our example, session F, and click on the Apply push button to choose the session to log on and run the host application. A shortcut for selecting the session ID is to double-click on the session in the list box. The selected session ID (the first time, the default session id) is always shown in the Connect to field.

The User ID and Password fields are provided to enter the userid and password for the host logon procedure. Because our CICS system works without security installed, the function behind these entry fields is not implemented.

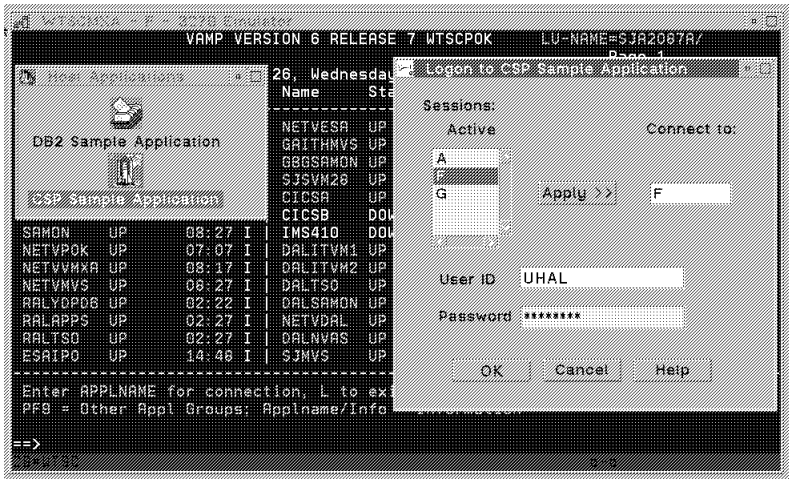

Figure 107. Logon to CSP Sample Application

After entering the data in this window, we click on the OK push button to start the logon procedure. Following this user action, the GUI application enters the string CICSA on the F session, and after the CICS good morning message panel is shown, activates the Clear key. The result is an empty host screen waiting for a transaction code to be entered.

7.3 Customer Application (Main) Window

The next window is the Customer Application (Main) window of the CSP sample application (see Figure 108). This window is a container view with icons to select the different application functions. By double-clicking on the appropriate icon, we can add a new customer, show a list of customers, or jump to the host session we selected in the logon window—session F in our case.

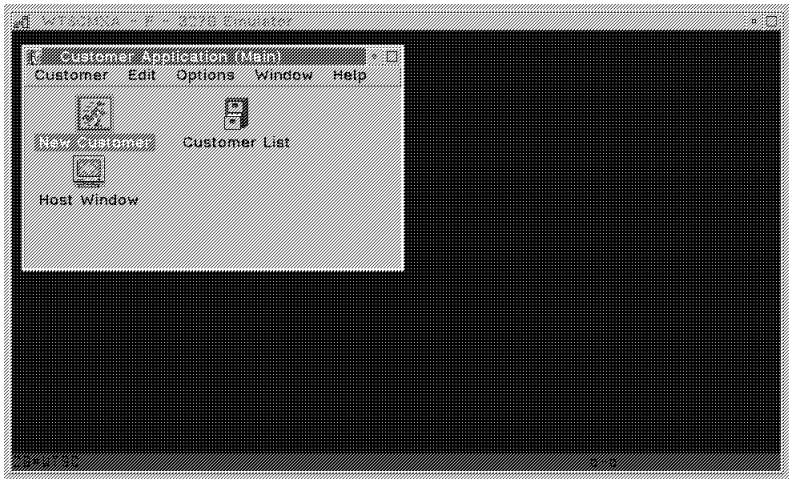

Figure 108. Customer Application (Main) Window

The Customer Application (Main) window provides an action bar with a Customer menu item. The Customer menu provides a pull-down with three entries: Add Customer, List Customer, and Exit. Selecting the menu items has

the same effect as selecting an icon. The other action bar menu items are not completely implemented; they are there to give you an idea of a full function CUA action bar.

The Customer Application (Main) window involves no host communication; the host session is still waiting for a transaction code.

7.4 Customer List Selection

Selecting the Customer List icon in the Customer Application (Main) window brings up the CustomerList Selection window (Figure 109) where the selection criteria for the customer list can be entered. The selection critera can be entered as a combination of:

- Start name
- End name
- Customer number.

At least one selection criterion must be entered. Wildcard searches (substring ended with an asterisk) are supported.

We enter * in the Name from field to get a list of all customers.

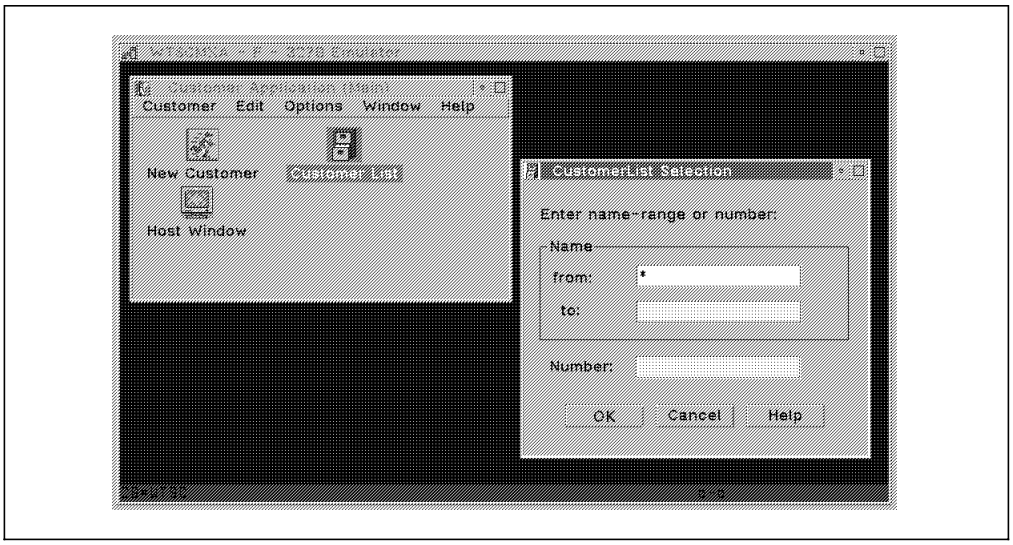

Figure 109. CustomerList Selection

After we click on the OK push button, the GUI application starts to talk to the host session. The CICS transaction code STLC is entered, the data for the search criteria is entered into the input fields of the host map, the enter key is pressed, and the resulting host list is scrolled using PF8 and read by the GUI application until the end of the host list is reached. After the whole list of customers is read by the GUI application, the GUI application activates the PF3 key to exit the host application. The host session shows the black CICS screen again, ready to receive the next transaction code. This is the common entry point, or SPOC, to our host application.

7.5 Customer List Window

The next window is the CustomerList window (Figure 110) with a list box and several buttons. The list box shows items with a customer number and a customer name. The result of our selection is a list starting from customer name A* to Z*. The items in the list box are sorted by name.

We can easily do some processing on the PWS, because we now have the list data on the PWS. If we click on the Sorted by Number radio button, the items in the list are sorted locally on the PWS and shown again in this new sorting sequence. This is a new function, provided only by the GUI application.

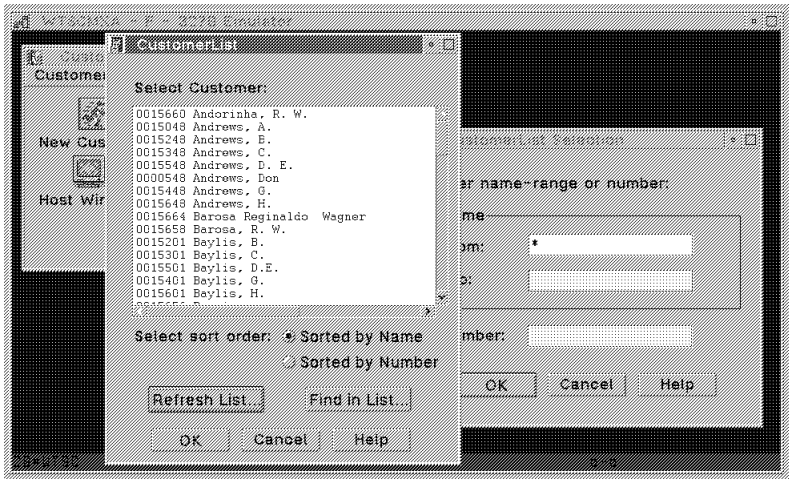

Figure 110. CustomerList Window

7.6 Find in Customer List

The GUI application has the advantage of having the customer list data available on the PWS. Therefore, it is possible to implement a function to find a specific substring in the customer list without any host communication.

Clicking on the Find in List... push button brings up the Find in CustomerList window (Figure 111 on page 109) where a substring for the local find function can be entered. We enter the string Ha (no * required) and click on the Find push button. The result is the subset of the list of items from the customer list where the Ha substring is found. Figure 111 on page 109 shows the Find in CustomerList window with two customers with Ha in their names.

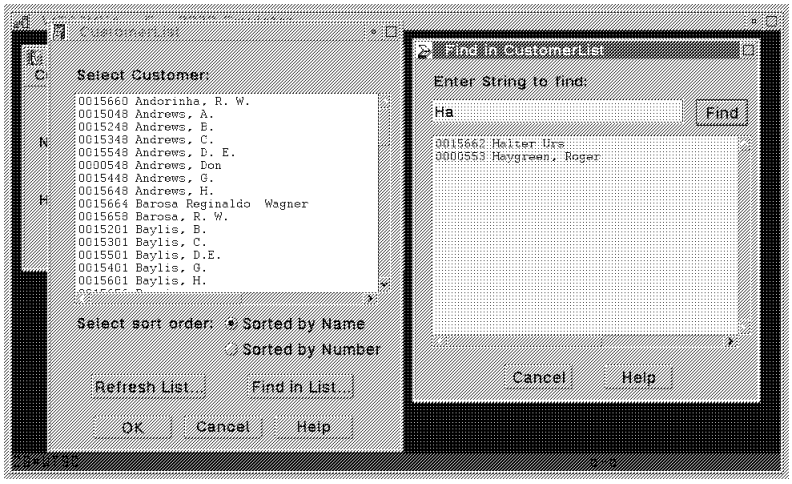

Figure 111. Find in CustomerList Window

7.7 Refresh Customer List

Because the customer list data is kept on the PWS for as long as the CustomerList window is open, it is necessary to provide the ability refresh the customer list data from the host. New customers can be added or existing customers can be deleted by other users while the list data is kept on the PWS. The refresh function rereads the customer list from the host.

To initiate the refresh function we click on the Refresh List... push button in the CustomerList window (Figure 110 on page 108). The Refresh CustomerList window (Figure 112 on page 110) is opened with the selection criteria from the customer list selection dialog kept in the entry fields. In our example, the * in the Name from field selects the entire list of customers. We could overtype the values in the entry fields of the Refresh CustomerList window and make a new selection.

Clicking on the Refresh push button starts the host communication, and the result is the refreshed list in the same CustomerList window (see Figure 112 on page 110).

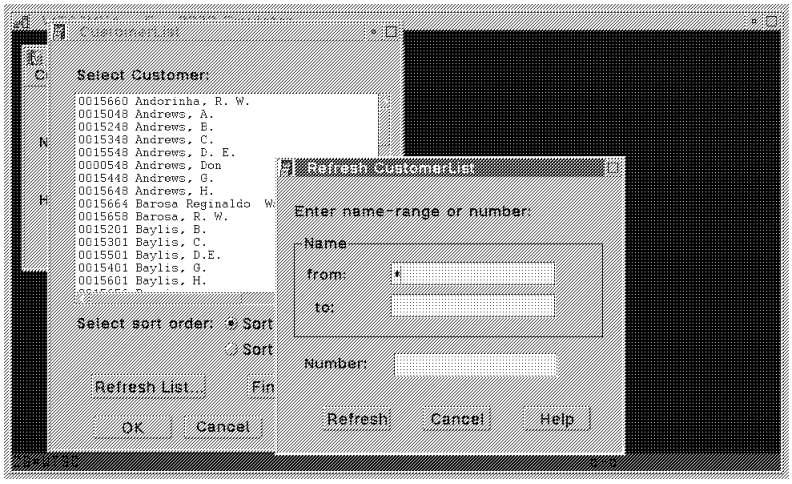

Figure 112. Refresh CustomerList Window

The host application sequence is the same as when the customer list was created initially.

7.8 Multiple Instances of Customer List Window

We implemented model 4 of the design approaches for mapping the host application to the GUI (refer to 4.2, “Design Models to Map Host Screens to GUI” on page 83). Therefore, we could open multiple instances of the GUI windows. Figure 113 shows a second instance of the Customer List Window to illustrate this capability.

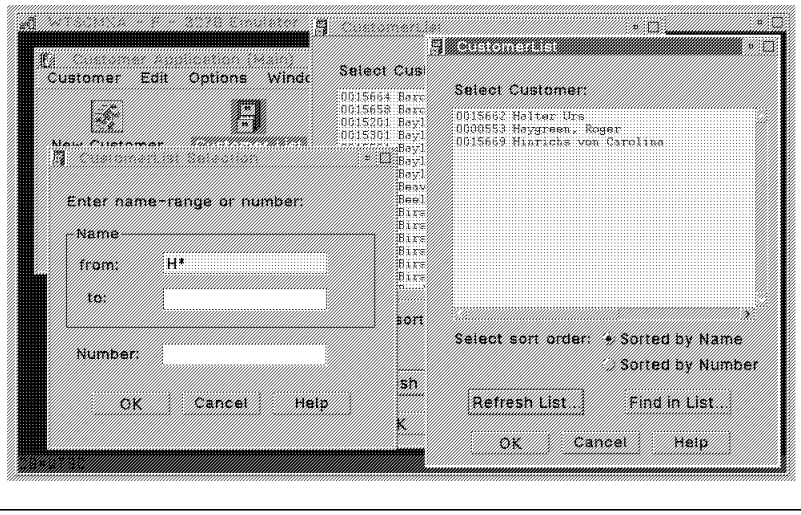

Figure 113. Multiple Instances of CustomerList Windows

We select the Customer List icon in the Customer Application (Main) window, which brings up the dialog to enter the customer list selection criteria for the second CustomerList window. We enter H* in the Name from field to select all customers with a name starting with H. The result is the second CustomerList window shown in Figure 113.

The host application sequence is the same as when the first customer list was created. In the end the host is on the CICS screen, waiting for the next transaction code.

7.9 Customer Detail Window (Address Page)

Now we select a customer from the customer list to show the detail information for this customer. We select customer Barosa, R.W. with customer number 0015658. The result of this selection is a window with a notebook with two pages containing the detail data for the customer address and contacts (see Figure 114).

Compared to the host screen for the customer detail information (see Figure 101 on page 99), the GUI application with the notebook lets us present the data in logical groups. Changing customer data is improved by providing drop-down lists for the title, state, zip code, and industry code fields. We can select one of the predefined items in the drop-down list instead of having to remember the correct values and typing those values in the entry fields of the host application.

The customer detail window also provides push buttons to refresh the detail information and update or delete the customer.

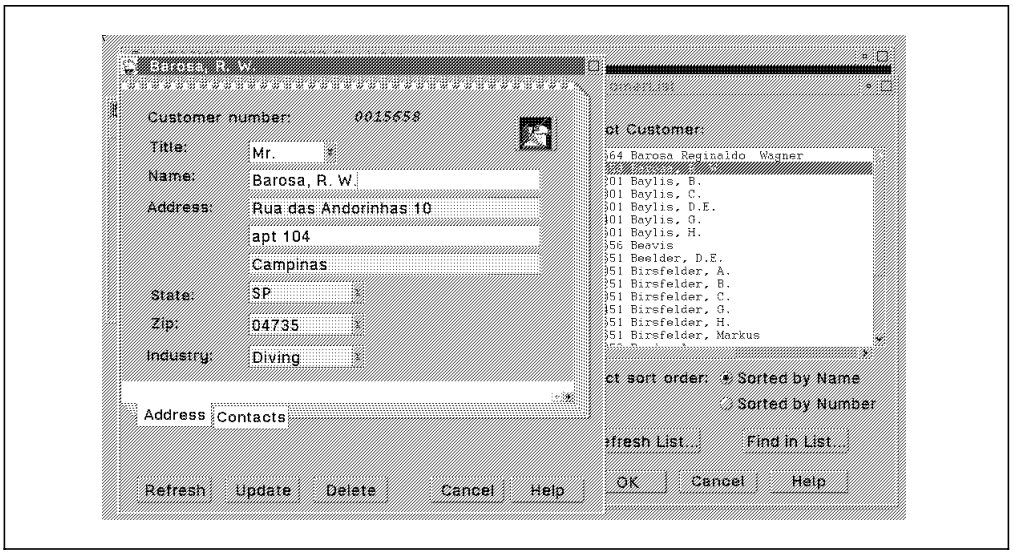

Figure 114. Customer Detail Window (Address Page)

Hidden behind the function of the GUI application that reads the details of a specific customer is a lot of EHLLAPI-to-host communication. The transaction code STLC is entered, the unique customer number for the search criteria is entered into the number input field on the host map, and the enter key is pressed, which results in a list with one item: the customer with the specified customer number. The GUI application types an S to select the details for the customer on the first line of the list, and the customer detail map appears. On this screen the GUI application reads all output fields and presses PF3 twice to go back to the blank CICS map, the entry point for the next host request.

7.10 Writing a Letter for the Customer

Another added value we can provide in a GUI application is the integration with other PWS applications. We can start a PWS application and pass data to that application. In our example, we show how to start a PWS editor to write a letter to a selected customer. The customer address is automatically inserted into the letterhead.

We click on the write a letter icon, and the GUI application starts the OS/2 EPM editor after we specify a file name through the standard file dialog. The result of this user action is shown in Figure 115. The GUI application has passed the address of our customer Barosa to the EPM editor.

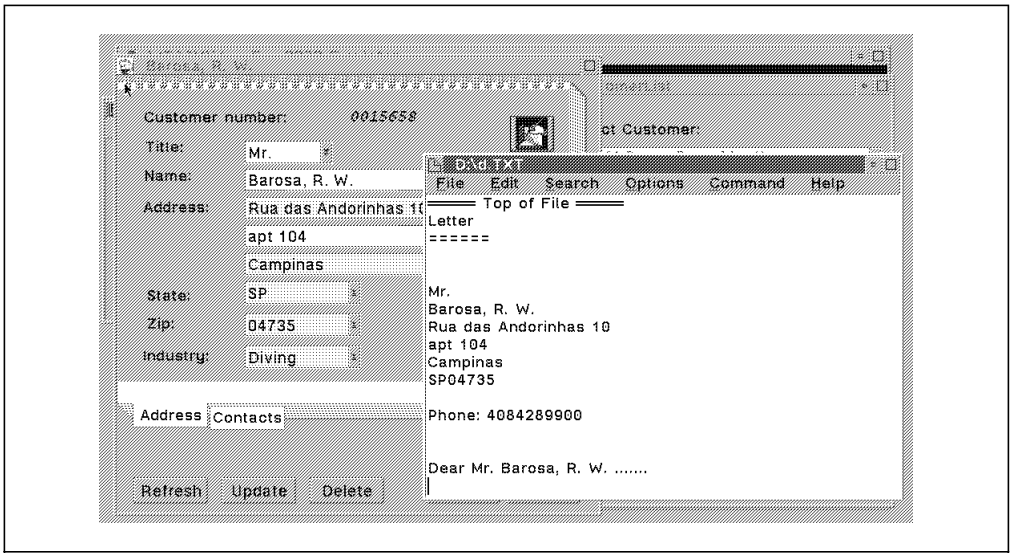

Figure 115. Writing a Letter for a Customer

We can now finish, print, and save the letter for our customer. This is just one example of application integration on the PWS. There are certainly many other situations where we could pass data from the host directly to standard software on the PWS.

7.11 Customer Detail Window (Contacts Page)

If you wanted to call or send a FAX to a customer you would select the tab with the Contacts labels in the customer detail notebook and get the second page of the notebook with the phone, FAX, and telex numbers (see Figure 116 on page 113). Similar to the push button to write a letter on the first page, there is an icon next to the phone number for initiating a call and an icon next to the FAX number for sending a FAX to the number in the entry field. We did not implement these two functions in our sample application. We mention them here to give you an idea of additional functions and application integration on the PWS that can improve the function of the host application.

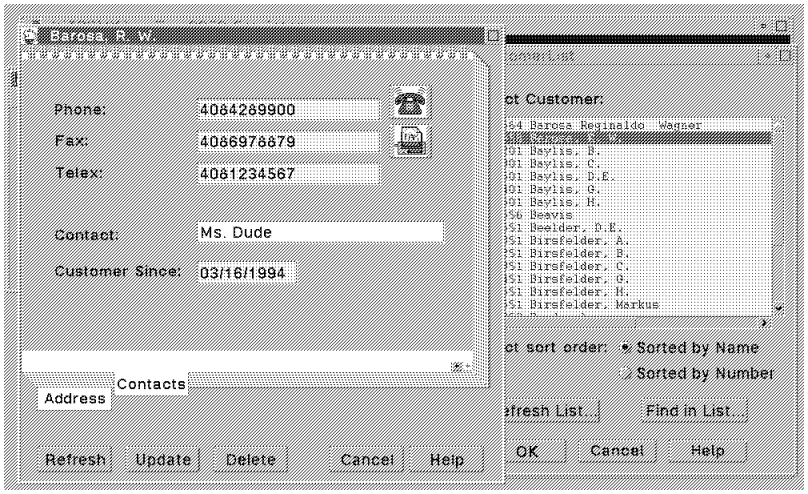

Figure 116. Customer Detail Window (Contacts Page)

7.12 Multiple Customer Detail Windows

It is possible to have multiple customer detail windows for different customers open at the same time (see Figure 117). To make it easy to distinguish the different detail windows we use the customer name as the window title.

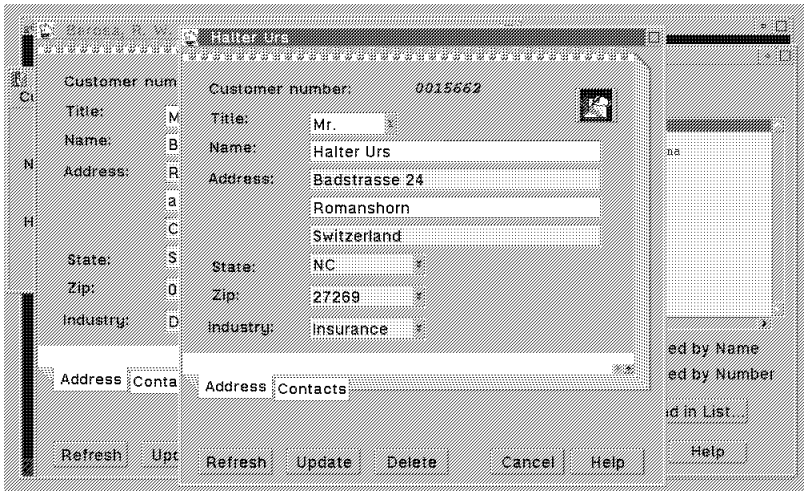

Figure 117. Multiple Customer Detail Windows

7.13 Add Customer Window

To add a new customer, we select the New Customer icon in the Customer Application (Main) window. The New Customer window (Figure 118 on page 114) is opened and shows the same notebook for entering customer data as we saw in the customer detail window. The difference between the two notebooks is that the new customer notebook has no customer number, because the customer number is created when the customer is added.

The Erase fields push button in the lower right-hand corner of the notebook is used to initialize all entry fields with a blank value. This function is useful when adding more than one customer; you do not need to overtype the values of a previously added customer. By default, the values entered for a customer are kept in the entry fields for situations where only a few of the entry fields change from one customer to another.

We click on the Add more push button to add more than one customer. This action keeps the New Customer window open after the customer is added successfully, and we can add additional customers. We click on the Add push button if we want to add one customer only. The New Customer window is closed automatically after the customer has been added.

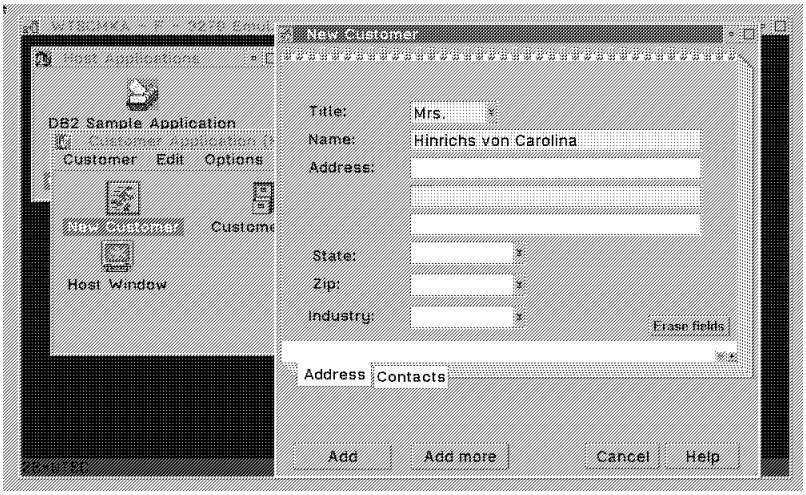

Figure 118. New Customer Window

The Add and Add more functions of the GUI application involve a lot of EHLLAPI-to-host communication. The transaction code STLC is entered. The host application does not implement the function to add a new customer with a separate transaction code that we can execute directly. Instead, we must select a list of customers and type N (for new) in front of any customer in the list. Therefore, our GUI application starts a selection of all customers by typing * in the Name from input field on the host map. On the first line of the resulting customer list, the GUI application types N to get to the detail window to add a new customer. The GUI application passes the data for the new customer from the notebook to the host application and presses enter to add the new customer. To finish and go back to the common entry point, the GUI application presses PF3.

Chapter 8. Implementation Walkthrough

In this chapter we describe the implementation of our sample application step-by-step by walking you through the parts. **To facilitate comparison the sequence of the walkthrough is the same as the application flow described in Chapter 7, "Running the Sample Application" on page 105.**

We use the following structure to describe the parts with their static and the dynamic behavior:

Part name	Smalltalk name of the part (class)
Category	Visual, nonvisual Abstract (used only for inheritance), basic, composite (built from parts)
Description	Short description of the part and its function.
Composition Editor view	Screen capture of the Composition Editor with labels or numbers to explain the connections
Part assembly	Picture showing the part's assembly structure. In the assembly structure, we show only the parts we implemented for the sample application, not all VisualAge standard parts that are also used as subparts
Public interface	Picture showing the part's public interface with the action, attribute, and event definitions
Used in part	Reference where this part is used to build another part
Superclass of	Picture showing where the class is inherited as a superclass.
Class definition	Smalltalk class definition with superclass, instance variables, class variables, and pool dictionaries
Scripts	Table with methods implemented in Smalltalk with a short description, followed by the code listings
Event trace	Table with user actions and the sequence of connections that are triggered from the user action
Special comments	Hints and tips for the part implementation and background information and reasons to explain why a specific solution was chosen.

During the walkthrough and the structured description of the parts, we also give some **how to** information that you can use as coding examples.

Before we go to the detailed description of each part we want to give some overview information about the parts of our application. When we open VisualAge's application browser for the sample application we see that the application is built from 23 different parts (see Figure 119 on page 116).

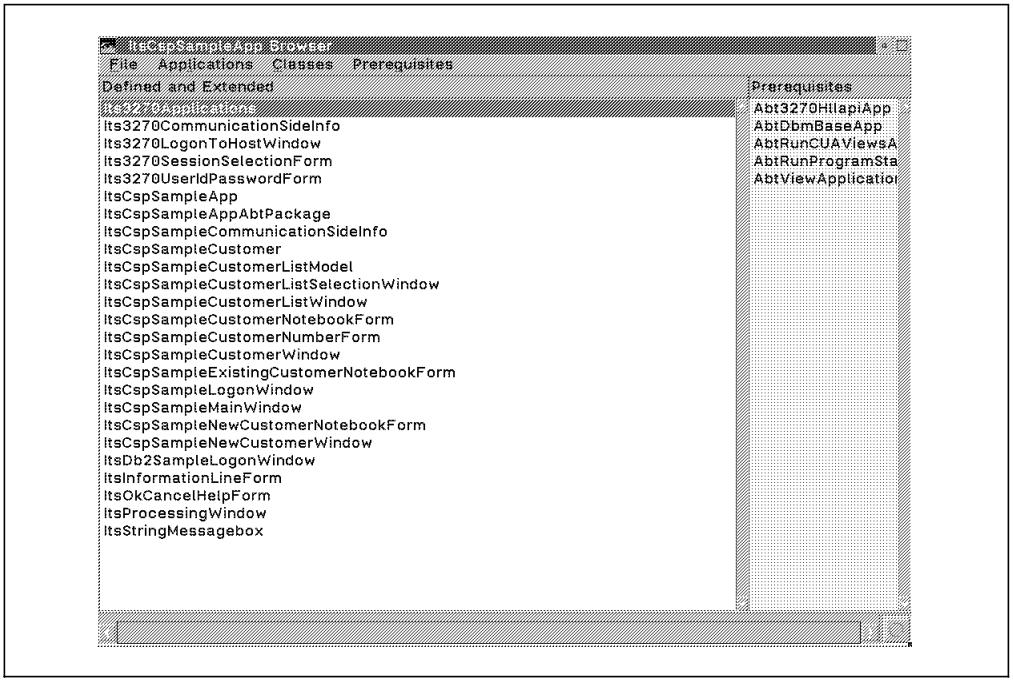

Figure 119. Application Browser for the CSP Sample Application

The ItsCspSampleApp and ItsCspSampleAppAbtPackage parts were generated automatically by VisualAge when we created the ItsCspSampleApp application. In the Prerequisites list box on the right-hand side of the application browser we can see the list of prerequisite applications required to run our application. Our application uses parts or methods from all of these applications, which are part of the VisualAge product.

Another way of looking at the parts of our sample application is from an inheritance point of view. Figure 120 on page 117 gives an overview of the Smalltalk class hierarchy for our CSP sample application.

Figure 120. Class Hierarchy for the CSP Sample Application

The parts inherited from the AbtAppBldrView class are the visual parts of our application. The parts inherited directly from the AbtAppBldrPart class are the nonvisual parts of our application. Figure 120 shows the result of the implementation of the model-view separation in the class hierarchy.

If we also distinguish between **abstract parts**, which are used only for inheritance, **basic parts**, which are the basic building blocks for other parts, and **composite parts**, which are built from basic parts, we can distinguish the following categories of parts implemented for the sample application:

1. **Visual parts**
   - Abstract parts
     - ItsCspSampleCustomerNotebookForm
     - Its3270LogonToHostWindow
   - Basic parts
     - Its3270SessionSelectionForm
     - Its3270UserIdPasswordForm
     - ItsCspSampleCustomerNumberForm

- ItsInformationLineForm
- ItsOkCancelHelpForm
- ItsProcessingWindow

- Composite parts
  - Its3270Applications
  - ItsCspSampleMainWindow
  - ItsCspSampleCustomerListSelectionWindow
  - ItsCspSampleCustomerListWindow
  - ItsCspSampleNewCustomerNotebookForm
  - ItsCspSampleExistingCustomerNotebookForm
  - ItsCspSampleCustomerWindow
  - ItsCspSampleNewCustomerWindow
  - ItsCspSampleLogonWindow
  - ItsDb2SampleLogonWindow

2. **Nonvisual parts**
   - Abstract parts
     - Its3270CommunicationSideInfo
   - Basic parts
     - ItsStringMessagebox
     - ItsCspSampleCommunicationSideInfo
   - Composite parts
     - ItsCspSampleCustomerListModel
     - ItsCspSampleCustomer

8.1 3270 Applications Window

The Its3270Applications part contains the Host Applications window that comes up when the application is started. The Host Applications window is just a folder that provides all of the 3270 applications for the user. It is a kind of master application from which the specific business applications can be started.

- **Part name**

  Its3270Applications

- **Category**

  Visual composite part

- **Description**

  This part has a primary window with a container of icons for each 3270 host application. For each application we add the part to log on to the host and the part for the main window of the application. This is what you see in the Composition Editor view. There is a logon window for the CSP sample application and its main window and a logon window for the DB2 sample application. There is no main window for the DB2 sample application because the GUI for this application is not implemented.

- **Composition Editor view**

  Figure 121 on page 119 shows the Composition Editor view for the 3270 Applications window.

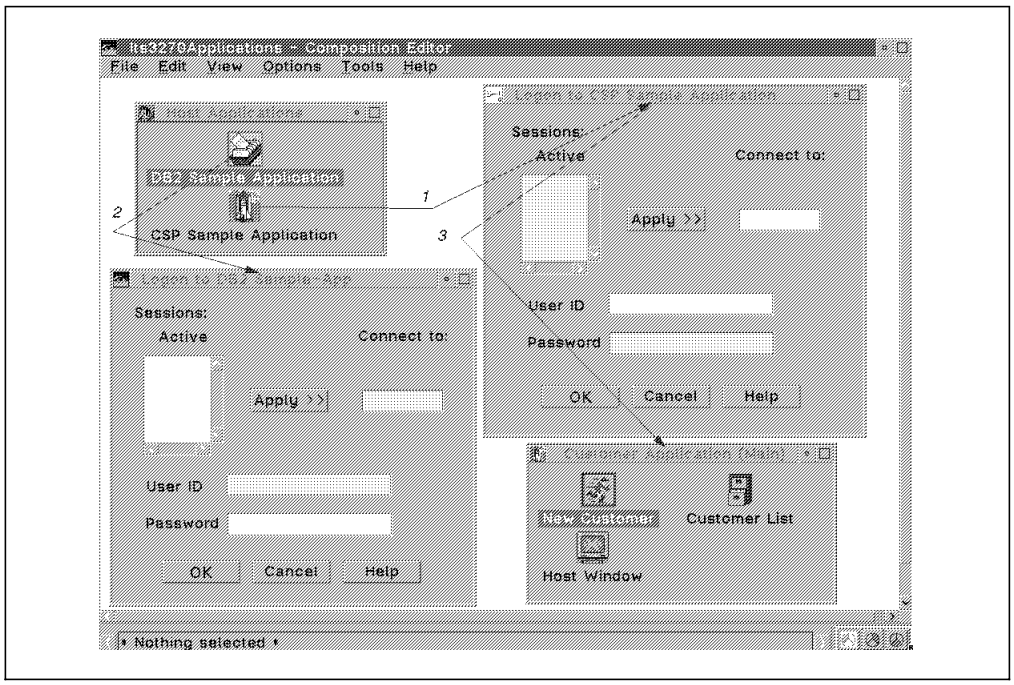

Figure 121. Composition Editor View: Its3270Applications

Note: The numbers next to the visual connections are for documentation purposes and correspond to the event numbers in the event trace in Table 7 on page 120.

- **Part assembly**

  Figure 122 shows the part assembly for the 3270 Applications window.

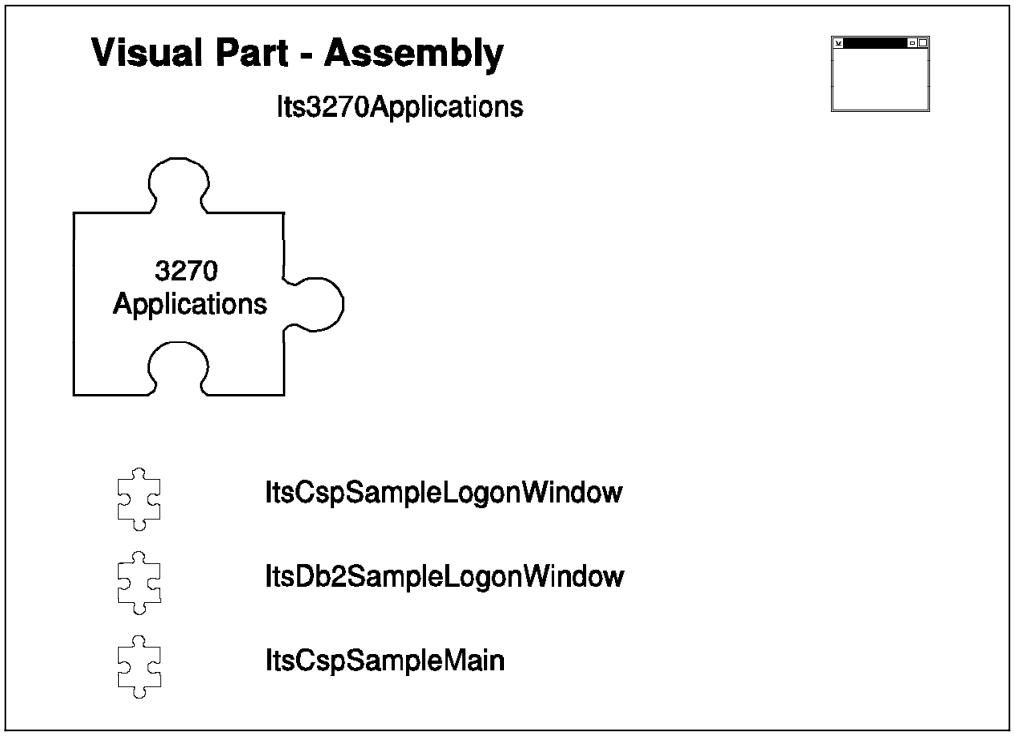

Figure 122. Part Assembly: Its3270Applications

- **Public interface**

None (this is the master window and therefore it is not used in other parts)

- **Used in part**

  None

- **Superclass of**

  None

- **Class definition**

  Figure 123 shows the class definition for the 3270 Applications window.

```
AbtAppBldrView subclass: #Its3270Applications
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
```

Figure 123. Class Definition: Its3270Applications

- **Scripts**

  None

- **Event trace**

  Table 7 shows the event trace for the 3270 Applications window.

Table 7. Event Trace: 3270 Applications Window

User Action	Sequence of Executed Connections
Double-click on CSP sample application icon	(1) defaultAction >> openWidget (CSP Sample Logon Window)
Double-click on DB2 sample application icon	(2) defaultAction >> openWidget (DB2 Sample Logon Window)
Logon dialog finished with OK, connected to host sessionId	(3) event: #SessionEstablished >> openWidget (CSP Sample Main Window)

- **Special comments**

  Providing a container window with all host applications in fact is comparable to the workplace shell folder concept. We opted to provide a container window because it was easier for us to deliver the sample application and show how to present the 3270 applications to the end user.

  Recommendation

  To group several GUI applications for existing host applications for the end user we would create a workplace shell folder for all packaged GUI applications. This solution looks the same as our implementation, but it is easier to add additional applications because there is no hardcoding in the 3270 application container window.

  To implement this approach for the sample application we would have to create separate applications for the CSP sample application and the DB2 sample application and implement the logon windows for the applications as primary parts. The logon windows would open the main window of the respective application after the logon request was successful.

8.2 3270 Logon to Host Window

After an application is selected, the user selects the session ID and performs the logon to the host. Because this procedure is the same for all host applications, we decided to implement an abstract class that provides the basic functions to select a session ID and enter a userid and password.

- **Part name**

  Its3270LogonToHostWindow

- **Category**

  Abstract, composite visual part

- **Description**

  This part is an abstract implementation of a generic logon dialog for an EHLLAPI application. It is used for inheritance. The way to use this part is to produce a logon dialog subpart for each EHLLAPI application. Creating a logon dialog subpart for each EHLLAPI application enables us to have each application connected to a different 3270 session. The logon dialog subpart has to store the selected session ID for the application in its communication side information (SideInfo) part.

- **Composition Editor view**

  Figure 124 shows the Composition Editor view for the Logon to Host window.

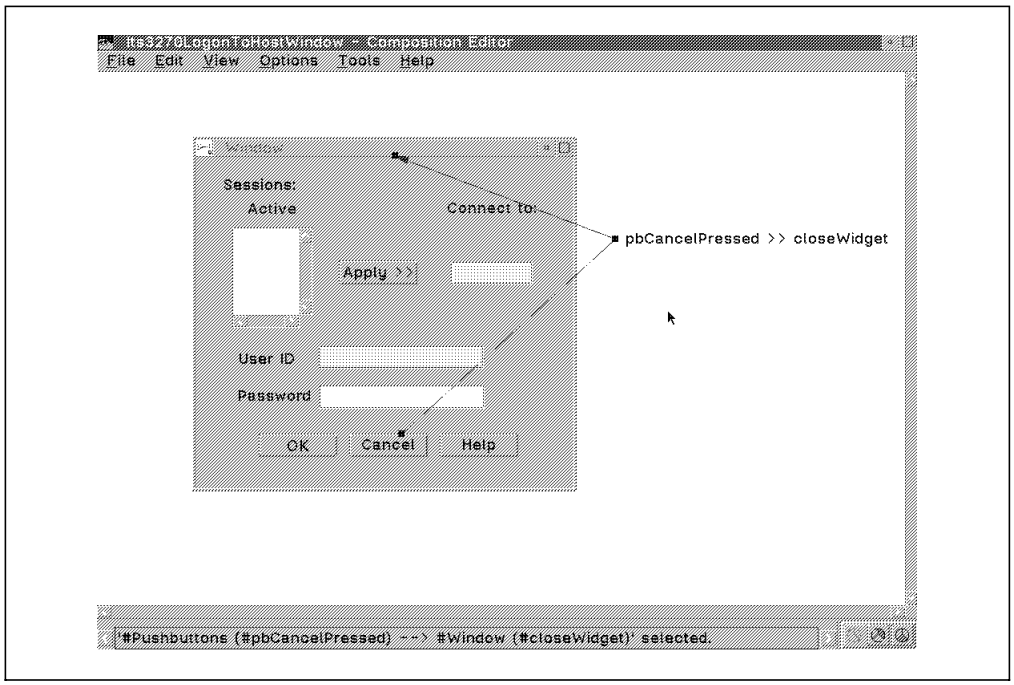

Figure 124. Composition Editor View: Its3270LogonToHostWindow

Note: We added comments to the free form surface to document the visual connections. This is a nice and easy way to document VisualAge applications but has the following drawbacks:

- Generates extra Smalltalk code, which could affect performance
- The comment positions are lost when you file-out/file-in your application.

- **Part assembly**

  Figure 125 shows the part assembly for the Logon to Host window.

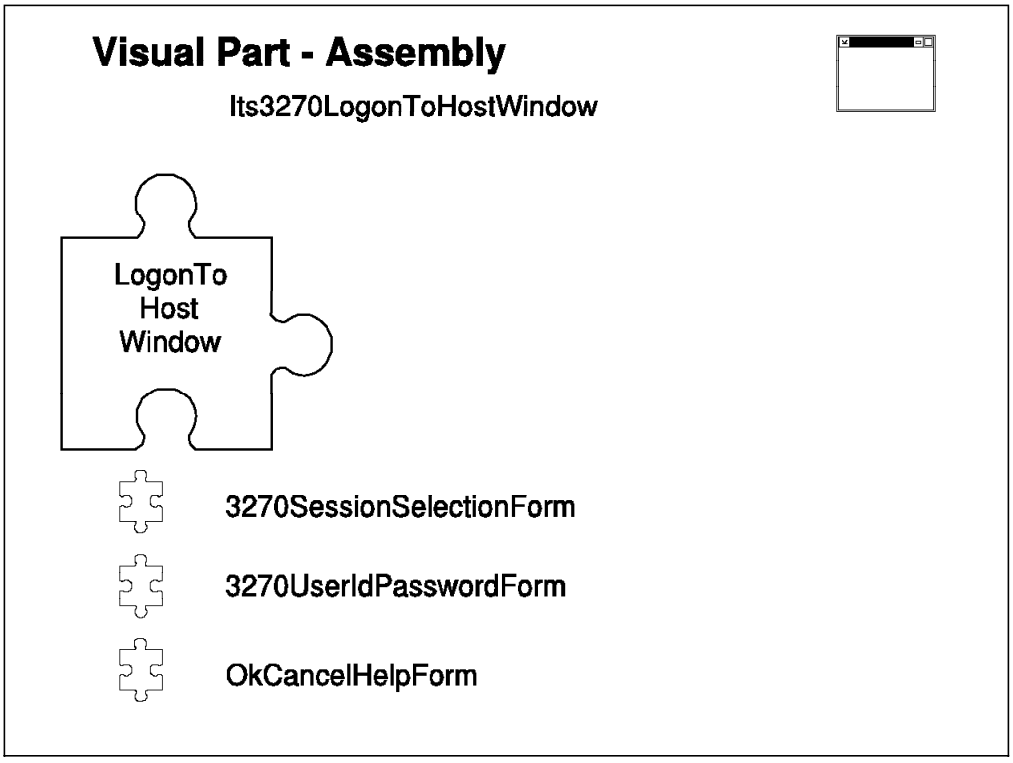

Figure 125. Part Assembly: Its3270LogonToHostWindow

- **Public interface**

  None

- **Used in part**

  None

- **Superclass of**

  Figure 126 on page 123 shows the inheritance hierarchy for the Logon to Host window.

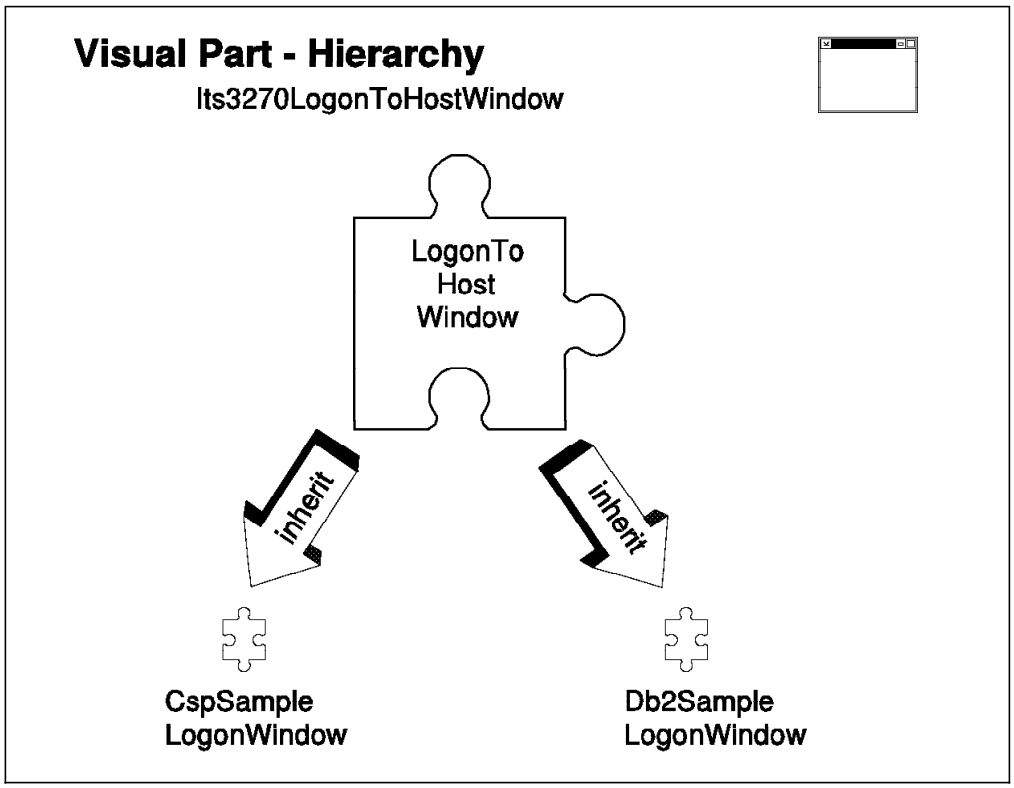

Figure 126. Inheritance Hierarchy: Its3270LogonToHostWindow

- **Class definition**

  Figure 127 shows the class definition for the Logon to Host window.

```
AbtAppBldrView subclass: #Its3270LogonToHostWindow
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
```

Figure 127. Class Definition: Its3270LogonToHostWindow

- **Scripts**

  None

- **Event trace**

  Table 8 shows the event trace for the Logon to Host window.

Table 8. Event Trace: Logon to Host Window

User Action	Sequence of Executed Connections
Click on Cancel push button	pbCancelClicked >> closeWidget

- **Special comments**

  In this implementation, our abstract part mainly provides the view for the logon window. Not much logic is implemented except to close the window when the Cancel push button is clicked on.

In a next iteration, we would probably implement a method to validate whether a session ID is selected or not and to signal an event when a session ID is selected. Now these functions are implemented in the CSP sample logon window (method: validateSession, event: SessionValidated).

There is no advantage, however, to provide attributes in the public interface of this abstract part. Because this part is built from other parts, we would have to put the attributes of the subparts, for example, the session ID, in the public interface. This is just overhead, and it is easier to directly access the public interface of the subparts.

8.3 3270 Session Selection Form

This part is used as a subpart in the Logon to Host Window (Figure 124 on page 121).

- **Part name**

  Its3270SessionSelectionForm

- **Category**

  Basic visual part

- **Description**

  This part provides the view and the logic to select a session for EHLLAPI communication. It provides a list of active sessions from which users can select a session. This generic part can be used in every logon or option dialog for 3270 communication.

- **Composition Editor view**

  Figure 128 shows the Composition Editor view for the 3270 Session Selection Form.

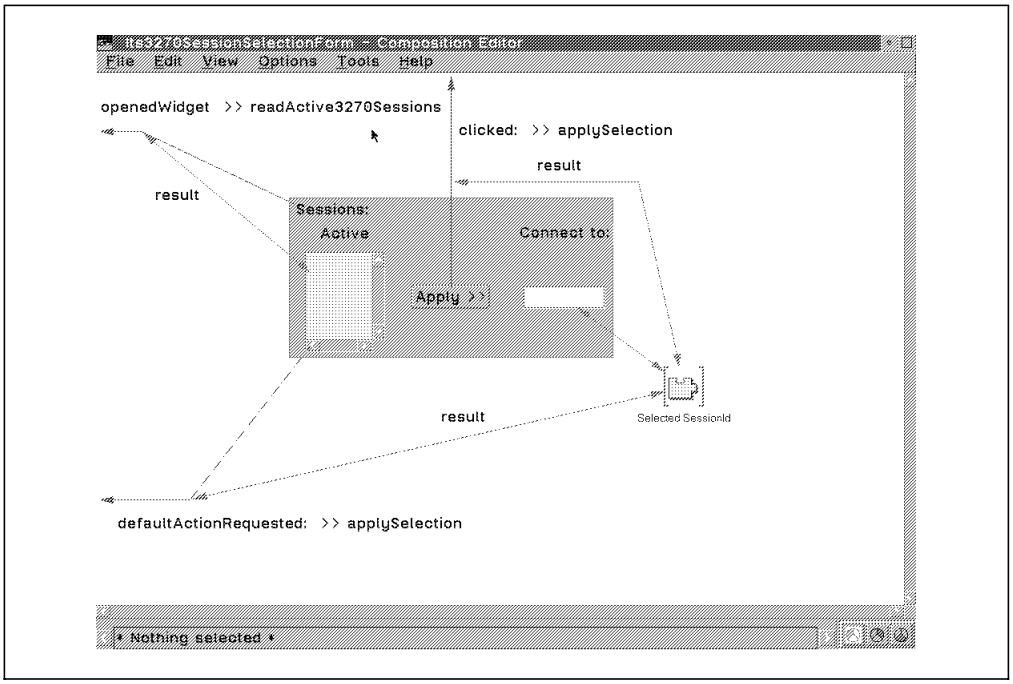

Figure 128. Composition Editor View: Its3270SessionSelectionForm

- **Part assembly**

None

- **Public interface**

  Figure 129 shows the public interface for the 3270 Session Selection Form.

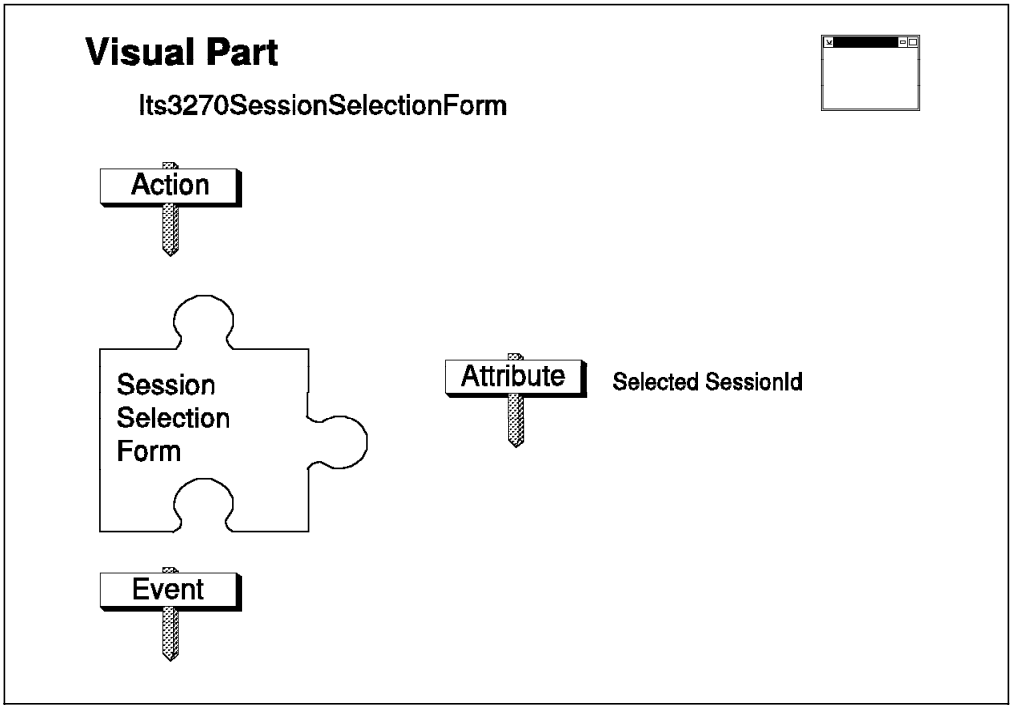

Figure 129. Public Interface: Its3270SessionSelectionForm

The Selected SessionId variable from the Composition Editor is added to the public interface.

- **Used in part**

  Its3270LogonToHostWindow

- **Superclass of**

  None

- **Class definition**

  Figure 130 shows the class definition for the 3270 Session Selection Form.

```
AbtAppBldrView subclass: #Its3270SessionSelectionForm
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
```

Figure 130. Class Definition: Its3270SessionSelectionForm

- **Scripts**

  Table 9 on page 126 shows the scripts for the 3270 Session Selection Form.

Table 9. Scripts: Session Selection Form	
Method	**Description**
readActive3270Sessions	Reads all active sessions and returns a sorted collection of session ids.
applySelection	Returns the value of the selected item (session ID) in the list box.

Figure 131 shows the readActive3270Sessions method.

```
readActive3270Sessions

   | sessions aDict |

   sessions := Abt3270Terminal allSessions select: [ :e |
               ((e at: #qsstSestype) = $D) | ((e at: #qsstSestype) = $F) ].

   aDict := Dictionary new.
   sessions do: [ :each | aDict at: (each at: #qsstShortname)
                              put: ((each at: #qsstShortname) asCharacter)].

   ^aDict asSortedCollection.
```

Figure 131. Method: readActive3270Sessions

Figure 132 shows the applySelection method.

```
applySelection

   ^self partAttributeValue: #(#List #selectedItem).
```

Figure 132. Method: applySelection

- **Event trace**

  Table 10 shows the event trace for the 3270 Session Selection Form.

Table 10. Event Trace: Session Selection Form	
User Action or Event	**Sequence of Executed Connections**
Event: openedWidget	openedWidget >> readActive3270Session (hook), put result in the list box
User action: double-click on list box item	defaultActionRequested >> applySelection (hook), put result in public interface variable and in entry field
User action: click on Apply push button	clicked >> applySelection (hook), put result in public interface variable and in entry field

- **Special comments**

  The prerequisite for any EHLLAPI application is a started 3270 session. There is currently a problem in VisualAge's EHLLAPI support. The method of asking whether EHLLAPI support is available does not work correctly when the Communications Manager is stopped after the EHLLAPI application is

started. The problem is in the isHllapiAvailable method of the Abt3270Hllapi class.

How to Read All Active 3270 Sessions

To find out which 3270 sessions are active at run time, the following code can be used:

```
| sessions aDict |

aDict := Dictionary new.

sessions := Abt3270Terminal allSessions select: [ :e |
     ((e at: #qsstSestype) = $D) | ((e at: #qsstSestype) = $F) ].

sessions do: [ :each | aDict at: (each at: #qsstShortname)
                     put: ((each at: #qsstShortname) asCharacter)].
^aDict asSortedCollection.
```

8.4 UserID Password Form

This part is also one of the subparts used in the Logon to Host window (Figure 124 on page 121).

- **Part name**

  Its3270UserIdPasswordForm

- **Category**

  Basic visual part

- **Description**

  This is a part with two entry fields for userid and password. The part is developed as a form that can be used as a subpart in a window.

  We had no security installed on our CICS system and therefore did not implement a userid and password check in our application. The entry field for the password does not hide the user input. Because there is no option to hide input available for an entry field, the code to hide user input must be provided in a script.

  For use in a customer application, this part must be enhanced with the function to check the userid and password that are administered in a security system, for example, in the user profile management system on the PWS.

- **Composition Editor view**

  Figure 133 on page 128 shows the Composition Editor view for the UserID Password Form.

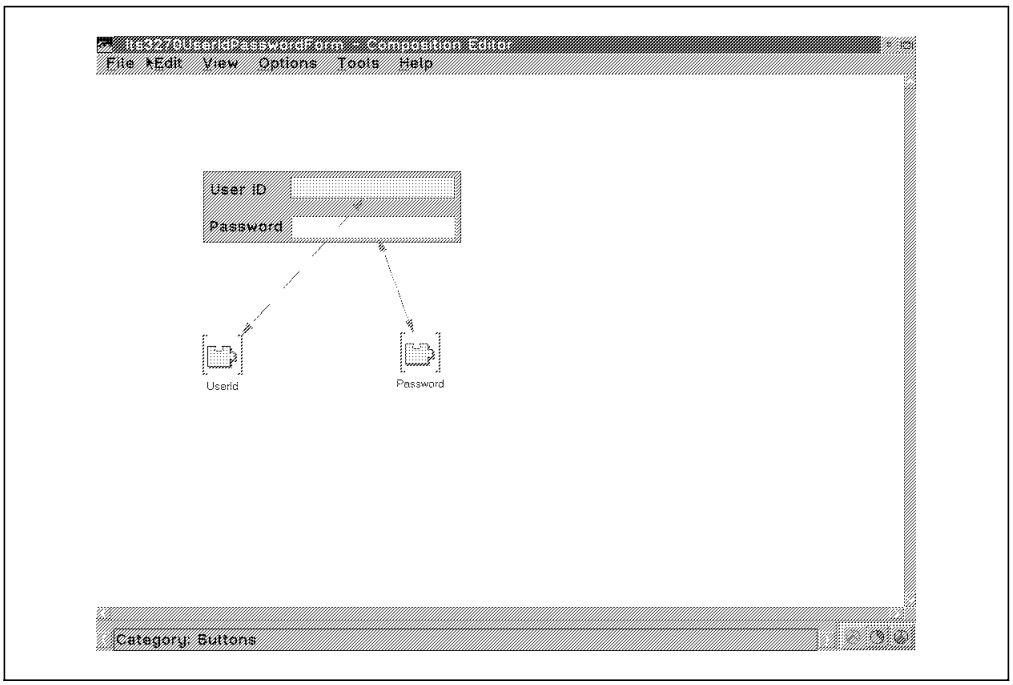

Figure 133. Public Interface: Its3270UserIdPasswordForm

The two variables UserId and Password in the Composition Editor are added to the public interface.

- **Part assembly**

  None

- **Public interface**

  Figure 134 on page 129 shows the public interface for the UserID Password Form.

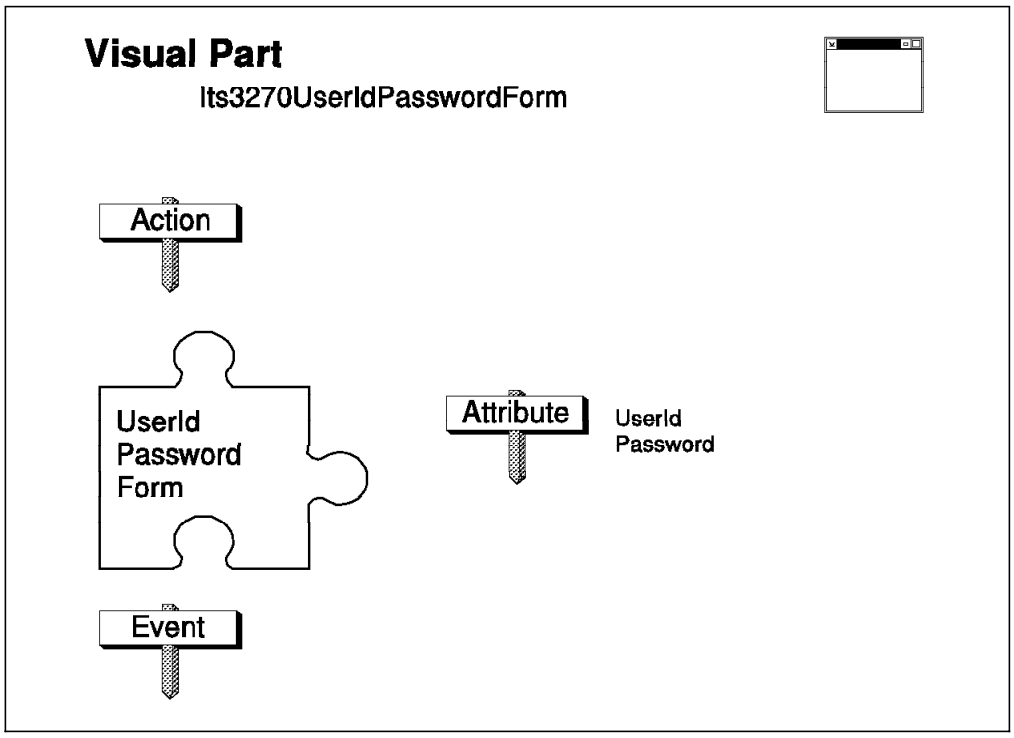

Figure 134. Public Interface: Its3270UserIdPasswordForm

- **Used in part**

  Its3270LogonToHostWindow

- **Superclass of**

  None

- **Class definition**

  Figure 134 shows the class definition for the UserID Password Form.

```
AbtAppBldrView subclass: #Its3270UserIdPasswordForm
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
```

Figure 135. Class Definition: Its3270UserIdPasswordForm

- **Scripts**

  None

- **Event trace**

  None

8.5 OkCancelHelp Form

This is a basic part with three standard CUA push buttons to be used in forms or windows.

- **Part name**

  ItsOkCancelHelpForm

- **Category**

  Basic visual part

- **Description**

  This part has the three push buttons—OK, Cancel, and Help—as proposed by the CUA 91 standard for windows and dialogs. This part shows how an enterprise GUI standard can be implemented by building parts. The approach to implementing parts for the standardized components in windows is powerful and makes the implementation of a standard easy. The developer can either use the standard part as is or extend the part using the standard part as a superclass to inherit all standard behaviors.

- **Composition Editor view**

  Figure 136 shows the Composition Editor view for the OkCancelHelpForm.

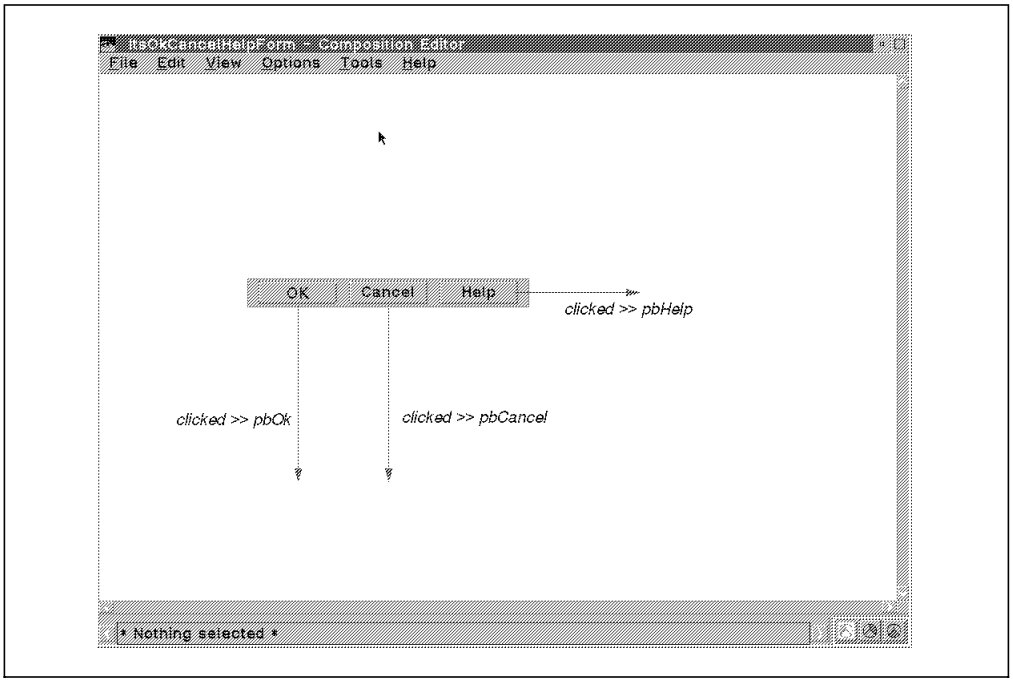

Figure 136. Composition Editor View: ItsOkCancelHelpForm

The Composition Editor shows three event-to-script connections.

- **Part assembly**

  None

- **Public interface**

  Figure 137 on page 131 shows the public interface for the OkCancelHelpForm.

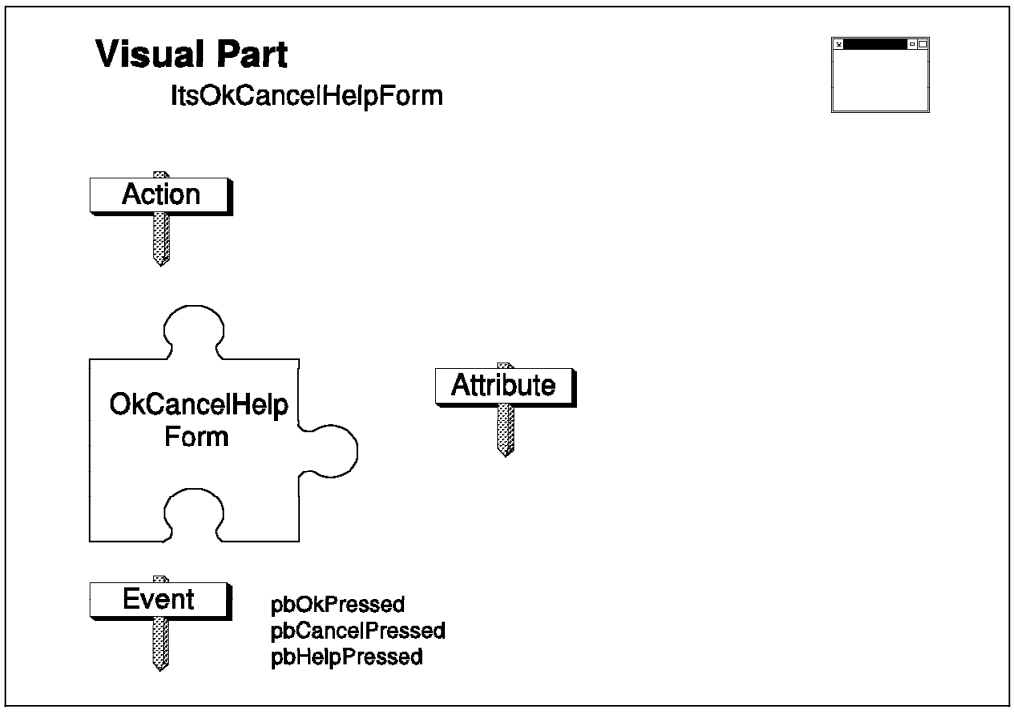

Figure 137. Public Interface: ItsOkCancelHelpForm

- **Used in part**
  - Its3270LogonToHostWindow
  - ItsCspSampleCustomerListSelectionWindow
  - ItsCspSampleCustomerListWindow
- **Superclass of**

  None
- **Class definition**

  Figure 138 shows the class definition for the OkCancelHelpForm.

```
AbtAppBldrView subclass: #ItsOkCancelHelpForm
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
```

Figure 138. Class Definition: ItsOkCancelHelpForm

- **Scripts**

  Table 11 shows the scripts for the OkCancelHelpForm.

Table 11. Scripts: OkCancelHelpForm

Method	Description
pbOk	Signals the pbOkPressed event
pbCancel	Signals the pbCancelPressed event
pbHelp	Signals the pbHelpPressed event

Figure 139 on page 132 shows the pbOk method.

```
pbOk

   ^self signalEvent: #pbOkPressed.
```

Figure 139. Method: pbOk

Figure 140 shows the pbCancel method.

```
pbCancel

   ^self signalEvent: #pbCancelPressed.
```

Figure 140. Method: pbCancel

Figure 141 shows the pbHelp method.

```
pbHelp

   ^self signalEvent: #pbHelpPressed.
```

Figure 141. Method: pbHelp

- **Event trace**

  Table 12 shows the event trace for the OkCancelHelpForm.

Table 12. Event Trace: OkCancelHelp Form

User Action	Sequence of Executed Connections
Click on OK push button	clicked >> pbOk (hook)
Click on Cancel push button	clicked >> pbCancel (hook)
Click on Help push button	clicked >> pbHelp (hook)

- **Special comments**

  Another way to implement an enterprise GUI standard is to build visual parts with a standard look and feel and use them as templates. Subparts that inherit from the template part can extend the standard part to add additional controls. Inheritance extends to the view, logic, and public interface of a part.

8.6 CSP Sample Logon Window

This part is the logon window for the CSP sample application. The part is implemented as a subpart of the abstract Logon to Host window part. This part is an example of a part that inherits its behavior from another part.

- **Part name**

  ItsCspSampleLogonWindow

- **Category**

  Composite visual part

- **Description**

  This is the only visual part in the sample application where a VisualAge communication part is used directly from the window. This is an exception to the principle of model-view separation, but in the case of an EHLLAPI logon window it is not essential to implement the window communication protocol independently.

  The logon window is specific for an EHLLAPI application because it asks for the session ID. Therefore, it cannot be used as a generic logon window for other communication protocols.

  This part contains the first window after a user starts the CSP sample application from the 3270 applications container window. It asks the user for the session ID, the userid, and password and stores the session ID as a global value for the entire application in the communication side information (SideInfo) part.

  While the logon and the initialization process for the communication side information part are in progress, the host processing window, another subpart, is shown to give the user feedback.

  Our application does not lock the push buttons while host communication is processing. A second logon attempt can cause problems. In a real customer application the push buttons should be disabled during host communication to serialize user interaction with the host.

- **Composition Editor view**

  Figure 142 shows the Composition Editor view for the CSP Sample Logon window.

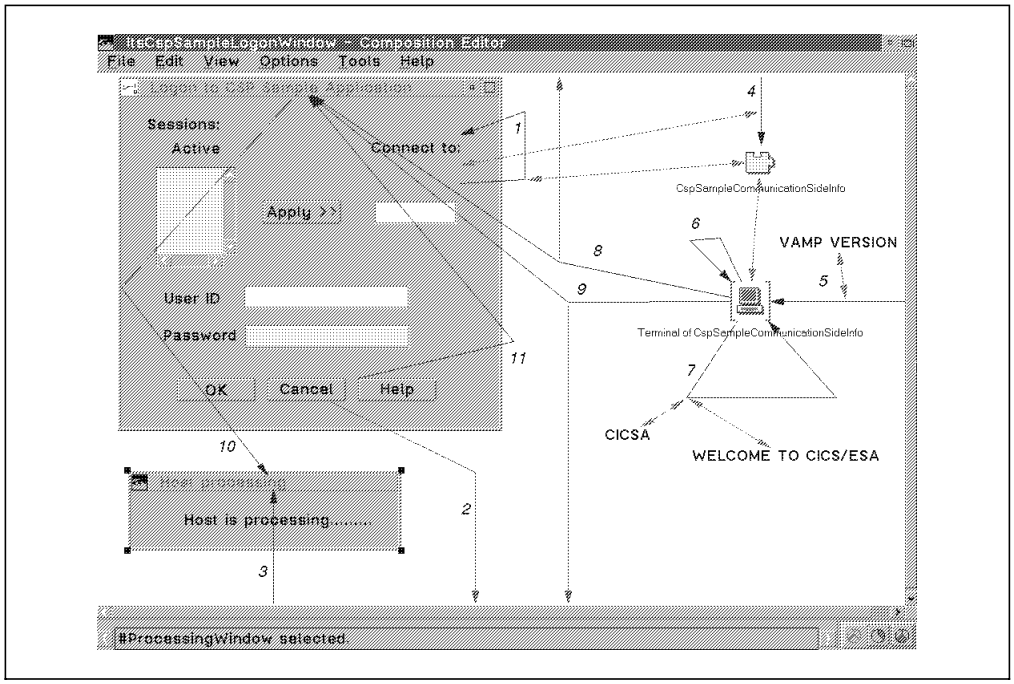

Figure 142. Composition Editor View: ItsCspSampleLogonWindow

- **Part assembly**

  Figure 143 on page 134 shows the part assembly for the CSP Sample Logon window.

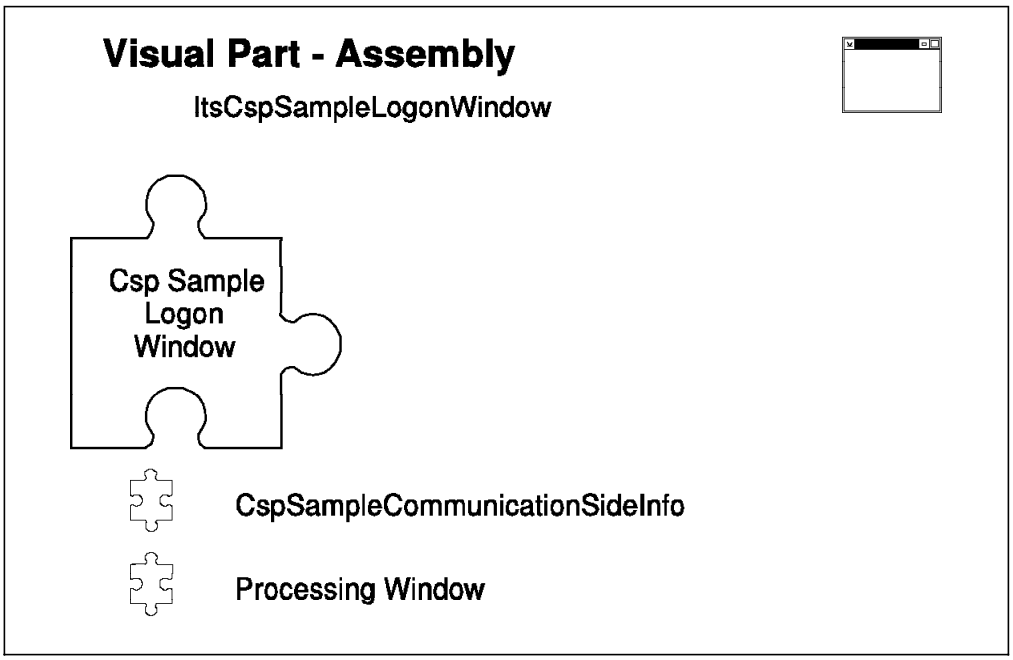

Figure 143. Part Assembly: ItsCspSampleLogonWindow

- **Public interface**

  Figure 144 shows the public interface for the CSP Sample Logon window.

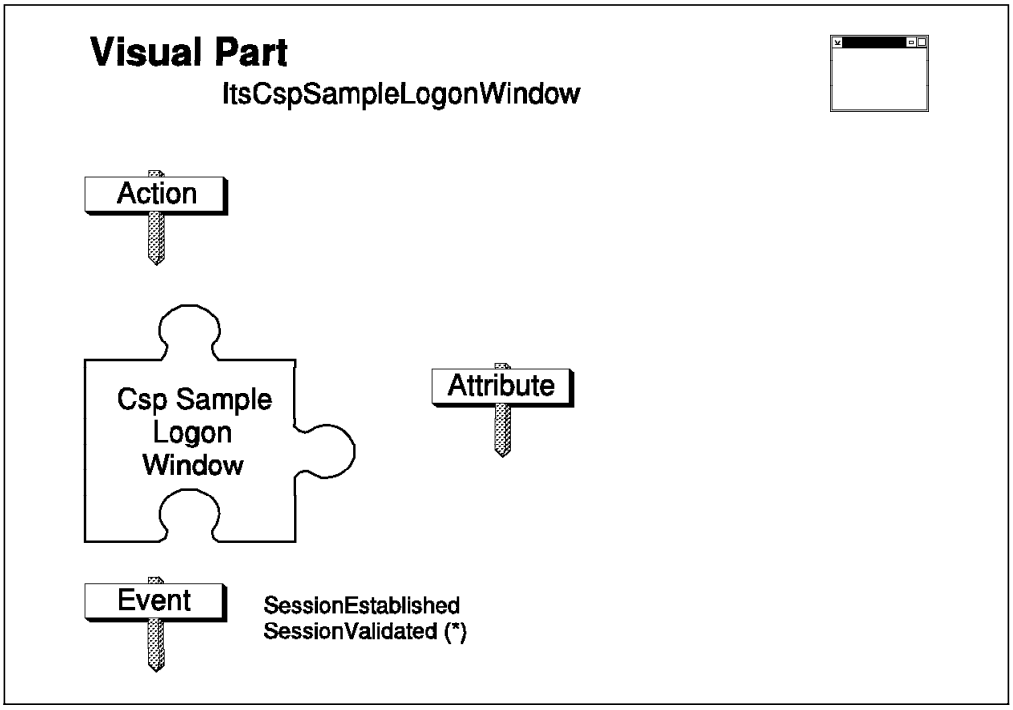

Figure 144. Public Interface: ItsCspSampleLogonWindow

Note: The SessionValidated (*) event is only used within this part to draw event-to-action connections in the Composition Editor.

- **Used in part**

  None

- **Superclass of**

None

- **Class definition**

  Figure 145 shows the class definition for the CSP Sample Logon window.

```
Its327OLogonToHostWindow subclass: #ItsCspSampleLogonWindow
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
```

Figure 145. Class Definition: ItsCspSampleLogonWindow

- **Scripts**

  Table 13 shows the scripts for the CSP Sample Logon window.

Table 13. Scripts: CSP Sample Logon Window

Method	Description
validateSession	Validates whether a session ID is selected. If not, a message box is shown telling the user to select a session ID. If a session ID is selected, the internal SessionValidated event is raised to trigger the logon to CICS processing.
sessionEstablished	Raises the public SessionEstablished event after the host communication session is established.

Figure 146 shows the validateSession method.

```
validateSession

   | selectedSession |

   selectedSession :=
      (self partAttributeValue: #(#SelectSessionId #'Selected SessionId')).

   (selectedSession isNil) ifTrue:
      [CwMessagePrompter
         errorMessage: 'No Session selected !'.
         ^self.
      ].

   ^self signalEvent: #SessionValidated.
```

Figure 146. Method: validateSession

Figure 147 on page 136 shows the sessionEstablished method.

```
sessionEstablished

   ^self signalEvent: #SessionEstablished.
```

Figure 147. Method: sessionEstablished

- **Event trace**

Table 14 shows the event trace for the CSP Sample Logon window.

Table 14. Event Trace: CSP Sample Logon Window

User Action or Event	Sequence of Executed Connections
Event: aboutToOpenWidget (initialization)	(1) aboutToOpenWidget >> action: read the selected session ID (the default) and pass the result to the SideInfo part.
User action: click on OK push button (pbOkPressed event)	(2) pbOkPressed >> validateSession (hook) and raise the #SessionValidated event if the selection is valid (3) event: #SessionValidated >> openWidget processing window (4) event: #SessionValidated >> initializeTerminalWithSessionId (SideInfo) (5) event: #SessionValidated >> findString: 'VAMP VERSION' (6) event: #SearchSuccessful >> keyClear (7) event: #SearchSuccessful >> enter: 'CICS' AndWaitForString: 'Welcome to CICS/ESA' (8) event: #SearchSuccessful >> closeWidget logon window (9) event: #SearchFailed >> closeWidget logon window (10) event: aboutToCloseWidget >> closeWidget processing window
User action: click on Cancel push button (pbCancelPressed event)	(11) pbCancelPressed >> closeWidget logon window

- **Special comments**

  The logic to implement the logon to CICS from the VAMP screen is done through visual programming. The terminal used is torn off from the communication side information part and already knows its session ID. The logic for the CICS logon is as follows:

  1. Find string 'VAMP VERSION'
  2. Enter 'CICSA' and wait for the CICS good morning message screen.

  The enter:andWaitForString: method of the Abt3270Terminal part is key for this EHLLAPI dialog, because we have to wait until the CICS good morning message is shown before we enter the next host command. If we do not wait and just enter the next host command, we are too fast and the synchronization with the host application is lost.

How to Show an Error Message Box

The following code can be used to show an error message box from a script:

```
CwMessagePrompter errorMessage: 'This is the errortext'.
```

8.7 Processing Window

The window in this part is always shown when the GUI application talks to the host.

- **Part name**

  ItsProcessingWindow

- **Category**

  Basic visual part

- **Description**

  This part is a window used to give the user feedback when the host is processing. It is added as a part in other visual parts, shown when the host is active, and hidden when communication to the host application has finished.

- **Composition Editor view**

  Figure 148 shows the Composition Editor view for the Processing window.

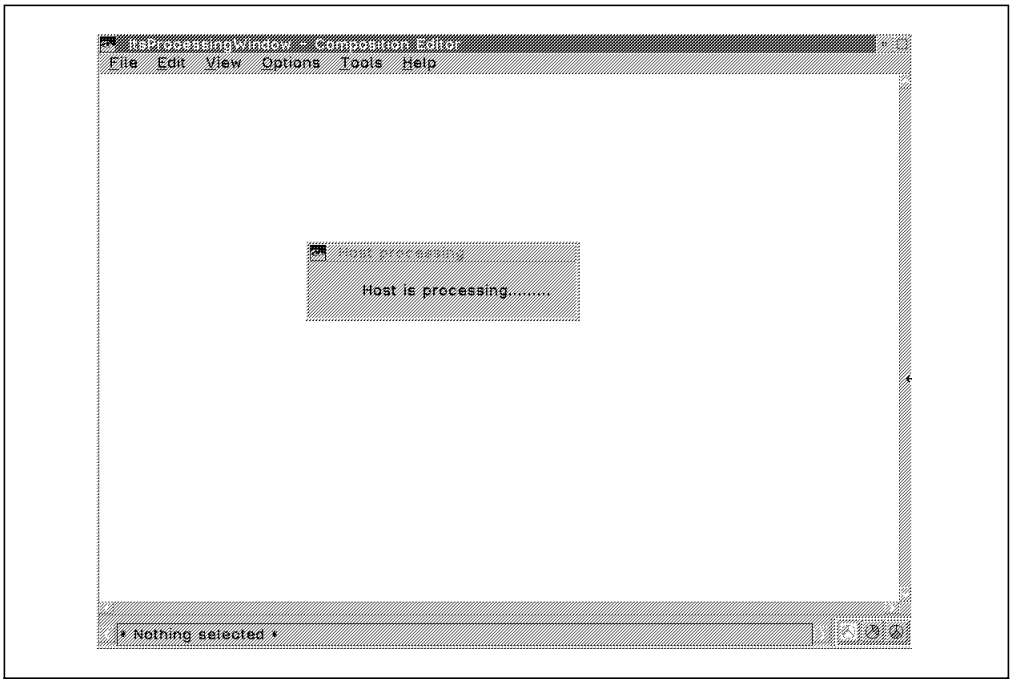

Figure 148. Composition Editor View: ItsProcessingWindow

- **Part assembly**

  None

- **Public interface**

  None

- **Used in part**

  – ItsCspSampleLogonWindow
  – ItsCspSampleCustomerListWindow
  – ItsCspSampleCustomerWindow
  – ItsCspSampleNewCustomerWindow

- **Superclass of**

  None

- **Class definition**

  Figure 149 shows the class definition for the Processing window.

```
AbtAppBldrView subclass: #ItsProcessingWindow
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
```

Figure 149. Class Definition: ItsProcessingWindow

- **Scripts**

  None

- **Event trace**

  None

- **Special comments**

  As a general recommendation for a client/server application, the client should start requests to the server in background whenever possible. An OS/2 client can use the multitasking features of the operating system and start a separate thread for each request to the server, for example.

  The distributed presentation model and the use of EHLLAPI as the communication protocol have some restrictions. The main restriction is the number of parallel sessions between the client and the server. Compared to other communication protocols, for example, APPC, which can handle several parallel sessions between a client and a server, this EHLLAPI restriction was the bottleneck in our client/server application.

  The requests had to be serialized, and we had to work step-by-step with the EHLLAPI application without using host application functions in parallel.

  Our recommendation is to give the user visual feedback whenever the host application is working in the background. The processing window is like a GUI implementation of the system sign in the operator interaction area of the emulation window. In addition, while the EHLLAPI session is busy, the controls, such as push buttons and menus, from which the user can start requests to the server should be disabled for any user interaction.

  Implementing functions to prevent the user from starting a host request while the session is busy does not mean that the GUI application does not allow the user to work with another PWS application in parallel. Any request or task other than an EHLLAPI request is allowed during this time.

8.8 3270 Communication SideInfo

Following our design for the sample application, we need a communication part that handles all communication related aspects. This is a model part that provides routing information for the communication protocol, such as the session ID or a transaction code.

- **Part name**

  Its3270CommunicationSideInfo

- **Category**

  Nonvisual, abstract part

- **Description**

  This abstract part encapsulates general information about the host communication. In our case it contains general information related to the EHLLAPI protocol.

  The implementation provides a global Abt3270Terminal part and Abt3270Screen part that can be torn off in the Composition Editor. The torn-off terminal and screen always know their session ID. We defined the terminal and the screen as class variables to be able to use them like global variables.

  In addition to being stored in the global terminal and screen, the session ID is stored separately in a class variable. The class variable can be used in attribute-to-attribute connections to set the session ID of separate screen or terminal parts in the Composition Editor.

  To migrate our sample application to another communication protocol, we would either replace the information in the SideInfo part with specific information for the new communication protocol or replace the entire part with another similar part.

- **Composition Editor view**

  None

- **Part assembly**

  The part assembly is not shown in the Composition Editor for this part. This part contains an Abt3270Terminal part, an Abt3270Screen part, and a variable (session ID). The assembly of the SideInfo part can be seen in the tear-off selection of the part. This same assembly approach is used in the implementation of the Abt3270Screen part, which contains an Abt3270Terminal part as an instance variable.

- **Public interface**

  Figure 150 on page 140 shows the public interface for the 3270 Communication SideInfo part.

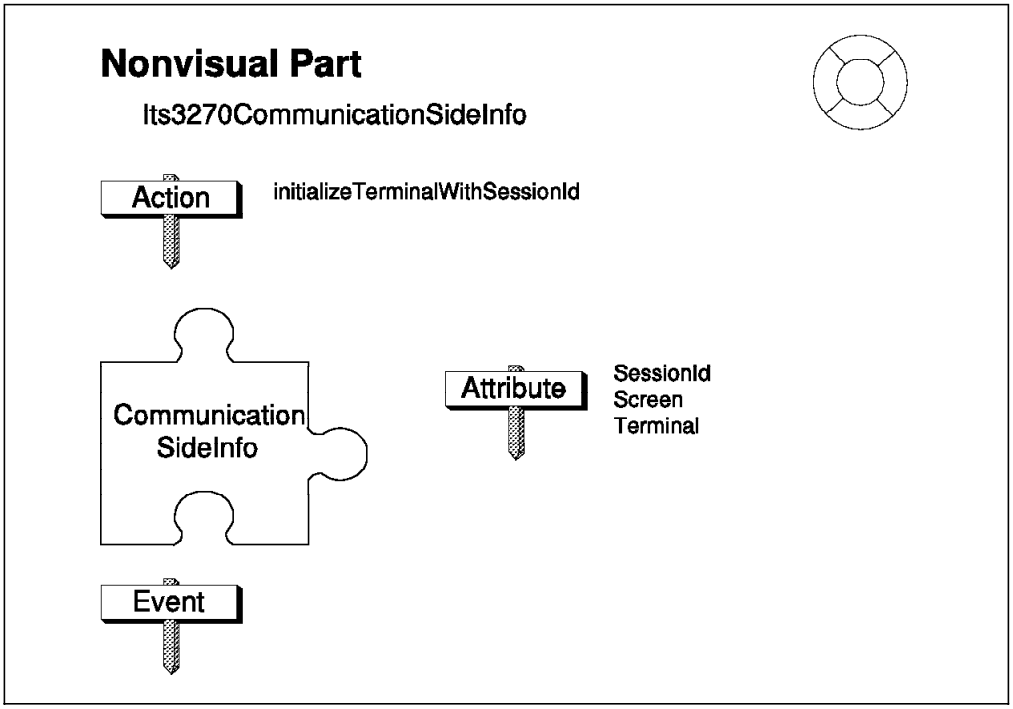

Figure 150. Public Interface: Its3270CommunicationSideInfo

- **Superclass of**

  Figure 151 shows the inheritance hierarchy for the 3270 Communication SideInfo part.

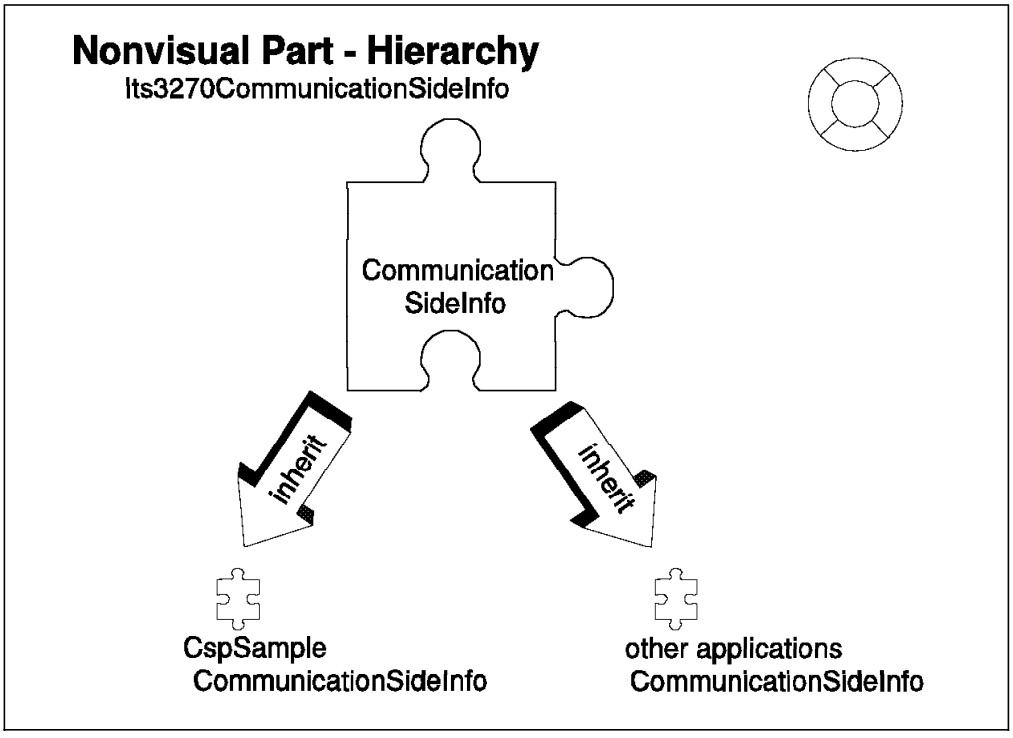

Figure 151. Inheritance Hierarchy: Its3270CommunicationSideInfo

- **Class definition**

Figure 152 shows the class definition for the 3270 Communication SideInfo part.

```
AbtAppBldrPart subclass: #Its3270CommunicationSideInfo
    instanceVariableNames: ''
    classVariableNames: 'Screen SessionId Terminal '
    poolDictionaries: ''
```

Figure 152. Class Definition: Its3270CommunicationSideInfo

- **Scripts**

  Table 15 shows the scripts for the 3270 Communication SideInfo part.

Table 15. Scripts: 3270 Communication SideInfo

Method	Description
initializeTerminal: aSessionIdCharOrString	Initializes the session ID screen and terminal variables with the character passed as parameter. Each time the 3270 Communication SideInfo part is added to another part in the Composition Editor, a new instance is created. To keep all 3270 Communication SideInfo instances in our application synchronized we must inform all instances when the session ID is changed or initialized. This is done by executing the *class basicAllInstances do:* method. This class method takes a while to complete.
isSessionIdChangedWith: aSessionIdCharOrString	Compares the old session ID with the new ID and returns the result of the comparison (true or false).
screen	Returns the value of the Screen class variable.
terminal	Returns the value of the Terminal class variable.
sessionId	Returns the value of the SessionId class variable.
sessionId: aSessionIdCharOrString	Saves the parameter value in the SessionId class variable.

Figure 153 shows the initializeTerminal method.

```
initializeTerminal: aSessionIdCharOrString

   Terminal := Abt3270Terminal new.
   Screen   := Abt3270Screen new.

   (self isSessionIdChangedWith: aSessionIdCharOrString) ifTrue:
   [self class basicAllInstances do:
      [ :each |
         each sessionId: aSessionIdCharOrString.
      ].
   ].

    self signalEvent: #Terminal.
    self signalEvent: #Screen.
```

Figure 153. Method: initializeTerminal

Figure 154 on page 142 shows the isSessionIdChangedWith: method.

```
isSessionIdChangedWith: aSessionIdCharOrString

   | firstCharacter |

   aSessionIdCharOrString isCharacter
   ifTrue: [firstCharacter := aSessionIdCharOrString]
   ifFalse: [firstCharacter := aSessionIdCharOrString first].

   ^(firstCharacter ~ = SessionId).
```

Figure 154. Method: isSessionIdChangedWith:

Figure 155 shows the screen method.

```
screen

   Screen isNil ifTrue:
      [Screen := Abt3270Screen new].

   ^Screen.
```

Figure 155. Method: screen

Figure 156 shows the sessionId method.

```
sessionId

   SessionId isNil ifTrue:
      [SessionId := $A.].          "default SessionId"

   ^SessionId.
```

Figure 156. Method: sessionId

Figure 157 shows the sessionId: method.

```
sessionId: aSessionIdCharOrString

   aSessionIdCharOrString isCharacter
   ifTrue: [SessionId := aSessionIdCharOrString]
   ifFalse: [SessionId := aSessionIdCharOrString first].

   (self terminal) shortSessionID: SessionId.
   (self screen)   shortSessionID: SessionId;
     settleTime: 1.

   self signalEvent: #SessionId
         with: aSessionIdCharOrString.

   self signalEvent: #Terminal.
   self signalEvent: #Screen.
```

Figure 157. Method: sessionId:

Figure 158 on page 143 shows the terminal method.

```
terminal

   Terminal isNil ifTrue:
      [Terminal := Abt3270Terminal new].

   ^Terminal.
```

Figure 158. Method: terminal

- **Event trace**

  None

- **Special comments**

  We used the tear-off possibility for the terminal part in most cases for our sample application. In some situations we connected the SessionId attribute of the SideInfo part to a separate Abt3270Terminal part to set its session ID.

  The initializeTerminal method is called from the logon window. When the session ID changes, all 3270 Communication Sideinfo part instances in our application must be informed of the change. In this way, the entire application works with the specified session ID. We provided A as the default session ID.

How to Use Class Variables in Parts

VisualAge provides a mapping between attributes in the public interface and instance variables of the Smalltalk class. Class variables are not directly connected to the public interface. We have to implement the mapping between a class variable and a public interface attribute with get and set methods. Also, if we define class variables as global variables for all instances of a class, we have to find a way to inform all instances when the class variable changes. This can be achieved as follows:

```
self class basicAllInstances do:
        [ :each |
                each sessionId: aSessionIdCharOrString.
        ].
```

8.9 Sample Communication SideInfo

This part is a specialization of the 3270 Communication SideInfo part. It is used in the CSP sample application to provide specific EHLLAPI-related information. Following the principle of model-view separation, this part is used as a subpart in several nonvisual parts of the sample application.

- **Part name**

  ItsCspSampleCommunicationSideInfo

- **Category**

  Nonvisual part

- **Description**

  This part contains side information for communication routing in a dictionary. In our case, it contains transaction codes for the different host transactions we front ended with our GUI application. The advantage is that this

communication routing information can be maintained centrally in this dictionary.

- **Composition Editor view**

  None

- **Part assembly**

  None

- **Public interface**

  Figure 159 shows the public interface for the Sample Communication SideInfo part.

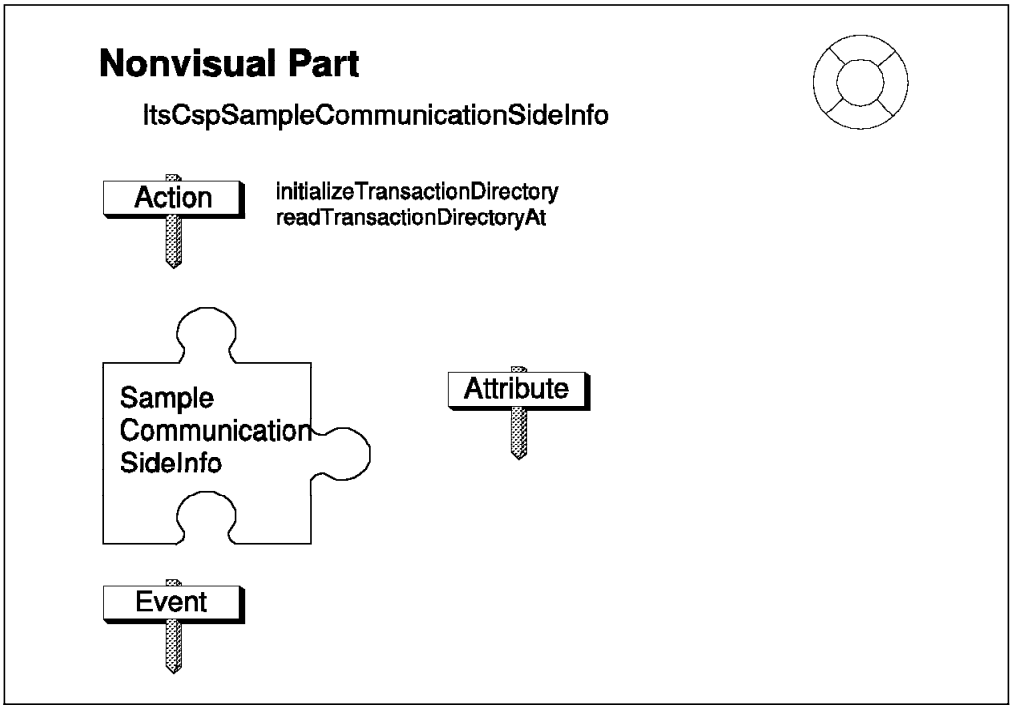

Figure 159. Public Interface: ItsCspSampleCommunicationSideInfo

- **Used in part**
  - ItsCspSampleLogonWindow
  - ItsCspSampleCustomer
  - ItsCspSampleCustomerListModel
- **Superclass of**

  None

- **Class definition**

  Figure 160 on page 145 shows the class definition for the Sample Communication SideInfo part.

```
Its3270CommunicationSideInfo subclass: #ItsCspSampleCommunicationSideInfo
    instanceVariableNames: ''
    classVariableNames: 'TransactionDirectory '
    poolDictionaries: ''
```

Figure 160. Class Definition: ItsCspSampleCommunicationSideInfo

- **Scripts**

  Table 16 shows the scripts for the Sample Communication SideInfo part.

Table 16. Scripts: Sample Communication SideInfo

Method	Description
initializeTransactionDirectory	Puts Tx codes for each operation into the class variable TransactionDirectory.
readTransactionDirectoryAt: aSymbol	Reads Tx codes from class variable TransactionDirectory with key.

Figure 161 shows the initializeTransactionDirectory method.

```
initializeTransactionDirectory

   TransactionDirectory := IdentityDictionary new.

   TransactionDirectory at: #AddCustomer        put: 'STLC'.
   TransactionDirectory at: #DeleteCustomer     put: 'STLC'.
   TransactionDirectory at: #ReadCustomer       put: 'STLC'.
   TransactionDirectory at: #UpdateCustomer     put: 'STLC'.
   TransactionDirectory at: #ReadCustomerList   put: 'STLC'.
```

Figure 161. Method: initializeTransactionDirectory

Figure 161 shows the readTransactionDirectoryAt: method.

```
readTransactionDirectoryAt: aSymbol

   ^(TransactionDirectory at: aSymbol).
```

Figure 162. Method: readTransactionDirectoryAt:

- **Event trace**

  None

- **Special comments**

  We did not use the TransactionDirectory in our application. The transaction code to update, list, and delete customers is hardcoded, which makes it easier to understand the code for beginners. For real customer applications we recommend using the TransactionDirectory to keep all application parameters in one place. The best implementation is to keep the application parameters in an external file that is read during application initialization. This approach allows maintenance of the parameters to be done externally without any code change.

How to Fill a Dictionary with Values

The following code can be used to fill a dictionary with values:

```
TransactionDirectory := IdentityDictionary new.

TransactionDirectory at: #AddCustomer          put: 'STLA'.
TransactionDirectory at: #DeleteCustomer       put: 'STLB'.
TransactionDirectory at: #ReadCustomer         put: 'STLC'.
TransactionDirectory at: #UpdateCustomer       put: 'STLD'.
TransactionDirectory at: #ReadCustomerList     put: 'STLE'.
```

8.10 DB2 Sample Logon Window

This part is the logon window for the DB2 sample application.

- **Part name**

  ItsDb2SampleLogonWindow

- **Category**

  Composite visual part

- **Description**

  This part is the same as the CSP sample logon window part described in 8.6, "CSP Sample Logon Window" on page 132. Because we experimented with different approaches during our project this part was not implemented exactly as the CSP sample logon window part.

  The GUI front end for the DB2 sample application is not implemented. Therefore, we do not explain the implementation of this part in detail.

8.11 CSP Sample Main Window

After a successful logon to the CSP sample application the main window of the application is opened. This part contains the main window for the sample application. From the main window users can start all functions of the application. This window has a parent relationship to all windows that are opened from here.

- **Part name**

  ItsCspSampleMainWindow

- **Category**

  Visual composite part

- **Description**

  This part has in its primary window a container with icon gadgets. Double-clicking on an icon starts an application function. The same functions can also be started from the action bar Customer menu item.

  The windows that are shown after a function is started are added as parts to this main part. The windows are the customer selection criteria window and the add a new customer window. The host window icon invokes a script that provides the function to jump to the emulation window of the selected session.

One decision to be made is how the windows that can be opened should be related to the main window:

- The **openWidget** action opens a window that is independent of the main window. Therefore, if the main window is minimized, the opened window remains open. However, if the main window is closed, the opened window is also closed.
- The **openOwnedWidget** action opens a window that is tightly related to the main window. Therefore, if the main window is minimized or closed, the opened window is also minimized or closed.

Another decision to be made is whether the opened window should allow multiple instances or not. Multiple instance windows are always independent of the main window, and the application has to provide the logic to close the window instances when the application main window is closed. To create single instance windows, a window can be drawn in the same Composition Editor view, or it can be implemented as a separate part and added as a subpart to the Composition Editor of another window. To create multiple instance windows, we can either write a script or use the VisualAge object factory part in the Composition Editor.

We decided to implement the New Customer window and the Customer List Selection window as single instance windows that are opened using the openWidget action. We added those two windows as subparts to the Composition Editor.

- **Composition Editor view**

  Figure 163 and Figure 164 on page 148 show the Composition Editor view for the CSP Sample Main window.

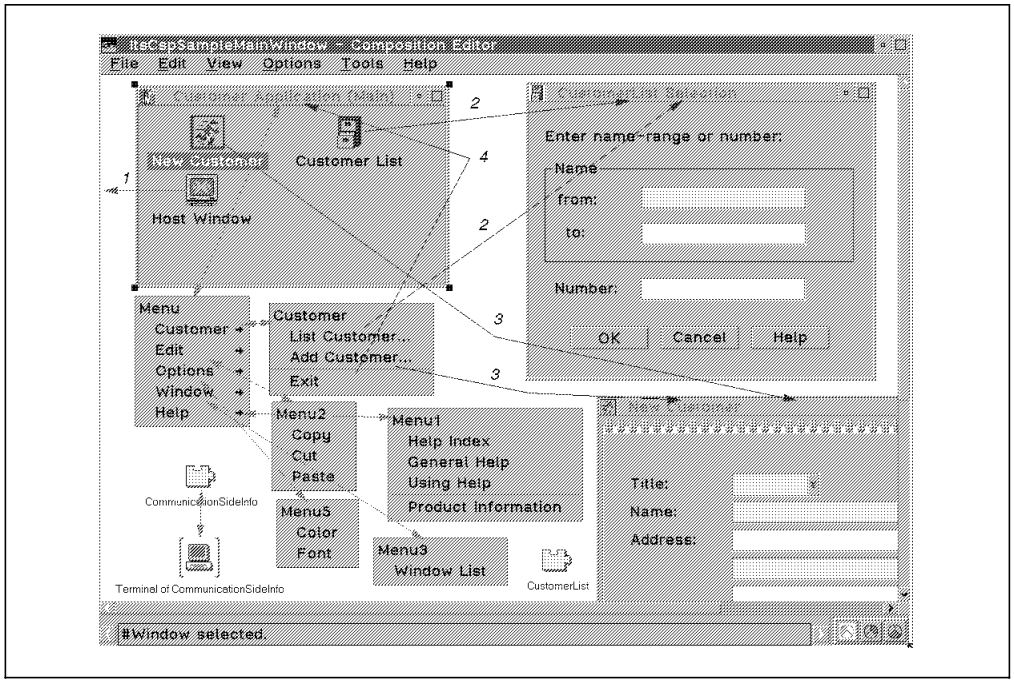

Figure 163. Composition Editor View: ItsCSPSampleMainWindow (Part 1)

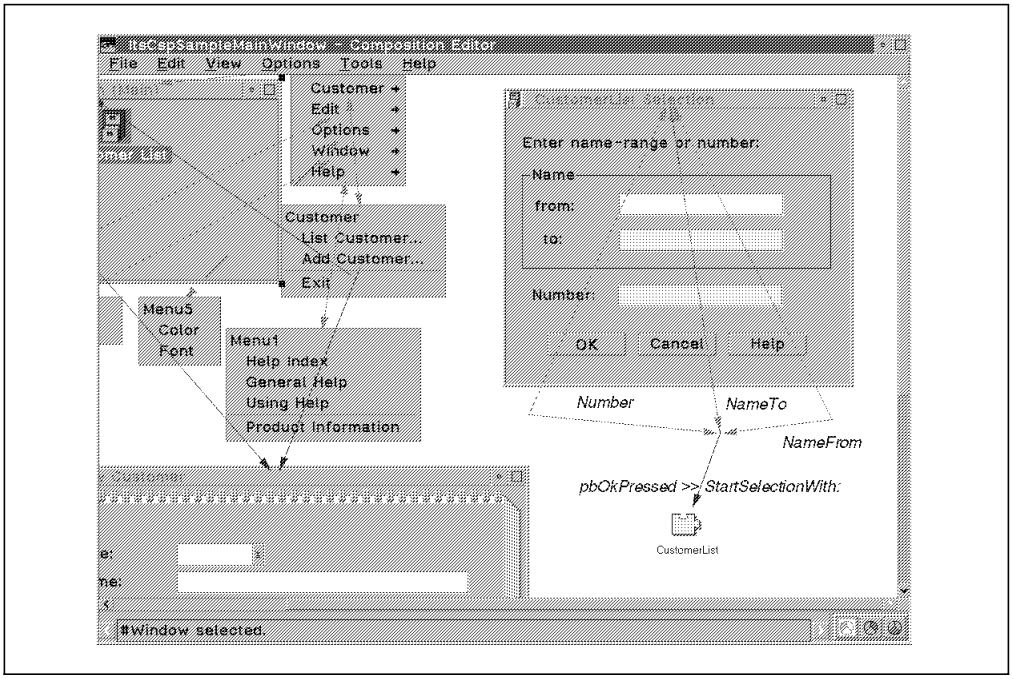

Figure 164. Composition Editor View: ItsCSPSampleMainWindow (Part 2)

- **Part assembly**

  Figure 165 shows the part assembly for the CSP Sample Main window.

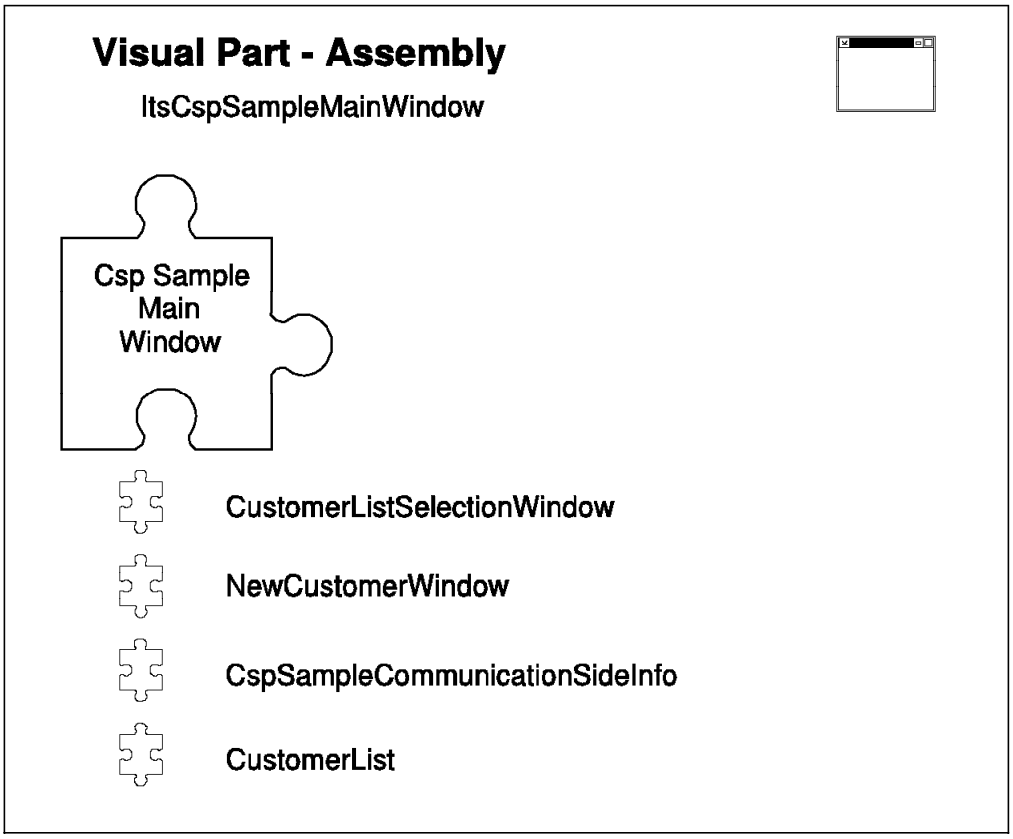

Figure 165. Part Assembly: ItsCSPSampleMainWindow

- **Public interface**

None

- **Used in part**
- Its3270Applications
- **Superclass of**

  None

- **Class definition**

  Figure 166 shows the class definition for the CSP Sample Main window.

```
AbtAppBldrView subclass: #ItsCspSampleMainWindow
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
```

Figure 166. Class Definition: ItsCSPSampleMainWindow

- **Scripts**

  Table 17 shows the scripts for the CSP Sample Main window.

Table 17. Scripts: CSP Sample Main Window

Method	Description
showHostWindow	Jumps to the emulation window with the session id stored in the communication side information (SideInfo) part.

Figure 167 shows the showHostWindow method.

```
showHostWindow

   | aTerminal |

   aTerminal  := self partAttributeValue:
                      #(#'Terminal of CommunicationSideInfo' #self).

   aTerminal windowRestore;
             windowJump.
```

Figure 167. Method: showHostWindow

- **Event trace**

  Table 18 shows the event trace for the CSP Sample Main window.

Table 18 (Page 1 of 2). Event Trace: CSP Sample Main Window

User Action	Sequence of Executed Connections
Double-click on Host Window icon	(1) defaultAction >> showHostWindow (hook)
Double-click on Customer List icon	(2) defaultAction >> openWidget (CustomerListSelectionWindow)

Table 18 (Page 2 of 2). Event Trace: CSP Sample Main Window

User Action	Sequence of Executed Connections
Click on List Customer menuItem	(2) clicked >> openWidget (CustomerListSelectionWindow)
Double-click on New Customer icon	(3) clicked >> openWidget (NewCustomerWindow)
Click on Add Customer menuItem	(3) clicked >> openWidget (NewCustomerWindow)
Click on Exit menuItem	(4) clicked >> closeWidget (Customer Application Main Window)
Press OK push button on CustomerListSelection dialog	pbOkPressed >> StartSelectionWith: (NameFrom, NameTo, Number)

- **Special comments**

  We provide a script to jump to the host session to which the application is connected in this part. The session ID is found in a Sample Communication SideInfo part.

How to Jump to a Host Session

The following code implements the function to jump to the host session:

```
| aTerminal |

aTerminal  := self partAttributeValue:
    #(#'Terminal of CommunicationSideInfo' #self).

aTerminal windowRestore;
          windowJump.
```

A window can be created and opened as a single instance or several times as multiple instances.

How to Create a Single Instance Window

If a window should exist as a single instance in an application, we can add the window as a subpart to another window and draw a connection to open the window. Using a subpart to add a window is good implementation style. However, we can achieve the same result if the second window is drawn in the Composition Editor of another window.

How to Create Multiple Instance Windows

To create multiple instances of a window, we can either use an object factory part or write a script.

When using the object factory, we have to make an event-to-action connection (1) to the new action of the factory part. The object factory type must be set to the window part to be created. The window part will become an instance variable of the object factory. We tear off the instance variable from the object factory and make a second event-to-action connection from the same event as connection (1) to the openWidget action of the instance variable.

The code to achieve the same result with a script looks as follows:

```
createWindowInstance: aWindowPartNameString

    | classView |

    classView:= Smalltalk at: (aWindowPartNameString asSymbol).
    classView newPart openWidget.
```

8.12 Sample Customer List Selection Window

We decided to have a single instance of the customer list selection window in the application. This window is developed as a separate part and used as a subpart in the main window.

- **Part name**

  ItsCspSampleCustomerListSelectionWindow

- **Category**

  Visual composite part

- **Description**

  This is a part on the view side according to the model-view separation design. It does not know anything about communication; it is communication independent. It is a dialog where users can enter their selection criteria to be passed as attributes through the public interface. These attributes can be used as parameters for the StartSelectionWith action that reads the customer list data from the host application.

- **Composition Editor view**

  Figure 168 on page 152 shows the Composition Editor view for the Customer List Selection window.

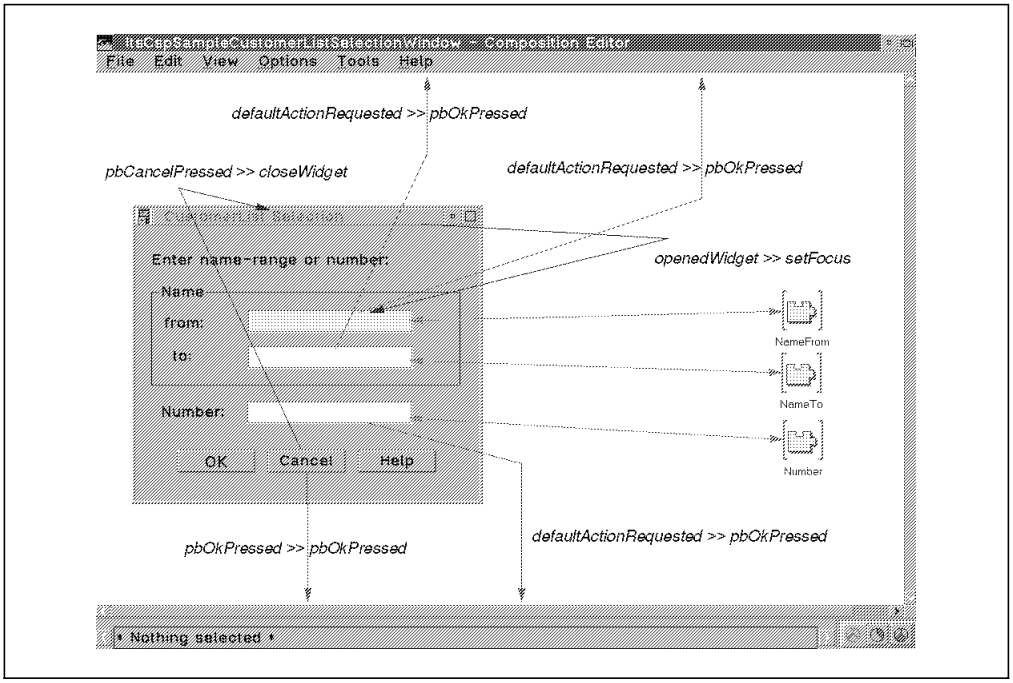

Figure 168. Composition Editor View: ItsCspSampleCustomerListSelectionWindow

- **Part assembly**

  Figure 169 shows the part assembly for the Customer List Selection window.

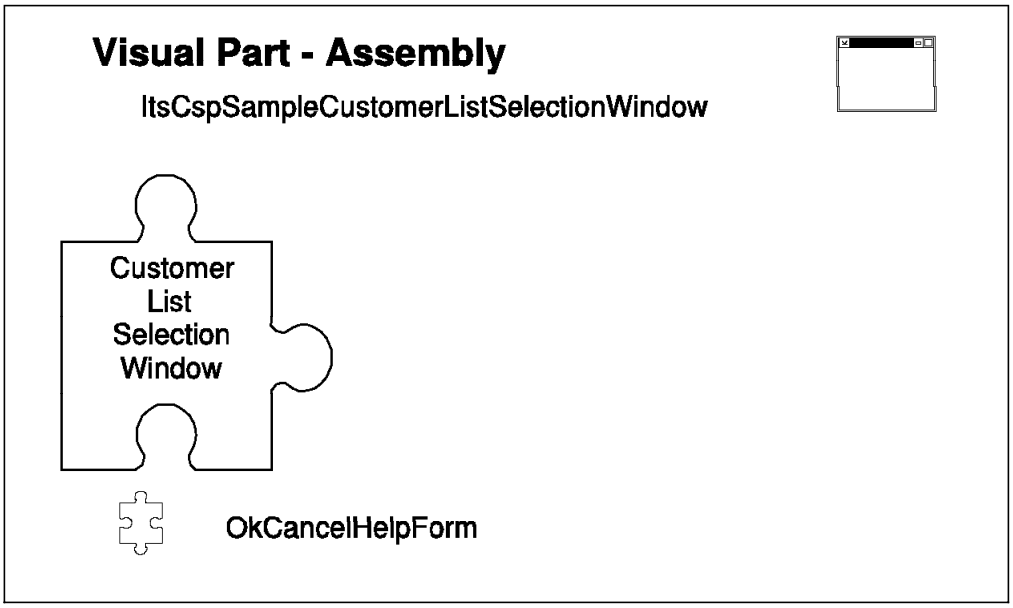

Figure 169. Part Assembly: ItsCspSampleCustomerListSelectionWindow

- **Public interface**

  Figure 170 on page 153 shows the public interface for the Customer List Selection window.

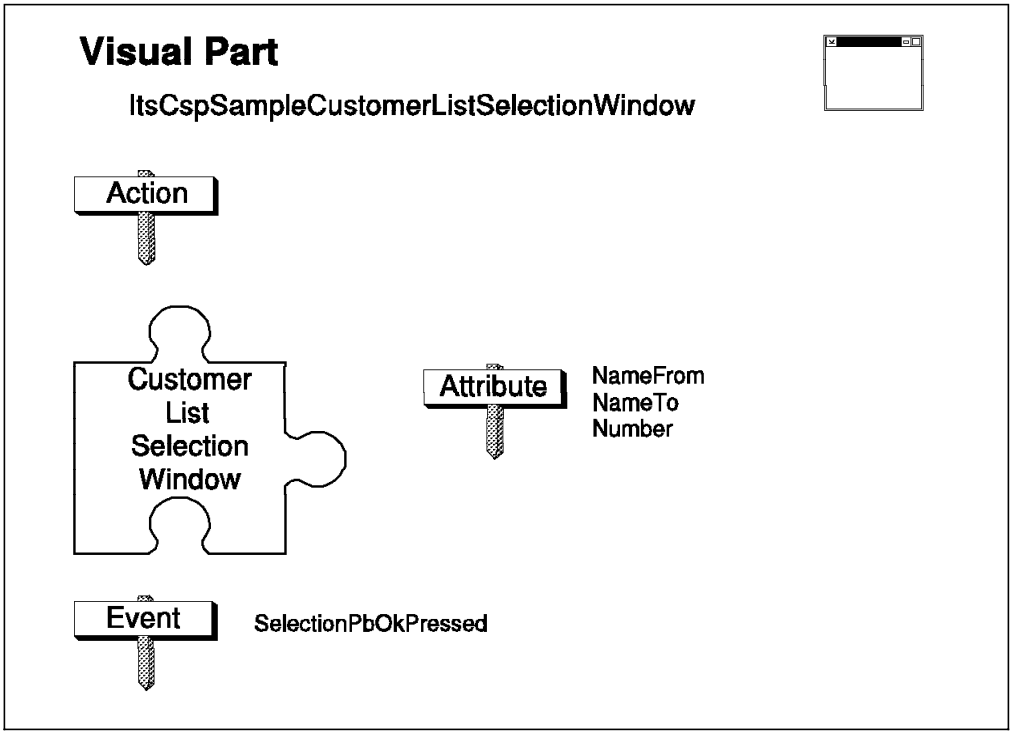

Figure 170. Public Interface: ItsCspSampleCustomerListSelectionWindow

- **Used in part**

  ItsCspSampleMainWindow

- **Superclass of**

  None

- **Class definition**

  Figure 171 shows the class definition for the Customer List Selection window.

```
AbtAppBldrView subclass: #ItsCspSampleCustomerListSelectionWindow
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
```

Figure 171. Class Definition: ItsCspSampleCustomerListSelectionWindow

- **Scripts**

  Table 19 shows the scripts for the Customer List Selection window.

Table 19. Scripts: Customer List Selection Window

Method	Description
pbOkPressed	Checks for nonblank selection criteria and raises SelectionPbOkPressed event if selection is made. Shows a message box if selection is blank.

Figure 172 on page 154 shows the pbOkPressed method.

```
pbOkPressed

   | sizeNameFrom sizeNameTo sizeNumber |

   sizeNameFrom:= (self partAttributeValue: #(#efFromName #string))
                  trimBlanks size.
   sizeNameTo:=   (self partAttributeValue: #(#efToName #string))
                  trimBlanks size.
   sizeNumber:=   (self partAttributeValue: #(#efNumber #string))
                  trimBlanks size.

   ((sizeNameFrom=0
      and: [sizeNameTo=0])
      and: [sizeNumber=0]) ifTrue:
         [CwMessagePrompter
            errorMessage: 'All fields are empty. No valid selection !!'.
          ^self.
         ].

   ^self signalEvent: #SelectionPbOkPressed.
```

Figure 172. Method: pbOkPressed

- **Event trace**

  Table 20 shows the event trace for the Customer List Selection window.

Table 20. Event Trace: Customer List Selection Window

User Action or Event	Sequence of Executed Connections
Event: openedWidget (initialization)	openedWidget >> setFocus (NameFrom entry field)
User action: click on OK push button	pbOkPressed (OkCancelHelpForm) >> pbOkPressed (hook)
User action: press Enter on NameFrom entry field	defaultActionRequested >> pbOkPressed (hook)
User action: press Enter on NameTo entry field	defaultActionRequested >> pbOkPressed (hook)
User action: press Enter on Number entry field	defaultActionRequested >> pbOkPressed (hook)
User action: click on Cancel push button	pbCancelPressed >> closeWidget

- **Special comments**

  We decided to make the link to the customer list in the application main part. Another way would be to add the customer list to this part. This is just another way to encapsulate things, but the advantage of our implementation is the granularity of the parts. To provide better part reusability, it is better to have many basic parts instead of a few composite parts.

How to Provide Entry Fields in the Public Interface

Perform the following sequence to pass the contents of the entry fields to the public interface as attributes:

1. Add a variable (Option Add Variable) for each entry field or go to the entry field and tear off #string to create a variable of data type string or #object to create a variable of another data type.
2. Define a name for or rename the variable to make it self-explanatory such as Number or NameFrom, and define the class name as the name of the data type class (String, Integer, Float) you want to provide.
3. Connect the variable (#self) to the entry field (#string) if the data type is string or to #object for another data type. This connection is done implicitly, if the variable is torn off from the entry field.
4. Select the variables (all together) in the Composition Editor and select Add to interface from the context menu.
5. Go to the public interface editor and select the attributes tab. Select Added Attributes to verify and apply the definition.

8.13 Sample Customer List

This part is one of the model parts of the application. It provides methods to access the host through EHLLAPI and creates instances of customer list windows to show the result of the list request in a window.

- **Part name**

  ItsCspSampleCustomerListModel

- **Category**

  Nonvisual composite part

- **Description**

  This part is the model representation of the customer list following the model-view separation design. The list selection criteria are passed to this part as parameters to an action. The implementation of this part is an example of extensive use of visual programming in a nonvisual part. We tried to define as much of the logic as possible with visual programming in this part. At first sight, the Composition Editor view of the part might look a little confusing. To understand the defined logic, it is important to follow the event trace described in Table 22 on page 162.

  The decision as to how much of the program logic should be built visually and how much should be provided through scripts depends on developer preference and coding style. From the software engineering point of view, a mix of visual programming and scripts allows a better structuring of the application logic, and the code is easier to understand and maintain as compared to a complex network of visual programming. The issue is to find the balance between visual programming and scripts when defining the logic of a part. Our implementation of this part is not a good example of the right balance.

- **Composition Editor view**

Figure 173 shows the Composition Editor view for the CustomerListModel part.

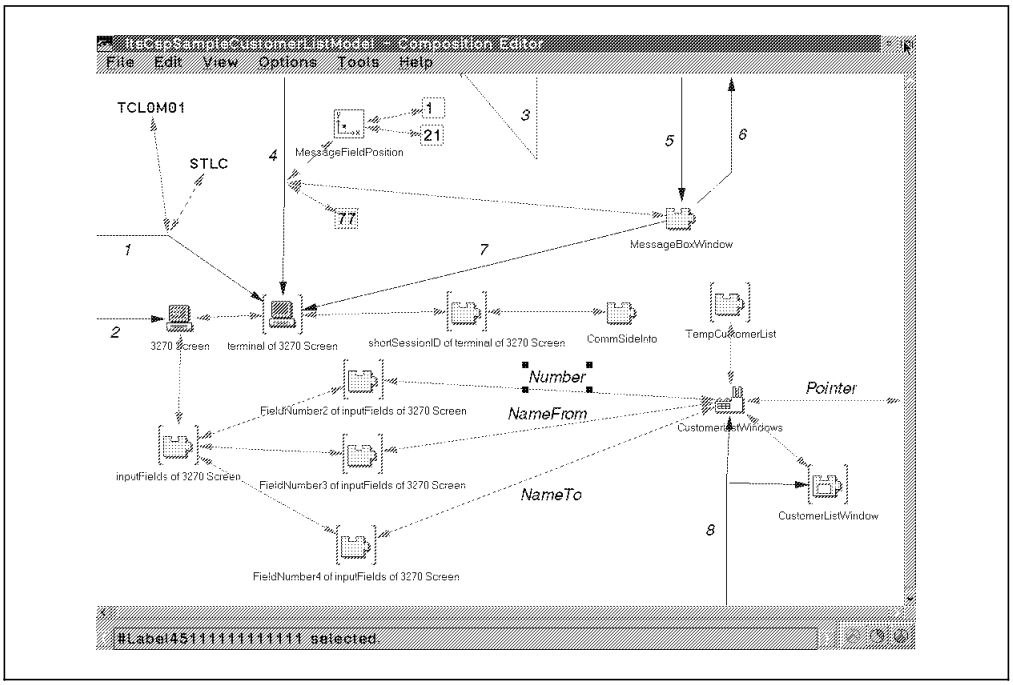

Figure 173. Composition Editor View: ItsCspSampleCustomerListModel

- **Part assembly**

  Figure 174 shows the part assembly for the CustomerListModel part.

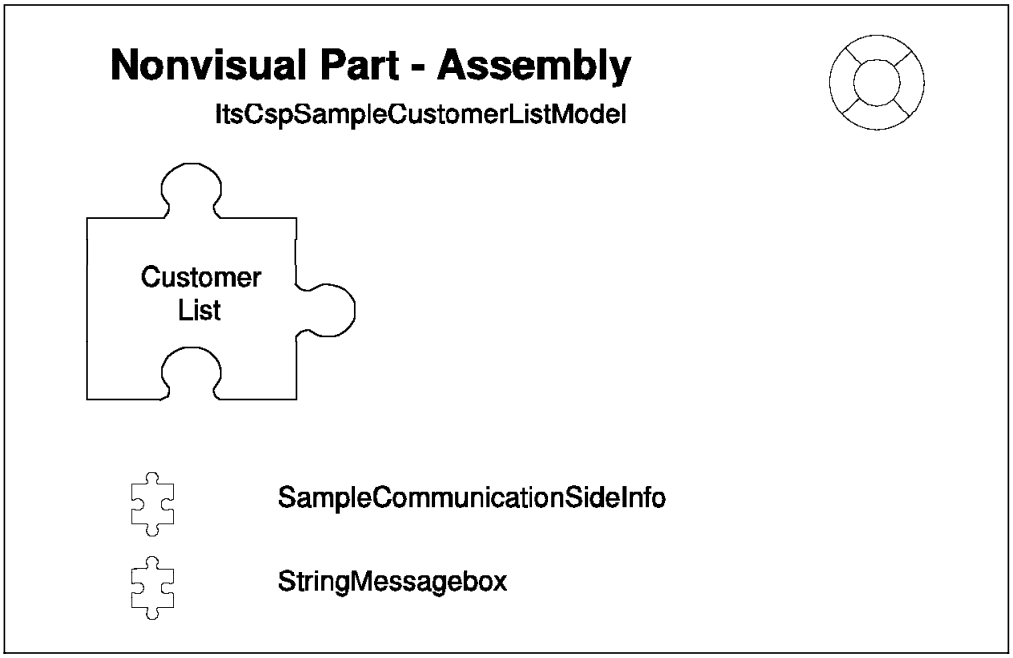

Figure 174. Part Assembly: ItsCspSampleCustomerListModel

- **Public interface**

  Figure 175 on page 157 shows the public interface for the CustomerListModel part.

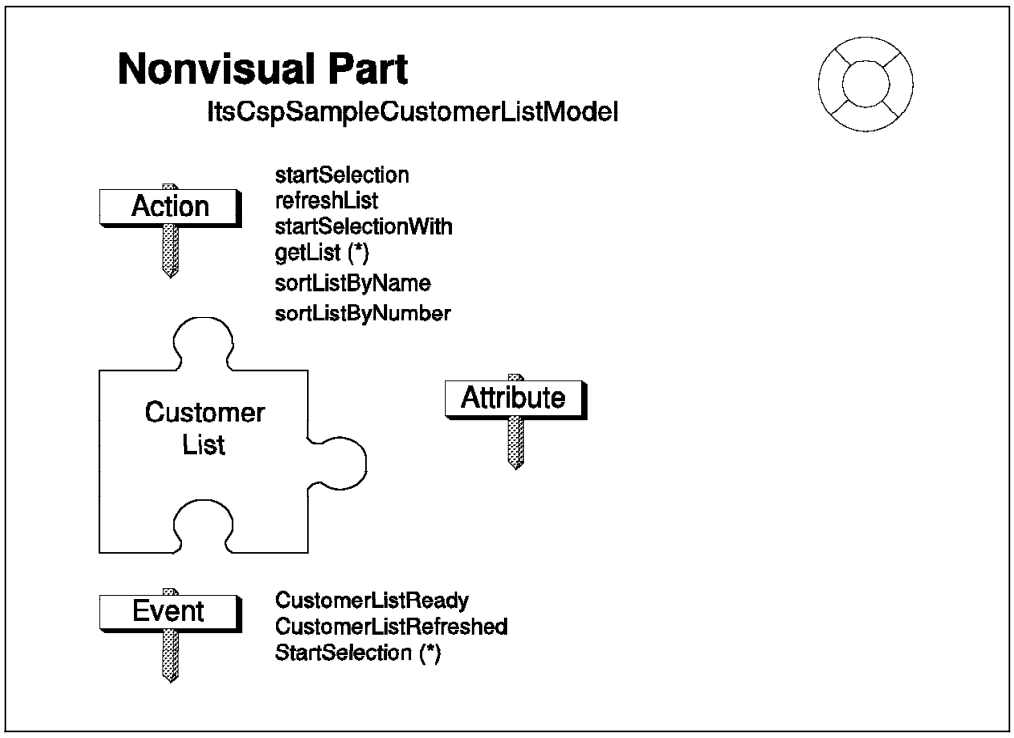

Figure 175. Public Interface: ItsCspSampleCustomerListModel

Note: The StartSelection(*) event and the getList(*) action are used only within this part to draw event-to-action connections in the Composition Editor. They are not used in other parts and could be called private in the public interface. Of course, calling something private in the public interface is a bit contradictory, but in this specific situation of visual programming, it is very useful.

- **Used in part**

  ItsCspSampleMainWindow

- **Superclass of**

  None

- **Class definition**

  Figure 176 shows the class definition for the CustomerListModel part.

```
AbtAppBldrPart subclass: #ItsCspSampleCustomerListModel
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
```

Figure 176. Class Definition: ItsCspSampleCustomerListModel

- **Scripts**

  Table 21 on page 158 shows the scripts for the CustomerListModel part.

Table 21. Scripts: CustomerListModel	
Method	**Description**
pressEnterandWaitfor CursorPositionChanged	Enters a blank string and waits until the cursor position is changed from position 1 @ 1.
readCustomerList AndBuildCollection	Loops through the whole list on the host screen using PF8 and returns the result in a sorted collection. The loop is stopped based on the message “No more rows” on the host screen. After finishing collecting the rows in the loop, the CustomerListReady and CustomerListRefreshed events are raised. This method is defined in the public interface as a getList action. It is used in an event-to-action connection within this part.
refreshCustomerWith: aNameFromString and: aNameToString and: aNumberFromString	Goes through the sequence of host screens and collects the items in a sorted collection. The complete logic for this function is coded in this script. There is no complementary logic on the Composition Editor. The method is defined as a refreshList action in the public interface.
sortListByName	Sorts the CustomerList variable in the Composition Editor by name.
sortListByNumber	Sorts the CustomerList variable in the Composition Editor by number.
startSelectionWith: aNameFromString and: aNameToString and: aNumberFromString	Puts the parameters into the torn-off input fields of the screen part and raises the StartSelection event.
startTransaction	Starts the STLC transaction. This is hardcoded and does not use the Communication SideInfo transaction dictionary.

Figure 177 shows the pressEnterandWaitforCursorPositionChanged method.

```
pressEnterandWaitforCursorPositionChanged

   | aTerminal |

   aTerminal := self partAttributeValue:
                        #(#'terminal of 3270 Screen' #self).
   aTerminal enter: ''
             andWaitForCursorPositionToChangeFrom: 1 @ 1.
```

Figure 177. Method: pressEnterandWaitforCursorPositionChanged

Figure 178 on page 159 shows the readCustomerListAndBuildCollection method.

```
readCustomerListAndBuildCollection

   | anArray aCollection aTerminal doLoop |

   aTerminal := self partAttributeValue:
                    #(#'terminal of 3270 Screen' #self).

   aCollection := SortedCollection sortBlock:
      [ :a :b |
         ((a copyFrom: 9 to: 30) trimBlanks)
         <=
         ((b copyFrom: 9 to: 30) trimBlanks)].

   doLoop := 1.

   [doLoop = 1] whileTrue:
      [
         doLoop := 0.
         anArray := aTerminal textFrom: 5 @ 4 to: 79 @ 16.

         anArray do:
            [:each | (each trimBlanks size = 0) ifFalse:
               [aCollection add: each].
            ].

         ((anArray at: 13) trimBlanks size = 0) ifFalse:
            [
               aTerminal keyPF:8.
               self pressEnterandWaitforCursorPositionChanged.
               ((aTerminal stringAt: 1@21 for: 77) abrIncludesString:
               'No more rows') ifFalse:
               [doLoop := 1.].
            ].
      ].

   self partAttributeValue:
         #(#TempCustomerList #self) put: aCollection.

   self signalEvent: #CustomerListReady.
   self signalEvent: #CustomerListRefreshed.

   aTerminal keyPF:3.

   ^aCollection.
```

Figure 178. Method: readCustomerListAndBuildCollection

Figure 179 on page 160 shows the refreshCustomerWith: and: and: method.

```
refreshCustomerWith: aNameFromString
      and: aNameToString and: aNumberFromString

   | anArray aCollection aTerminal doLoop aScreenRecord aString |

   aTerminal := self partAttributeValue: #(#'terminal of 3270 Screen' #self).

   aCollection := SortedCollection sortBlock:
      [ :a :b |
         ((a copyFrom: 9 to: 30) trimBlanks)
         <=
         ((b copyFrom: 9 to: 30) trimBlanks)].

   self startTransaction.

   aScreenRecord := (aTerminal buildFieldDefsOfType: 'Unprotected')
                    newRecord.
   aTerminal getStringsIntoFieldRecord: aScreenRecord.

   aScreenRecord at: #FieldNumber2 put: (aNumberFromString).
   aScreenRecord at: #FieldNumber3 put: (aNameFromString).
   aScreenRecord at: #FieldNumber4 put: (aNameToString).
   aTerminal putStringsFromFieldDefs: aScreenRecord.

   self pressEnterandWaitforCursorPositionChanged.

   aString := ((aTerminal stringAt: 1@21 for: 77) trimBlanks).
   (aString isEmpty) ifFalse:
   [
      (aString abrIncludesString: 'No rows meet') ifTrue:
         [
            CwMessagePrompter message: aString.
            aTerminal keyPF:3.
            ^aCollection.
         ].
   ].

   doLoop := 1.

   [doLoop = 1] whileTrue:
      [
         doLoop := 0.
         anArray := aTerminal textFrom: 5@4 to: 79@16.

         anArray do:
            [:each | (each trimBlanks size = 0) ifFalse:
               [aCollection add: each].
            ].

         ((anArray at: 13) trimBlanks size = 0) ifFalse:
```

Figure 179 (Part 1 of 2). Method: refreshCustomerWith: and: and:

```
            [
               aTerminal keyPF:8.
               self pressEnterandWaitforCursorPositionChanged.
               ((aTerminal stringAt: 1@21 for: 77) abrIncludesString:
               'No more rows') ifFalse:
                  [doLoop := 1.].
            ].
      ].

   self signalEvent: #CustomerListRefreshed.

   aTerminal keyPF:3.

   ^aCollection.
```

Figure 179 (Part 2 of 2). Method: refreshCustomerWith: and: and:

Figure 180 shows the sortListByName method.

```
sortListByName

   | aSortedCollect |

    aSortedCollect := self partAttributeValue: #(#CustomerList #self).

   aSortedCollect sortBlock: [ :a :b |
      ((a copyFrom: 9 to: 30) trimBlanks)
      <=
      ((b copyFrom: 9 to: 30) trimBlanks)].

   ^self partAttributeValue: #(#CustomerList #self) put: aSortedCollect.
```

Figure 180. Method: sortListByName

Figure 181 shows the sortListByNumber method.

```
sortListByNumber

   | aSortedCollect |

    aSortedCollect := self partAttributeValue: #(#CustomerList #self).

    aSortedCollect sortBlock: [ :a :b |
      (a copyFrom: 1 to: 7 )
      <=
      (b copyFrom: 1 to: 7)].

   ^self partAttributeValue: #(#CustomerList #self) put: aSortedCollect.
```

Figure 181. Method: sortListByNumber

Figure 182 on page 162 shows the startSelectionWith: and: and: method.

```
startSelectionWith: aNameFromString
   and: aNameToString and: aNumberFromString

   self partAttributeValue:
      #(#'FieldNumber2 of inputFields of 3270 Screen' #self)
      put: aNumberFromString.
   self partAttributeValue:
      #(#'FieldNumber3 of inputFields of 3270 Screen' #self)
      put: aNameFromString.
   self partAttributeValue:
      #(#'FieldNumber4 of inputFields of 3270 Screen' #self)
      put: aNameToString.

   self signalEvent: #StartSelection.
```

Figure 182. Method: startSelectionWith: and: and:

Figure 183 shows the startTransaction method.

```
startTransaction

   | aTerminal |

   aTerminal := self partAttributeValue:
                    #(#'terminal of 3270 Screen' #self).

   aTerminal   keyHome;
               enterCommand: 'STLC'.
```

Figure 183. Method: startTransaction

- **Event trace**

  Table 22 shows the event trace for the CustomerListModel part.

Table 22 (Page 1 of 2). Event Trace: CustomerListModel

Event	Sequence of Executed Connections
event: StartSelection raised from StartSelectionWith: (NameFrom, NameTo, Number), which is called in the customer main window as a connection between ListSelectionWindow and CustomerList	• (1) event: StartSelection >> terminal enter: 'STLC' andWaitForString: 'TCL0M01' • (2) event: StartSelection >> screen putData (the fields are filled in the StartSelectionWith method called first) • (3) event: StartSelection >> pressEnterAndWaitForCursorPositionChanged (hook) • (4) event: StartSelection >> terminal execute: stringAt: 21@1 for: 77 and put result in StringMessagebox (public interface attribute messageString) • (5) event: StartSelection >> open (MessageboxWindow)
event: ErrorNotFound raised from MessageboxWindow	(6) event: ErrorNotFound >> getList (hook) (this method is in the public interface and is mapped to the script readCustomerListAndBuildCollection)

Table 22 (Page 2 of 2). Event Trace: CustomerListModel	
Event	**Sequence of Executed Connections**
event: ErrorFound raised from MessageboxWindow	(7) event: ErrorFound >> terminal keyPF:3 (to end the transaction)
event: CustomerListReady raised from script readCustomerList AndBuildCollection	• (8) event: CustomerListReady >> new CustomerWindowsFactory • result of new >> openWidget (CustomerListWindow)

- **Special comments**

How to Use a Factory to Create Multiple Instances

You can use a factory to create multiple instances of customer windows as follows:

- Add a factory part to the Composition Editor.
- Define the instance class name.
- Connect variables on the Composition Editor to the instance variables provided in the public interface of the class for which you defined the factory. In our example, we connect the NameFrom, NameTo, Number, TempCustomerList (SortedCollection), and the Pointer to the CustomerList part to the factory.
- Add another variable that has the class (or data type) of the factory instances to the Composition Editor.
- Connect instance (factory) >> self (variable) to put the result into this temporary variable.
- Connect the follow-on logic to the variable. In our example, the variable is a visual part and we send the openWidget message to it.

How to Write Data to the Terminal from a Script

A very efficient way of writing directly to the presentation space is the following:

```
aScreenRecord :=
    (aTerminal buildFieldDefsOfType: 'Unprotected') newRecord.

aScreenRecord at: #FieldNumber2 put: (aNumberFromString).
aScreenRecord at: #FieldNumber3 put: (aNameFromString).
aScreenRecord at: #FieldNumber4 put: (aNameToString).

aTerminal putStringsFromFieldDefs: aScreenRecord.
```

First, we built a record with the field definitions of the unprotected fields found in the actual presentation space. Then, we put the contents into the fields. Last, we wrote the whole record to the terminal presentation space.

8.14 String Messagebox

This part is used in the CustomerListModel part to show a message box with an error string passed to it through an attribute-to-attribute connection to the messageString attribute.

- **Part name**

  ItsStringMessagebox

- **Category**

  Basic nonvisual part

- **Description**

  This is a part for a customized message box. It takes a string as a parameter and, depending on the string contents, brings up a message box with the contents of the parameter string and raises the errorFound event. If the string is blank or is not an error string, no message box is shown, and the errorNotFound event is raised.

  This part demonstrates how to implement a message box that can be used in the Composition Editor for visual programming.

- **Composition Editor view**

  Figure 184 shows the Composition Editor view for the String Messagebox part.

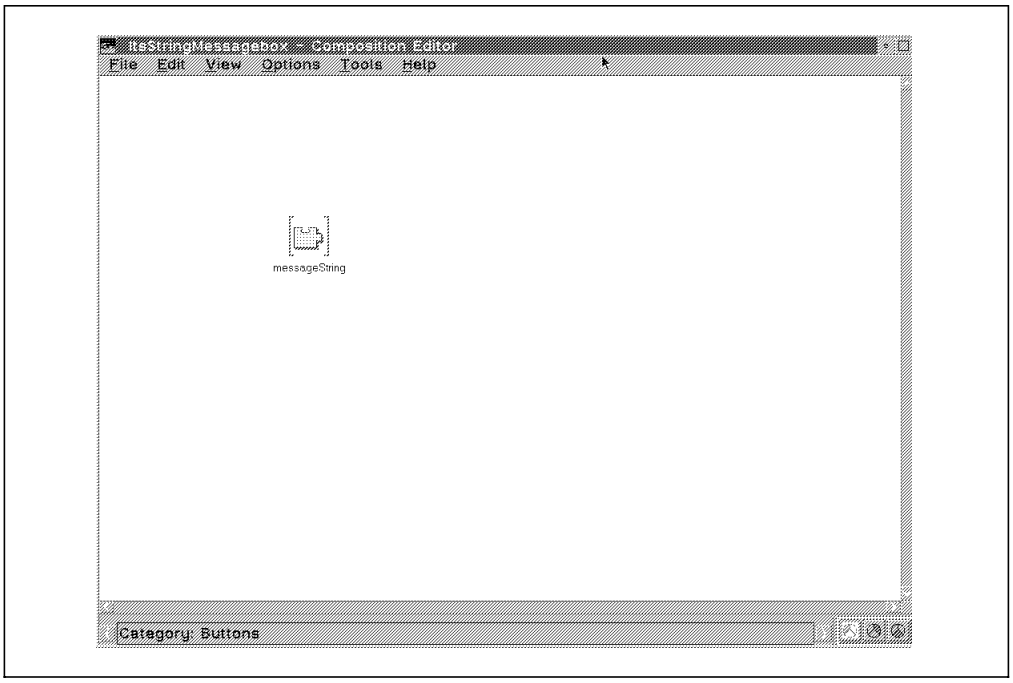

Figure 184. Composition Editor View: ItsStringMessagebox

  The messageString variable is added to the public interface as an attribute.

- **Part assembly**

  None

- **Public interface**

Figure 185 shows the public interface for the String Messagebox part.

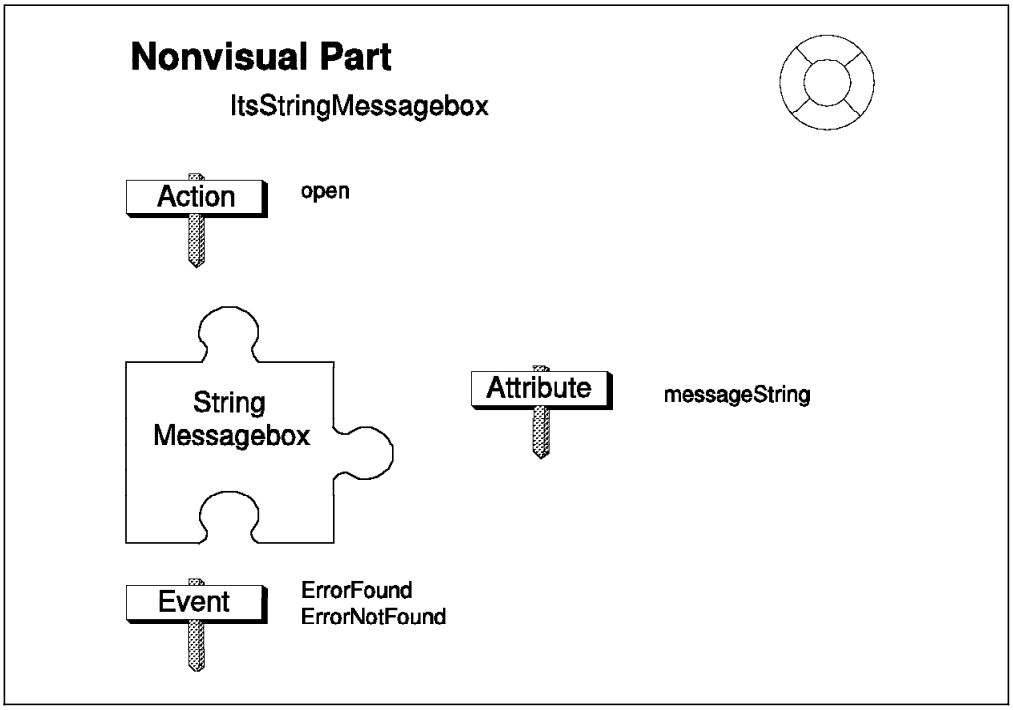

Figure 185. Public Interface: ItsStringMessagebox

- **Used in part**

  Customer List

- **Superclass of**

  None

- **Class definition**

  Figure 186 shows the class definition for the String Messagebox part.

```
AbtAppBldrPart subclass: #ItsStringMessagebox
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
```

Figure 186. Class Definition: ItsStringMessagebox

- **Scripts**

  Table 23 shows the scripts for the String Messagebox part.

Table 23. Scripts: String Messagebox

Method	Description
open	Checks the messageString passed by the public interface as an attribute and decides whether or not a messagebox should be shown. The ErrorFound or ErrorNotFound events are raised accordingly.

Figure 187 on page 166 shows the open method.

```
open

   | aString |

   aString := self partAttributeValue: #(#messageString #self).
   aString := aString trimBlanks.

   "no MessageString"
   (aString isEmpty) ifTrue:
      [^self signalEvent:#errorNotFound.].

   "MessageString: Single item selection list ..."
   (aString abrIncludesString: 'Single item selection list') ifTrue:
      [^self signalEvent:#errorNotFound.].

   "MessageString: Invalid function key,(caused by timing problems)"
   (aString abrIncludesString: 'Invalid function key') ifTrue:
      [^self signalEvent:#errorNotFound.].

   "otherwise"
   CwMessagePrompter message: aString.
   ^self signalEvent:#errorFound.
```

Figure 187. Method: open

- **Event trace**

  None

- **Special comments**

  Instead of passing the messagestring as a parameter directly to the method, we passed the string as an attribute to the public interface and used it in the Composition Editor of the nonvisual part.

8.15 Customer List Window

After the CustomerListModel part reads the data from the host application, it creates a Customer List Window instance using an object factory part. The Customer List Window is a window with a list box where the selected customers are listed as strings of customer number and name. By default, the customer list is sorted by number. That is the only sort order the existing host application provides. The GUI application provides the additional functionality of sorting by name entirely on the workstation. No communication to the host is necessary to do the sort.

Another added function that the GUI application provides is a local subselect with a search string in the customer list. The result of this subselect is shown in a separate window.

The refresh function can be used to refresh the information in the customer list from the host application. The customer list remembers the former selection criteria, because they are stored in variables defined in both the Composition Editor and the public interface. In the refresh dialog, the user can change the selection criteria.

The GUI application supports multiple instances of Customer List Windows. Multiple Customer List Windows can be opened by providing new selection criteria in the customer list selection window.

- **Part name**

  ItsCspSampleCustomerListWindow

- **Category**

  Composite visual part

- **Description**

  The implementation of this visual part is a typical example of visual programming. The visual programming approach is very powerful, especially for window handling and the passing of attributes with connections from one part to another. This part does not have many scripts.

  We decided to keep the Customer List Window together with the subdialogs to find a string and to refresh the list when designing the parts. The two dialog windows are single instance windows and closely related to the context of the Customer List Window. Therefore, there is not much potential to reuse those windows, and they can be implemented in the Customer List Window part.

  There is nothing that prevents dividing this part into three separate parts, one for each window. The public interface for each part would need to be defined to allow for synchronization of the list boxes in the Customer List and Find in Customer List windows.

- **Composition Editor view**

  We show the different aspects of the visual program definition in different pictures to reduce the complexity of the Composition Editor view (see Figure 188 on page 168 through Figure 190 on page 169). If we open the Composition Editor with this part, the three pictures are all in the same view.

  Figure 188 on page 168 shows the visual program definition required to show the ordered collection of customers in the list box and to read the details of a selected customer with an event-to-action connection to the nonvisual part customer.

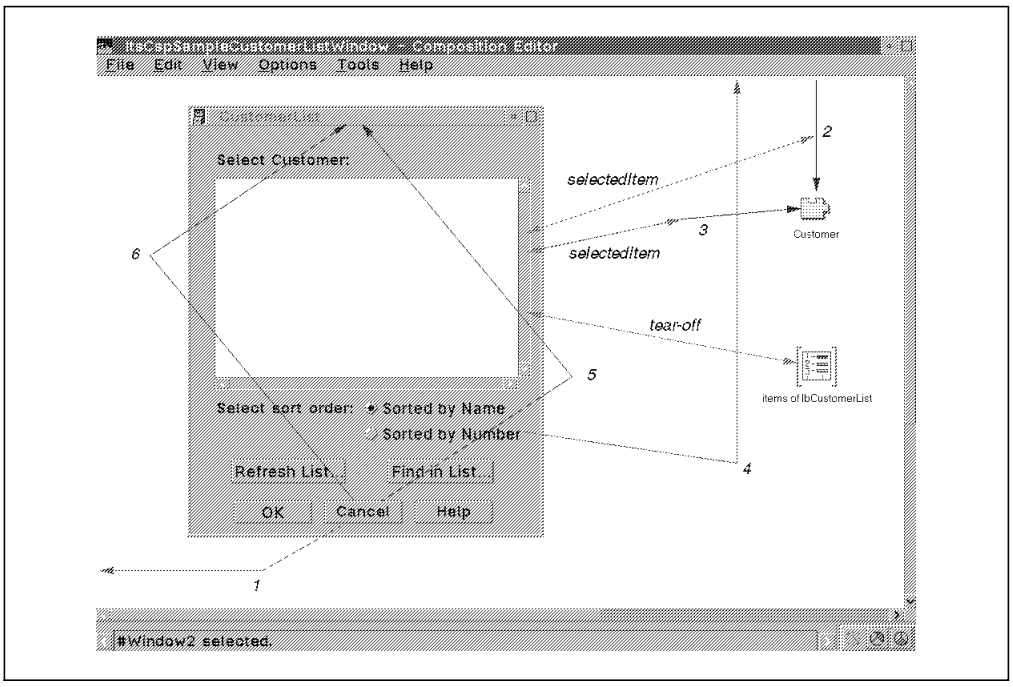

Figure 188. Composition Editor View: ItsCspSampleCustomerListWindow (Part 1)

Figure 189 shows the visual programming required to open the Find in Customer List Window to start a subselection and read the details of a selected customer in the subselection list.

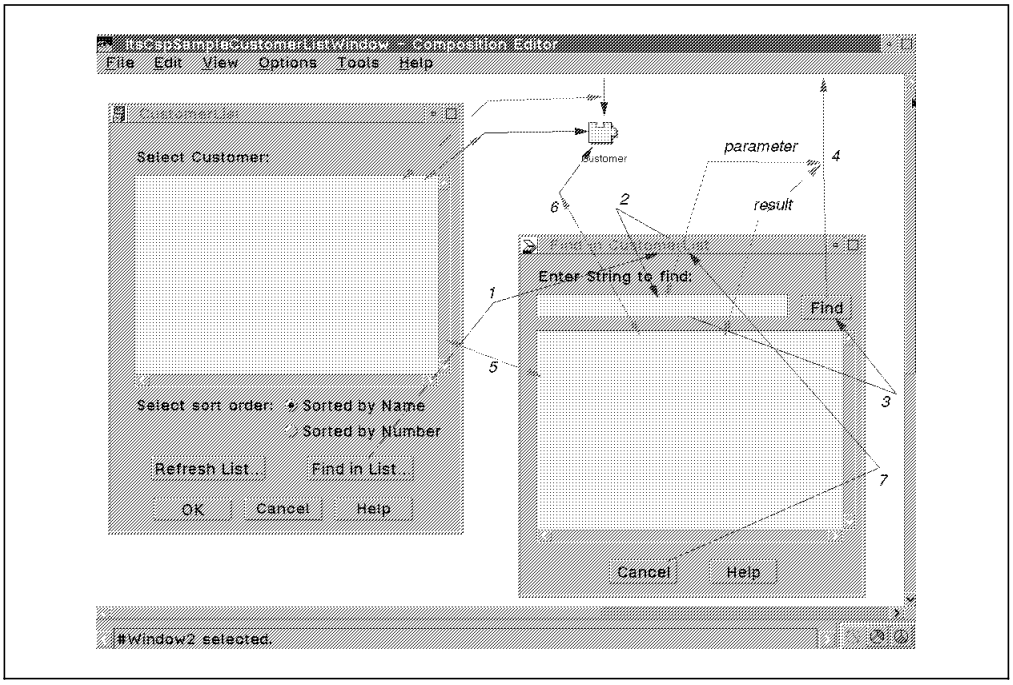

Figure 189. Composition Editor View: ItsCspSampleCustomerListWindow (Part 2)

Figure 190 on page 169 shows the window to refresh the contents of the list box in the Customer List window. The connections are made to the same customer list model object that created the instance of this Customer List window. The pointer to the model object is stored in the Pointer to CustomerList variable, which is a public interface attribute and is filled

during the instantiation with the factory in the Customer List part. The same approach to provide values is used for the NameFrom, NameTo, and Number variables that are connected to the entry fields.

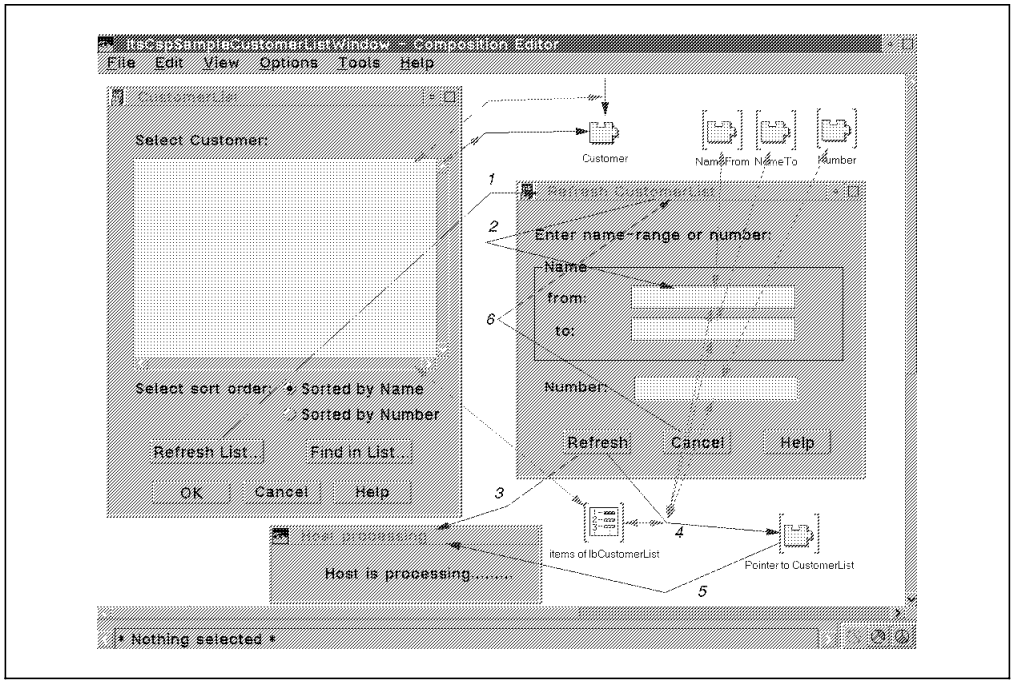

Figure 190. Composition Editor View: ItsCspSampleCustomerListWindow (Part 3)

- **Part assembly**

  Figure 191 shows the part assembly for the Customer List Window part.

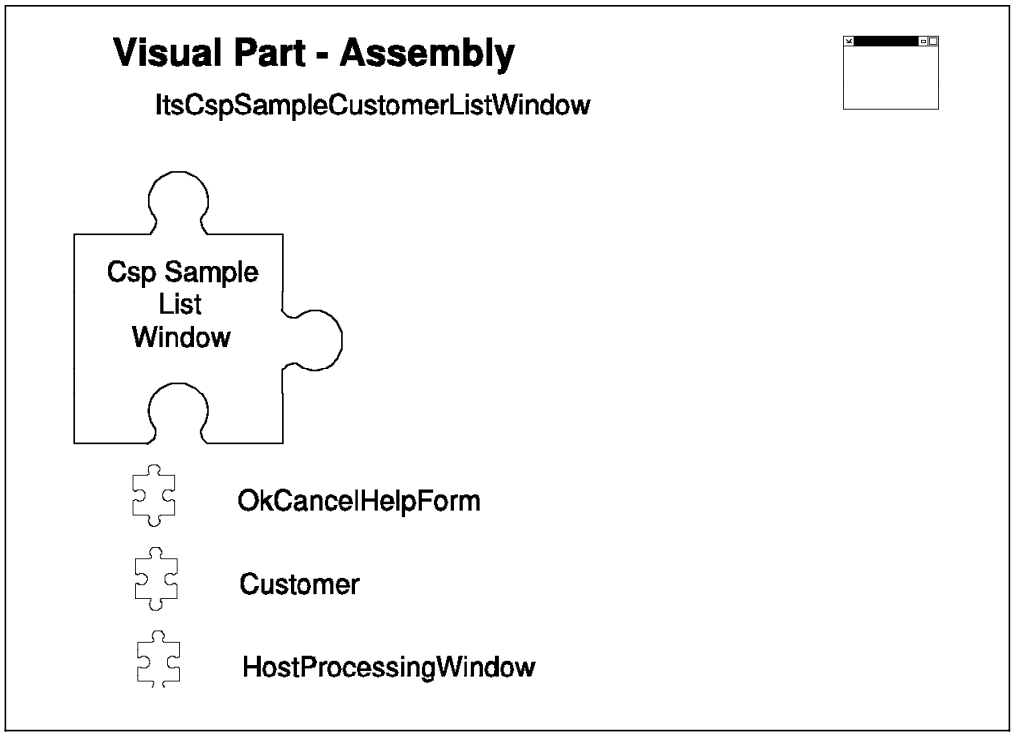

Figure 191. Part Assembly: ItsCspSampleCustomerListWindow

- **Public interface**

Figure 192 shows the public interface for the customer list window part.

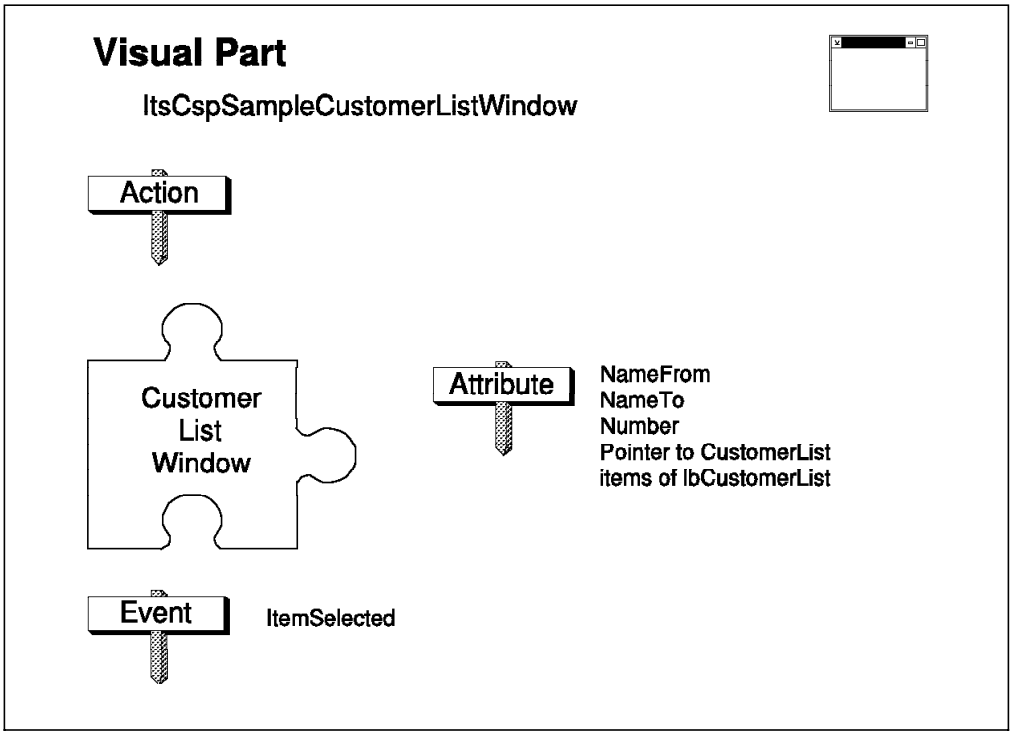

Figure 192. Public Interface: ItsCspSampleCustomerListWindow

- **Used in part**

  None (created dynamically by the factory defined in Customer List)

- **Superclass of**

  None

- **Class definition**

  Figure 193 shows the class definition for the Customer List Window part.

```
AbtAppBldrView subclass: #ItsCspSampleCustomerListWindow
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
```

Figure 193. Class Definition: ItsCspSampleCustomerListWindow

- **Scripts**

  Table 24 shows the scripts for the Customer List Window part.

Table 24 (Page 1 of 2). Scripts: Customer List Window

Method	Description
doLocalSubselect: searchString	Finds the searchString in the list box and returns a sortedCollection with the results.
isItemSelected	Tests whether an item is selected and raises the ItemSelected event if it is.

Table 24 (Page 2 of 2). Scripts: Customer List Window	
Method	**Description**
sortListbox	Sorts the contents of the list box by name or by number depending on the radio button selection and returns the result as a sorted collection in the list box.

Figure 194 shows the doLocalSubselect: method.

```
doLocalSubselect: searchString

   | destinationCollection sourceCollection |

   destinationCollection := OrderedCollection new.

   sourceCollection := self partAttributeValue:
                         #(#'items of lbCustomerList' #self).

   sourceCollection do: [ :each |
      (each abrIncludesString: searchString) ifTrue:
         [destinationCollection add: each].
      ].

   ^destinationCollection.
```

Figure 194. Method: doLocalSubselect

Figure 195 shows the isItemSelected method.

```
isItemSelected

   | anItem |

   anItem := self partAttributeValue: #(#lbCustomerList #selectedItem).

   (anItem isNil) ifTrue:
      [CwMessagePrompter
         errorMessage: 'No Customer selected !!'.
         ^self.
      ].

   ^self signalEvent: #ItemSelected.
```

Figure 195. Method: isItemSelected

Figure 196 on page 172 shows the sortListbox method.

```
sortListbox

   | aSortedCollect selectionIndex |

    aSortedCollect := self partAttributeValue:
                        #(#'items of lbCustomerList' #self).

   selectionIndex := self partAttributeValue:
                        #(#rbListSortOrder #selectionIndex).
   selectionIndex = 1 ifTrue:
      [aSortedCollect sortBlock: [ :a :b |
         ((a copyFrom: 9 to: 30) trimBlanks)
         <=
         ((b copyFrom: 9 to: 30) trimBlanks)].
      ].
   selectionIndex = 2 ifTrue:
      [aSortedCollect sortBlock: [ :a :b |
         (a copyFrom: 1 to: 7)
         <=
         (b copyFrom: 1 to: 7)].
      ].

   ^self partAttributeValue: #(#'items of lbCustomerList' #self)
         put: aSortedCollect.
```

Figure 196. Method: sortListbox

- **Event trace**

  Table 25 shows part 1 of the event trace for the Customer List Window part.

Table 25. Event Trace: Customer List Window (Part 1)

User Action or Event	Sequence of Executed Connections
User action: click on OK push button	(1) pbOkPressed (OkCancelHelpForm) >> isItemSelected (hook)
Event: ItemSelected from the hook after OK push button was pressed	(2) event: ItemSelected >> readNewCustomerWithId: (selectedItem)
User action: double-click on item in the list box	(3) defaultActionRequested >> readNewCustomerWithId: (selectedItem)
User action: change the radio button for the sort order	(4) selectedItemChanged >> sortListbox (hook)
User action: click on Cancel push button	(5) pbCancelPressed (OkCancelHelpForm) >> closeWidget
User action: click on Cancel push button	(6) pbCancelPressed (OkCancelHelpForm) >> destroyPart (because it was dynamically created)

Table 26 on page 173 shows part 2 of the event trace for the Customer List Window part.

Table 26. Event Trace: Customer List Window (Part 2)	
User Action or Event	**Sequence of Executed Connections**
User action: click on FindInList... push button	(1) clicked >> openOwnedWidget (Find in CustomerList)
Event: openedWidget (initialization for the window)	(2) event: openedWidget >> setFocus (entry field)
User action: press Enter on entry field	(3) defaultActionRequested >> click (Find push button)
User action: click on Find push button	(4) clicked >> doLocalSubselect: (entry field) and put result into the list box
User action: click on item in the list box to mark the item	(5) selectedItem (customerListbox) >> selectedItem (findListbox) (attribute-to-attribute connection)
User action: double-click on item in the list box to select and open detail	(6) defaultActionRequested >> readNewCustomerWithId: (selectedItem)
User action: click on Cancel push button	(7) clicked (pbCancel) >> closeWidget Find in CustomerList

Table 27 shows part 3 of the event trace for the Customer List Window part.

Table 27. Event Trace: Customer List Window (Part 3)	
User Action or Event	**Sequence of Executed Connections**
User action: click on Refresh List... push button	(1) clicked >> openOwnedWidget (Refresh CustomerList)
Event: openedWidget (initialization for the window)	(2) event: openedWidget >> setFocus (NameFrom entry field)
User action: click on Refresh push button	• (3) clicked >> openWidget (HostProcessingWindow) • (4) clicked >> refreshList (CustomerList action) with NameFrom, NameTo, and Number as parameters and put the result as items of the list box collection • (5) event: CustomerListRefreshed (CustomerList) >> closeWidget (host processing window)
User action: click on Cancel push button	(6) clicked >> closeWidget (Refresh CustomerList)

- **Special comments**

  The model-view separation concept requires a solution to link the view objects to the model objects. In the customer list implementation, the customer list object exists as a single instance in the application. It creates new Customer List Windows that must know the model object (customer list). Therefore, we want to keep a pointer to the model object in the view object.

How to Keep a Pointer to a Model Object

- Define a variable in the free form surface of the view object and add the variable to the public interface.
- During the instantiation of a new view object in the model object, assign the value #self to the public interface variable of the view object created with a factory.

Another useful hint is to synchronize two list boxes that show the same ordered collection as items. In our part, one list box shows all items, and the other list box shows a subset of items. If we wanted to synchronize the selection in the two list boxes, that is, the user clicks on an item in one list box and the same item is selected in the other list box, we would have used the visual programming described in the box below.

How to Synchronize Selection in Two List Boxes

- To synchronize the selection of the same item when the two list boxes contain the same items:

  Make an attribute-to-attribute connection #selectedItem >> #selectedItem between the two list boxes.
- To synchronize the selection of the same index when the two list boxes contain different items:

  Make an attribute-to-attribute connection #selectionIndex >> #selectionIndex between the two list boxes.

8.16 Customer

A subpart in the Customer List Window part is the Customer part. The Customer part is a nonvisual part and belongs to the model side of the application. This part is responsible for the customer details and provides actions to read, add, update, and delete the details of a customer. The part is not visible in the running application, but it is involved in all of the customer detail windows.

- **Part name**

  ItsCspSampleCustomer
- **Category**

  Nonvisual composite part
- **Description**

  Communication to the host in this part is started from the SPOC, that is, the CICS screen from which the STLC transaction is started to read, add, update, and delete a customer.

  The sequence on the host is the same for all methods that initiate a read, add, update, or delete function on the host. After the host transaction is started, the customer number is entered, which brings up a single item list where we enter the action code (S = read, U = update, D = delete). The only exception is the method to add a new customer. Because the customer number is unknown, we select a list of all customers with customer name = * and enter the action code N on the first item of the list.

We go directly to the terminal part and write scripts for the host communication.

The customer part provides attributes for all of the fields on the host screen in the public interface. All of these attributes are defined as instance variables, and the get and set methods are generated with VisualAge. The public interface also provides events to inform other parts of the successful execution of a method.

- **Composition Editor view**

  Figure 197 shows the Composition Editor view for the customer part.

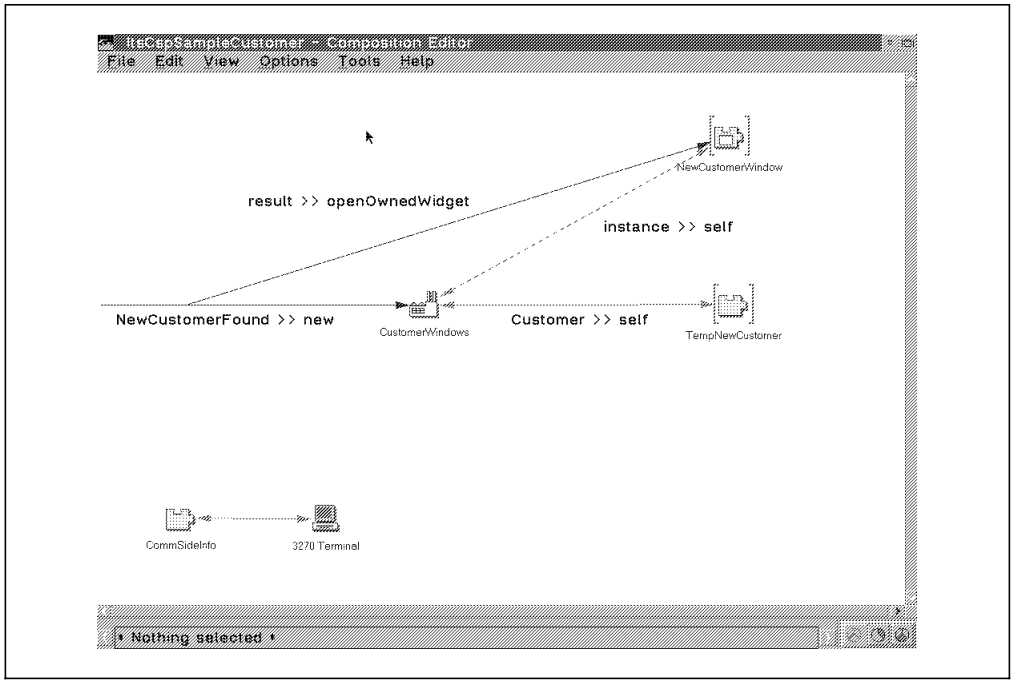

Figure 197. Composition Editor View: ItsCspSampleCustomer

Visual programming is used to implement the instantiation of a new customer window part. This logic is defined in the upper half of the Composition Editor view and is implemented with a factory part. After the readNewCustomerWithId action is executed successfully, the NewCustomerFound event is raised. This is the trigger to create a new instance of a Customer Window part with a customer defined as a variable in the public interface. After the new method, the openOwnedWidget message is sent to open the new instance.

The visual programming logic in the lower half of the Composition Editor view is a 3270 terminal connected to the Communication SideInfo part that keeps the value of the session ID. Instead of adding a separate Abt3270Terminal part to the free form surface, we could also tear off the terminal from the Communication SideInfo part.

The 3270 terminal is used by scripts. The first time a script is executed from a customer instance (readNewCustomerWithId), the 3270 terminal is assigned to the terminal instance variable.

- **Part assembly**

  Figure 198 on page 176 shows the part assembly for the Customer part.

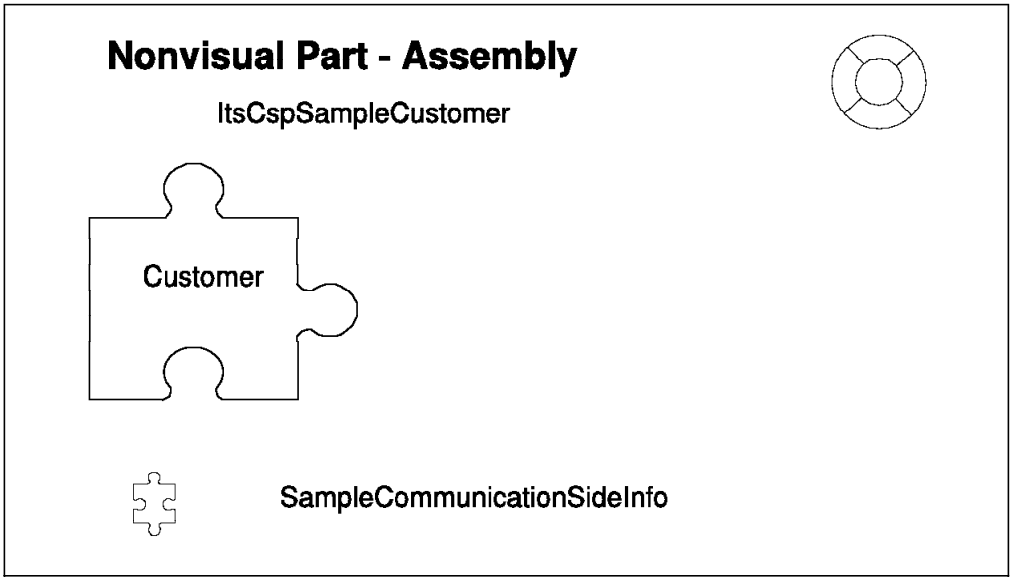

Figure 198. Part Assembly: ItsCspSampleCustomer

- **Public interface**

  Figure 199 shows the public interface for the Customer part.

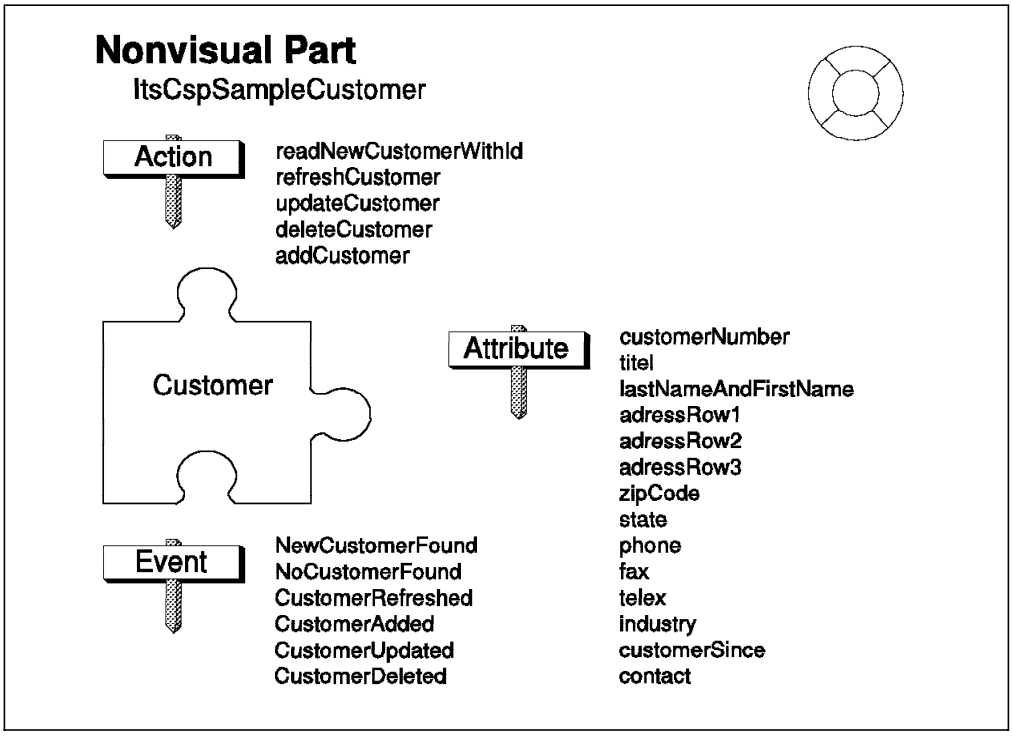

Figure 199. Public Interface: ItsCspSampleCustomer

- **Used in part**
  - ItsCspSampleCustomerListWindow
  - ItsCspSampleNewCustomerWindow
- **Superclass of**

  None
- **Class definition**

Figure 200 shows the class definition for the customer part.

```
AbtAppBldrPart subclass: #ItsCspSampleCustomer
    instanceVariableNames: 'terminal
                            titel
                            lastNameAndFirstName
                            customerRecord
                            zipCode
                            contact
                            customerSince
                            adressRow2
                            telex
                            state
                            phone
                            industry
                            fax
                            customerNumber
                            adressRow3
                            adressRow1 '
    classVariableNames: ''
    poolDictionaries: ''
```

Figure 200. Class Definition: ItsCspSampleCustomer

- **Scripts**

  Table 28 shows the scripts for the customer part.

Table 28. Scripts: Customer

Method	Description
readNewCustomerWithId: anIdString	Reads a customer with the number passed as a substring in the parameter. The string passed as a parameter comes directly from the Customer List Window list box and contains number and name. The NewCustomerFound event is raised if the search for the customer was successful. The NoCustomerFound event is raised if the search for the customer was not successful.
updateCustomer	Updates a customer with the values stored in the instance variables of the Customer part and raises the CustomerUpdated event.
deleteCustomer	Deletes the Customer with the number stored in the instance variable of the Customer part and raises the CustomerDeleted event.
refreshCustomer	Refreshes the customer with the number stored in the instance variable of the Customer part and raises the CustomerRefreshed event.
addCustomer	Adds a customer using the values stored in the instance variables of the Customer part.
startTransaction	Starts transaction STLC. The transaction code is hardcoded here and does not use the capabilities of the communication SideInfo part.

The get and set methods for the instance variables are not mentioned in Table 28, because they are generated with VisualAge and modified to perform a lazy initialization as described in 3.5.1.2, “Generating Scripts for the Attributes” on page 71.

Figure 201 shows the readNewCustomerWithId: method.

```
readNewCustomerWithId: anIdString

   | aFieldDefRecord aScreenRecord aNewCustomer
     theFirstSubStringOfIdString |

   theFirstSubStringOfIdString := anIdString subStrings at: 1.

   terminal := self partAttributeValue: #(#'3270 Terminal' #self).
   self startTransaction.

   "first Selection Map up"
   terminal enter: theFirstSubStringOfIdString
               andWaitForCursorPositionToChangeFrom: 1 @ 1.

   "List with one Item shown"
   terminal keyHome.
   terminal enter: 'S'
               andWaitForCursorPositionToChangeFrom: 1 @ 1.

   "Customer Details Map shown"
   aFieldDefRecord := terminal buildFieldDefsOfType: 'Protected'.
                                                  "Protected Fields"
   aScreenRecord   := terminal buildScreenRecordForFields: aFieldDefRecord.

   aNewCustomer := self class new.

   aNewCustomer   customerNumber:        theFirstSubStringOfIdString;
                  titel:                 (aScreenRecord at:#FieldNumber5);
                  lastNameAndFirstName:  (aScreenRecord at:#FieldNumber7);
                  adressRow1:            (aScreenRecord at:#FieldNumber9);
                  phone:                 (aScreenRecord at:#FieldNumber11);
                  adressRow2:            (aScreenRecord at:#FieldNumber13);
                  fax:                   (aScreenRecord at:#FieldNumber15);
                  adressRow3:            (aScreenRecord at:#FieldNumber17);
                  telex:                 (aScreenRecord at:#FieldNumber19);
                  state:                 (aScreenRecord at:#FieldNumber21);
                  zipCode:               (aScreenRecord at:#FieldNumber23);
                  industry:              (aScreenRecord at:#FieldNumber25);
                  contact:               (aScreenRecord at:#FieldNumber27);
                  customerSince:         (aScreenRecord at:#FieldNumber29).

   "Store terminal for update, delete, refresh"
   aNewCustomer  terminal: terminal.

   terminal keyPF:3.
   terminal keyPF:3.

   "store object in temporary variable to use as factory instance variable"
```

Figure 201 (Part 1 of 2). Method: readNewCustomerWithId

```
   self partAttributeValue: #(#TempNewCustomer #self) put: aNewCustomer.

   "signal Customer found"
   self signalEvent: #NewCustomerFound.

   ^aNewCustomer.
```

Figure 201 (Part 2 of 2). Method: readNewCustomerWithId

Figure 202 shows the updateCustomer method.

```
updateCustomer

   | aFieldDefRecord |

   self startTransaction.

   "first Selection Map up"
   terminal enter: (self customerNumber)
               andWaitForCursorPositionToChangeFrom: 1 @ 1.

   "List with one Item shown"
   terminal keyHome.

   "Input: U for Update on firstLine"
   terminal enter: 'U'
               andWaitForCursorPositionToChangeFrom: 1 @ 1.

   "Customer Details Map shown"
   aFieldDefRecord := terminal buildFieldDefsOfType: 'Unprotected'.

   terminal copyString: (self titel)
               toField: (aFieldDefRecord fieldAt: #FieldNumber1);
            copyString: (self lastNameAndFirstName)
               toField: (aFieldDefRecord fieldAt: #FieldNumber2);
            copyString: (self adressRow1)
               toField: (aFieldDefRecord fieldAt: #FieldNumber3);
            copyString: (self phone)
               toField: (aFieldDefRecord fieldAt: #FieldNumber4);
            copyString: (self adressRow2)
               toField: (aFieldDefRecord fieldAt: #FieldNumber5);
            copyString: (self fax)
               toField: (aFieldDefRecord fieldAt: #FieldNumber6);
```

Figure 202 (Part 1 of 2). Method: updateCustomer

```
            copyString: (self adressRow3)
               toField: (aFieldDefRecord fieldAt: #FieldNumber7);
            copyString: (self telex)
               toField: (aFieldDefRecord fieldAt: #FieldNumber8);
            copyString: (self state)
               toField: (aFieldDefRecord fieldAt: #FieldNumber9);
            copyString: (self zipCode)
               toField: (aFieldDefRecord fieldAt: #FieldNumber10);
            copyString: (self industry)
               toField: (aFieldDefRecord fieldAt: #FieldNumber11);
            copyString: (self contact)
               toField: (aFieldDefRecord fieldAt: #FieldNumber12);
            copyString: (self customerSince)
               toField: (aFieldDefRecord fieldAt: #FieldNumber13).

   terminal enter: ''
               andWaitForCursorPositionToChangeFrom: 1@1.

   terminal keyPF:3.

   ^self signalEvent: #CustomerUpdated.
```

Figure 202 (Part 2 of 2). Method: updateCustomer

Figure 203 shows the deleteCustomer method.

```
deleteCustomer

   | aFieldDefRecord aScreenRecord aMessagebox |

   self startTransaction.

   "first Selection Map up"
   terminal enter: (self customerNumber)
               andWaitForCursorPositionToChangeFrom: 1 @ 1.

   "List with one Item shown"
   terminal keyHome.

   "Enter 'D' for Delete on first line"
   terminal enter: 'D'
               andWaitForCursorPositionToChangeFrom: 1 @ 1.

   "Customer Details Map shown"
   aFieldDefRecord := terminal buildFieldDefsOfType: 'Protected'.
   aScreenRecord   := terminal buildScreenRecordForFields: aFieldDefRecord.

   self  titel:                  (aScreenRecord at:#FieldNumber5);
         lastNameAndFirstName:   (aScreenRecord at:#FieldNumber7);
         adressRow1:             (aScreenRecord at:#FieldNumber9);
```

Figure 203 (Part 1 of 2). Method: deleteCustomer

```
        phone:                  (aScreenRecord at:#FieldNumber11);
        adressRow2:             (aScreenRecord at:#FieldNumber13);
        fax:                    (aScreenRecord at:#FieldNumber15);
        adressRow3:             (aScreenRecord at:#FieldNumber17);
        telex:                  (aScreenRecord at:#FieldNumber19);
        state:                  (aScreenRecord at:#FieldNumber21);
        zipCode:                (aScreenRecord at:#FieldNumber23);
        industry:               (aScreenRecord at:#FieldNumber25);
        contact:                (aScreenRecord at:#FieldNumber27);
        customerSince:          (aScreenRecord at:#FieldNumber29).

   (CwMessagePrompter confirm: ('Delete Customer: "',
            ((self lastNameAndFirstName) trimBlanks),'" ?')
            title: 'Delete Customer')
      ifTrue:
      [
         terminal keyEnter.
         self signalEvent: #CustomerDeleted.
      ]
      ifFalse:
         [terminal keyPF: 3].

   terminal keyPF:3.
```

Figure 203 (Part 2 of 2). Method: deleteCustomer

Figure 204 shows the refreshCustomer method.

```
refreshCustomer

   | aFieldDefRecord aScreenRecord |

   self startTransaction.

   "first Selection Map up"
   terminal enter: (self customerNumber)
               andWaitForCursorPositionToChangeFrom: 1 @ 1.

   "List with one Item shown"
   terminal keyHome.
   terminal enter: 'S'
               andWaitForCursorPositionToChangeFrom: 1 @ 1.

   "Customer Details Map shown"
```

Figure 204 (Part 1 of 2). Method: refreshCustomer

```
    aFieldDefRecord := terminal buildFieldDefsOfType: 'Protected'.
    aScreenRecord   := terminal buildScreenRecordForFields: aFieldDefRecord.

    self  titel:                  (aScreenRecord at:#FieldNumber5);
          lastNameAndFirstName:   (aScreenRecord at:#FieldNumber7);
          adressRow1:             (aScreenRecord at:#FieldNumber9);
          phone:                  (aScreenRecord at:#FieldNumber11);
          adressRow2:             (aScreenRecord at:#FieldNumber13);
          fax:                    (aScreenRecord at:#FieldNumber15);
          adressRow3:             (aScreenRecord at:#FieldNumber17);
          telex:                  (aScreenRecord at:#FieldNumber19);
          state:                  (aScreenRecord at:#FieldNumber21);
          zipCode:                (aScreenRecord at:#FieldNumber23);
          industry:               (aScreenRecord at:#FieldNumber25);
          contact:                (aScreenRecord at:#FieldNumber27);
          customerSince:          (aScreenRecord at:#FieldNumber29).

    terminal keyPF:3.
    terminal keyPF:3.

    ^self signalEvent: #CustomerRefreshed.
```

Figure 204 (Part 2 of 2). Method: refreshCustomer

Figure 205 shows the addCustomer method.

```
addCustomer

    | aFieldDefRecord |

    terminal := self partAttributeValue: #(#'3270 Terminal' #self).

    self startTransaction.

    "first Selection Map up"
    terminal enter: '*'
             andWaitForCursorPositionToChangeFrom: 1 @ 1.

    "List with one Item shown"
    terminal keyHome.

    "Input: N for New on firstLine"
    terminal enter: 'N'
             andWaitForCursorPositionToChangeFrom: 1 @ 1.

    "Customer Details Map shown"
    aFieldDefRecord := terminal buildFieldDefsOfType: 'Unprotected'.
                                    "Unprotected Fields"

    terminal copyString: (self titel)
```

Figure 205 (Part 1 of 2). Method: addCustomer

```
               toField: (aFieldDefRecord fieldAt: #FieldNumber1);
            copyString: (self lastNameAndFirstName)
               toField: (aFieldDefRecord fieldAt: #FieldNumber2);
            copyString: (self adressRow1)
               toField: (aFieldDefRecord fieldAt: #FieldNumber3);
            copyString: (self phone)
               toField: (aFieldDefRecord fieldAt: #FieldNumber4);
            copyString: (self adressRow2)
               toField: (aFieldDefRecord fieldAt: #FieldNumber5);
            copyString: (self fax)
               toField: (aFieldDefRecord fieldAt: #FieldNumber6);
            copyString: (self adressRow3)
               toField: (aFieldDefRecord fieldAt: #FieldNumber7);
            copyString: (self telex)
               toField: (aFieldDefRecord fieldAt: #FieldNumber8);
            copyString: (self state)
               toField: (aFieldDefRecord fieldAt: #FieldNumber9);
            copyString: (self zipCode)
               toField: (aFieldDefRecord fieldAt: #FieldNumber10);
            copyString: (self industry)
               toField: (aFieldDefRecord fieldAt: #FieldNumber11);
            copyString: (self contact)
               toField: (aFieldDefRecord fieldAt: #FieldNumber12);
            copyString: (self customerSince)
               toField: (aFieldDefRecord fieldAt: #FieldNumber13).

   terminal keyEnter.
   terminal keyPF:3.

   ^self signalEvent: #CustomerAdded.
```

Figure 205 (Part 2 of 2). Method: addCustomer

Figure 206 shows the startTransaction method.

```
startTransaction

   terminal keyHome;
            enterCommand: 'STLC'.
```

Figure 206. Method: startTransaction

Figure 207 shows the titel method.

```
titel
    "Return the value of titel."

   titel isNil ifTrue:
      [self titel: ''].
   ^titel
```

Figure 207. Method: titel

Figure 208 on page 184 shows the titel: method.

```
titel: aString
    "Save the value of titel."

    titel := (aString trimBlanks).
    self signalEvent: #titel
         with: aString.
```

Figure 208. Method: titel:

The get and set methods shown in Figure 207 on page 183 and Figure 208 are examples of the generated methods to read and write an instance variable. The other instance variables have the analogous get and set methods.

- **Event trace**

  Table 29 shows the event trace for the customer part.

Table 29. Event Trace: Customer

Event	Sequence of Executed Connections
event: NewCustomerFound (raised from hook readNewCustomerWithId)	• event: NewCustomerFound >> new (factory of CustomerWindows) • result of new >> openOwnedWidget

- **Special comments**

  In the Customer part, we write directly to the EHLLAPI interface from the terminal part. We want to show different ways to build record objects that can be used to access the terminal presentation space.

  How to Read from the Terminal Presentation Space

  The easiest way to read from the terminal presentation space is to use methods of the Abt3270Terminal part to build a record object that contains the protected fields. We can address a field at a specific offset with a defined length in the presentation space through the field names (#FieldNumberX) as shown in the following code fragment:

```
aFieldDefRecord := terminal buildFieldDefsOfType: 'Protected'.
aScreenRecord   := terminal buildScreenRecordForFields: aFieldDefRecord.

self titel: (aScreenRecord at:#FieldNumber5).
```

  How to Write to the Terminal Presentation Space

  We build a record object that contains all unprotected fields of the host map. We can address a field at a specific offset with a defined length in the presentation space through the field names (#FieldNumberX) as shown in the following code fragment:

```
aFieldDefRecord := terminal buildFieldDefsOfType: 'Unprotected'.

terminal copyString: (self titel)
            toField: (aFieldDefRecord fieldAt: #FieldNumber1);
```

To update the fields on the host map, the Abt3270Terminal part provides the copyString: toField: method. This method can be used to write to the fields in the presentation space. The copyString: toField: method is implemented such that fields are overwritten and not padded with blanks. We decided to change the method to pad the fields with blanks as shown in the box below.

Change to the copyString: toField: Method

```
copyString: aString toField: aFieldDefItem

   "Write the string on the host screen at the location and
    length specified by aFieldDefItem."

   | rc |

   rc := self doHostPS: [
      self copy: (aString abrPadWithBlanks:(aFieldDefItem length))
           toFieldAtOffset: aFieldDefItem offset + 1].
   ^self handleRC: rc locus: #copyString:
                       toField: ifNonZero: [ self lastError ].

   "------------------ ORIGINAL CODE -----------------------"
   "
   rc := self doHostPS: [
      self copy: aString
           toFieldAtOffset: aFieldDefItem offset + 1].
   ^self handleRC: rc locus: #copyString:
                      toField: ifNonZero: [ self lastError ].
   "
   "------------------END OF ORIGINAL CODE -----------------"
```

How to Create a Model Object

The best way to create model objects is a step-by-step approach using the VisualAge generate default scripts function as follows:

- Add all attributes that represent the data of your model object to the public interface.
- Select the generate default scripts function in the File menu and generate instance variables and the methods to get and set the instance variables.
- Modify the get method to provide a lazy initialization for the instance variables.

8.17 Customer Window

The windows created by the Customer part are instances of the Customer Window part. The Customer Window part belongs to the view side and contains the details of a customer in a notebook. The window presents all instance variables of the customer object that are defined as public interface attributes in the entry fields.

An additional function in the GUI application is to start a PWS editor to write a letter to the selected customer and pass the data of the customer as an address to the letter.

- **Part name**

  ItsCspSampleCustomerWindow

- **Category**

  Visual composite part

- **Description**

  This part again demonstrates the power of visual programming. Only the method to write to the letter file is implemented with a script.

  The Customer Window part contains a customer model object in a variable. This variable is defined in the public interface and is filled during instantiation of a Customer Window.

- **Composition Editor view**

  We show the part logic in two different pictures where only half of the connections are visible and the others are hidden, because there are many connections between the different parts in the Composition Editor view.

  Figure 209 shows the logic required to initialize the window, start the editor with the program starter, and close and destroy the part.

  The subpart InformationLine is a bit difficult to recognize in the picture. It is positioned above the push buttons and is selected, as the message at the bottom of the Composition Editor indicates.

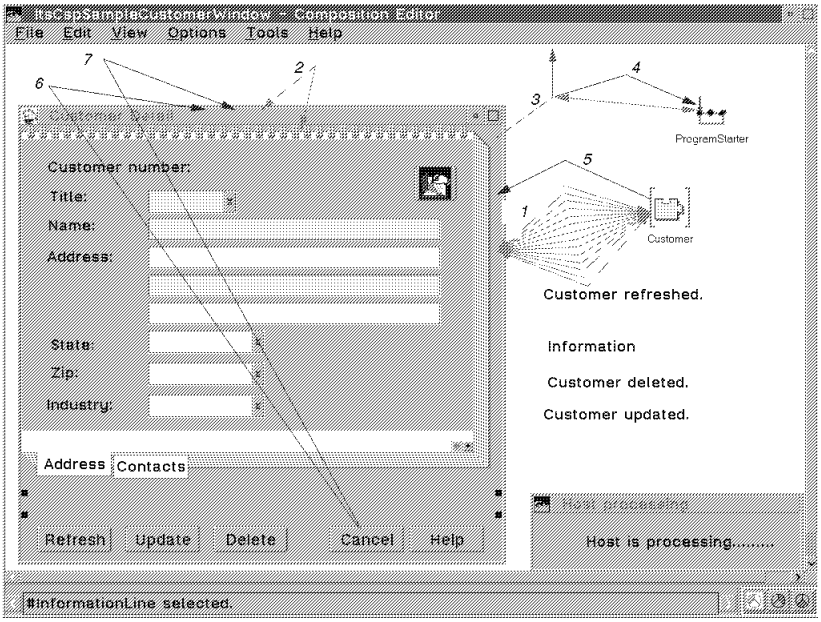

Figure 209. Composition Editor View: ItsCspSampleCustomerWindow (Part 1)

Figure 210 on page 187 contains the logic to trigger the refresh, update, and delete actions of the customer object each time the user clicks on the respective push button. The information line in the window provides feedback for the user about the execution of the requested action.

The host processing window is shown during host processing and hidden when the host is idle. This part should be locked when the Customer part is

communicating with the host. This locking function was not implemented in our sample application.

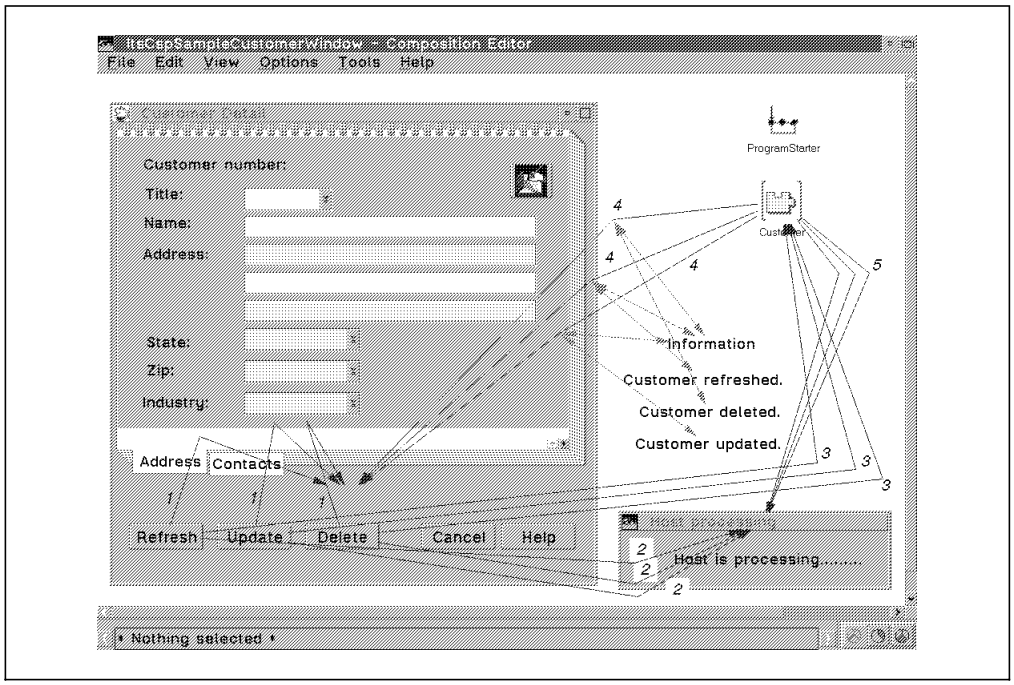

Figure 210. Composition Editor View: ItsCspSampleCustomerWindow (Part 2)

- **Part assembly**

  Figure 211 on page 188 shows the part assembly for the Customer Window part.

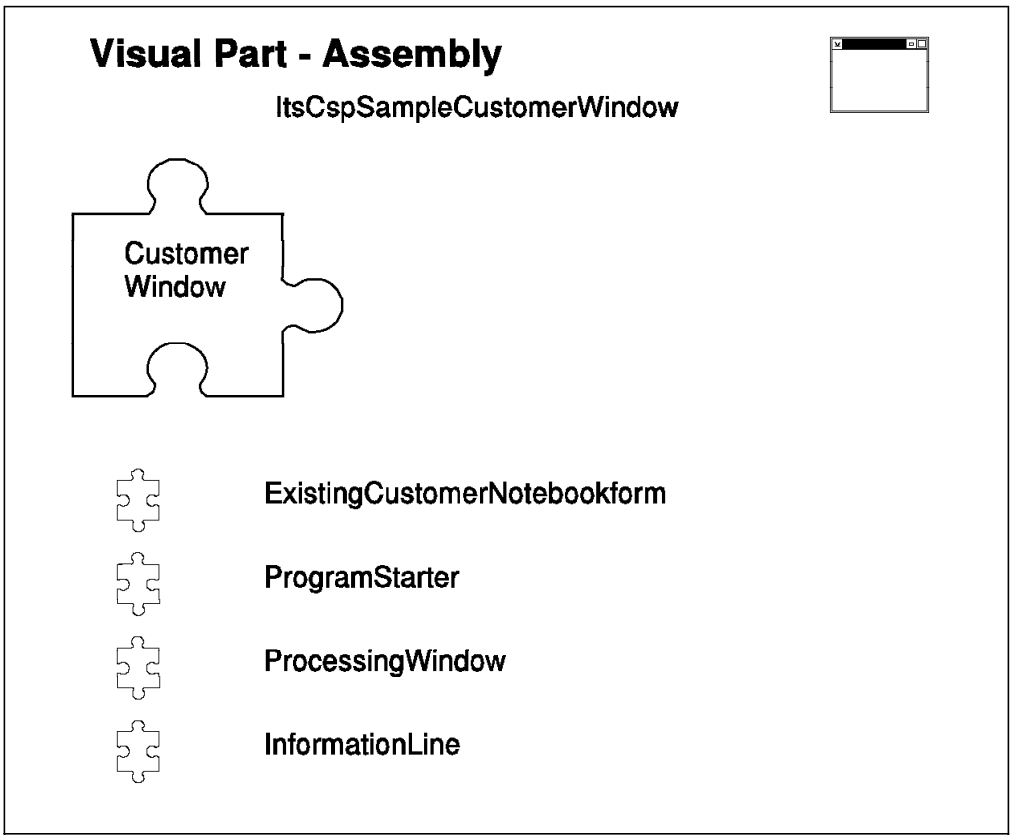

Figure 211. Part Assembly: ItsCspSampleCustomerWindow

- **Public interface**

  Figure 212 on page 189 shows the public interface for the Customer Window part.

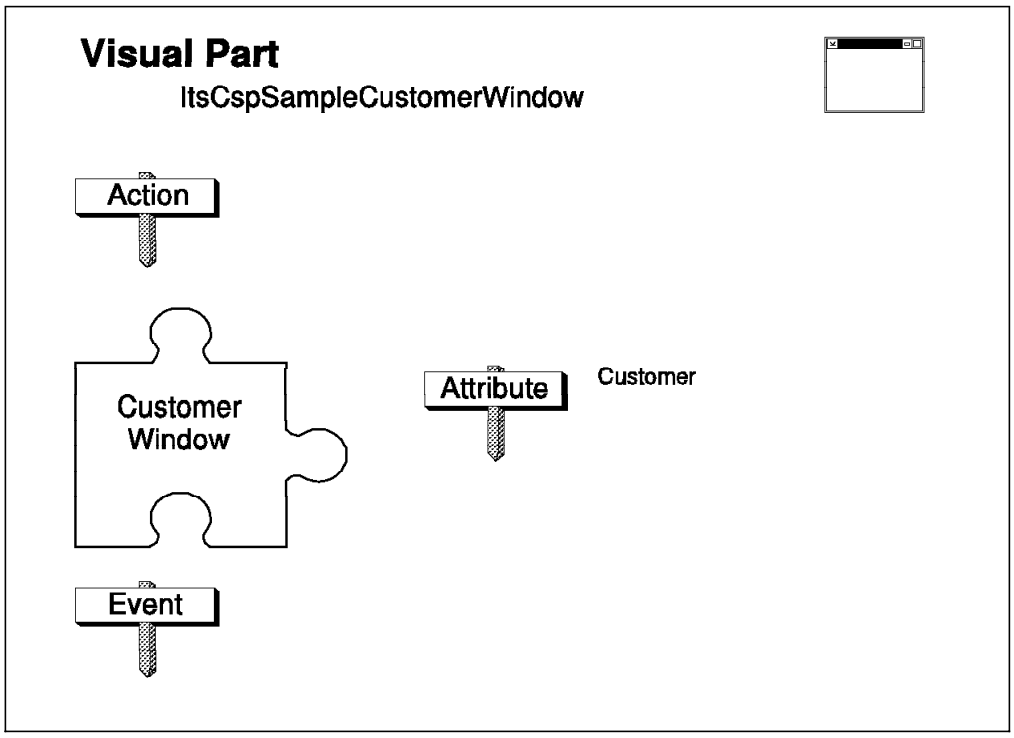

Figure 212. Public Interface: ItsCspSampleCustomerWindow

- **Used in part**

  None (created dynamically by the factory in the customer part)

- **Superclass of**

  None

- **Class definition**

  Figure 213 shows the class definition for the Customer Window part.

```
AbtAppBldrView subclass: #ItsCspSampleCustomerWindow
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
```

Figure 213. Class Definition: ItsCspSampleCustomerWindow

- **Scripts**

  Table 30 shows the scripts for the Customer Window part.

Table 30. Scripts: Customer Window

Method	Description
openFile	Opens a file dialog and writes the address of the customer to the file. The name of the file is returned.

Figure 214 on page 190 shows the openFile method.

```
openFile

   | dialog file fileStream text |

   dialog := CwFileSelectionPrompter new initialize;
               searchMask: '*.TXT'.
   file := dialog prompt.

   fileStream := CfsWriteFileStream openEmpty: file.

   fileStream
      nextPutAll: 'Letter';
      cr;
      nextPutAll: '======';
      cr;cr;cr;
      nextPutAll:
      (self partAttributeValue: #(#Customer #titel)) ;
      cr;
      nextPutAll:
         (self partAttributeValue: #(#Customer #lastNameAndFirstName));
      cr;
      nextPutAll: (self partAttributeValue: #(#Customer #adressRow1));
      cr;
      nextPutAll: (self partAttributeValue: #(#Customer #adressRow2));
      cr;
      nextPutAll: (self partAttributeValue: #(#Customer #adressRow3));
      cr;
      nextPutAll: (self partAttributeValue: #(#Customer #state)),
         (self partAttributeValue: #(#Customer #zipCode));
      cr;cr;cr;
      nextPutAll: 'Phone: ',
         (self partAttributeValue: #(#Customer #phone));
      cr;cr;cr;cr;
      nextPutAll: 'Dear ',( self partAttributeValue: #(#Customer #titel)),
         ' ',
         (self partAttributeValue: #(#Customer #lastNameAndFirstName)),
         ' .......';
      cr;cr.

   fileStream close.

   ^file.
```

Figure 214. Method: openFile

- **Event trace**

  Table 31 shows the event trace (part 1) for the customer window part.

Table 31 (Page 1 of 2). Event Trace: Customer Window (Part 1)

	Sequence of Executed Connections
Initialization and synchronization of notebook	(1) attribute-to-attribute connections between notebook entry fields and customer instance variables to link the values

Table 31 (Page 2 of 2). Event Trace: Customer Window (Part 1)	
	Sequence of Executed Connections
Initialization and synchronization of title bar	(2) attribute-to-attribute connection between name entry field and title text to link the values
User action: click on Write Letter push button	• (3) pbWriteLetterPressed (ExistingCustomerNotebook) >> openFile (hook) • (4) result of openFile (hook) >> action: startProgram (ProgramStarter) 'EPM.EXE' and pass the file name (result of 3) as a parameter
user action: click on Cancel push button	• (6) clicked >> closeWidget • (7) clicked >> destroyPart (because the part was created dynamically)

Table 32 shows the event trace (part 2) for the customer window part.

Table 32 (Page 1 of 2). Event Trace: Customer Window (Part 2)	
User Action	**Sequence of Executed Connections**
User action: click on Refresh push button	• (1) clicked >> initializeInformationLine • (2) clicked >> openWidget (HostProcessingWindow) • (3) clicked >> refreshCustomer (Customer variable) • (4) event: CustomerRefreshed (Customer variable) >> writeInformationLine (Label strings: 'Information', 'Customer refreshed') • (5) event: CustomerRefreshed (Customer variable) >> closeWidget (HostProcessingWindow)
User action: click on Update push button	• (1) clicked >> initializeInformationLine • (2) clicked >> openWidget (HostProcessingWindow) • (3) clicked >> updateCustomer (Customer variable) • (4) event: CustomerUpdated (Customer variable) >> writeInformationLine (Label strings: 'Information', 'Customer updated') • (5) event: CustomerUpdated (Customer variable) >> closeWidget (HostProcessingWindow)

Table 32 (Page 2 of 2). Event Trace: Customer Window (Part 2)	
User Action	**Sequence of Executed Connections**
User action: click on Delete push button	• (1) clicked >> initializeInformationLine • (2) clicked >> openWidget (HostProcessingWindow) • (3) clicked >> deleteCustomer (Customer variable) • (4) event: CustomerDeleted (Customer variable) >> writeInformationLine (Label strings: 'Information', 'Customer deleted') • (5) event: CustomerDeleted (Customer variable) >> closeWidget (HostProcessingWindow) • (5 of part 1) event: CustomerDeleted (Customer variable) >> initializeFields (ExistingCustomerNotebook)

- **Special comments**

  We write to a sequential file in this part. The methods to open a file selection dialog and to write to the file are shown in the box below.

How to Write to a Sequential File

- Select a file name with a file dialog:

```
dialog := CwFileSelectionPrompter new initialize;
              searchMask: '*.TXT'.
file := dialog prompt.
```

- Open the file in overwrite mode:

```
fileStream := CfsWriteFileStream openEmpty: file.
```

- Write a line to the file:

```
fileStream nextPutAll: 'Text......';
           cr.
```

- Close the file:

```
fileStream close.
```

8.18 Customer Notebook Form

In the Customer Window part, the ExistingCustomerNotebookForm is a subpart inside the window. This subpart is an inherited part from the Customer Notebook Form described here. The idea of this abstract part is to provide a common view, common logic, and common public interface definitions for the specialized parts: ExistingCustomerNotebookForm and NewCustomerNotebookForm. This part shows that the inheritance mechanism in VisualAge extends to the public interface definition.

Because this part is an abstract part, used for inheritance, it is not directly visible in the running application.

- **Part name**

  ItsCspSampleCustomerNotebookForm

- **Category**

Abstract visual part

- **Description**

  This part provides the logic to show the actual date in a date entry field and to clear all entry fields.

- **Composition Editor view**

  Figure 215 shows the variables that are added as attributes to the public interface. The variables are created using the tear-off function from the entry fields. The variables on the right-hand side are connected to the entry fields on the second page of the notebook, and we do not show the connections.

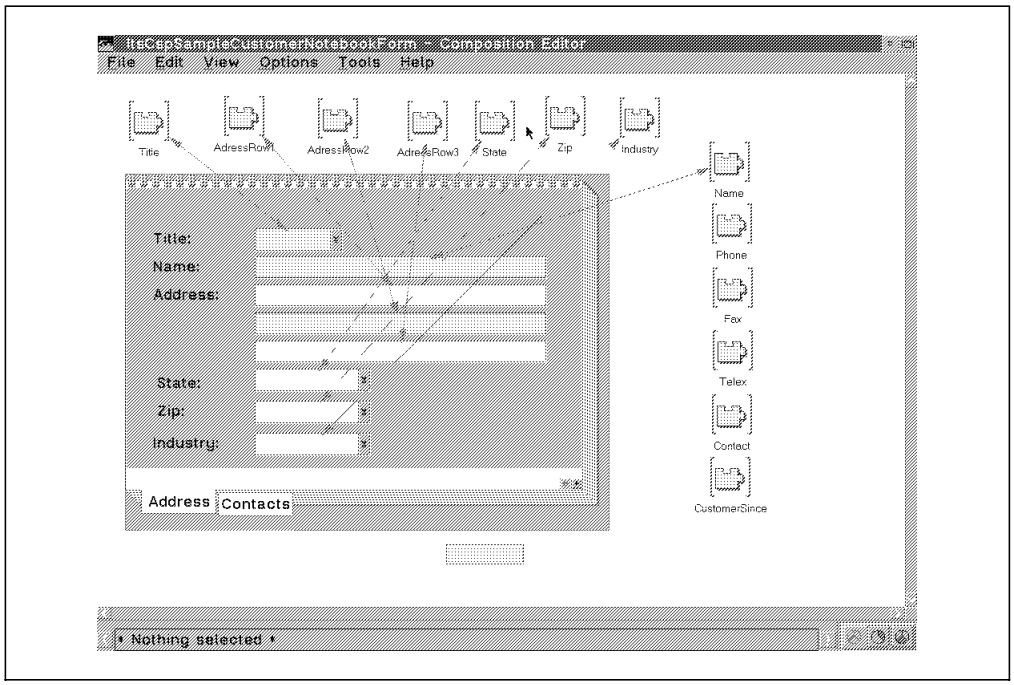

Figure 215. Composition Editor View: ItsCspSampleCustomerNotebookForm (Part 1)

Figure 216 on page 194 shows the initialization for the entry fields.

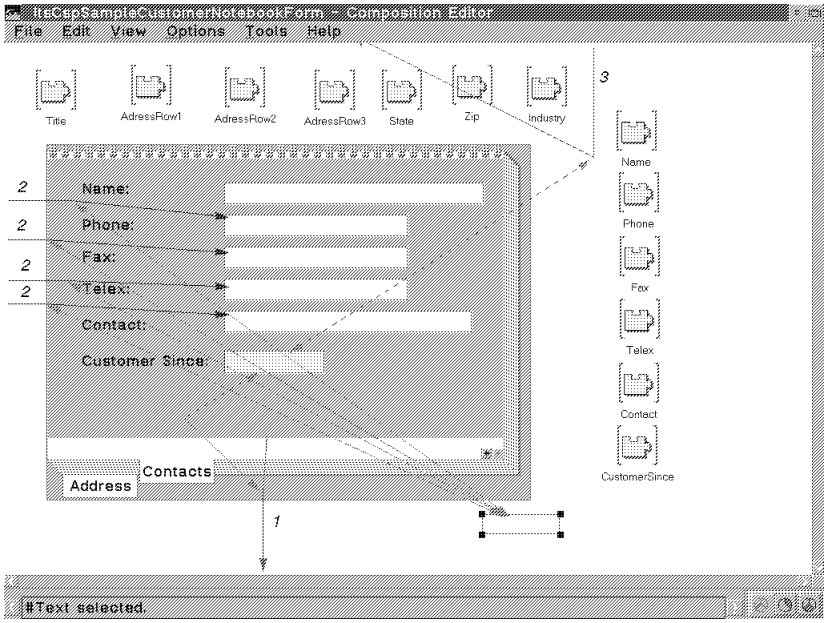

Figure 216. Composition Editor View: ItsCspSampleCustomerNotebookForm (Part 2)

Figure 217 shows the sequence of connections executed when the #InitializeFields event is raised. The event is raised by the initializeFields action.

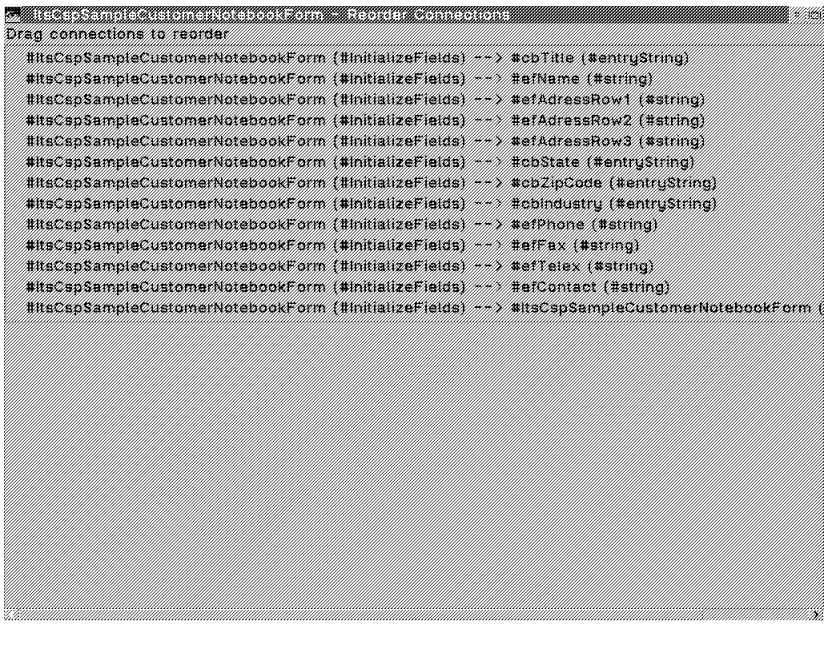

Figure 217. Composition Editor View: ItsCspSampleCustomerNotebookForm (Part 3)

- **Part assembly**

  None

- **Public interface**

Figure 218 shows the public interface for the Customer Notebook Form.

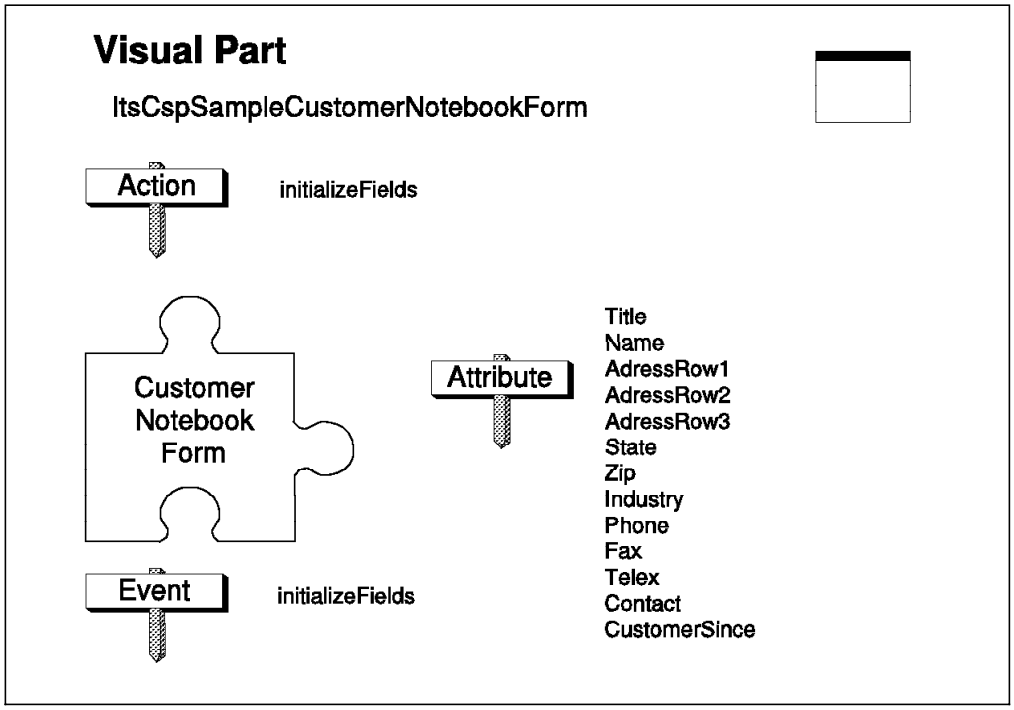

Figure 218. Public Interface: ItsCspSampleCustomerNotebookForm

- **Used in part**

  None (abstract part used for inheritance)

- **Superclass of**

  Figure 219 on page 196 shows the inheritance hierarchy for the Customer Notebook Form.

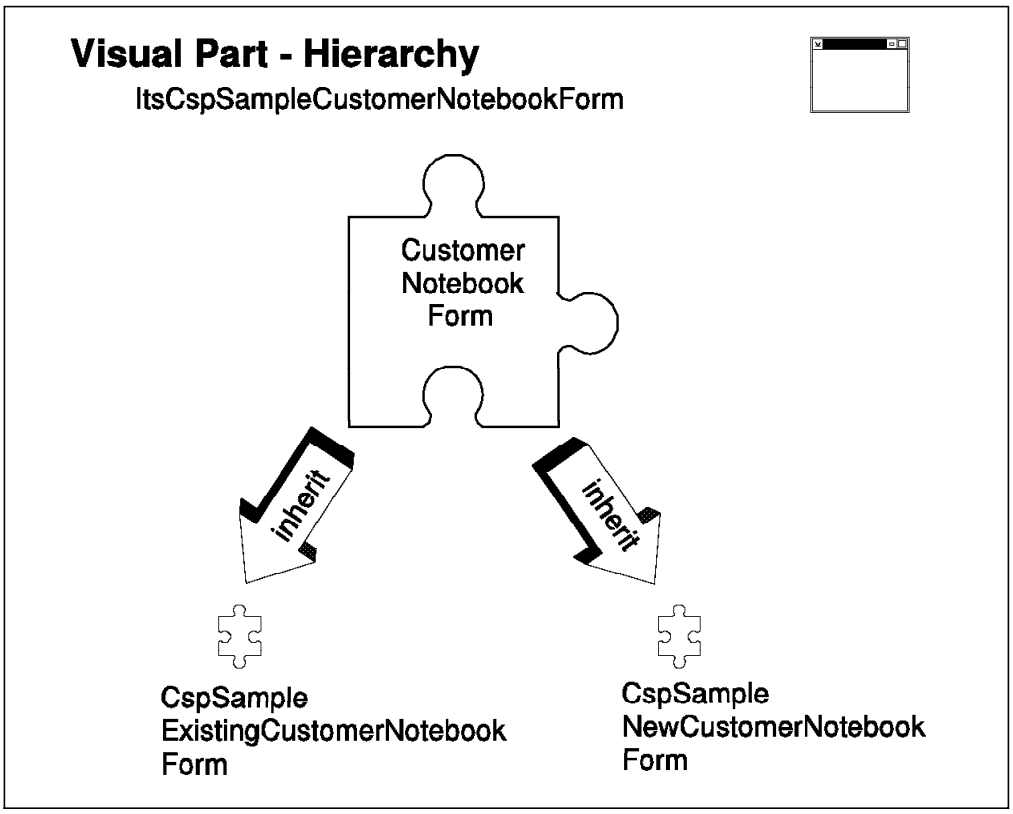

Figure 219. Inheritance Hierarchy: ItsCspSampleCustomerNotebookForm

- **Class definition**

  Figure 220 shows the class definition for the Customer Notebook Form.

```
AbtAppBldrView subclass: #ItsCspSampleCustomerNotebookForm
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
```

Figure 220. Class Definition: ItsCspSampleCustomerNotebookForm

- **Scripts**

  Table 33 shows the scripts for the Customer Notebook Form.

Table 33. Scripts: Customer Notebook Form

Method	Description
actualDate	Returns the actual date of today.
initializeFields	Raises the InitializeFields event.

Figure 221 on page 197 shows the actualDate method.

```
actualDate

   ^Date today.
```

Figure 221. Method: actualDate

Figure 222 shows the initializeFields method.

```
initializeFields

   self signalEvent: #InitializeFields.
```

Figure 222. Method: initializeFields

- **Event trace**

  Table 34 shows the event trace for the Customer Notebook Form.

Table 34. Event Trace: Customer Notebook Form

Events	Sequence of Executed Connections
Part initialization	(1) aboutToOpenWidget >> actualDate (hook), pass returned date to date field
InitializeFields (triggered from the part where this part is used)	(2) event: InitializeFields >> see Figure 217 on page 194 for the connections
InitializeFields (triggered from the part where this part is used)	(3) event: InitializeFields >> actualDate (hook) (see also Figure 217 on page 194 for the connections)

- **Special comments**

  We show how an external action can trigger follow-up events inside the part (InitializeFields action -> InitializeFields event).

  How to Initialize Entry Fields with Blanks

  The following sequence demonstrates how entry fields can be initialized using visual programming:

  1. Add an entry field with a blank string to the free form surface in the Composition Editor.
  2. Connect this entry field to the parameter of the entry field action string, which sets the contents of an entry field.

8.19 Existing Customer Notebook Form

The Existing Customer Notebook Form is a specialization of the Customer Notebook Form. It is visible inside the Customer Window part, where it is used as a subpart.

- **Part name**

  ItsCspSampleExistingCustomerNotebookForm

- **Category**

  Composite visual part

- **Description**

This part is created as a subclass of the Customer Notebook Form. A Customer Number Form part is added to the part and to the public interface of the part.

The push buttons to start writing a letter, initiate a phone call, and start a FAX application are added to the notebook. The events to be raised when the user clicks on one of those push buttons are defined in the public interface.

- **Composition Editor view**

  Figure 223 shows the first part of the Composition Editor view for the Existing Customer Notebook Form.

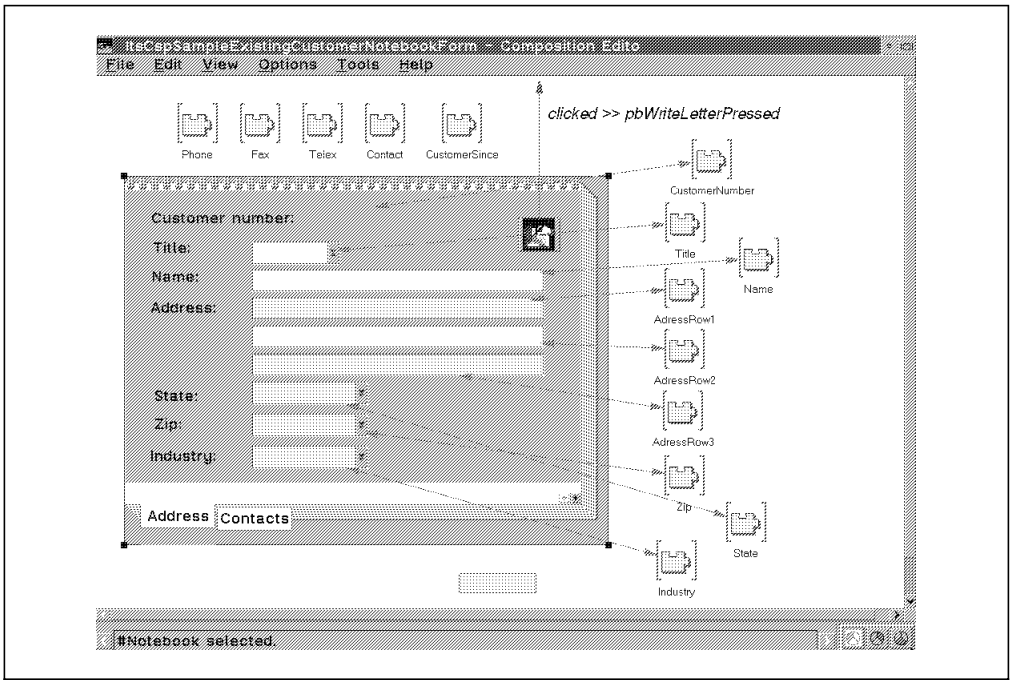

Figure 223. Composition Editor View: ItsCspSampleExistingCustomerNotebookForm (Part 1)

Figure 224 on page 199 shows the second part of the Composition Editor view for the Existing Customer Notebook Form.

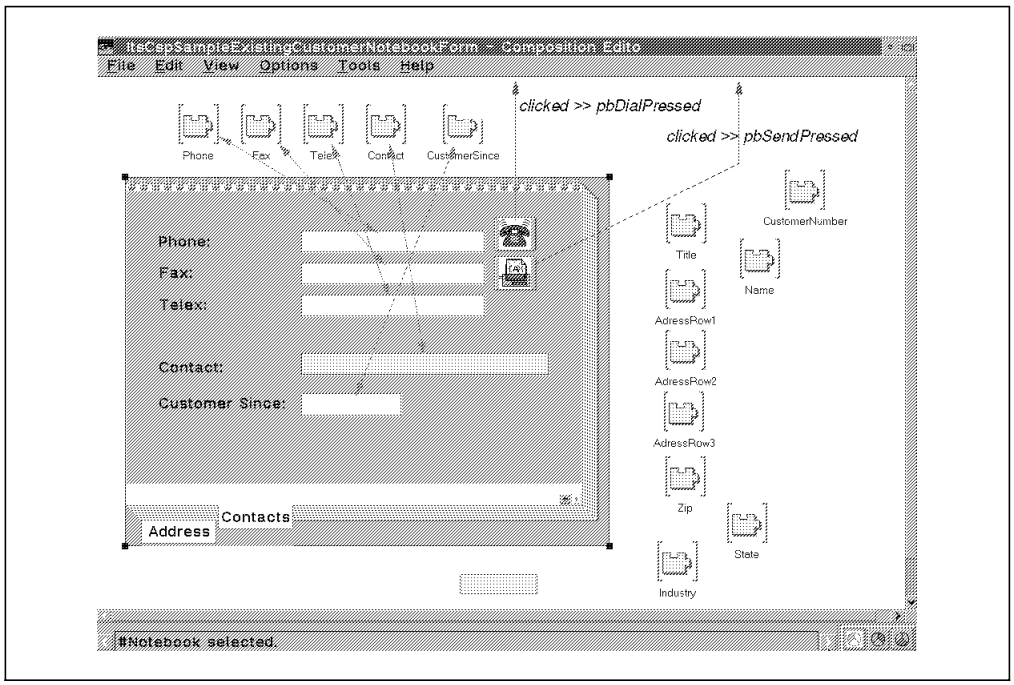

Figure 224. Composition Editor View: ItsCspSampleExistingCustomerNotebookForm (Part 2)

- **Part assembly**

  Figure 225 shows the part assembly for the Existing Customer Notebook Form.

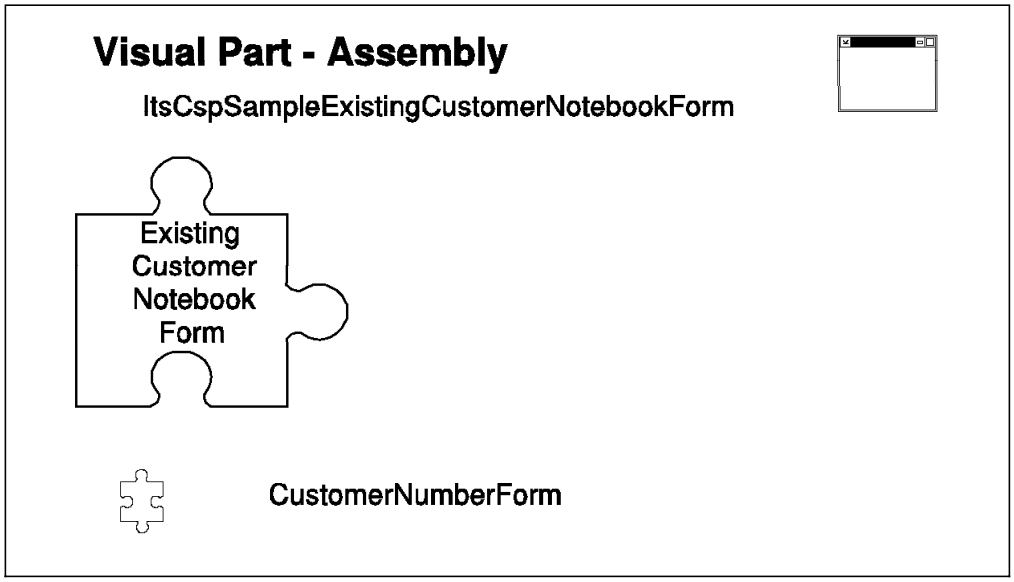

Figure 225. Part Assembly: ItsCspSampleExistingCustomerNotebookForm

- **Public interface**

  Figure 226 on page 200 shows the public interface for the Existing Customer Notebook Form.

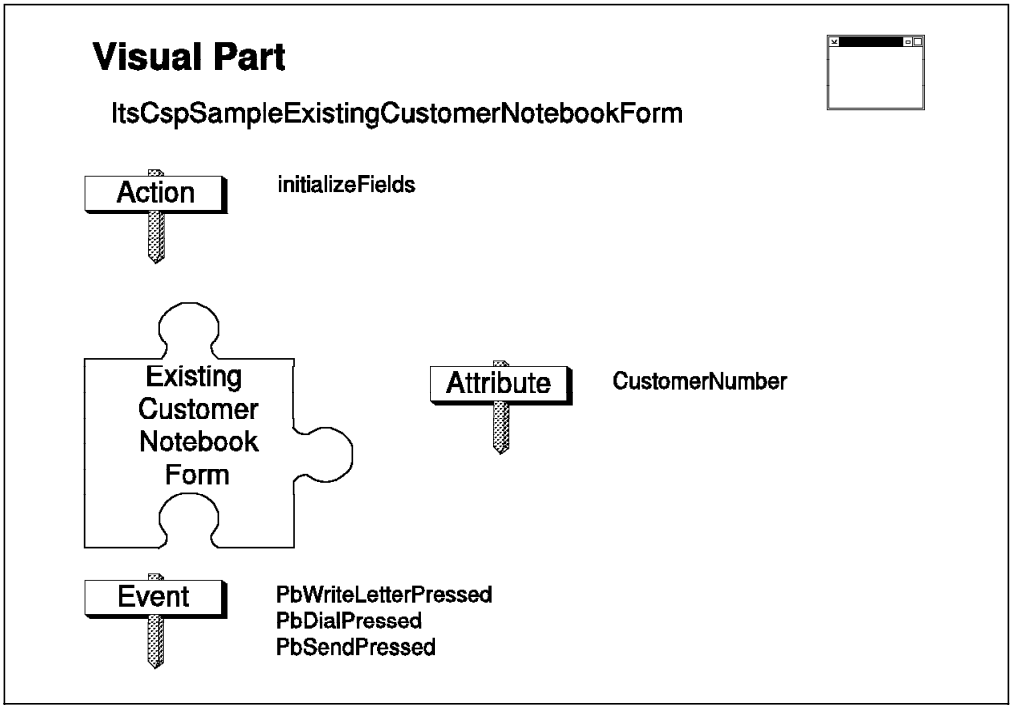

Figure 226. Public Interface: ItsCspSampleExistingCustomerNotebookForm

- **Used in part**

  ItsCspSampleCustomerWindow

- **Superclass of**

  None

- **Class definition**

  Figure 227 shows the class definition for the Existing Customer Notebook Form.

```
ItsCspSampleCustomerNotebookForm subclass:
                          #ItsCspSampleExistingCustomerNotebookForm
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
```

Figure 227. Class Definition: ItsCspSampleExistingCustomerNotebookForm

- **Scripts**

  Table 35 shows the scripts for the Existing Customer Notebook Form.

Table 35 (Page 1 of 2). Scripts: Existing Customer Notebook Form

Method	Description
pbDialPressed	Push button for phone dialing is pressed and the PbDialPressed event is raised.
pbSendPressed	Push button for sending a FAX is pressed and the PbSendPressed event is raised.

Table 35 (Page 2 of 2). Scripts: Existing Customer Notebook Form	
Method	**Description**
pbWriteLetterPressed	Push button for writing a letter is pressed and the PbWriteLetterPressed event is raised.

Figure 228 shows the pbDialPressed method.

```
pbDialPressed

   self signalEvent: #PbDialPressed.
```

Figure 228. Method: pbDialPressed

Figure 229 shows the pbSendPressed method.

```
pbSendPressed

   self signalEvent: #PbSendPressed.
```

Figure 229. Method: pbSendPressed

Figure 230 shows the pbWriteLetterPressed method.

```
pbWriteLetterPressed

   self signalEvent: #PbWriteLetterPressed.
```

Figure 230. Method: pbWriteLetterPressed

- **Event trace**

  Table 36 shows the event trace for the Existing Customer Notebook Form.

Table 36. Event Trace: Existing Customer Notebook Form	
User Action	**Sequence of Executed Connections**
Click on Write Letter push button	clicked >> pbWriteLetterPressed (hook)
Click on Dial Phone push button	clicked >> pbDialPressed (hook)
Click on Send Fax push button	clicked >> pbSendPressed (hook)

- **Special comments**

  We discuss here the difference between inheriting from a part and adding a part.

  If we inherit from a part we can change and extend the subpart (subclass) without changing the look and feel of the part's superclass.

Adding a part should be treated as a read-only situation. The added part should be used as is. If we edit the part, the part is changed in every place where it is used, not just in the application or part where it is edited.

8.20 Customer Number Form

This part illustrates the granularity that can be achieved when applications are constructed from parts. The Customer Number Form part is a form with two labels.

The same approach could be used for an entry field with a sophisticated validation or formatting algorithm assigned. Such a part could be reused as an isolated part.

- **Part name**

  ItsCspSampleCustomerNumberForm

- **Category**

  Basic visual part

- **Description**

  This is a very simple part that contains a prompt and an empty label string. The empty label string (#laNumber) is connected to a variable created using tear-off. The variable is added to the public interface so that the label string can be set using visual programming.

- **Composition Editor view**

  Figure 231 shows the Composition Editor view for the Customer Number Form.

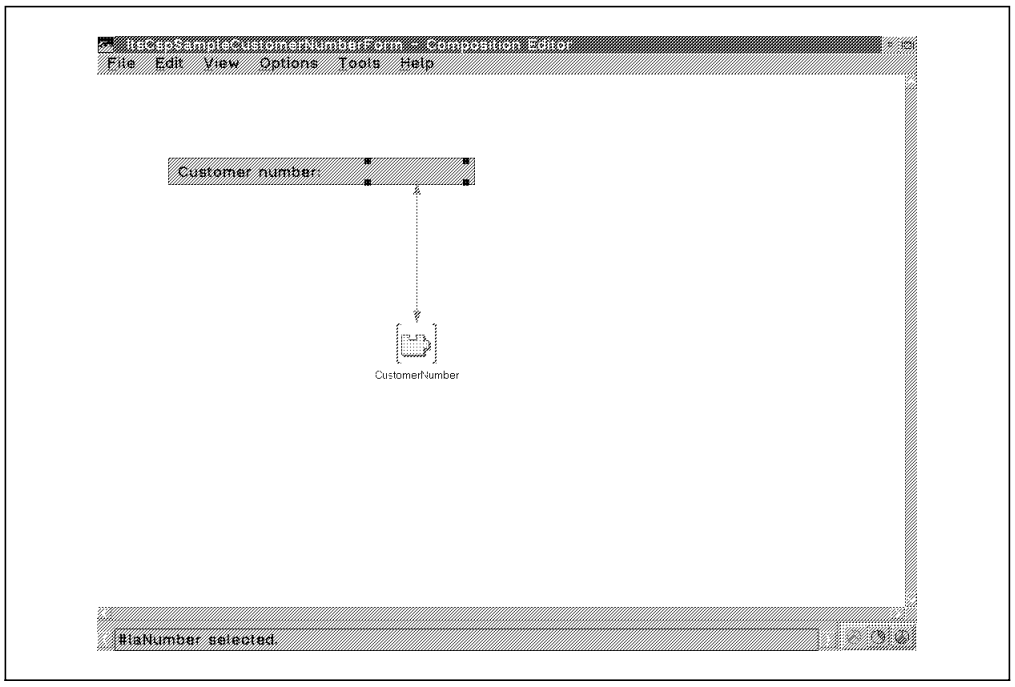

Figure 231. Composition Editor View: ItsCspSampleCustomerNumberForm

- **Part assembly**

None

- **Public interface**

  Figure 232 shows the public interface for the Customer Number Form.

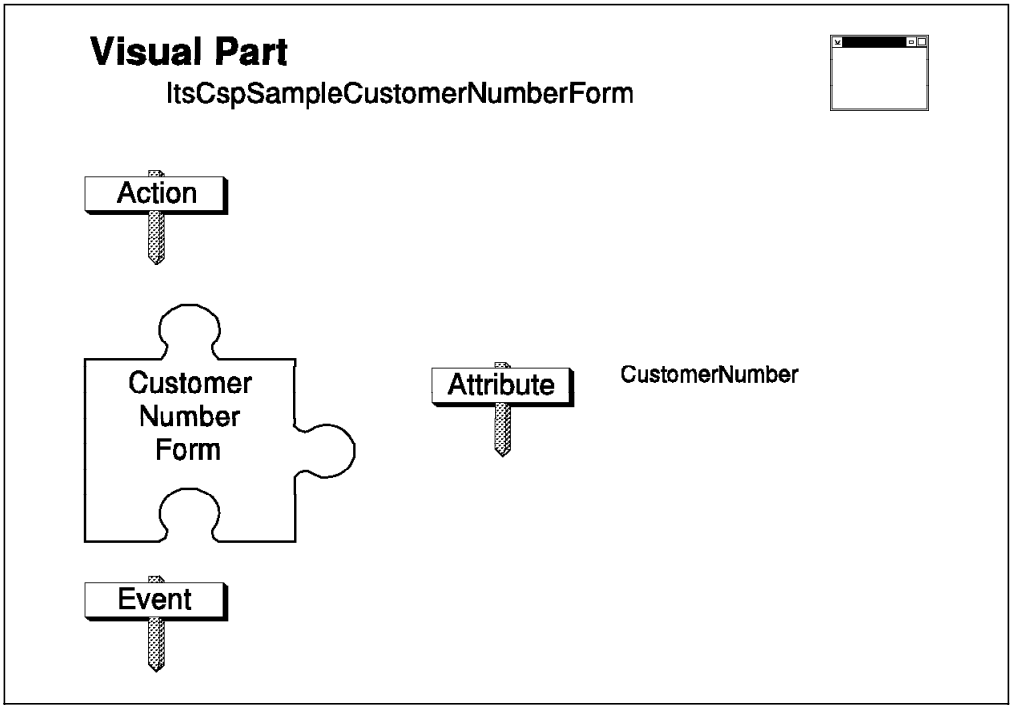

Figure 232. Public Interface: ItsCspSampleCustomerNumberForm

- **Used in part**

  ItsCspSampleExistingCustomerNotebookForm

- **Superclass of**

  None

- **Class definition**

  Figure 233 shows the class definition for the Customer Number Form.

```
AbtAppBldrView subclass: #ItsCspSampleCustomerNumberForm
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
```

Figure 233. Class Definition: ItsCspSampleCustomerNumberForm

- **Scripts**

  None

- **Event trace**

  None

- **Special comments**

  None

8.21 Information Line Form

The Information Line Form part is a very simple, generic part. It provides a standardized way of giving users textual feedback about the current status of an application.

Building basic parts, such as this one, can provide a common look and feel for the windows of the entire application.

- **Part name**

  ItsInformationLineForm

- **Category**

  Basic visual part

- **Description**

  This simple part contains an empty label string. The empty label string (#IaInformationText) can be initialized and set with actions available in the public interface.

- **Composition Editor view**

  Figure 234 shows the Composition Editor view for the Information Line Form.

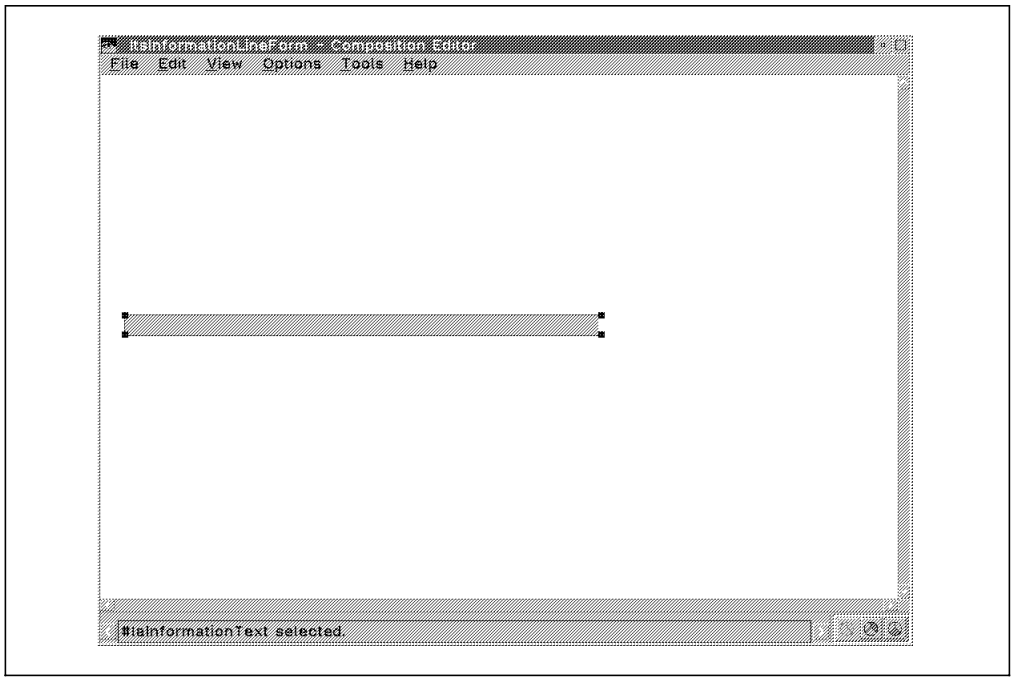

Figure 234. Composition Editor View: ItsInformationLineForm

- **Part assembly**

  None

- **Public interface**

  Figure 235 on page 205 shows the public interface for the Information Line Form.

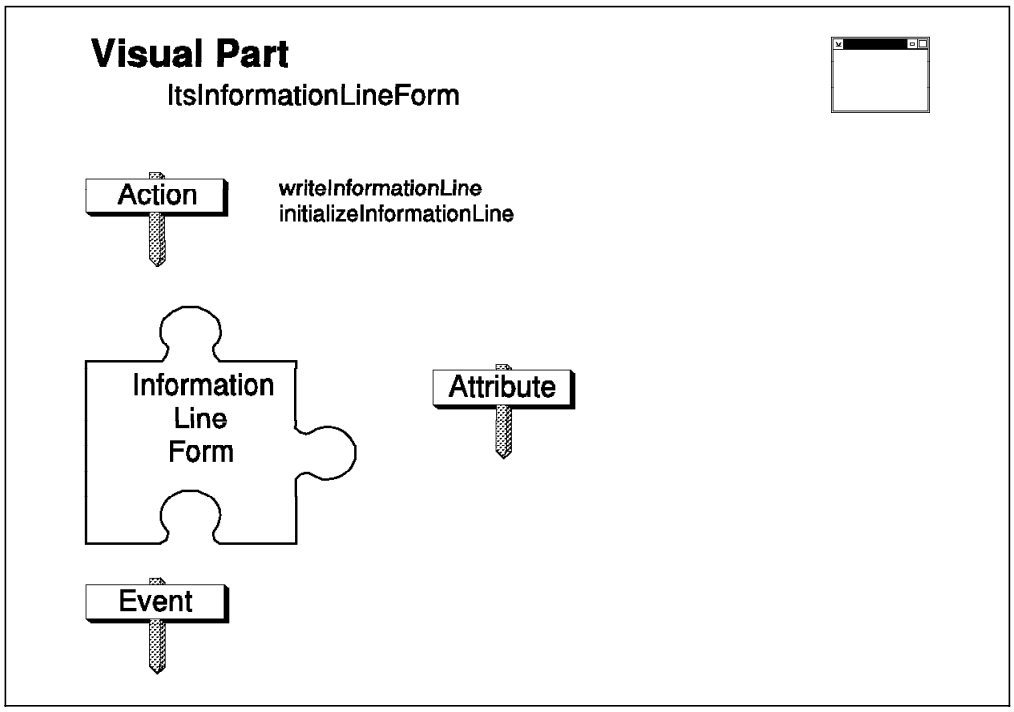

Figure 235. Public Interface: ItsInformationLineForm

- **Used in part**
  - ItsCspSampleCustomerWindow
  - ItsCspSampleNewCustomerWindow
- **Superclass of**

  None
- **Class definition**

  Figure 236 shows the class definition for the Information Line Form.

```
AbtAppBldrView subclass: #ItsInformationLineForm
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
```

Figure 236. Class Definition: ItsInformationLineForm

- **Scripts**

  Table 37 shows the scripts for the Information Line Form.

Table 37. Scripts: Information Line Form

Method	Description
initializeInfo	Initializes the text in the information line with blanks.
showInfo: aPromptString text: aString	Sets the prompt and the text in the information line.

Figure 237 on page 206 shows the initializeInfo method.

```
initializeInfo

    self partAttributeValue: #(#laInformationText #labelString) put: ''.
```

Figure 237. Method: initializeInfo

Figure 238 shows the showInfo: text: method.

```
showInfo: aPromptString text: aString

    self partAttributeValue: #(#laInformationText #labelString)
      put: (aPromptString, ': ',aString).
```

Figure 238. Method: showInfo: text:

- **Event trace**

  None

- **Special comments**

  None

8.22 New Customer Window

The New Customer Window is opened to add a new customer. It is implemented as a single instance window in the application.

The window has two push buttons to allow for two different modes of operation for adding customers. If the Add push button is selected, the window is closed automatically after a new customer is added through the host application. If the Add more push button is selected, the window remains open after a customer is added. To implement this mechanism, we keep information in a flag to remember which push button was selected.

- **Part name**

  ItsCspSampleNewCustomerWindow

- **Category**

  Composite visual part

- **Description**

  This part contains a single instance window, and it is not created by a model object as is the Customer Window. An instance of this part is created on the view side of our application. The New Customer Window is added as a subpart to the CSP sample application Customer Application (Main) window (see Figure 108 on page 106) and is opened after the user double-clicks on the new customer icon. Therefore this view object has the model object, customer, directly added as a subpart in its Composition Editor view.

  The visual programming logic to know which push button was clicked is implemented with a checkbox on the free form surface. The push button #clicked event is connected to the #selection action of the toggle button with a parameter, set to checked or unchecked, in its settings. The value of the checked parameter is queried from a script that is triggered after the addCustomer method is executed and the CustomerAdded event is raised.

- **Composition Editor view**

  Figure 239 shows the Composition Editor view for the New Customer Window.

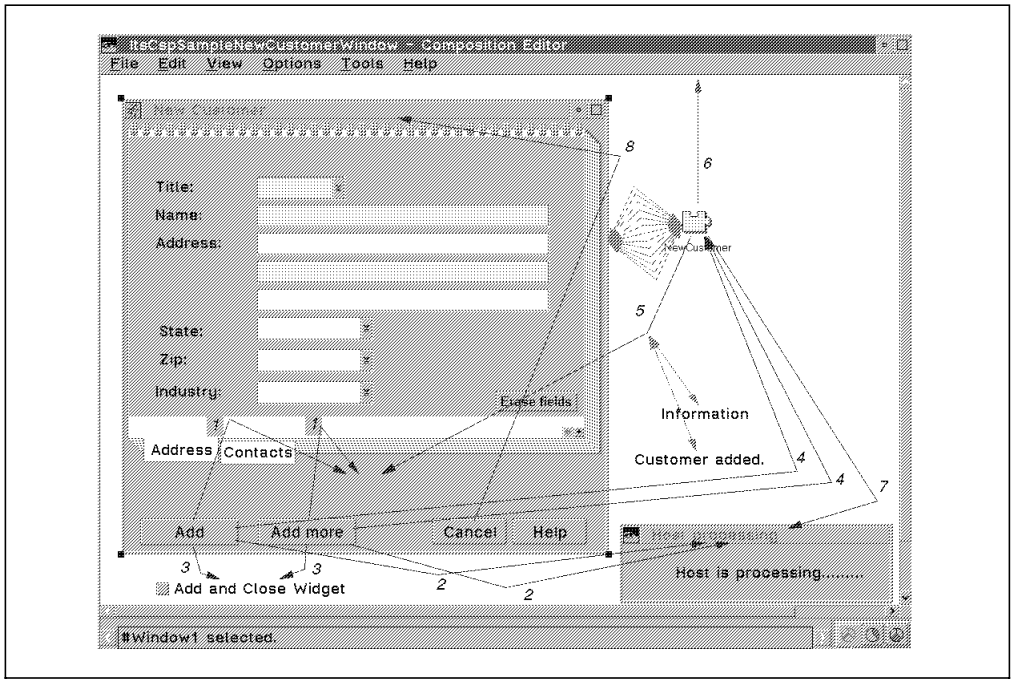

Figure 239. Composition Editor View: ItsCspSampleNewCustomerWindow

- **Part assembly**

  Figure 240 on page 208 shows the part assembly for the New Customer Window.

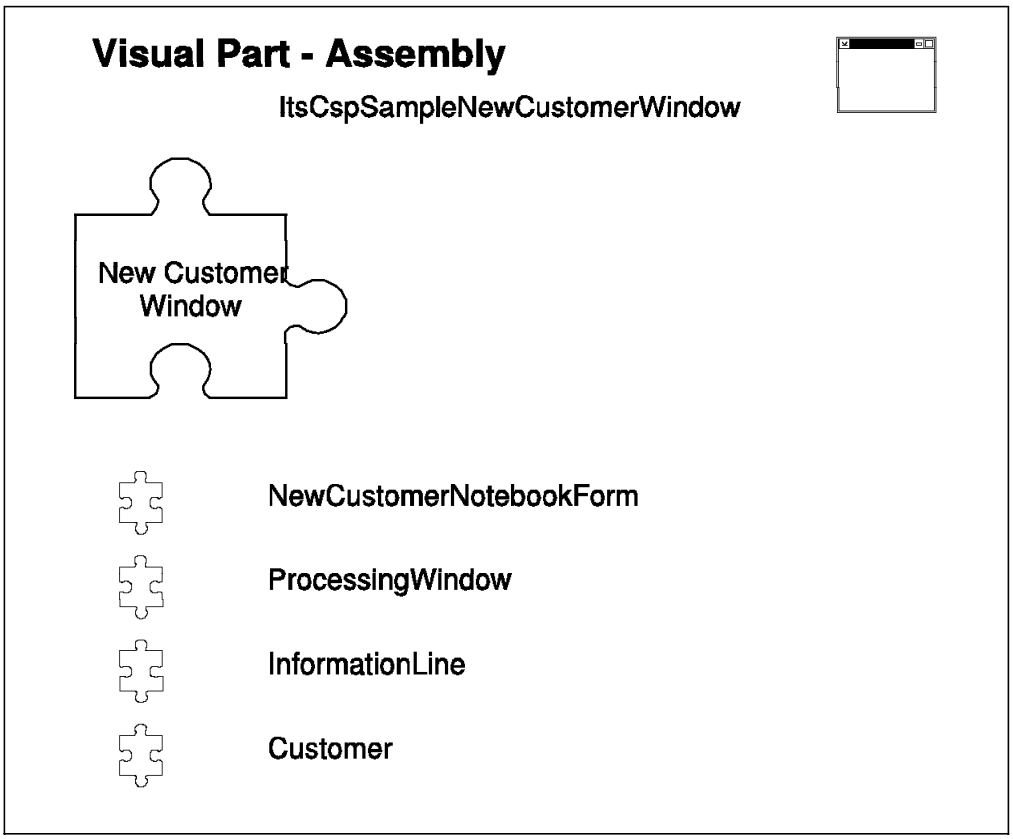

Figure 240. Part Assembly: ItsCspSampleNewCustomerWindow

- **Public interface**

  None

- **Used in part**

  ItsCspSampleMainWindow

- **Superclass of**

  None

- **Class definition**

  Figure 241 shows the class definition for the New Customer Window.

```
AbtAppBldrView subclass: #ItsCspSampleNewCustomerWindow
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
```

Figure 241. Class Definition: ItsCspSampleNewCustomerWindow

- **Scripts**

  Table 38 on page 209 shows the script for the new customer window.

Table 38. Script: New Customer Window	
Method	**Description**
addOneAndCloseWidget	Closes the widget after the customer is added successfully when invoked through the Add push button. Whether the Add or the Add more push button was pressed can be determined by querying whether the toggle button (Add and Close Widget) is checked or not.

Figure 242 shows the addOneAndCloseWidget method.

```
addOneAndCloseWidget

   (self partAttributeValue: #(#tbAddOneAndCloseWidget #selection))
   ifTrue:
      [(self subpartNamed: #Window1) performActionNamed: #closeWidget.].
```

Figure 242. Method: addOneAndCloseWidget

- **Event trace**

  Table 39 shows the event trace for the new customer window.

Table 39 (Page 1 of 2). Event Trace: New Customer Window	
User Action	**Sequence of Executed Connections**
Click on Add push button	• (1) clicked >> initializeInformationLine • (2) clicked >> openWidget (HostProcessingWindow) • (3) clicked >> selection with value **checked** as a parameter defined in the connection settings • (4) clicked >> addCustomer (Customer part) • (5) event: CustomerAdded (Customer part) >> writeInformationLine (Label strings: 'Information', 'Customer added') • (6) event: CustomerAdded (Customer part) >> addOneAndCloseWidget (hook) • (7) event: CustomerAdded (Customer variable) >> closeWidget (HostProcessingWindow)

Table 39 (Page 2 of 2). Event Trace: New Customer Window

User Action	Sequence of Executed Connections
Click on Add more push button	• (1) clicked >> initializeInformationLine • (2) clicked >> openWidget (HostProcessingWindow) • (3) clicked >> selection with value **unchecked** as a parameter defined in the connection settings • (4) clicked >> addCustomer (Customer part) • (5) event: CustomerAdded (Customer part) >> writeInformationLine (Label strings: 'Information', 'Customer added') • (6) event: CustomerAdded (Customer part) >> addOneAndCloseWidget (hook) • (7) event: CustomerAdded (Customer variable) >> closeWidget (HostProcessingWindow)
Click on Cancel push button	(8) clicked >> closeWidget

- **Special comments**

  In the box below we explain how to keep previous events.

How to Keep information about previous events

- Add a toggle button to the free form surface with the value unchecked in its settings.
- Connect the event that must be kept (for example, push button clicked) to the #selection action of the toggle button.
- Open the settings of the connection and define the value checked. (This value also could be set visually with a connection to the parameter value of the connection.)
- Write a script and use value of the #selection attribute of the toggle button in a Boolean expression to define the follow-on logic.
- Reset the value checked with a visual connection or from the script.

8.23 New Customer Notebook Form

The New Customer Notebook Form part is used as a subpart in the New Customer Window. It is a basic part and mainly uses the inherited function. The added function to erase the contents of the entry fields is done visually.

- **Part name**

  ItsCspSampleNewCustomerNotebookForm

- **Category**

  Basic visual part

- **Description**

This part is created as a subpart (subclass) of the Customer Notebook Form part. This part adds the Erase fields push buttons on both pages of the notebook.

- **Composition Editor view**

  Figure 243 shows the Composition Editor view for the New Customer Notebook Form. The figure does not show the connections for the initializeFields event. Because this part inherits from the Customer Notebook Form part, all connections for the initialization are also inherited.

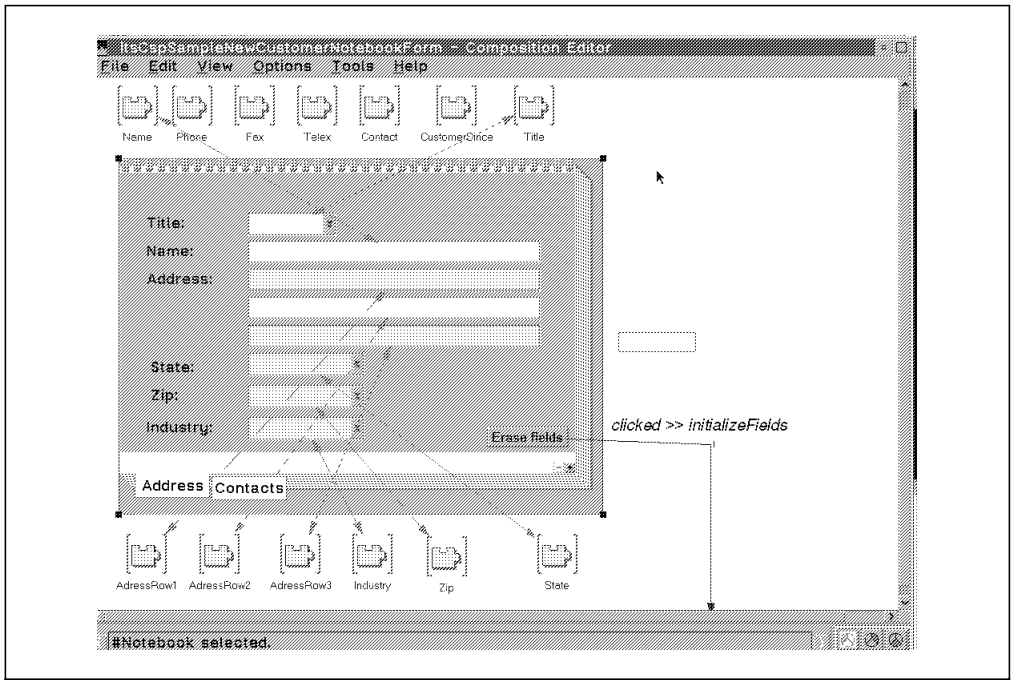

Figure 243. Composition Editor View: ItsCspSampleNewCustomerNotebookForm

- **Part assembly**

  None

- **Public interface**

  None (inherited from the Customer Notebook Form part)

- **Used in part**

  ItsCspSampleNewCustomerWindow

- **Superclass of**

  None

- **Class definition**

  Figure 244 on page 212 shows the class definition for the New Customer Notebook Form.

```
ItsCspSampleCustomerNotebookForm subclass:
                  #ItsCspSampleNewCustomerNotebookForm
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
```

Figure 244. Class Definition: ItsCspSampleNewCustomerNotebookForm

- **Scripts**

  None

- **Event trace**

  Table 40 shows the event trace for the new customer notebook form.

Table 40. Event Trace: Existing Customer Notebook Form

User Action	Sequence of Executed Connections
Click on Erase fields push button	clicked >> initializeFields (hook to inherited script)

  The connections for the initialization after the initializeFields script raises the InitializeFields event are also inherited. The inherited connections are hidden in our pictures to reduce the complexity of the Composition Editor view.

- **Special comments**

  The inheritance in VisualAge extends to the class definition with variables and methods and the public interface definition.

Appendix A. Screen Field Monitor Tool

One of the challenges developers of VisualAge EHLLAPI applications face is the mapping between the fields on the host screen and the fields that can be torn off from an Abt3270Screen part as a result of VisualAge′s parsing of the host screen. We developed a tool during our project that helps developers meet that challenge. Our tool is a VisualAge application that can be executed against any host screen and shows the input and output fields from the host screen in two list boxes with field number, length, position, and contents.

This appendix describes the problem, explains VisualAge′s parsing of the host screen, and gives detailed information about our tool.

A.1 Problem Description

To understand the problem, one must first understand how the data from the host screen is represented on the PWS and how VisualAge parses the host screen to build input and output records that can be used to send data to and receive data from the host using VisualAge′s EHLLAPI support. Figure 245 on page 214 shows the problem domain.

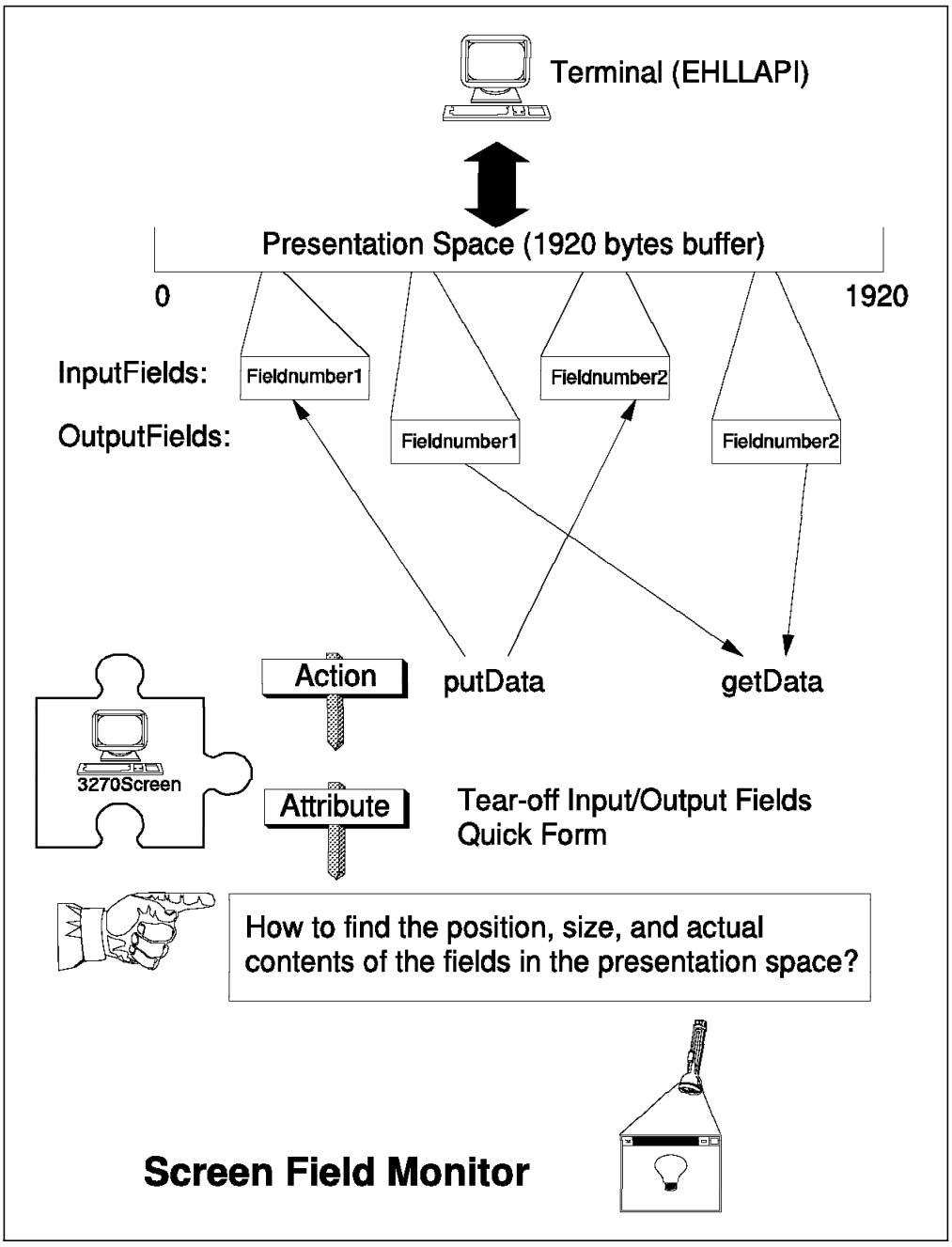

Figure 245. The Find Fields Problem

The interface between the 3270 host session and the Abt3270Terminal part, which is also used as an instance variable in the Abt3270Screen part, is the presentation space. The presentation space is a buffer, usually 1920 bytes long, that can be displayed on the screen of a 3270 emulation.

Data transmitted to and from a host application is kept in the presentation space. Data in the presentation space must be addressed with an offset in the presentation space.

The Abt3270Terminal part provides a method of parsing the presentation space to build a record of input and output fields. The parser considers unprotected fields as input fields and protected fields as output fields. The distinction is based on the field attributes.

The parser is called from the settings view of an Abt3270Screen part when the Build Records push button is selected. The results of this parsing method are record objects that contain field objects. The field objects know their offset position in the presentation space and their length.

VisualAge generates names for the input and output fields as FieldNumberX, where X is a sequence number.

The input and output fields are visible in the attribute list of the Abt3270Screen part and can be either torn off to get variables or used to generate the entry fields for a window using quick form.

The input and output fields built by VisualAge's parsing mechanism can be manipulated using the Abt3270Screen part's putData (to write data) and getData actions (to read data).

Figure 246 shows the host screen used to illustrate the problem and the solution for the problem.

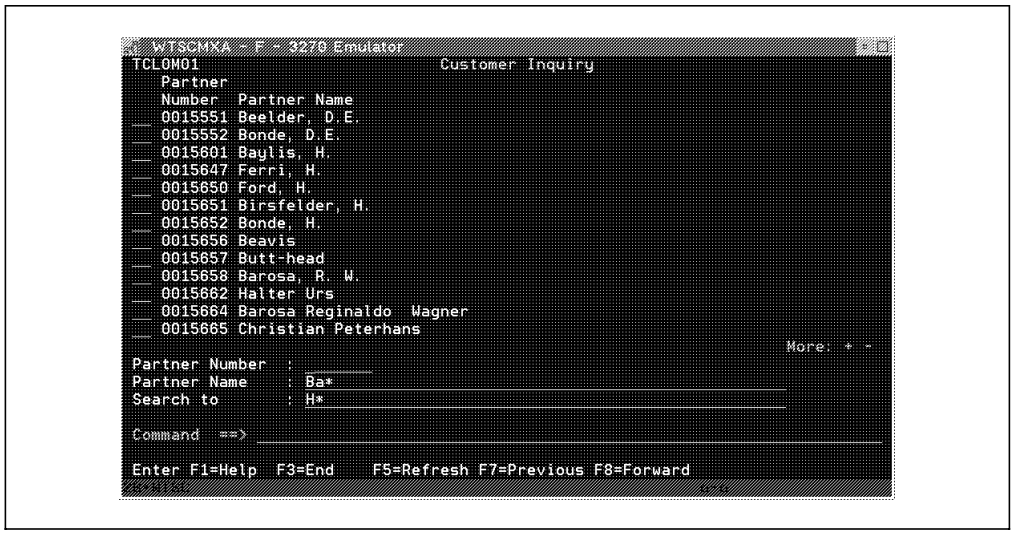

Figure 246. Host Screen

Because of the way the parser works and the way VisualAge generates field names, the challenge for the developer is to know which input or output field maps to which field on the host map.

Developers can use the quick form function to create a default form for all input and output fields and to display those fields in a window to help with the field mapping. However, if a host screen has many fields, it may not be easy to fit all fields on a single window. Remember that parsing a host screen for protected and unprotected fields may result in many more fields than one might expect. For example, constant text on the host screen would result in additional output fields.

If we parse the host screen shown in Figure 246 and use quick form to build entry fields in a window for the output fields of the host screen, we get the result shown in Figure 247 on page 216.

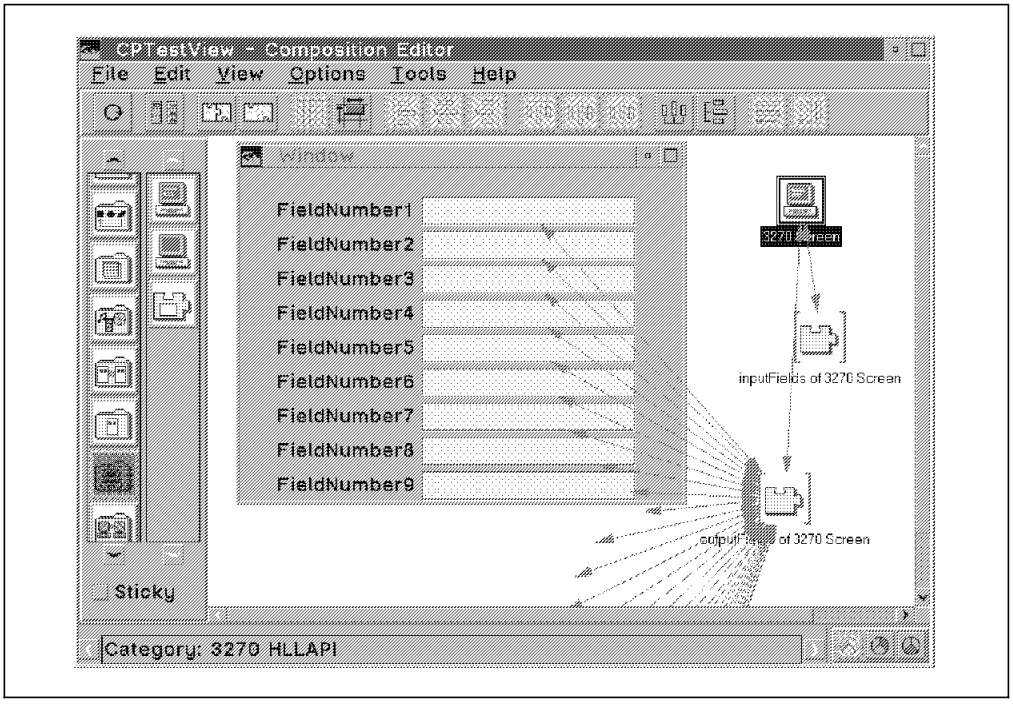

Figure 247. Quick Form with Many Fields

We created a tool—the screen field monitor—during our project to help us map the input and output fields from VisualAge′s parsing to the fields on the host screens. The screen field monitor provides the number of input and output fields in two window fields and the field name, size, start position, and contents of the input and output fields in two list boxes. Figure 248 shows the output from the tool for the sample host screen shown in Figure 246 on page 215.

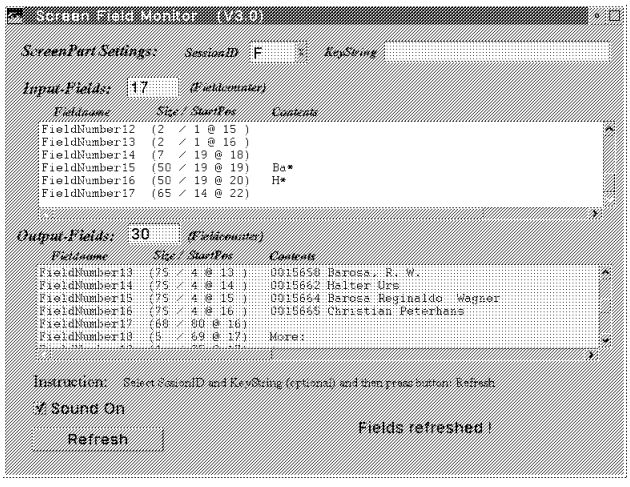

Figure 248. Fields Collected by Screen Field Monitor Tool

A.2 Tool Implementation

The tool implementation basically had to solve two problems:

- Using a screen part whose session ID is not hardcoded but dynamically defined from the list of available sessions
- Getting the information from the fields of the composite AbtRecord object.

Figure 249 shows the Composition Editor view with the visual programming definitions for the screen field monitor tool.

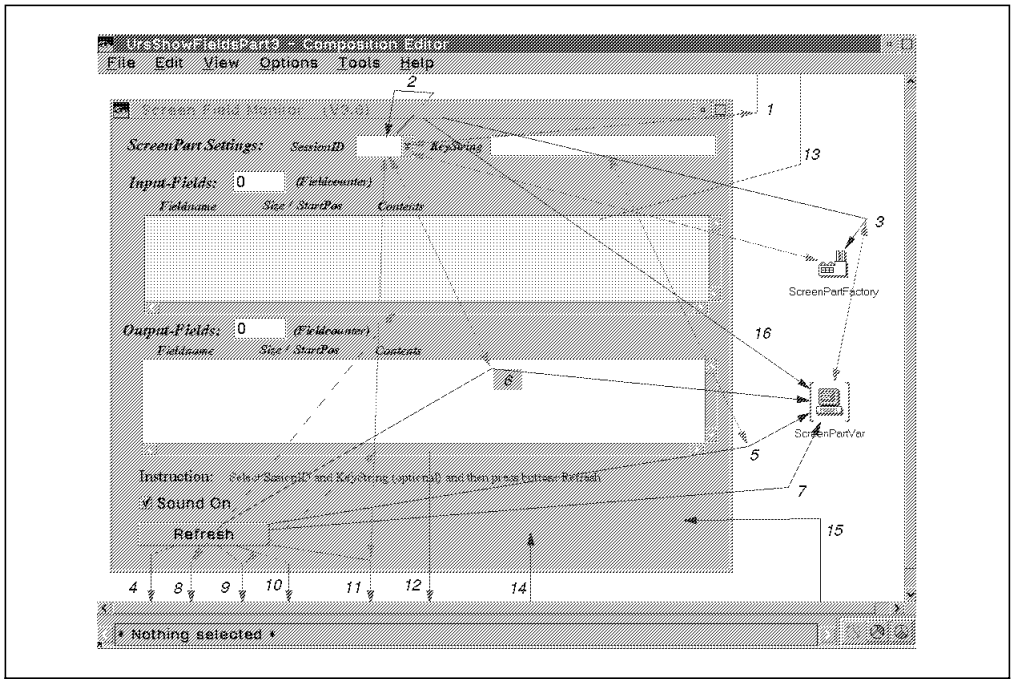

Figure 249. Composition Editor View: Screen Field Monitor

We used an object factory part to dynamically create an Abt3270Screen part. During initialization of the Screen Field Monitor window (aboutToOpenWidget), the factory part creates an instance of an Abt3270Screen part. The session ID of this Abt3270Screen part is set to the first session ID found in the list of available sessions.

Note: Although our approach allows for dynamic host session selection at run time, the session used during development—A in our case—must be available at run time.

Table 41 shows the event trace for the Screen Field Monitor window initialization.

Table 41 (Page 1 of 2). Event Trace: Initialization of the Screen Field Monitor

	Sequence of Executed Connections
Window initialization	(1) aboutToOpenWidget (window) >> listActiveSessions (hook), put result into the SessionID drop down list
Drop-down list initialization	(2) aboutToOpenWidget (drop down list) >> selectionIndex: 1 (setting), put result into entry field

Table 41 (Page 2 of 2). Event Trace: Initialization of the Screen Field Monitor	
	Sequence of Executed Connections
Following window initialization	• (3) openedWidget (window) >> new (ScreenPartFactory), put result into variable (ScreenPartVar) • attribute-to-attribute connection: selectedItem (drop down list) >> shortSessionId (ScreenPartFactory)

Users can refresh the contents of the two list boxes in the Screen Field Monitor window to refresh the information from the same host screen or gather information from another host screen. Table 42 shows the connections for the Refresh push button.

Table 42. Event Trace: Refresh Push Button Clicked	
User Action	**Sequence of Executed Connections**
Click on Refresh push button	Refer to Figure 250 • (4) clicked >> initializeRefresh (hook) • (5) clicked >> keyString (ScreenPartVar) • (6) clicked >> shortSessionId (ScreenPartVar) • (7) clicked >> refreshFieldDefs (ScreenPartVar) • (8) clicked >> refreshFields (hook) • (9) clicked >> createItemsFromInputFields (hook) • (10) clicked >> createItemsFromOutputFields (hook) • (11) clicked >> listActiveSessions (hook)

The sequence of these connections is very important for the programming logic. Figure 250 shows the connection sequence.

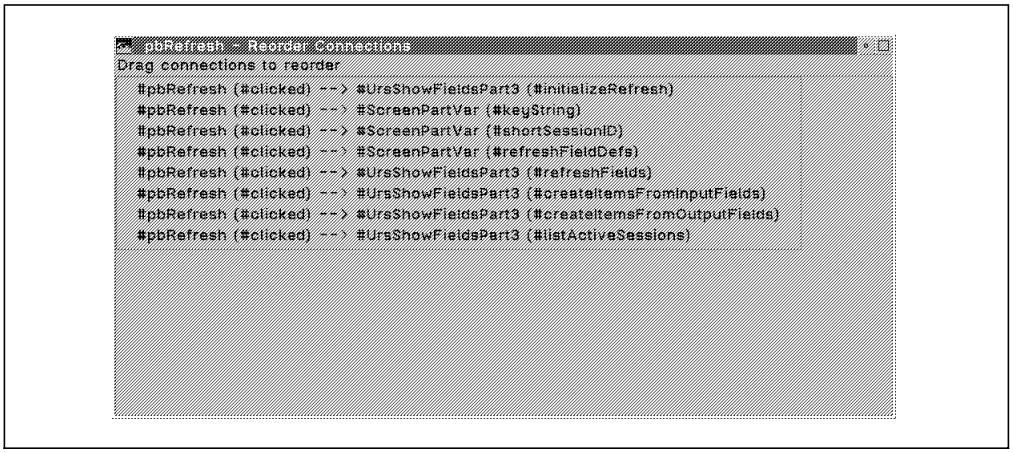

Figure 250. Connection Sequence for the Refresh Push Button

The last four required connections (12 to 14 in Figure 249 on page 217) define the logic to count the actual number of items in the list box and show and hide a message, for instance, Fields refreshed, in an entry field. The message field is not visible in Figure 249 on page 217.

It is good programming style to destroy dynamically created parts when they are no longer needed. We destroyed the ScreenPartVar variable when the window closed. However, in our case this was optional because the application terminates when the Screen Field Monitor window is closed.

The event trace for the remaining connections is shown in Table 43.

Table 43. Event Trace: Screen Field Monitor Additional Logic

User Action or Event	Sequence of Executed Connections
event: output field items changed	(12) items >> countOutputFieldItems (hook), put result in entry field
event: input field items changed	(13) items >> countInputFieldItems (hook), put result in entry field
event: showProcessingMessage raised from hooks	(14) event: showProcessingMessage >> show (invisible entry field)
event: hideProcessingMessage raised from hooks	(15) event: hideProcessingMessage >> hide (invisible entry field)
user action: close widget from the system menu	(16) closedWidget >> destroyPart (ScreenPartVar)

Figure 251 shows the definition of the class in the Smalltalk class hierarchy.

```
AbtAppBldrView subclass: #UrsShowFieldsPart3
    instanceVariableNames: 'screen '
    classVariableNames: 'SessionExists '
    poolDictionaries: ''
```

Figure 251. Class Definition: Screen Field Monitor

To implement the rest of the logic, in addition to the visual connections, we implemented the scripts shown in Table 44.

Table 44 (Page 1 of 2). Scripts: Screen Field Monitor

Method	Description
countInputFieldItems	Returns the number of items in the input field list box.
countOutputFieldItems	Returns the number of items in the output fields list box.
createItemsFromFields: aFieldsRecord	Reads the presentation space with the definition of the fields passed as a parameter (aFieldsRecord). It creates a sorted collection.
createItemsFromInputFields	Calls the createItemFromFields method for the input fields and returns a sorted collection.
createItemsFromOutputFields	Calls the createItemFromFields method for the output fields and returns a sorted collection.

Table 44 (Page 2 of 2). Scripts: Screen Field Monitor	
Method	**Description**
initializeRefresh	Stores the ScreenPartVar variable value into the screen instance variable and deletes the items in the list boxes and the fields in AbtRecord.
listActiveSessions	Returns a dictionary with all active sessions.
playInputFieldrefreshedMusic	Plays a tune in a separate thread when the input fields are refreshed.
playOutputFieldrefreshedMusic	Plays a tune in a separate thread when the output fields are refreshed.
playRefreshFieldsMusic	Plays a tune in a separate thread when the input and output fields are refreshed.
refreshFields	Refreshes the fields by requesting getFieldData from the linked screen.
screen	Returns the value of screen.
screen: anObject	Saves the value of screen.
screenKeyString	Returns the value of screen KeyString.
screenSessionId	Returns the value of screen SessionId.

Figure 252 shows the countInputFieldItems method.

```
countInputFieldItems

   | aCollection result |

   aCollection := self partAttributeValue: #(#lbInputFields #items).

   result := aCollection size.

   ^self partAttributeValue: #(#efInputFieldCount #object) put: result.
```

Figure 252. Method: countInputFieldItems

Figure 253 shows the countOutputFieldItems method.

```
countOutputFieldItems

   | aCollection result |

   aCollection := self partAttributeValue: #(#lbOutputFields #items).

   result := aCollection size.

   ^self partAttributeValue: #(#efOutputFieldCount #object) put: result.
```

Figure 253. Method: countOutputFieldItems

Figure 254 on page 221 shows the createItemsFromFields: method.

```
createItemsFromFields: aFieldsRecord

   | aFieldsDict aString aSortedCollect aValueString |

   aFieldsDict := aFieldsRecord arrayOf arrayType fields.

   aSortedCollect := SortedCollection new.

   aSortedCollect sortBlock: [ :a :b |
      ((a copyFrom: 12 to: 14) trimBlanks asNumber)
      <=
      ((b copyFrom: 12 to: 14) trimBlanks asNumber)].

   aFieldsDict keysDo: [  :key |

      aValueString := aFieldsRecord at: key.
      aString := key.
      aString := aString abrPadWithBlanks: 15.
      aString := aString ,'(', (aFieldsDict at: key) count printString.
      aString := aString abrPadWithBlanks: 19.
      aString := aString ,'/ ',
            (screen terminal convertOffset: ((aFieldsDict at: key) offset))
            printString.
      aString := aString abrPadWithBlanks: 28.
      aString := aString ,')'.
      aString := aString abrPadWithBlanks: 32.
      aString := aString ,aValueString.

      aSortedCollect add: aString.
   ].

   ^aSortedCollect.
```

Figure 254. Method: createItemsFromFields

Figure 255 shows the createItemsFromInputFields method.

```
createItemsFromInputFields

   | aSortedCollect |

   (self subpartNamed: #txtProcessingMessage)
                  labelString: 'Grabbing Input Fields .....'.

   screen inputFields = nil ifFalse:
      [aSortedCollect := self createItemsFromFields: (screen inputFields)].

   self playInputFieldrefreshedMusic.

   ^aSortedCollect.
```

Figure 255. Method: createItemsFromInputFields

Figure 256 on page 222 shows the createItemsFromOutputFields method.

```
createItemsFromOutputFields

   | aSortedCollect |

   (self subpartNamed: #txtProcessingMessage)
            labelString: 'Grabbing Output Fields .....'.

   screen outputFields = nil ifFalse:
      [aSortedCollect := self createItemsFromFields: (screen outputFields)].

   self playOutputFieldrefreshedMusic.
      (self subpartNamed: #txtProcessingMessage)
         labelString: 'Fields refreshed !!'.

   ^aSortedCollect.
```

Figure 256. Method: createItemsFromOutputFields

Figure 257 shows the initializeRefresh method.

```
initializeRefresh

   "initialize instance variable: screen"
   screen := self partAttributeValue: #(#ScreenPartVar #self).

   "initialize input and output fields"
   screen inputFields:  nil.
   screen outputFields: nil.

   "initialize list boxes"
   self partAttributeValue: #(#lbInputFields #items) put: nil.
   self partAttributeValue: #(#lbOutputFields #items) put: nil.
```

Figure 257. Method: initializeRefresh

Figure 258 shows the listActiveSessions method.

```
listActiveSessions

   | sessions aDict |

   sessions := Abt3270Terminal allSessions select: [ :e |
      ((e at: #qsstSestype) = $D) | ((e at: #qsstSestype) = $F) ].

   aDict := Dictionary new.
```

Figure 258 (Part 1 of 2). Method: listActiveSessions

```
   sessions do: [ :each | aDict at: (each at: #qsstShortname)
                    put: ((each at: #qsstShortname) asCharacter)].

   aDict size = 0 ifTrue:
   [ SessionExists := 0.
     CwMessagePrompter message: 'No Session active. Start session first.'.
     " exit Application here !!!!!! "
   ]
   ifFalse: [SessionExists := 1].

   ^aDict asSortedCollection.
```

Figure 258 (Part 2 of 2). Method: listActiveSessions

Figure 259 shows the playInputFieldrefreshedMusic method.

```
playInputFieldrefreshedMusic

   (self partAttributeValue: #(#tbSoundOn #selection)) ifTrue:
   [
      (PlatformFunctions at: 'DosBeep')
         coroutineCallWith: 270 with: 100;
         coroutineCallWith: 440 with: 100;
         coroutineCallWith: 660 with: 100.
   ].
```

Figure 259. Method: playInputFieldrefreshedMusic

Figure 260 shows the playOutputFieldrefreshedMusic method.

```
playOutputFieldrefreshedMusic

   (self partAttributeValue: #(#tbSoundOn #selection)) ifTrue:
   [
      (PlatformFunctions at: 'DosBeep')
          coroutineCallWith: 100 with: 500.
   ].
```

Figure 260. Method: playOutputFieldrefreshedMusic

Figure 261 shows the playRefreshFieldsMusic method.

```
playRefreshFieldsMusic

   (self partAttributeValue: #(#tbSoundOn #selection)) ifTrue:
   [
      (PlatformFunctions at: 'DosBeep')
          coroutineCallWith: 880 with: 250.
   ].
```

Figure 261. Method: playRefreshFieldsMusic

Figure 262 on page 224 shows the refreshFields method.

```
refreshFields

   self playRefreshFieldsMusic.
   (self subpartNamed: #txtProcessingMessage)
      labelString: 'Getting Data from Screen .....';
      show.

   screen getFieldData.
```

Figure 262. Method: refreshFields

Figure 263 shows the screen method.

```
screen

   "Return the value of screen."

   ^screen
```

Figure 263. Method: screen

Figure 264 shows the screen: method

```
screen: anObject

   "Save the value of screen."

   screen := anObject.
   self signalEvent: #screen
      with: anObject.
```

Figure 264. Method: screen:

Figure 265 shows the screenKeyString method.

```
screenKeyString

   "Return the value of screen KeyString."

   ^screen keyString.
```

Figure 265. Method: screenKeyString

Figure 266 shows the screenSessionId method.

```
screenSessionId

   "Return the value of screen SessionId."

   ^screen shortSessionID.
```

Figure 266. Method: screenSessionId

Appendix B. Changing Host Sessions Dynamically without Using Scripts

In our simple Abt3270Screen part example, described in 3.3.1, “Abt3270Screen Part Only” on page 21, we assumed that the host session was always fixed (G). You may want to allow the users of a VisualAge EHLLAPI application implemented using the Abt3270Screen part to switch host sessions dynamically at run time.

Note: Although the approach described in this appendix allows users to switch host sessions dynamically at run time, the session used during development—G in our case—must be available at run time.

This appendix explains how we used the visual part created in the example and expanded it to provide for the dynamic switching of host sessions. We used the action bar of the first GUI window to provide a session selection menu. We performed the following activities:

1. Tear off the *shortSessionId* attribute for both Abt3270Screen parts as shown in Figure 267.

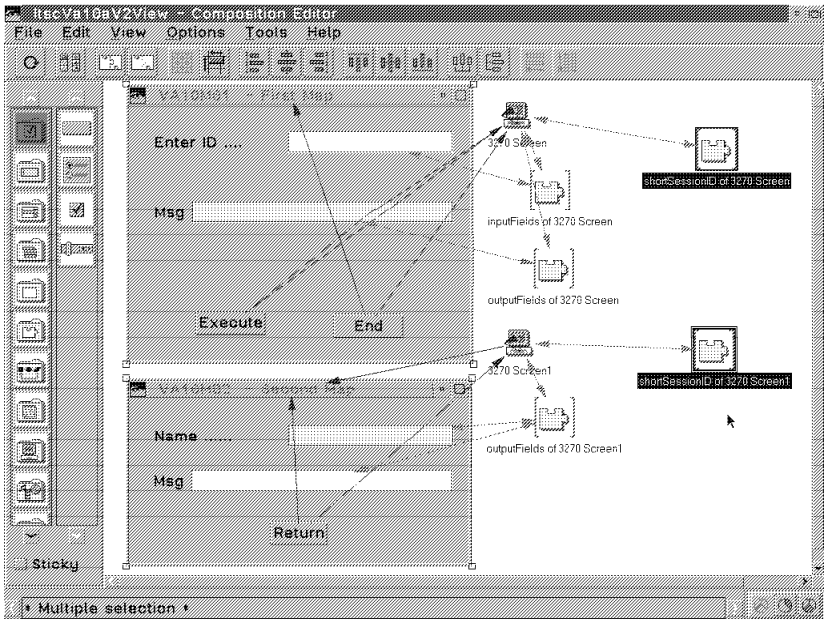

Figure 267. Tearing Off shortSessionId

2. Implement the action bar with the session ID options. The *VisualAge User's Guide and Reference* provides detailed instructions on how to implement an action bar menu in *Adding Menu Parts*. Figure 268 on page 226 shows the menus with the session ID options.

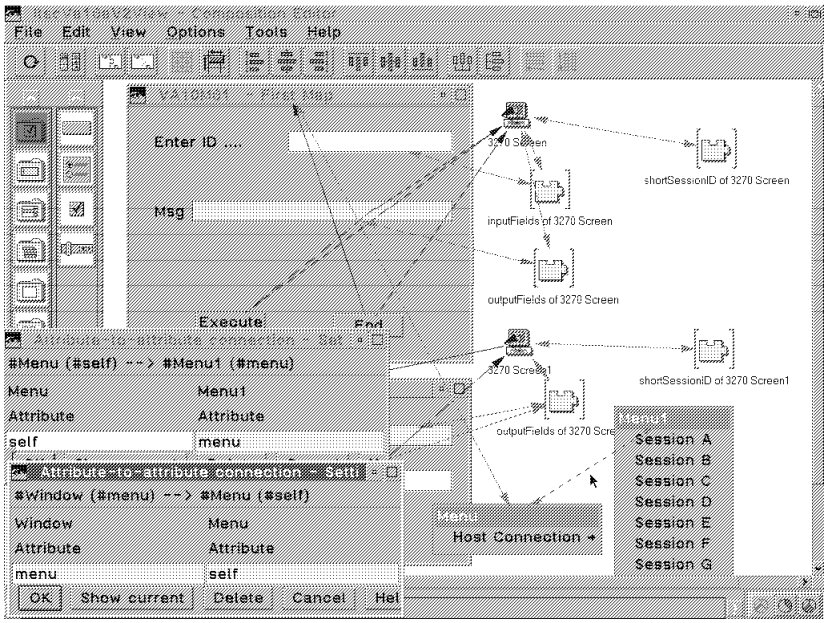

Figure 268. Adding a Menu with Session Id Options

3. Add a variable part to the free form surface to pass the session ID to the Abt3270Screen part. Connect each session to the variable as follows:

   - *Push Button1* (#clicked) to Object (#self)
   - *Push Button2* (#clicked) to Object (#self)
   - ....

   The connections are incomplete (dashed), and you must supply a parameter value.

4. Select a connection, double-click on it, and select *Set parameters*. Figure 269 on page 227 shows the parameter setting for session A. Repeat this step for each session ID (B, C, D..., G).

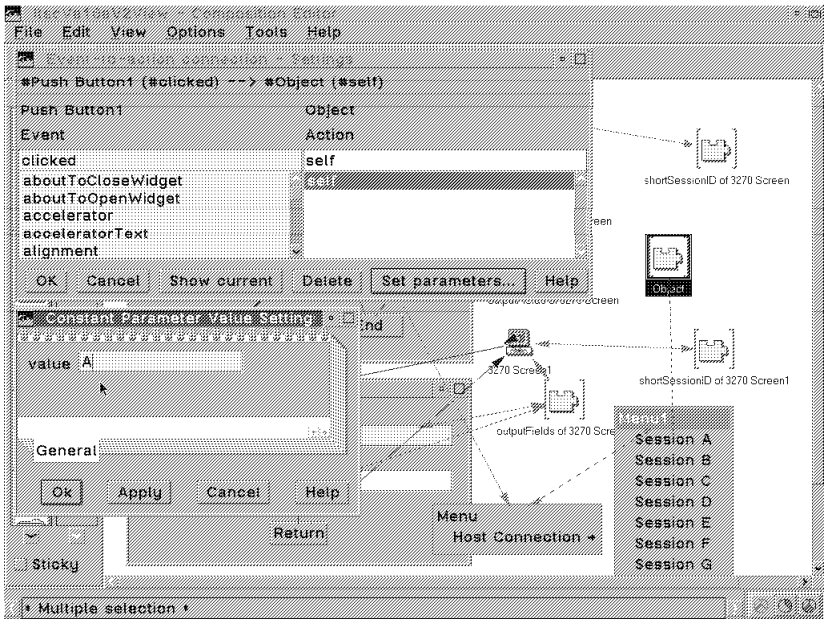

Figure 269. Setting Parameters for a Connection

5. Finally, connect the variable to the torn-off *shortSessionID* attributes of the Abt3270Screen parts as follows:
   - *#Object* (#self) to *#shortSessionID of 3270 Screen* (#self)
   - *#Object* (#self) to *#shortSessionID of 3270 Screen1* (#self)

Figure 270 shows all connections.

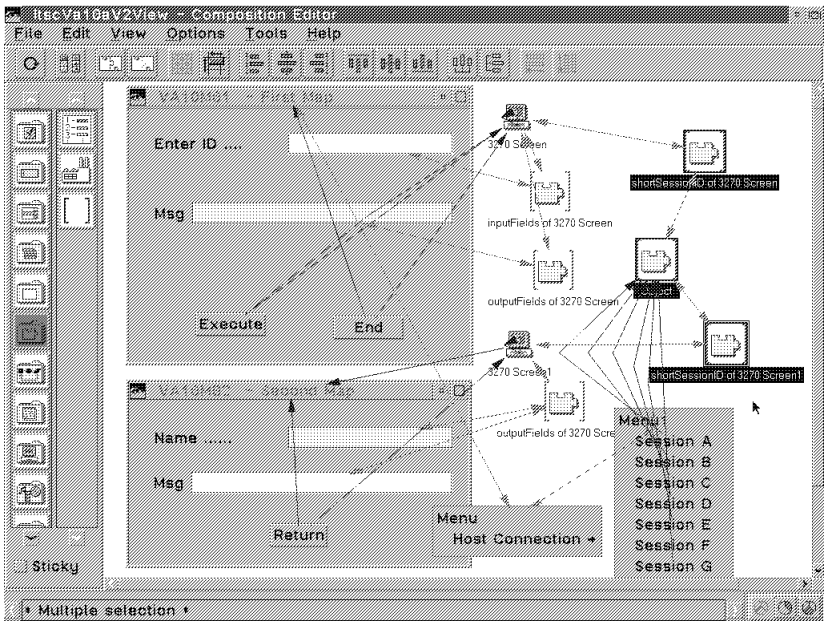

Figure 270. All Connections Required to Switch Host Sessions

Figure 271 shows the session ID options menu at run time.

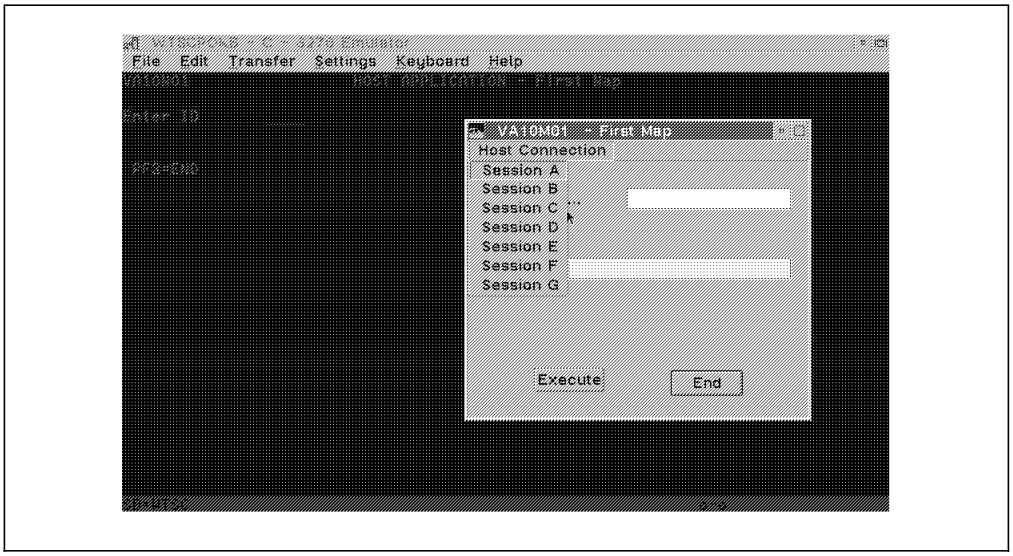

Figure 271. Session Id Options Menu

Appendix C. Jumping to the 3270 Screen

Suppose you have to build a GUI interface for a specific application, but, at the same time, users of that GUI interface want to be able to use existing, text-based host applications from the same PWS.

For example, PWS users have access to text-based host applications through a 3270 emulation. For a specific host application a GUI interface is available. If that specific application is started, the GUI application should be in control automatically.

To implement this example with VisualAge's EHLLAPI support you must write an application with host control, that is, using the Abt3270Screen part. You must also write some VisualAge scripts.

This appendix explains how we implemented the function described above using the application described in Chapter 3, "Applying the VisualAge EHLLAPI Parts with a Simple Example" on page 17.

Figure 272 shows the completed application with all visual connections in the Composition Editor window. The sections that follow explain the relevant steps to implement this application.

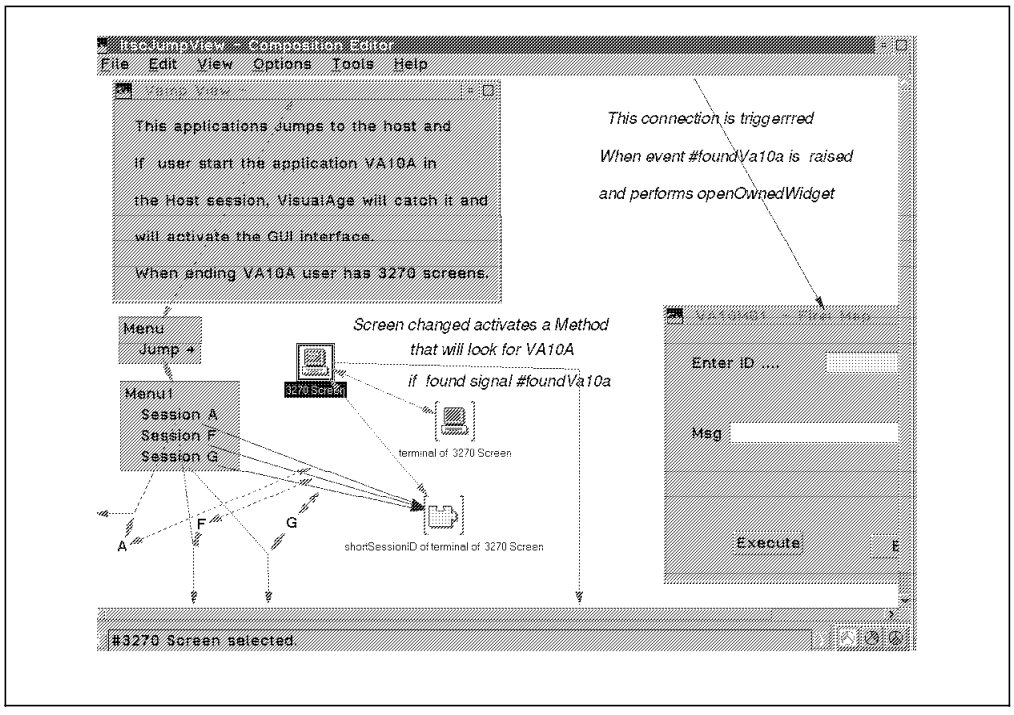

Figure 272. Completed Application

Note: We added comments (in *italic style*) to the free form surface to document the visual connections. This is a nice and easy way to document VisualAge applications but has the following drawbacks:

- Generates extra Smalltalk code, which could affect performance
- The comment positions are lost when you file-out/file-in your application.

C.1.1 Adding the Parts to the Free Form Surface

Perform the following steps to add the required parts to the free form surface:

1. Create the window and the menus and add an Abt3270Screen part to the free form surface.
2. Tear off the *terminal* and the *shortSessionID* attributes from the Abt3270Screen part.
3. Set the session ID for the Abt3270Screen part. It does not matter which session you choose; just be sure that the session is available and active on your PWS and on the PWSs of all users.

   You do not need to specify a key string because a script is used to verify the correct host screen.

   The *Screen Settle Time* must be specified to avoid multiple *screenChanged* events when a host screen is modified.

Figure 273 shows the settings for the Abt3270Screen part.

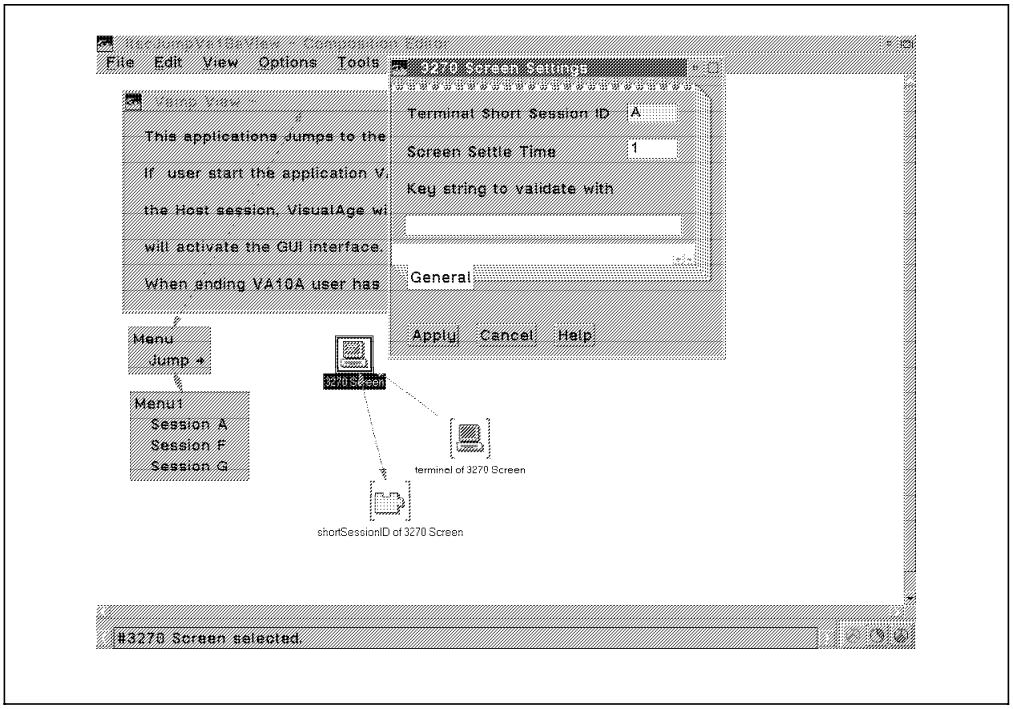

Figure 273. Abt3270Screen Part Settings

C.1.2 Creating the jumpSession: Method

The *jumpSession:* method is executed when the user selects a session from Menu1. You should use the Script Editor to create that method. Figure 274 on page 231 shows the *jumpSession:* method.

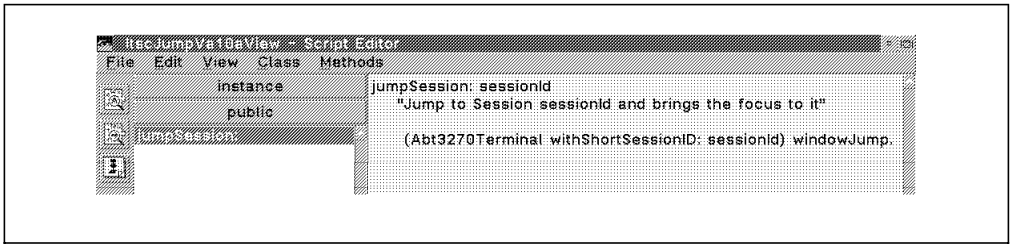

Figure 274. jumpSession: Method

C.1.3 Creating the Event-to-Script Connection

Create an event-to-script connection for each push button on Menu1 as follows:

- *#PbSessionA* (#clicked) to method (#jumpSession:).
  The event-to-script connection is incomplete at this point, and you must provide a session ID as a parameter. You will do this in the next step.

Repeat this step for all push buttons on Menu1. Figure 275 shows the settings for one event-to-script connection.

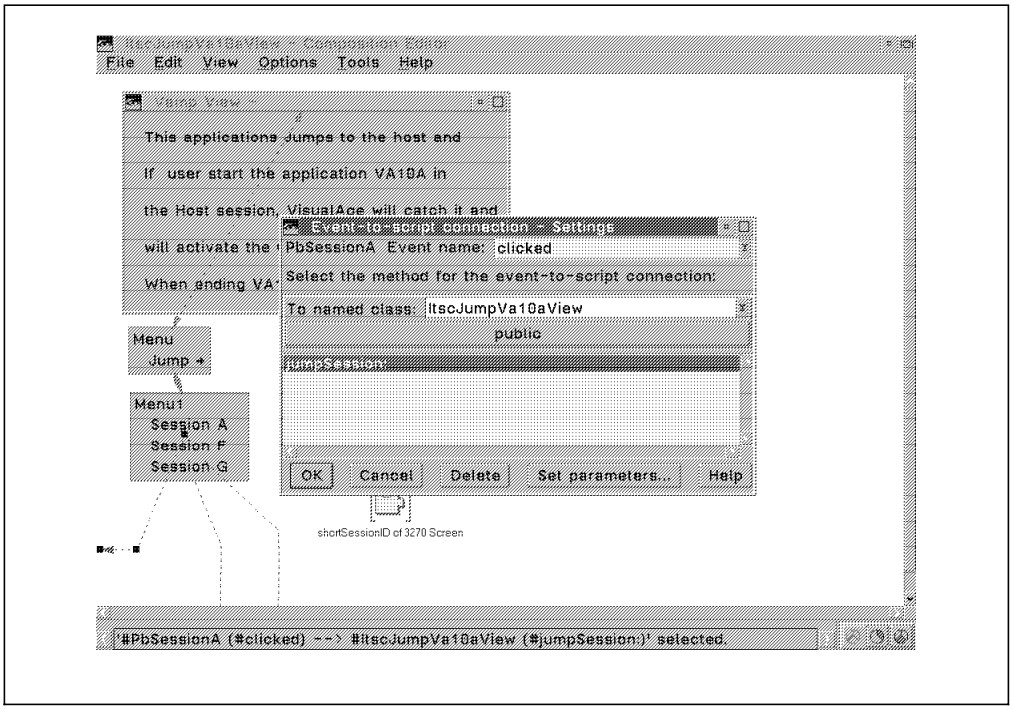

Figure 275. Event-to-Script Connection for the jumpSession: Method

C.1.4 Assigning Session IDs

To create the parameter (session ID) for the *jumpSession:* method follow these steps:

1. Drop a *Label* part for each session ID onto the free form surface.
2. Set the labels to the session ID characters using the Settings dialog.
3. Connect the labels to the incomplete event-to-script connections to provide the required parameters. The connections are now complete. Figure 276 on page 232 shows one complete event-to-script connection.

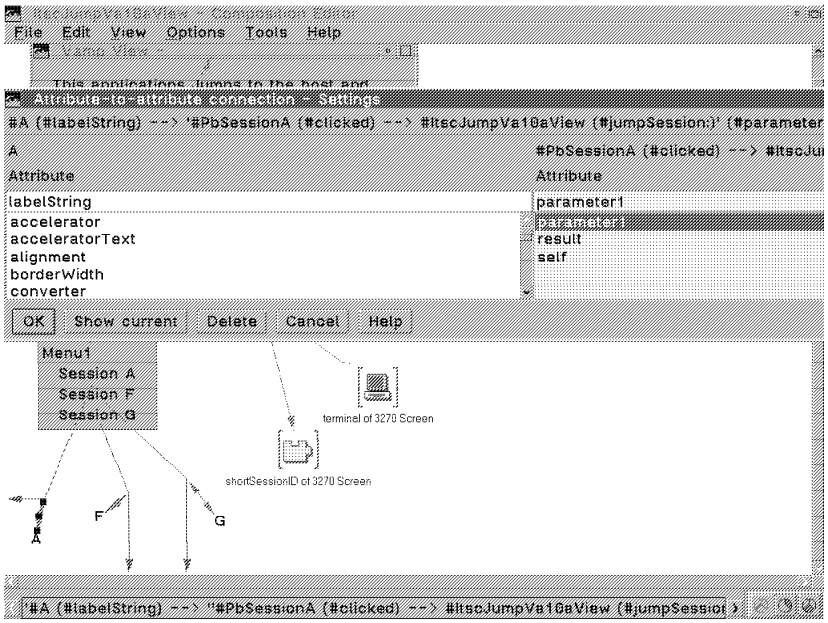

Figure 276. Providing the Session ID Parameter

4. You must also pass the session ID to the torn-off *shortSessionID* attribute of the Abt3270Screen part. Make the following connections:
   - Connect *PbSessionA* (#clicked) to shortSessionId of 370 Screen (#self)
   - Connect *A* (#labelString) to the above connection (#value).

   Repeat this step for all session IDs. Figure 277 shows the connections.

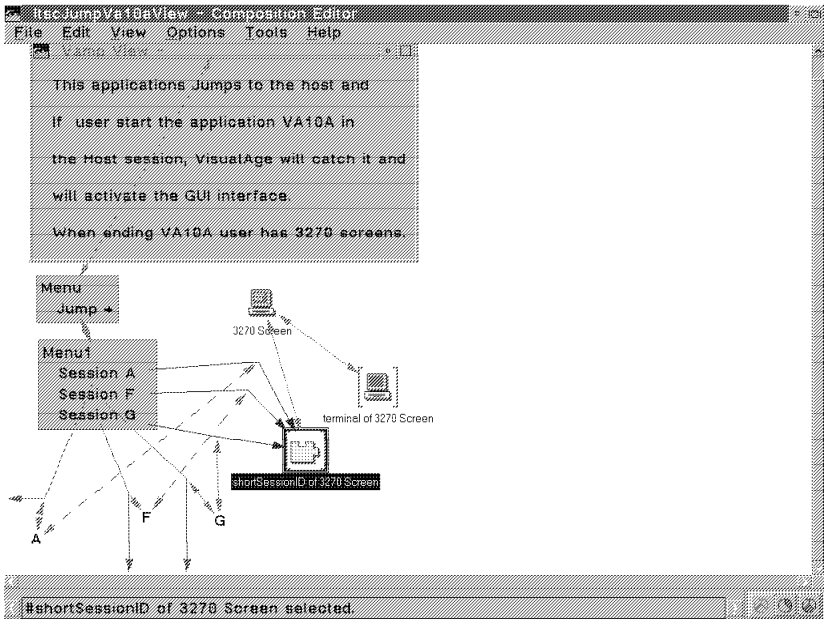

Figure 277. Assigning Session IDs to the Abt3270Screen Part

C.1.5 Identifying the Host Screen

Every time the host screen changes you must check to see whether the map that triggers the GUI application is shown on the host. You can implement this check in different ways. We decided to create a method to find the key string on the host screen because that allowed us to implement our example with just one Abt3270Screen part and to implement more GUI windows for other host applications by just adding code to the method.

We implemented the *findMapInHost* method, which checks each host screen for a defined key string and raises the *foundVa10a* event when the key string is found. Figure 278 shows the *findMapInHost* method.

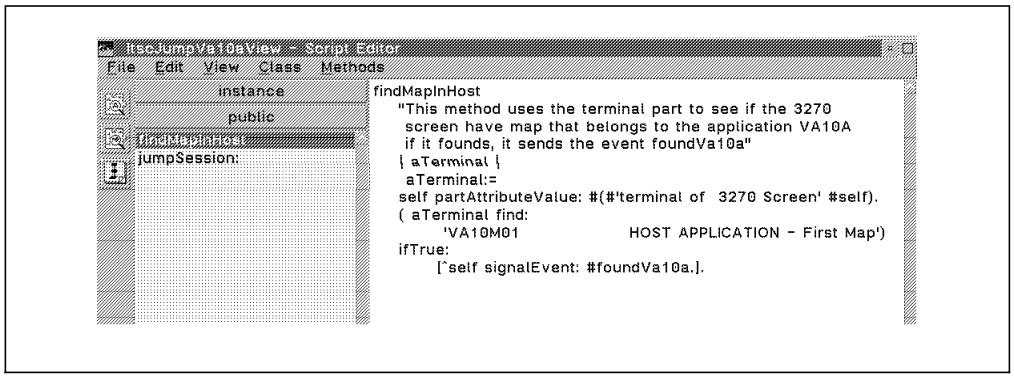

Figure 278. findMapInHost Method

C.1.6 Creating the Event-to-Script Connection for the Abt3270Screen Part

Each time the *screenChanged* event is raised, the *findMapInHost* method must be executed. We defined that with an event-to-script connection as follows:

- *3270 Screen* (#screenChanged) to method (#findMapInHost).

Figure 279 on page 234 shows the event-to-script connection.

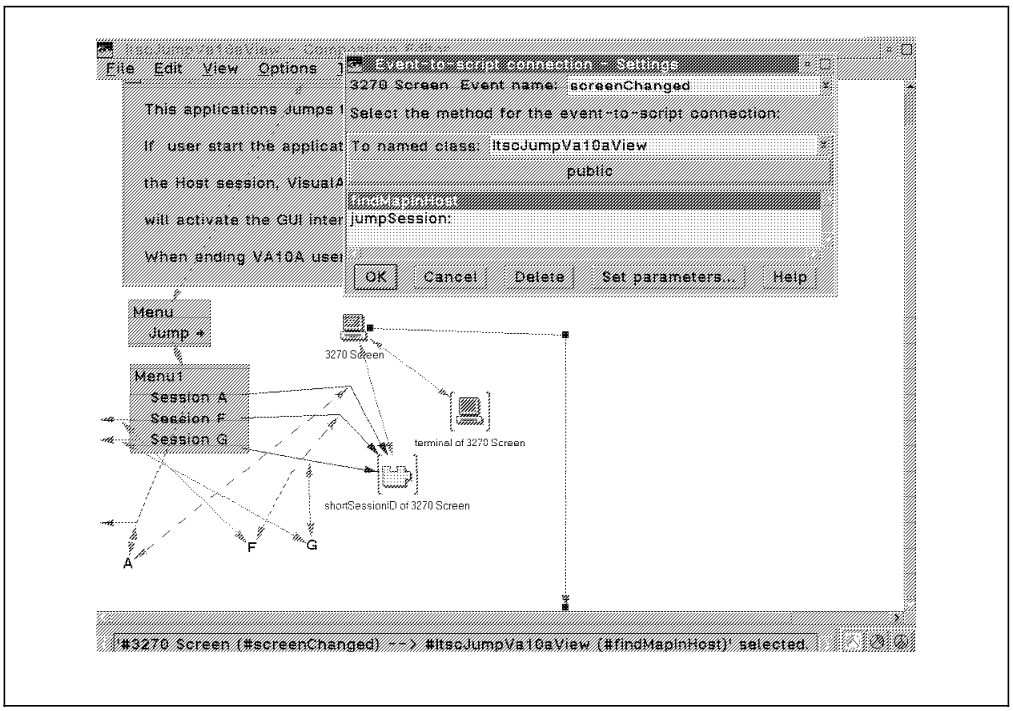

Figure 279. Event-To-Script Connection for the findMapInHost Method

C.1.7 Adding the foundVa10a Event to the Public Interface

We added the *foundVa10a* event to the public interface of the visual part (ItscJumpVa10aView) so that we could use this event to open the first GUI window of our application with a visual connection. Perform the steps below to add the *foundVa10a* event to the public interface:

1. Switch to the public interface editor.
2. Select the *Event* tab.
3. Type *foundVa10a* in the *Event name* field.
4. Press the *Add with defaults* push button.

Figure 280 on page 235 shows the results of performing these steps.

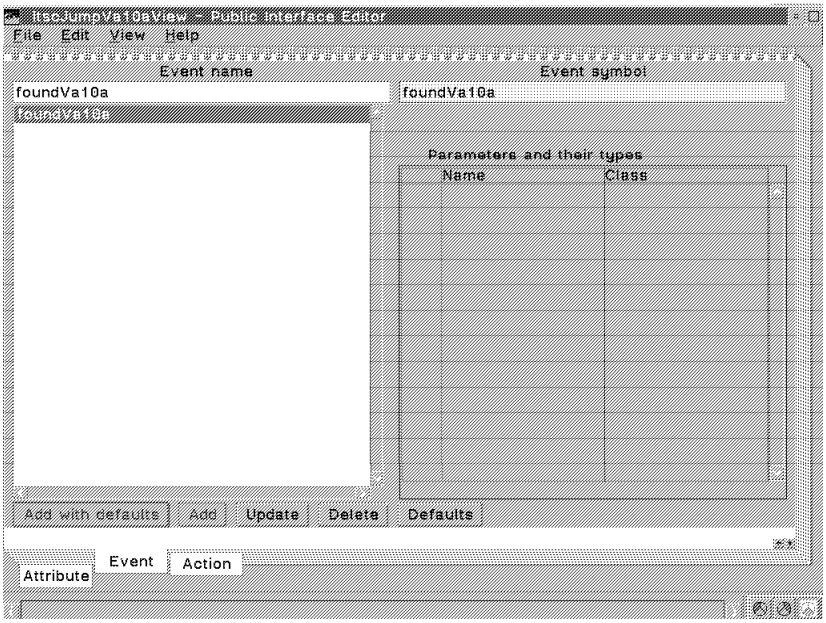

Figure 280. Adding the foundVa10a Event to the Public Interface

C.1.8 Adding the GUI Application to the Free Form Surface

We added to the free form surface the already coded GUI application that handled the host Va10A application. Figure 281 shows the dialog to add the GUI application to the free form surface.

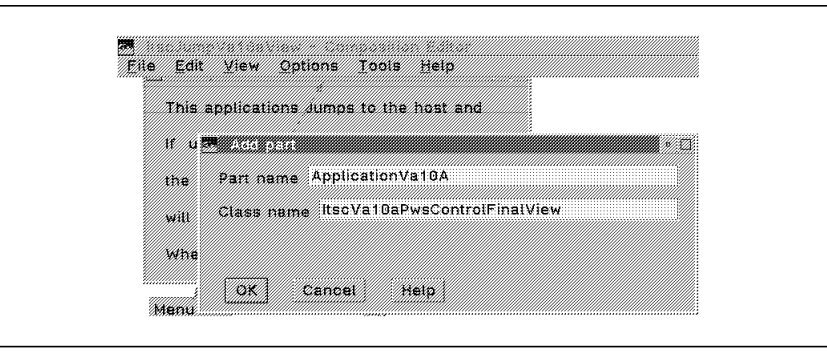

Figure 281. Adding the GUI Application

C.1.9 Connecting the Parts

We added the following connection to open the first GUI window of our application whenever the *foundVa10a* event is raised:

- Connect *#itsJumpVa10aView*(#foundVa10a) to *ApplicationVa10A* (#openOwnedWidget) (see Figure 282 on page 236).

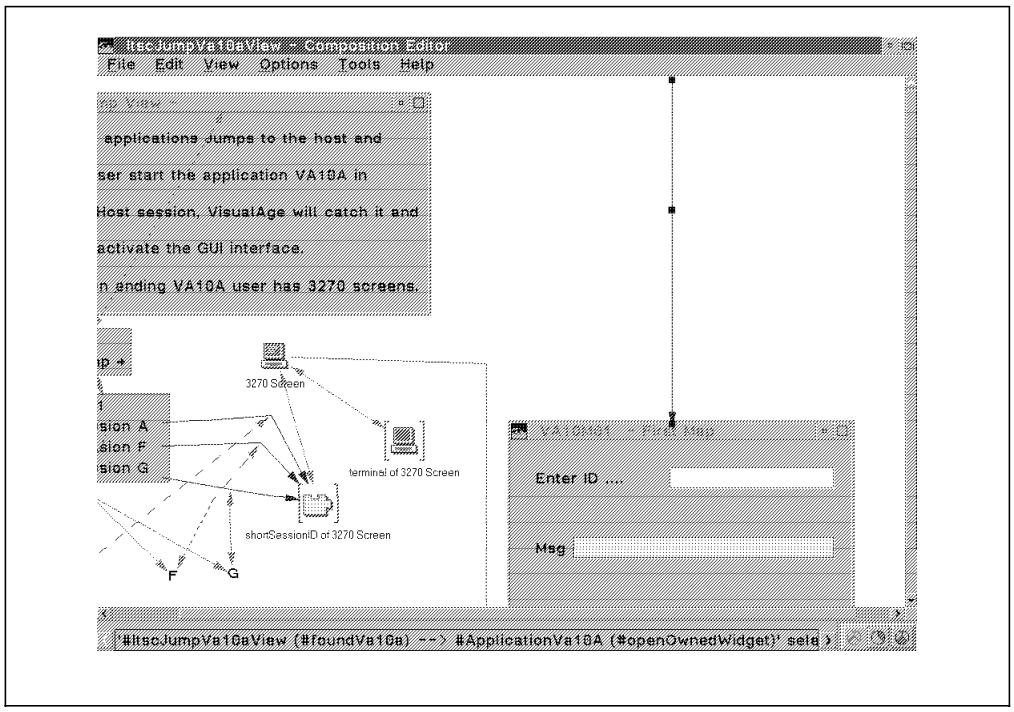

Figure 282. Connection foundVa10a to openOwnedWidget

C.1.10 Testing the Application

Figure 283 through Figure 285 on page 237 show the application at run time.

The user jumps to session G by selecting Session G from the drop-down menu (see Figure 283).

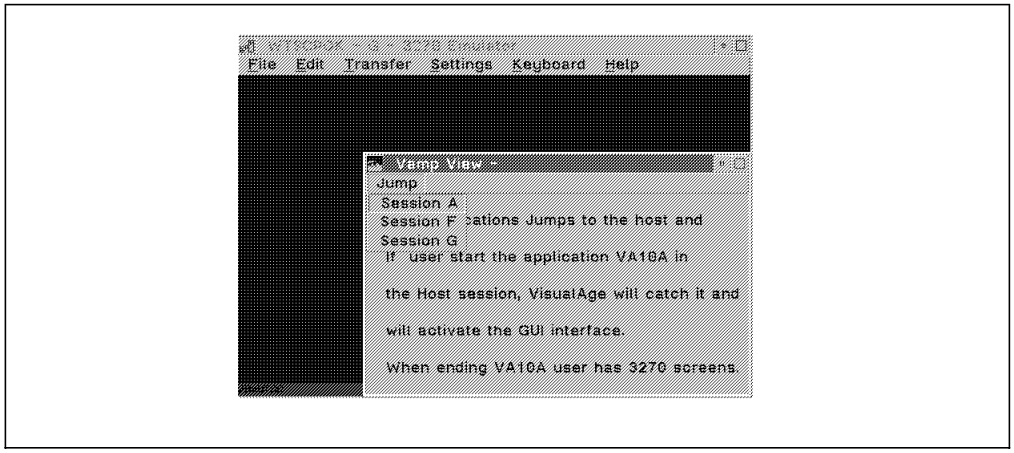

Figure 283. User Selects Host Session G

Session G is in control. The user works with the host, and the VisualAge GUI application is an active OS/2 task looking for application VA10A. Figure 284 on page 237 shows host session G active, running a host application.

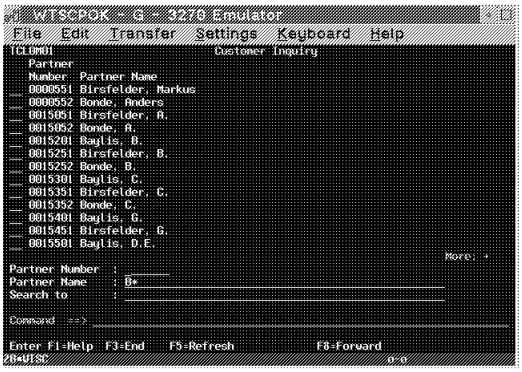

Figure 284. User Uses Host Session G

When application VA10A executes on the host, the VisualAge application takes control and shows the GUI window (see Figure 285).

Figure 285. GUI Application in Control

Control goes back to the 3270 session when the user selects the *End* push button.

List of Abbreviations

APPC	Advanced Program-to-Program Communications
CICS	Customer Information Control System
CSP/AD	Cross System Product/Application Development
CUA	Common User Access
DB2	DATABASE 2
DLL	dynamic link library
DRDA	Distributed Relational Database Architecture
DUW	distributed unit of work
EHLLAPI	emulator high-level language application programming interface
GUI	graphical user interface
host PS	host presentation space
IBM	International Business Machines Corporation
IMS	Information Management System
ITSO	International Technical Support Organization
LUW	logical unit of work
MVC	Model-View-Controller
NPT	nonprogrammable terminal
OSF	Open System Foundation
OS/2	Operating System/2
PWS	programmable workstation
RUW	remote unit of work
SAA	Systems Application Architecture
SPOC	single point of control
SQL	Structured Query Language
TSO	Time Sharing Option

Index

I

J

K

L

M

N

O

P

R

S

T

U

V

ITSO Technical Bulletin Evaluation RED000

VisualAge: Building GUIs for Existing Applications

Publication No. GG24-4244-00

Your feedback is very important to help us maintain the quality of ITSO Bulletins. **Please fill out this questionnaire and return it using one of the following methods:**

- Mail it to the address on the back (postage paid in U.S. only)
- Give it to an IBM marketing representative for mailing
- Fax it to: Your International Access Code + 1 914 432 8246
- Send a note to REDBOOK@VNET.IBM.COM

Please rate on a scale of 1 to 5 the subjects below.
(1 = very good, 2 = good, 3 = average, 4 = poor, 5 = very poor)

Overall Satisfaction ____

Organization of the book	____	Grammar/punctuation/spelling	____
Accuracy of the information	____	Ease of reading and understanding	____
Relevance of the information	____	Ease of finding information	____
Completeness of the information	____	Level of technical detail	____
Value of illustrations	____	Print quality	____

Please answer the following questions:

a) If you are an employee of IBM or its subsidiaries:

Do you provide billable services for 20% or more of your time? Yes____ No____

Are you in a Services Organization? Yes____ No____

b) Are you working in the USA? Yes____ No____

c) Was the Bulletin published in time for your needs? Yes____ No____

d) Did this Bulletin meet your needs? Yes____ No____

If no, please explain:

_____

_____

What other topics would you like to see in this Bulletin?

_____

_____

What other Technical Bulletins would you like to see published?

_____

Comments/Suggestions: (THANK YOU FOR YOUR FEEDBACK!)

Name	Address
Company or Organization	
Phone No.	

Cut or Fold
Along Line

Fold and Tape **Please do not staple** Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

IBM International Technical Support Organization
Department 471, Building 070B
5600 COTTLE ROAD
SAN JOSE CA
USA 95193-0001

Fold and Tape **Please do not staple** Fold and Tape

Cut or Fold
Along Line

Printed in U.S.A.