Take Note!

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

First Edition (March 1994)

This edition applies to Version 1.0 of VisualAge Team, Program Number 5621-388 for use with the operating system OS/2 Version 2.1.

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

An ITSC Technical Bulletin Evaluation Form for readers’ feedback appears facing Chapter 1. If the form has been removed, comments may be addressed to:

IBM Corporation, International Technical Support Center
Dept. 91J, Building 235-2 Internal Zip 4423
901 NW 51st Street
Boca Raton, Florida 33431-1328

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

© Copyright International Business Machines Corporation 1994. All rights reserved.
Note to U.S. Government Users — Documentation related to restricted rights — Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.
Abstract

This document describes VisualAge, a high-productivity application development tool for workstation applications in a client/server environment, based on the new construction from parts technology. Applications are created by laying out reusable components on a design surface, and then developing functional relationships among parts by drawing lines between them. The user interface is constructed in a similar way. Connections are made between user interface parts and from user interface parts to nonvisual parts to specify the behavior of the application when it runs. This document provides a description of the concepts and features of the product from a high-level overview to its more detailed aspects, with focus on its most innovative aspects: visual programming and construction from parts.

This document was written for people involved with application development who need information about new tools and directions, and need to build high-function client/server applications.

PS

(117 pages)
# Contents

Abstract ............................................................... iii  
Figures ................................................................. ix  
Tables ........................................................................ xi  
Special Notices ......................................................... xiii  
Preface ........................................................................ xv  
How This Document Is Organized ................................... xv  
Related Publications .................................................. xvii  
International Technical Support Center Publications .......... xvii  
Acknowledgments ......................................................... xviii  

Chapter 1. Introduction .................................................. 1  
1.1 Application Scenarios .............................................. 1  
1.1.1 Workstations ..................................................... 3  
1.1.2 The Information System Department ......................... 4  
1.2 VisualAge: A Client/Server Application Development Tool .... 5  
1.2.1 VisualAge Highlights .......................................... 6  
1.2.2 VisualAge in Application Development ..................... 8  

Chapter 2. Application Development Themes ..................... 11  
2.1 Client/Server ......................................................... 11  
2.2 Iterative Development Process .................................... 13  
2.2.1 VisualAge in the Iterative Development Process ........... 14  
2.3 Visual Programming ............................................... 15  
2.4 Client/Server Power Tools ........................................ 17  
2.5 Construction from Parts Paradigm ............................... 18  
2.6 Object-Oriented Technology ...................................... 20  
2.7 Roles in Application Development ............................... 21  
2.7.1 Professional Environment Support in VisualAge .......... 23  

Chapter 3. Visual Tools Overview ..................................... 25  
3.1 Construction from Parts .......................................... 25  
3.1.1 Parts .............................................................. 25  
3.1.2 Application ...................................................... 29  
3.2 Visual Tools ......................................................... 29  
3.2.1 Part Composition Editor ...................................... 29  
3.2.2 Part Interface Editor .......................................... 32
6.2 Terminology and Concepts ......................................................... 76
  6.2.1 Public Interface of Parts ..................................................... 76
  6.2.2 Implementation Concepts ................................................... 77
6.3 Adding Elements to the Interface ................................................ 77
  6.3.1 Defining Primitive Parts ................................................... 78
6.4 Examples ................................................................................... 78
  6.4.1 Creating a Nonvisual Primitive Part ....................................... 79
  6.4.2 Defining the Interface for AddressBook .................................. 79
  6.4.3 Adding an Attribute to AddressBook ........................................ 80
  6.4.4 Numeric Key Pad ................................................................. 81
  6.4.5 Notes on Selecting Names ..................................................... 86

Chapter 7. Visual Tools: Customization ................................................. 87
7.1 A Quick Walk through the Script Language .................................... 87
  7.1.1 Variables .............................................................................. 88
  7.1.2 Practicing ............................................................................. 89
  7.1.3 Messages ............................................................................. 90
  7.1.4 Methods ............................................................................. 94
  7.1.5 Conditional and Iterative Control Structures ......................... 95
  7.1.6 Creating Instances of Parts from Code .................................. 98
7.2 Notes on Using the Script Editor ................................................... 99
7.3 Writing Scripts ........................................................................... 101
  7.3.1 Adding Code to an Action Selector in AddressBook ............... 102
  7.3.2 A Sorted List of People’s Names ............................................ 102
  7.3.3 Numeric Key ........................................................................ 104
  7.3.4 Numeric Key Pad ................................................................. 105

Chapter 8. VisualAge Supporting Components: Overview ....................... 107
8.1 Tools to Interface COBOL and C Programs .................................. 107
  8.1.1 COBOL and C Parsers .......................................................... 108
  8.1.2 Remote Programs (Foreign Structures) ................................. 108
8.2 Relational Database ..................................................................... 108
  8.2.1 Access to Transactions and Programs in a Network ................ 109
8.3 Client/Server ............................................................................. 109
  8.3.2 Facelift: EHLLAPI Support .................................................... 110

Appendix A. Visual Tool AddressBook Class ........................................ 111

List of Abbreviations ........................................................................ 115

Index ............................................................................................. 117
Figures

1. Typical Scenario ........................................ 2
2. End User’s Perspective .................................. 4
3. VisualAge Overview ....................................... 7
4. VisualAge in Application Development ................. 8
5. Application Development Themes ......................... 11
6. Client/Server Areas ...................................... 12
7. Visual Programming ....................................... 16
8. Construction from Parts .................................. 18
9. Application Developers and Users ....................... 22
10. Team Programming ....................................... 24
11. Examples of Parts ....................................... 26
12. Public Interface of Parts ................................ 27
13. Visual Tools .............................................. 30
14. Part Composition Editor ................................ 31
15. Parts Library .............................................. 35
16. Model/View Separation Advantages ..................... 38
17. View for the Address Book Application ............... 42
18. Address Book: Visual Notes ............................. 43
19. Address: Visual Notes .................................... 44
20. Composition Editor: AddressView Create Visual Part 48
21. Composite Part ........................................... 57
22. Constructing Parts: an Iterative Process ............. 59
23. Composition Editor: Variables, Attributes, Tearing Off Attributes 64
24. Making Connections: Plugs and Sockets ................. 68
25. Connections ............................................... 69
26. Connections to Scripts ................................... 71
27. List of Names of People in the Address Book .......... 72
28. Part Interface Editor: an Example ...................... 80
29. Numeric Key Pad: Visual Notes ......................... 82
30. NumericKey: Event with Parameter .................... 83
31. NumericKeyPad: Actions ................................ 85
32. Script Editor .............................................. 100
33. Sorted List of Names of People in AddressBook ....... 103
34. NumericKey: Event Code Hook ......................... 104
35. NumericKeyPad: disableDecimalKey Method ........... 105
## Tables

1. Tutorial Application: Parts, Attribute Names and Class Types ... 46
2. AddressBook Part: Actions ........................................ 46
3. Unary Expression ................................................. 90
4. Unary Expressions: Examples ...................................... 90
5. Binary Expressions ................................................ 91
6. Binary Expressions: Examples ..................................... 91
7. Keyword Expressions ............................................... 92
8. Keyword Expressions: Examples ................................... 92
This publication is intended to help customers use VisualAge to develop high-function client-server applications. 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 use of this information or the implementation of any of these techniques is a customer responsibility and depends on the customer’s ability to evaluate and integrate them into the customer’s operational environment. While each item may have been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the same or similar results will be obtained elsewhere. Customers attempting to adapt these techniques to their own environments do so at their own risk.

The following terms, 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
Common User Access
CUA
DB2
DB2/2
IBM
OS/2
SQL/DS
SQL/400
VisualAge

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

SYBASE SQL Server is trademark of Sybase Inc.
ORACLE is trademark of Oracle Corporation
Preface

This document is intended to provide information about VisualAge. The document opens with a high-level description of the product in relation to today’s application scenarios with regard to current themes and trends in the application development arena. The document also provides more detailed information on the process of building applications and on the usage of the visual tools.

This document is intended for people:

- Who need to understand trends and directions in application development.
- Who need a high-level description and an introduction to VisualAge.
- Who want to use the visual tools to construct applications.
- Who want to understand more of the design considerations.

How This Document Is Organized

The document is organized as follows:

- Chapter 1, “Introduction”
  This chapter opens with a brief description of the elements that affect computing scenarios, from the role of information system departments, to the kinds of applications that must be provided and to application development in general. The major market requirement is to quickly build client/server, user-oriented applications. The last sections of the chapter introduce VisualAge and highlight both its powerful functions and role in application development.

- Chapter 2, “Application Development Themes”
  This chapter introduces and describes some of the major themes and trends in the application development arena. It also outlines how VisualAge addresses many of these themes including client/server computing, low risk development, rapid application development, graphical user interface, object-oriented technology, and construction from parts.

- Chapter 3, “Visual Tools Overview”
  This chapter provides an overview of VisualAge’s library of reusable parts and its visual programming tools. After a more detailed description of construction from parts and the definition of a part in VisualAge, the
chapter focuses on the application builder that lets you compose, customize parts, and assemble applications in a visual way by using its 2-D graphical editor.

The Part Interface Editor, that is used to create and modify the interface of a part, and the Script Editor, that is used to enter scripts, are also introduced.

- Chapter 4, “A Sample Application”
  This chapter introduces an address book application that will be used as an example throughout the document to illustrate components of the visual tools. The chapter contains a description of the parts for this application as well as a step-by-step guide on how to build the visual parts.

- Chapter 5, “Visual Tools: Assembling”
  This chapter addresses aspects related to the assembly from parts. The scope of this chapter is to give some insights that may help in the part assembly phases, that include the search for parts in the catalog, the composition of parts and the making of connections among parts.

- Chapter 6, “Visual Tools: Using the Part Interface Editor”
  This chapter contains some hints on how to use the Part Interface Editor and concludes with some examples.

- Chapter 7, “Visual Tools: Customization”
  This chapter relates to the customization of parts by writing scripts to add or modify code.
  The chapter opens with an overview of the major elements and the syntax of Smalltalk language that is used to write scripts. The focus is on the language, not on the overall Smalltalk system, knowledge of which is not necessary at this stage.
  The editing of scripts, in the script language provided by VisualAge, is performed by using the Script Editor, as described in this chapter. Examples of scripts are given at the end of the chapter.

- Chapter 8, “VisualAge Supporting Components: Overview”
  This chapter contains an overview of the framework that VisualAge provides for application development. The chapter is intended to be an introduction for people who will fabricate parts that make use of the VisualAge components. It can also be read by those people who want to have a better understanding of the VisualAge components that support generic parts.
Appendix A, “Visual Tool AddressBook Class”

This appendix contains the sample code to fabricate the part AddressBook used in the address book sample application.

Related Publications

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

- *Object-Oriented Interface Design*, SC34-4399
- *Object-Oriented Software Development, A Practical Guide*, ZH20-9093

International Technical Support Center Publications

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

- *Cooperative Processing in an Object-Oriented Environment*, GG24-3801
- *Smalltalk Portability: A Common Base*, GG24-3903
- *Object Technology In Application Development*, GG24-4290
Acknowledgments

The advisor for this project was:

Dieter Neumann
International Technical Support Center, Boca Raton

The author of this document is:

Anna Ottobelli
IBM SEMEA

This publication is the result of a residency conducted at the International Technical Support Center, Boca Raton.

Thanks to all people in the VisualAge development team for all the support provided.

Special thanks to the following people for the invaluable advice, guidance and support provided in the production of this document:

Greg Bonadies - IBM Cary
Martin Nally - IBM Cary
Andy So - IBM Canada

Thanks to the following people for the advice, and support provided in the production of this document:

Markus Birsfelder - IBM Switzerland
Frank Haynes - IBM Cary
Hayden Lindsey - IBM Cary
Osamu Ochiai - IBM Japan
John O’Keefe - IBM Cary
Yen-Ping Shan - IBM Cary
Chapter 1. Introduction

The company that employs the average American [in general any individual worldwide] in the future will be flatter, leaner, and more aggressive than the company he or she works for today. It will have to be that in order to have the flexibility to respond to rapidly changing customer demands.

The processing, analysis, and decision making...will be moved to lower levels who will be aided in the performance of these tasks by new, sophisticated, and more “intelligent” software. [Workplace 2000. The Revolution of Reshaping American Business, Joseph H. Boyett and Henry P. Conn.]

There are major changes taking place in various areas of our society, mainly dictated by the need to make a profit in a rapidly changing environment. Companies must be flexible, adapt to rapid changes, integrate easily with other enterprises, and have a work force that can easily move from one job to another.

1.1 Application Scenarios

The changes noted above will have profound effects on computing scenarios, from the role of information system (IS) departments, to the kind of applications we must be able to provide and to application development in general. Figure 1 on page 2 shows a scenario that is becoming more and more common in the enterprise environment, namely, more powerful personal workstations with the need and ability to access various other systems and resources.

Most of the changes share one common element: large rigid entities are broken up into smaller autonomous elements that can still cooperate to form a complex system. Local and remote data and programs must be accessed and existing legacy programs must be reused and integrated. There are also new applications that must be integrated.

Another interesting phenomenon that is affecting the business is the rapid metamorphosis of companies. They form new subsidiaries, incorporate other companies, make alliances, etc. with no regard for national boundaries. Due to these new forms of cooperation among companies and subsidiaries, information must be effectively and rapidly shared among heterogeneous environments, thus raising the challenge of application solutions in an
inter-enterprise perspective and requiring applications to span systems connected via different protocols.

The new scheme requires very dynamic reconfigurations of elements in order to adapt to the fast changing environment. In this perspective we need more than ever to be able to design and implement applications that can be changed more easily and allow extensions to be added quickly and with low impact on the application itself.

The dynamic of the business environment also requires big changes in the organization of labor. People, for instance, will have to move more rapidly from one job to a different one, as required. As a consequence, they will have to quickly learn how to accomplish the tasks associated with their new jobs. To accomplish these transitions, applications must provide transparent access to both local and remote resources and other applications, with a consistent and seamless interface to the end user.

An approach to application development that would satisfy these requirements is to design and implement applications that are split among work-
stations and host systems, and are able to get information from heterogeneous servers. The workstation portion of the application, in most cases more subject to change, will support end users’ needs. The host portion, less subject to change, will provide access to resources, guarantee their integrity and enforce the enterprise’s business logic.

1.1.1 Workstations

We must be able to support the end user’s view of a workstation. For an end user, a workstation is the tool that helps him do his job in the most productive way. In other words, the workstation is the window into the enterprise which gives transparent access to information wherever it is located and however it is provided.

Usually, end users are not the individuals that write programs on the workstation, because conventional programming languages are difficult to learn and use. Nevertheless, the programmer of a workstation application is required to take the part of the individual end user. The success of the application will be largely judged by the extent to which it meets the end users’ needs. The goal of workstation applications is to provide the end user with the most flexible and powerful data access possible in the easiest fashion possible. Consequently, applications in the workstations must be user-centered, giving them access to the information they need in the most natural (for them), usable, and simple way. Applications must also provide intuitive user interfaces, that allow the end user to use new applications without lengthy and costly learning phases. Thus, there is an increasing need for powerful and innovative methods that let end users customize and tailor the application to suit specific tastes and needs without requiring programming skills.
1.1.2 The Information System Department

The information system department is clearly affected by all of the above mentioned changes. The IS cannot afford to be a self-centered department within the organization, or it will collapse under the burden of its own procedures, which are most often dictated by old-fashioned programs and systems. The IS must be a creative part of the organization and provide the means to support the organization’s needs and those of the end users.

Surveys taken in various countries show that an average of 80% of the developer’s time is dedicated to maintenance. In the current market, we cannot afford such a rate. Can we?

The dynamics of the market further increase the challenge for the IS. Incorporating new companies, for instance, implies that totally different IS departments may be acquired and integrated. On the other hand, users who change jobs require intuitive user interfaces that minimize the effort of learning new procedures. There is the need for flexible and consistent ways to access information within the enterprise that quickly adapt to its changes.
Host systems must start playing the role of servers, driven by requests, and workstation programs must be user-centered to allow optimum productivity.

Adoption of new technologies and a new approach in building software may help the IS. Emerging capabilities of devices and technologies will dominate the way people interact with information handling and with computers. They may also help the IS to play an active role within their companies. Client/server computing, enterprise networking, distributed databases, object-oriented technology, visual programming, multimedia, and pen-based systems, are some of these devices and technologies.

1.1.2.1 The Information System Role
The IS role is to provide support in this new trend. On the one hand, it ought to provide support to that part of the application focused toward the end user, with rapid and flexible application development. On the other hand, it must guarantee the physical, and, most importantly, the logical data integrity of the enterprise’s resources and the security of these assets. It must ensure that the enterprise’s business policies are applied.

The major challenges for the IS are to play a proactive role within the enterprise and make software a catalyst to rapidly exploit new opportunities and enable the enterprise’s growth, rather than be a bottleneck that stifles flexibility and responsiveness.

1.2 VisualAge: A Client/Server Application Development Tool
The ultimate goals of the challenges outlined above are products and tools for application development that can increase customers’ business productivity, reduce operating costs and allow application developers to better meet the needs of their end users.

One of the major market requirements is to quickly build client/server, user-oriented applications. This need is addressed by new tools that have started to emerge, that provide visual programming as well as the power to easily connect to remote applications, servers, and legacy code.

VisualAge is a client/server power tool, which means it is a high-productivity application development tool for workstation applications in a client/server environment (for a more detailed description of power tools, see 2.4, “Client/Server Power Tools” on page 17). It focuses on line of business applications, including online transaction processing (OLTP) applications as well as decision support applications. VisualAge enables professional developers to build quickly the client portions of line of business applications,
complete with a graphical user interface (GUI), application logic, and local and remote resource access. VisualAge provides a set of interactive development tools, a library of already-constructed parts, a set of powerful components for client/server computing, and a pure object-oriented development environment that all combine to provide a state-of-the-art environment.

1.2.1 VisualAge Highlights

Figure 3 on page 7 highlights the variety of components and tools provided by VisualAge that individuals in a development team would utilize during application analysis, design and implementation:

- **Visual Tools**: VisualAge provides a visual programming tool that allows you to create complete applications non-linearly using the exciting new technology called construction from parts.

- **Library of parts**: Already-constructed parts that are delivered include support for graphical user interfaces and generic parts for database queries, transactions, remote and local functions.

- **Graphical User Interface support**: The GUI support included in the library of parts enables the development of applications that support the 1991 version of the Common User Access* (CUA*) specifications with extensions to support smart entry fields, tables and forms.

- **Client/server and communication support**: VisualAge provides comprehensive support for client/server computing that is made possible over multiple protocols and programming interfaces, such as:
  - APPC (Advanced Program to Program Communications)
  - TCP/IP (Transmission Control Protocol/Internet Protocol)
  - NetBIOS (Network Basic Input Output Services)
  - ECI (CICS * External Call Interface)
  - EHLLAPI (Emulator High-Level Language Application Programming Interface)

- **Relational database support**: VisualAge framework includes support for local relational database support and queries. Remote databases can also be accessed transparently through this function. This support is used by VisualAge to provide visual programming parts that enable generic queries. Databases (DBs) currently supported are: DB2/2*, DB2*, SQL/DS*, SQL/400*, Oracle**, SYBASE SQL Server**.

- **Enhanced DLL (Dynamic Link Library) support**: This feature automates the definitions that are needed to interface a local C or COBOL DLL by building the necessary objects and behaviors for you. This feature is
used by VisualAge to provide the generic DLL visual programming part. The DLL enhancements also provide full multithreading support.

- **Records to objects mapping**: Whenever information must be exchanged between an object-oriented application and an application written with a traditional language, flat record structures must be mapped to objects and vice versa. VisualAge provides a tool that simplifies the building of the objects that can provide the mapping.

- **Team programming**: VisualAge provides advanced and comprehensive support for team programming with a central library of parts and classes in a networked development environment.

- **Configuration management**: Besides team programming, VisualAge provides support for version and release control with verification of prerequisites.

VisualAge exists in two editions:

1. VisualAge, an entry level product for individual programmers.

2. VisualAge Team, that provides support for team programming and configuration management.
Both editions include advanced test facilities, as well as performance and packaging tools.

1.2.2 VisualAge in Application Development

VisualAge is an extremely powerful application development tool that allows you to write applications that provide sophisticated functions with minimum coding. By customizing generic parts and interactively building the user interface, you will quickly implement applications with advanced graphic user interfaces in client/server environments.

The power of VisualAge goes further. For advanced applications with complex local business logic, VisualAge provides a pure object-oriented language, Smalltalk, that can be used to enhance and extend the applications that you create with the visual tools. From this perspective, VisualAge is also a tool that makes the transition toward the adoption of the object-oriented technology smoother and easier. The technology can be acquired and introduced gradually by steps and at the pace that is best for your organization.

![Figure 4. VisualAge in Application Development](image-url)
1.2.2.1 Transition to Object-Oriented Technology

The application builder included in VisualAge is an object-oriented visual programming tool. When you work visually with the application builder, you are creating true object-oriented applications composed of object-oriented classes. Using VisualAge’s DLL and networking interfaces, application developers can access code written in any programming language.

If you do not want to invest immediately in Smalltalk and object-oriented skills, you may decide to implement logic in another programming language, or you can invoke logic that is already written in another programming language. In this case, you use VisualAge to build views for your application that are directly connected to external resources and functions. Your application will be composed of visual parts, and parts that are easily customized from generic parts, and provide access to the database, to the business logic written in C or COBOL DLLs, and to your remote transaction programs.

As you progress, you may realize that business objects that implement the business logic and act as agents between views and external resources (DB, DLLs, transactions, etc.) increase the flexibility and reuse of your applications more than using connections that go directly from the views to external resources. At this stage, you may want to start designing and implementing the business logic within VisualAge that allows you to fabricate reusable custom parts in the parts catalog. Clearly, these parts will also be able to use the full set of VisualAge services and tools. Therefore, as you get more proficient and more ambitious with VisualAge, you will find that you can implement very sophisticated applications.

Surprisingly perhaps, the entire VisualAge environment is implemented using Smalltalk and the visual tools. The fact that VisualAge was implemented using its own language and tools also means that the knowledgeable user can use VisualAge to modify and enhance the functionality of the VisualAge system itself. This means that you are never limited in what you can develop, and you are not dependent on a vendor to provide you with missing functionality.
Chapter 2. Application Development Themes

VisualAge addresses (and is an answer to) many of today’s major themes and trends in the application development arena. They are: client/server computing, low risk development, rapid application development, graphical user interface (GUI), object-oriented technology, and construction from parts.

This chapter introduces and describes the aforementioned themes, while a more detailed description of VisualAge’s support can be found in VisualAge Supporting Components: Overview and Visual Tools Overview.

2.1 Client/Server

From the generic perspective of writing applications that span more than one system, there are three major models in client/server computing, as shown in Figure 6 on page 12.

1. Client applications that provide a new look to the user interface of old host applications, sometimes called a “facelift.”
There is indeed a great need to provide an advanced and user-friendly interface to old legacy applications that currently use character-based, non-graphical user interfaces. This may be needed, for instance, for applications that are not going to be renewed, or to start providing better user interfaces without changing the host programs and enabling new applications to be delivered in stages. The client part can make use of terminal emulator programming interfaces to simulate the operator operations. The legacy application runs unchanged, while the client can provide the desired user interface.

The penalty may be in degraded performance and in the complexity that derives from the rules imposed by the application to the user interface. This complexity becomes even greater if you want to map the heavily modal, application-driven flow of control to a flexible end-user driven flow of control. Last but not least, changes to the user interface in the server application require equivalent changes at the client side.

2. Remote resource sharing. The most typical shared resource is data.

The big demand for this comes from decision support applications, in which users need to freely read data from the enterprise’s databases and
use it in their local decision support tools. The answer to this demand is both in decision support tools and in programming tools. On the one hand, the decision support tools will extend their capabilities to support various distributed resource managers. On the other hand, programming tools will let you interface to those resource managers from your applications.

Importantly, all the business logic resides on the client side, unless the database manager supports features such as stored procedures that let you add business logic to database access.

3. New client/server applications with split logic and access to remote transactions and programs.

In this case the client application communicates with the application logic on the server side. These applications make use of an appropriate program-to-program communication protocol. The demand for this comes from applications that are critical for the enterprise operations. Examples may be order entry, inventory control, customer information systems, financial transactions, reservation systems, and so on. The major requirement is the enforcement of enterprise-wide business rules that can only be reached through business logic in the server side.

VisualAge provides support for applications in the three areas described above with its client/server and relational database components.

2.2 Iterative Development Process

The iterative development process (IDP) reduces risk in development. IDP is not a new approach in application development, but it is made more easily and successfully applicable whenever object-oriented application development is used.

Delivering a product in a timely manner is important, but of equal or greater importance is delivering the correct product. Since the definition of correct changes as the project progresses and requirements are better understood and may change, the development process must be able to accommodate these changes. The traditional waterfall approach freezes requirements early in the development process with resulting products that do not match end users’ needs.

The iterative approach allows progress by stages, at the end of which the result can be verified by end users. Even at early stages in development, prototypes can be delivered to end users. In this way, requirements do not have to be fully specified at the beginning; instead, they are dynamically
identified and refined and the product under development can easily incorpo-
rate new and better understood requirements. Usability and functions of
applications can be assessed early and throughout the development process.
The resulting product is more likely to be a valid solution to users’ needs.

2.2.1 VisualAge in the Iterative Development Process

VisualAge can be used effectively in the various stages of an iterative devel-
opment process.

The term prototype, as we use it, does not refer to some “quick and dirty”
code that is developed just to prove some concepts and then is thrown away.
A prototype, in object-oriented application development, is the first step in
building an application. You should be willing to discard part of it, redo
some parts, and reorganize others. But you should be aware that you need
not throw everything away.

The process we describe is iterative and starts from a subset of the overall
application scenario.

During the analysis steps, rapid prototypes can be built by:

- Interactively constructing the user interface from parts.
- Constructing customized parts that represent existing external resources
  (DB, transactions, etc.) from VisualAge generic parts.
- Defining parts that represent business objects by simply specifying their
  interface, that is defining the elements and behaviors that are necessary
to interact with a part.
- Making the connections among the various parts to specify the behavior
  of the application.

All of the above tasks are done using the Visual Tool to produce a running
prototype that can be verified iteratively with end users.

During the design and implementation steps you can:

- Refine the user interface.
- Design and implement the necessary external programs.
- Replace direct connections from views to DBs with transactions when-
ever some central business logic must be enforced.
- Introduce business objects that replace direct connection from views to
  external resources.
• Design and fabricate the necessary parts that represent business objects and design the supporting objects.

These tasks are done using Visual Tool, other VisualAge components, and Smalltalk, as extensions and refinements to the prototype, without disrupting the user interface or anything that was built during the analysis stages.

2.3 Visual Programming

Visual programming tools have started to emerge in the market in response to two major requirements:

1. Facilitate the building of advanced user interfaces.
2. Lower the programming skill necessary to assemble and customize applications.

These tools make intensive use of metaphors and icons for computing. Metaphor in computing relates to the usage of visual representations that, for implicit comparison or analogy, give the user an immediate understanding of the entity, function, object, or computer processing. The term icon is used to refer to a pictorial representation of an object or a selection choice. Icons can represent objects that users want to work on or actions that users want to perform. A visual programming tool can be defined as a tool that provides users with a means to interactively specify programs in a highly graphical fashion. For example, routines, programs, and data have graphical representations, such as metaphors and icons. Relationships among these components are depicted graphically as well. The construction of programs is done graphically; that is, the programmer “writes” programs by manipulating and articulating graphical representations of components in an application, as shown in Figure 7 on page 16.

Visual programming tools differ significantly from those tools that provide program visualization, in which case programs are still written with traditional techniques and the tool is able to show a graphical view of them. Program visualization tools use graphics only to illustrate either some aspects of a program or its execution. These kinds of tools are commonly used for debugging and teaching.

Visual programming lets individuals take advantage of a larger spectrum of the capacities of the human brain than the one-dimensional textual form of traditional programming. The visual representation of problems is considered closer to people’s mental representations of problems. In addition to the graphical construction, visual programming tools usually provide scripts, as a way to describe those functions that cannot be expressed graphically.
Figure 7. Visual Programming

Scripts often are declared in fourth generation languages, and they come in different varieties. Some of these tools use proprietary languages, while others make use of, or are derived from, standard languages available in the market.

Almost all the visual programming tools available in the market offer an object-oriented interface to the user: programs, data, and routines are objects that the user selects and connects. However, not all the tools are based on object-oriented technology and not all of them integrate with an application development platform.

Visual programming tools acquire an even more interesting flavor when they, like VisualAge, are based on object-oriented technology and integrate with an object-oriented development environment. In this case, the tools provide a comprehensive and consistent approach to application development, in which everything (user interface, business and computing entities) at every stage is an object, thus avoiding the need to map from the conceptual view of problems to procedural representations.

For example, if we have to implement an invoice and its function using a traditional approach, we have to describe it, with a semantic gap between the
conceptual view of the invoice and the procedural way it is enabled by the traditional language. With visual programming tools this gap is eliminated, because the enabling procedures are embedded within the conceptual view.

2.4 Client/Server Power Tools

Several of the visual programming tools available today mainly address end users and focus only on helping them build graphical user interfaces. Often they provide database access for building an interface to a query result. Sometimes they help integrate local applications. Furthermore, the typical development environment addresses single programmers. It is clear from these observations that tools of this kind could hardly be used to implement complete applications that include business logic in a client/server environment.

We consider client/server power tools as programming tools that let you quickly write client/server applications with advanced graphical user interfaces. They allow the building of complete, industrial-strength line of business (LOB) applications.

A power tool meets the requirements for rapidly building user interfaces, customization and assembly of applications without the need of professional programming skills. It also provides a professional-level application development environment with the ability to integrate business logic and client/server kinds of applications and integrated support for team programming.

Characteristics of a power tool may include:

- Visual programming, for the construction of the user interface and the assembly of the application.
- Fourth generation scripting language.
- Support for implementing local business logic.
- Support for connecting to databases, preferably from multiple vendors.
- Support for the complete spectrum of client/server application models, using multiple communication protocols.
- Rapid application development by prototyping.
- Team programming.
- Configuration management.
- Packaging.
2.5 Construction from Parts Paradigm

Construction from parts is an application development paradigm in which the applications are assembled from reusable and existing software components (parts).

A part is a software object that has a number of external features that allow it to be connected to other parts in order to implement application scenarios. A part is not just an elementary component; it can be composed of multiple interacting subparts. We call this a composite part.

The process of building the application consists of:

• Selecting predefined parts that are necessary.
• Using them unmodified or tailoring them for specific requirements.
Establishing the connections among parts to create the application or a new part.

The process could be performed by writing code; however, visual programming tools are much more suitable for supporting the phases of the construction.

Even though these concepts are rather new to software development, they are not new to the industry and are commonly used in manufacturing. Figure 8 on page 18 shows the analogies between the construction of a personal computer and the construction of an application. For instance, parts correspond to chips, composite parts to cards and the application to the complete personal computer.

To build a new personal computer, who would ever design and construct every single component from scratch? Who would do so in software to build a new application? Typically, only a few standard subroutines and system services are likely to be reused. Most of the application is developed from scratch and most of the effort is expended in re-writing code that already exists somewhere, often within the same company.

The benefits of the construction from parts paradigm include:

- Reduction of application development costs.
  The assembly and the tailoring of parts does not require a professional programmer. Programming will be done by exceptions and the skill of professional programmers can be applied to build innovative components when necessary.

- Enhanced application quality.
  The reuse of existing parts reduces the chance of errors. Within a short time, parts will become more and more solid, and almost error free. Based on the obvious idea that if we do not write new code, then we do not introduce new errors, we can conclude that the less new code we have to write for a new application, the fewer errors we will encounter.

- Reduced cycle time with faster and better response to end users’ needs.
  The rapid development of applications made possible by visual programming tools and existing parts is invaluable to quickly verify user requirements and deliver applications in a short time.

The success of the paradigm in software development depends on various factors. First, interactive tools for visual constructions must be available. The tools must integrate with the development platform to design and build...
parts and frameworks. Second, interfaces and messaging protocols must be specified and supported by an architecture for interoperability of tools and component parts. Finally, a set of standard parts must be available and the software providers must move towards the building of components.

### 2.6 Object-Oriented Technology

Without a new approach to application development, the application development effort will hardly be able to deal with the current complex environments that include graphical user interfaces, remote data access, client/server computing, heterogeneous environment and reuse of legacy systems. Object-oriented technology simplifies the software realization of complex environments because the global problem can be approached by isolating smaller and easier to understand self-contained subproblems rather than decomposing the global problem into functions. Furthermore, developing object-oriented applications occurs through the construction of reusable components that become assets and speed up the development cycle.

Object-oriented technology is receiving a great deal of attention in the software development community because it simplifies our view of the problem space and helps translate our view of business entities into software systems. As a result, uniform ideas and concepts are applied throughout the software development cycle, from the requirements phase to the implementation. This avoids the contrived representations and discontinuities commonly imposed by traditional development methodologies, in which the conceptual view of the world must be mapped to a procedural and descriptive paradigm.

The fundamental concepts of object-oriented technology are summarized below:

- **Objects** are self-contained software entities that are characterized by their attributes and behavior, while hiding their implementation details.

- Objects interact by sending **messages** to each other. Objects execute the requests by executing operations within them that are called methods. The same message can be sent to different objects. The object receiving the message knows how to behave, while the object sending the message does not have to be aware of the type of the object to which it is sending the message (**polymorphism**). This is an extremely powerful mechanism, which is key to the realization of interchangeable components.
• Objects can be classified, based on common behaviors and attributes, ignoring details and aspects that are not of interest at the moment. A **class** can be seen as a model that describes the structure and the behavior of the object. Classes are organized into hierarchical trees in which lower level classes are specializations of the classes above them.

Object-oriented technology has been used successfully and has been proven to be a technology that enables rapid application development, low risk iterative development, as well as the design and implementation of graphical user interfaces. It also facilitates the handling of the higher degree of complexity inherent, for instance, to client/server application development.

Some of the new visual programming tools ease the transition to the adoption of object-oriented technology and concretely provide a major step towards the most significant promise of the technology: the construction of applications from existing reusable components.

### 2.7 Roles in Application Development

Among the effects of the construction from parts paradigm and the adoption of visual programming tools, there are changes we foresee among individuals involved with, or somehow related to, applications and application development. New roles will also be identified, because the new tools address non-programmers as well as programmers.
Figure 9. Application Developers and Users

The left part of Figure 9 shows the current dichotomy of software developers and end users. The population of application developers and application users can be represented as a pyramid with a line that clearly separates the two worlds. Above the line are computer professionals, who know computer languages and how to write applications. Below the line are individuals that use applications. Most likely, they know what applications should do, but they do not know how to develop one and cannot afford to learn complex programming languages and techniques that are beyond the scope of their responsibilities.

The object-oriented technology alone certainly increases the productivity of application developers and, if properly applied, gives end users better and more timely results, but it does not change the position of the line.

The advent of power tools in object-oriented environments, however, will help to lower the line and expand the number of people who will be able to assemble and customize their own applications. Two new lines appear in the pyramid on the right side of the figure. The bottom line shows how individuals with less skill in programming will be able to assemble applications.
The upper line shows how analysts, user interface specialists and individuals within IS that support end users will be able to build prototypes and perform more sophisticated customizations by using existing parts. High-level programming skill is required only for the “fabricators” of parts.

As a result, we foresee three different and complementary uses of power tools that provide visual programming and construction from parts in an object-oriented environment:

1. The assembly of an application and the construction of composite parts from existing parts as they are. At this level, users must know the application environment, must be able to find existing parts and know how to establish connections using the construction tool.

2. Addition of custom logic while refining and customizing parts. In this case, users must know more about the script language and have some notion of object-oriented technology.

3. Fabrication of new parts. Individuals responsible for this must be skilled in object-oriented technology and focus on reuse. Even though the new technology helps, reuse does not happen by accident.

2.7.1 Professional Environment Support in VisualAge

VisualAge provides an advanced environment for team programming and configuration management.

It is generally accepted that large teams of programmers have difficulties developing consistent and valid software. However, the team cannot be reduced to a single programmer, which is the environment most commonly addressed by visual programming tools available in the market today.

Here are some highlights of VisualAge’s support:

- A central repository in the local network that stores classes and allows multiple programmers to share classes and update them. In a traditional development environment, the unit of sharing is a file. In an object-oriented environment a natural unit of sharing is the class, on which multiple programmers may have the need to work concurrently.

- Aggregation of multiple classes into subsystems (called applications) with an enhanced browsing tool that shows only the classes and the methods defined and extended within the subsystem.

- Ownership of classes and subsystems by a developer who has the final responsibility for the integrity of them.
Support for version control and releases of classes and subsystems, plus support for prerequisites.
Chapter 3. Visual Tools Overview

VisualAge’s visual programming tool lets you compose, customize parts, and assemble applications in a visual way by using its 2-D graphical editor. By allowing you to combine parts visually, without writing procedural code, the visual tools take away much of the tedium and error-prone detail from application programming, especially user interface programming, allowing you to concentrate on the essential capabilities of your application.

3.1 Construction from Parts

Construction from parts refers to the ability to create application programs by combining existing software components rather than creating the logic of the application from scratch (see 2.5, “Construction from Parts Paradigm” on page 18) . Before describing the visual tools, let us expand on the construction from parts paradigm and describe what parts are in VisualAge.

Whenever you build an application from parts, you select those parts (most likely from the parts palette) that are necessary and meaningful for the application. You decide which interactions should take place among the selected parts in order to provide the desired results, and establish the necessary links by making connections among them. Usually, the process is an iterative one that consists of building parts, which will be reused.

Often, in a production development environment, where custom parts are available together with standard vendors’ parts, the task of composing and assembling is all that is needed to implement a new application scenario.

In some cases, however, you may need to customize parts, either by adding some features to the interface of an existing part, or by using scripts to add some simple behavior to the part you are building.

3.1.1 Parts

In VisualAge a part is a class with a well-defined public interface, which supports a simple and architectured messaging protocol. We use the term subpart to refer to a part taken from the palette and used to build a composite part.

Parts can be very simple or highly sophisticated to provide a wide range of functions, as shown in Figure 11 on page 26. Parts, for instance, can be as simple as a text-entry field or a default window. Often, parts are composed of multiple interacting subparts. Thus, they can be a little more complex,
Figure 11. Examples of Parts

such as a person view that may include multiple text-entry fields for names and telephone numbers and, possibly, views for addresses. Furthermore, they can be as complex as a mail system, for instance, or as a protocol-independent client/server component. Parts can also represent (wrap) programs written in COBOL or C language, thus allowing the reuse of existing code in a construction from parts paradigm.

3.1.1.1 Public Interface of Parts

The public interface of parts refers to the features that are used to connect parts among them.

To specify the public interface of parts, the VisualAge introduces three clearly defined features: attributes, actions and events that correspond to a natural way of seeing parts and objects in general, and expressing the possible interactions among objects. The way to access a part is through the names of its attributes, actions and events, as shown in Figure 12 on page 27.

1. Attributes

Attributes are the logical properties of parts. At a conceptual level, attributes are an important and integral aspect of the objects’ semantic defi-
Attributes are objects that the part can return or set upon request. The part may also need to signal other parts that the attribute has been changed.

2. Actions

Actions are the behaviors of the parts, which means the services or operations the part may be requested to perform.

3. Events

Events provide a notification mechanism. They are used to signal that something has happened to the part. For user interface (UI) parts, they are often related to some user interaction, such as the clicking of the mouse on a push button, the selection of a check box, the opening of a window, and so on. Events are used to trigger some action. As an example, you may want a detail window to be shown whenever the user selects an item in a list box.

The introduction and support of the three types of features of the public interface facilitate the design of parts and the establishing of interactions among them.
3.1.1.2 Types of Parts
Parts can be grouped into two major types: visual parts and nonvisual parts. The major difference between them is the capability of visual parts to present a graphical "view" to the end user at run time.

1. Visual parts:
   - Have a run-time view, such as a list box, a window, a view of an address or person, and so on.
   - Understand a protocol that lets them be "edited" in their visual run-time form.

2. Nonvisual parts:
   - Usually do not have a run-time view. Examples are business logic objects, such as an address or a person, or parts that represent generic database queries or generic DLLs.

   Note: Parts that have a run-time view but do not support the editing protocol (such as a window not built with the application builder) are treated as nonvisual parts. This implies that any part built outside the visual tools is treated as nonvisual by the visual tools.

3.1.1.3 Primitive Parts
There are parts that constitute the basic units from which the other parts are constructed. We call these parts primitive parts. Examples are the basic visual parts, such as the text-entry field, the default window, the push button, the list box, some nonvisual parts, etc.

When a new primitive part is required, it has to be fabricated using a programming language, and its part interface defined. At this point, the new part is available for reuse. There are no limits on how many or which kinds of primitive parts can be fabricated.

If the primitive part is written in a traditional language, such as C or COBOL, a simple wrapper has to be made available in Smalltalk. The wrapper can be easily constructed by customizing the generic part provided by VisualAge (see 3.3.2, "Nonvisual Parts" on page 36).
3.1.2 Application

The term application is commonly used to refer to a collection of software components used to perform user-oriented tasks on a computer.

VisualAge supports the concepts of application also from an application development perspective. An application is a collection of parts that can be managed as a whole, and can be packaged to produce the run-time application that will be distributed to end users.

The VisualAge Team provides further support to manage applications. Developers can arrange and group their software in clearly organized components by specifying prerequisites, versions, and editions.

3.2 Visual Tools

Figure 13 on page 30 shows the three editors that are available within the visual tools. They provide editing facilities to perform the three different steps described in 3.1, “Construction from Parts” on page 25:

1. Part Composition Editor, used to edit a part built with the Visual Tool.
2. Part Interface Editor, used to edit the interface of parts.
3. Script Editor, used to edit scripts, that are fragments of textual code.

When you edit a part that was built and composed with the visual tools, VisualAge recognizes it and opens on the Composition Editor. If needed, interface and script editors can then be activated with a simple selection. If you edit a part not built with the Composition Editor, such as a primitive part, the visual tools open on the Public Interface Editor. In this case, only the script editor can be selected.

An important advantage is that application builder is also able to handle parts that are not built with application builder itself, and lets you use them to compose your new parts. In fact, any class can be used as a part, after defining its public interface.

3.2.1 Part Composition Editor

The Composition Editor allows you to compose parts and intermediate parts that are necessary for the application by selecting parts from the parts palette and establishing the necessary connections. Furthermore, the editor lets you edit an existing part, add or delete subparts, and add, delete, or modify connections.
You create parts for your applications by laying out reusable parts on a design surface, and connecting them together by drawing lines. You draw the user interface of the application by laying out user interface parts relative to each other. Connections are made between user interface parts and from user interface parts to nonvisual parts to specify the behavior of the application when it runs.

The editor has a free-form surface in which you place parts selected from the palette (see Figure 14 on page 31). Whenever you place a visual part onto the free-form surface, the editor can display its run-time view, and let you act on it, by specifying options, making connections, and adding other visual parts. Nonvisual parts are represented by icons since they do not have a run-time view. If you need to modify the implementation of a subpart, you just select the option Edit part from the context menu. Changes applied to the subpart will be immediately available to all the parts that make use of it, that implies that changes made locally are applied globally.

**Nested Visuals:** When you construct the visuals of your part, you can nest them. This means that a larger visual component can include more than one visual part. For instance, the default window in Figure 14 on page 31 contains a text-entry field, a list box, and two push button parts. One advantage
of this approach is that the same view can be reused in different visual parts, by nesting it within different contexts.

**Connections:** After you have put the subparts in the layout, you can start making connections among interface features of subparts, that is attributes, actions and events.

- **Attribute-to-attribute:** This connection ties attributes together and keeps them synchronized at run time. As an example, you may connect the attribute name in the business object Person, to a text-entry field in a visual part. This will allow the view, at run time, to display the content of the business object attribute and request the business object to set the new value when it is entered by the end user. In addition, if the attribute in the business object changes, the view (and all the views connected to it) will be notified.

- **Event-to-action:** This connection is used to trigger an action in a subpart as a consequence of an event. For instance, when the user clicks on a...
push button, you may want a new window with a list of the possible choices, or you may want an address book to position on its first entry.

- **Event-to-attribute**: This connection is used to set an attribute whenever an event occurs. As an example, when the user clicks on a numeric key, you want an accumulator to be set to the numeric value of the key.

- **Attribute-to-action**: These connections are used to trigger an action whenever an attribute changes.

**Connections to scripts**: In some cases you may need to “attach” some code to an event or to an attribute. To support these situations, VisualAge introduces the concept of **scripts** and lets you make connections to code you may write using the Script Editor. The Composition Editor provides two ways to hook to some code in a part, which satisfy two possible cases:

1. **Event-to-script connection**: These connections are used when some processing needs to be performed when an event occurs either in the part itself or in any subpart. This may be the case if you want to provide a sort option that would display a sorted list of items without sorting the collection of items.

2. **Attribute-to-script connection**: These connections are used when you need some processing related to an attribute. With a slightly different example from the one described above, you may want a view to always display a sorted list of some items without sorting the collection of items.

### 3.2.2 Part Interface Editor

Whenever a new primitive part is added, its interface must be defined and specified. Furthermore, in some cases the interface of a part needs to be updated.

The Public Interface Editor lets you create and modify the interface (attributes, events, and actions) of a part. The Public Interface Editor is commonly used for nonvisual parts, because the visual tools automatically define the interface for visual parts when you build them with the Composition Editor. There are three different cases in which the interface editor will likely be used:

1. Some features must be added to the interface of an existing part. For example, you want to add the attribute “country” to an Address nonvisual part.

2. The interface must be defined for an existing class. For example, in your environment you may have an existing Customer business object that
does not have a part interface defined and, as such cannot be used for connection in building the application.

3. The need for a new part arises during your analysis or design. The new part must be defined and its interface must be specified.

The Public Interface Editor uses a CUA notebook with a page for each type of interface feature, that is attributes, events and actions.

• Attributes

The attributes page lists the names of all the attributes defined in the part. For each attribute, the following may be listed:

– The operations that can set and get the attribute; the names of the operations are get and set selector respectively.

– The symbol that is used to signal other connected parts that the attribute has changed.

Usually, the change event symbol as well as the get and set selectors are defined and implemented by the fabricators of the part.

• Events

This page lists the names of the events defined in the part and the corresponding event symbols that are used within the code to signal connected parts that the event has happened.

For visual parts, all the events related to user interactions are provided by the application builder and are not listed here. Only special custom events are listed.

• Actions

The actions page lists the names of all the actions and, for each action, the name of the operation (called a selector) associated with it.

Whenever a new feature is added, the Public Interface Editor provides defaults for selectors and event symbols. Alternatively, it may also list the available selectors within a part to choose from. The editor can generate standard code for selectors, providing a useful feature when set and get selectors are needed. If some customized processing is necessary, it can be added later, by using the Script Editor.
3.2.3 Script Editor

The term script is becoming quite common among visual programming tools. It refers to a description, done by using a simple language, of how elements within an application have to interact in those cases in which the visual language cannot be used effectively. For example, scripts allow you to specify application logic when you do not have a suitable visual part to use, or when the logic you are creating is easier to express programmatically than visually. The term script suggests an analogy between an application and a theatrical performance, in which scripts describe the desired behavior of actors, by specifying how they should interact.

VisualAge provides Smalltalk as the language that can be used to write small scripts, called methods. Smalltalk is a non-proprietary, pure object-oriented language, that offers a rather simple syntax. It is important to emphasize that you are not required to be skilled in object-oriented programming in order to write these scripts. The object-oriented skill becomes necessary only when designing and fabricating classes and parts.

The Script Editor browses the underlying class of a part and lets you act on variables, list the selectors and scripts, and edit their contents. The various elements related to the part are also listed, such as attributes and actions of subparts. When you have to enter or modify a script, the editor helps you enter the most common language constructs and the code to access the elements mentioned above.
3.3 Library of Parts

IBM* provides an extensive set of reusable parts with VisualAge including user interface parts for constructing CUA '91 user interfaces, and nonvisual parts that allow you to access databases, remote programs and DLL entry-points. Parts delivered with VisualAge are shown in Figure 15.

Other parts may be created by composing the parts provided and sometimes customizing them using the visual tools, or by programming in the language of your choice.

3.3.1 User Interface

The CUA '91 controls are included, for example: window, check box, list box, combination box (combo box), push button, etc. In addition, VisualAge provides enhancements to the user interface with support for smart entry fields, table control and form processing.
3.3.1.1 Smart Entry Field
An entry field provides support for displaying and updating a string. Smart entry fields provide flexible support for displaying and updating objects other than strings in entry fields. Number, integer, floating point number, date and monetary amount are all examples of supported objects. Smart entry field support provides:

- Conversion of objects to displayable format and vice versa.
- Input validation including user definable input checking, such as minimum and maximum value.
- Entry editing, with support for user definable options, conversions, monetary symbols, etc.

3.3.1.2 Table Control
This provides support for the display and update of objects in a tabular form. It supports entry and protected fields, tabbing and scrolling, in place addition, deletion, and modification of rows.

3.3.1.3 Form Processing
This provides support for nested forms, in which the view is composed of multiple views. One of the requirements is, for instance, that the tabbing works correctly across the nested views. Form support provides correct tabbing between fields even when the fields are nested inside different views and the current position is modified with the mouse.

3.3.2 Nonvisual Parts
The application builder parts palette contains commonly used nonvisual parts, such as collections, dictionaries, files, etc.

There are also generic nonvisual parts available that greatly simplify the task of accessing local and remote external functions, local and remote databases, and so on. Among these generic parts are:

- Generic C and COBOL DLLs
- Generic transaction
- Generic query

From the generic parts, you can create specific parts unique to your environment. You can customize generic parts to allow access to DLLs, databases and remote transaction programs available in your environment, by setting attributes without writing any application-specific code.
The generic parts are based on the supporting components that are included in VisualAge. An overview of these components is given in VisualAge Supporting Components: Overview. However, you are not required to know about VisualAge's supporting components in order to use the generic parts.

3.4 Support for Applications

VisualAge provides a framework for application development that enables advanced structures for interactive applications to be implemented, such as model/view separation and deferred updates.

3.4.1 Application Model

In the past, applications have been designed and implemented with a centralized and monolithic structure. In these applications, the flow of control is in the application itself, which determines and governs communication, synchronization and user interactions.

There are new models for application design that, on one hand facilitate the division of the application into elements that are processed in different systems. On the other hand, they can provide more flexible user interfaces and reflect the trend towards applications in which the user is in control of the flow of operations. The top part of Figure 16 on page 38 shows the elements of an application: user interface, business logic, and data access. Let us focus on the user interface and business logic and make some comments about them.

The business logic for an application defines the rules and relationships governing business entities (for example, customers, invoices, and line-items, in an order-entry application) without regard for how they are to be presented or manipulated by the user. These entities are known as models, because they typically model real-world objects in the specific target domain. The goal of the business logic is to implement the objects that comprise the user's conceptual model of the underlying system.

The user interface defines how these entities are presented to the user (views) using windows, dialog boxes, menus, etc., and manipulated by typing, clicking with the mouse, etc. The overall purpose of the user interface is to present the business logic model to the end user and to interpret the user's inputs to notify this model. It does not implement the behaviors of the business entities being displayed.
Developers find that separating the business logic from the user interface has several important advantages, some of which are illustrated in Figure 16 on page 38:

- **Parallelism in development**
  Prototyping and evolution of the user interface can occur without requiring changes to the programming of the underlying business objects and can occur independently of the development or the business logic.

- **Multiple views**
  Multiple user interfaces can be presented concurrently for the same business objects.

- **Specialization of skills**
  User interface specialists can concentrate on user interfaces while analysts and developers concentrate on business logic.

- **Integration of programming environment**
  Business logic developed in different technologies can be integrated with event-driven, object-oriented interfaces.
• Business logic split

Business logic implemented on a server or mainframe system can be integrated with a user interface and business logic running on the workstation.

3.4.2 Model and View Separation

When you design and write an application it is important to keep views separated from models. Views are related to the display of information and user entry and actions, while models are related to the business logic. Views are more likely to change than the models are. If you maintain a clear distinction between them, you will be able to provide an application that can be easily reused and extended with a minimum impact on the models whenever views need to be changed or new views are required.

VisualAge helps you construct your parts with a clear separation between views and models. However, it is important to keep this separation when making customizations by writing scripts.

3.4.3 Deferred Updates

Updates change the state of parts and objects and are logically dissimilar from queries of their state. Furthermore, an update may affect multiple attributes within a part and, potentially, multiple parts. The view that supports the user input for an update may be composed of multiple fields and nested visuals. We do not want to change the state of the object(s) until the user has finished entering all the information, considers all the entries valid, and confirms the overall update. Also, we need to keep track of the potential changes, do input validation on each field, and help the user with lists of choices and conflict resolution, whenever it is appropriate.

As an example, consider the multiple fields that can be entered when a customer has to be updated: name, address (with street, city, state and zip code), telephone number, credit limit, special category, customer preferences, and so on. We do not want to affect or change the state of the customer object immediately, that is, when the user enters a new value for a field. We want to let the user enter the various fields and help him, whenever it is appropriate, by validating the input and listing choices. When the user is satisfied with the entry and confirms it, we want to inform the model object of the updates and have them applied.

The visual tools enable the implementation of deferred updates while building your parts. Deferred update operations act as intermediate agents between the view and the model object(s) to handle all the complexity.
related to the logic necessary to support the user input. When the user confirms the update, the operation notifies the model object(s) about the new values.

3.4.3.1 Undo and Redo
Deferred update operations include the behavior necessary to support “infinite” undo and redo. With minimum additional coding, your applications will be able to provide end users with this high-level mechanism.
Chapter 4. A Sample Application

This chapter describes an address book sample application that will be used as an example in the following chapters to illustrate components of the visual tools.

Building the overall example will take you through several of the key components of the visual tools, even though it does not use all of the functions of the visual tools. However, our purpose is not to provide a tutorial on how to use the visual tools. We want, rather, to have an example we can use to illustrate VisualAge’s concepts and features. For a complete description of the features of VisualAge and information on how to use it, please refer to the VisualAge manuals.

Though we will proceed in a step-by-step way, we would like to emphasize that there is no single step-by-step method to creating applications. Whenever you build an application using the visual tools, you can select from where to start and how to proceed. This is in line with the ability of object-oriented programming in allowing iterative development.

Note: When we implemented this sample application, we prefixed all the parts’ names with “AA” that we omit here for easy reading. We will use the full names when describing sample code.

4.1 Description

Our address book is a simple application similar to a standard store-bought address book that contains the names and addresses of several people. The purpose of this application is to display a person’s name and address and to be able to page forward and backward through the address book.

One way to present people’s addresses would be to create a table or list. However, realizing that the number of people in the address book could get relatively large and the amount of information for each entry (name and all address attributes) could be considerable, we prefer a different presentation that will allow us to scroll forward and backward within the address book. To achieve this we decide on a view that presents one person to the user and provides buttons for paging forward and backward within the address book.

Realizing that most people have two addresses, a home and work address, the address book application will present information with a view similar to the one illustrated in Figure 17 on page 42.
4.2 Design

Now that we have a description of the application and its purpose, we can start to consider the objects necessary to build it. In general the objects that model the problem space map to the models used for the problem description and closely relate to the user’s environment.

Visual notes can be sketched on paper or on a white board and may help in finding objects that are necessary, their characteristics and their relationships. These visual notes we are proposing are an informal technique that can be useful in supporting the early stages of thinking and design processes. They are quite similar to the sketches that architects draw when they start working on a new project. Figure 18 on page 43 and Figure 19 on page 44 show examples of visual notes for the address book application.

1. The first model that comes to mind is the address book (AddressBook); however, the address book is simply a collection of people’s addresses that must be able to position to a selected page and return the current person on that page, while the contents themselves do not naturally belong to this model.

2. Thus a person model (Person) is a second natural model object for our application.
3. Each person has two addresses, and different people may have the same working address. For this reason we introduce a third model object: the address (Address).

4. The address itself is shown twice within the address book view. Therefore we create a view for the attributes of an address (AddressView). After creating the address view, we can embed it within the address book view twice to represent the home and work address. (See Figure 19 on page 44.)
To conclude, we need the following nonvisual parts:

- AddressBook
- Person
- Address

Now that we have an idea of what we want our application to look like, we have to consider more carefully the various parts that make up our models.

### 4.2.1 Nonvisual Parts

This section contains a description of the nonvisual parts used by the application and their interface.

#### 4.2.1.1 Address

The address model (Address) is a separate object since it is reused twice inside the person model. The person model holds both a home address and a work address.

The address model is also reusable in another sense. Since several people often share the same address (for example, work in the same building), they can, in our application, share the same address model object. Furthermore,
it is even likely that the address model can be reused in other applications that are concerned with addresses.

**Attributes**

The following properties must be accessible by other parts, so they are defined as attributes of the part Address:

- **street** String
- **city** String
- **state** String
- **zip code** Number

### 4.2.1.2 Person

The person model (Person) is a separate object since it is instantiated multiple times inside the address book. Each entry in the address book is of type Person.

**Attributes**

The following properties must be accessible by other parts, so they are defined as attributes of the part Person:

- **name** String
- **home address** Address
- **home phone** String
- **work address** Address
- **work phone** String

### 4.2.1.3 Address Book

The address book model is a collection of people’s addresses that allows writing and reading addresses on pages of the book, with paging capabilities through the book to retrieve an address.

**Attributes**

In our application we are only concerned with showing one person at a time; thus the AddressBook must be able to return and set the current person. Therefore the property that is defined as an attribute is:

- **current person** Person
The address book model must also support the function of paging forward and backward through its collection of persons. Therefore, among others, the AddressBook has the following actions defined:

get next person scrolls forward
get previous person scrolls backward

4.2.1.4 Summary: Attributes and Actions
The attributes of the nonvisual parts (Address, Person and AddressBook) and the objects that compose these nonvisual parts are listed in Table 1.

<table>
<thead>
<tr>
<th>Part Name</th>
<th>Attribute Name</th>
<th>Class Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address</td>
<td>street</td>
<td>String</td>
</tr>
<tr>
<td></td>
<td>city</td>
<td>String</td>
</tr>
<tr>
<td></td>
<td>state</td>
<td>String</td>
</tr>
<tr>
<td></td>
<td>zip code</td>
<td>Number</td>
</tr>
<tr>
<td>Person</td>
<td>name</td>
<td>String</td>
</tr>
<tr>
<td></td>
<td>home telephone number</td>
<td>String</td>
</tr>
<tr>
<td></td>
<td>home address</td>
<td>Address</td>
</tr>
<tr>
<td></td>
<td>work phone number</td>
<td>String</td>
</tr>
<tr>
<td></td>
<td>work address</td>
<td>Address</td>
</tr>
<tr>
<td>AddressBook</td>
<td>current person</td>
<td>Person</td>
</tr>
</tbody>
</table>

The following actions are defined in the AddressBook part:

<table>
<thead>
<tr>
<th>Action Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>go to first entry</td>
<td>Position to the first entry in the address book.</td>
</tr>
<tr>
<td>go to last entry</td>
<td>Position to the last entry in the address book.</td>
</tr>
<tr>
<td>get next person</td>
<td>Position to the next person in the address book.</td>
</tr>
<tr>
<td>get previous person</td>
<td>Position to the previous person in the address book.</td>
</tr>
</tbody>
</table>
4.2.2 Visual Parts

This section contains a brief description of the visual parts needed for the application.

4.2.2.1 AddressView

As said earlier, the visual part AddressView is created as a separate object since it is reused twice inside the address book view.

The address view is also created as a separate object with the expectation that it could be reused in other applications that need to display an address.

Attributes

The address view will display the contents of addresses that are supplied at run time. Therefore, the address view needs a variable \texttt{(address)} that will hold instances at run time (see Figure 19 on page 44). The variable \texttt{address} is also added to the interface as an attribute that can be accessed when AddressView is reused (see Figure 18 on page 43).

4.2.2.2 AddressBookView

The visual part AddressBookView displays the information concerning one person at a time.

The view also includes push buttons to page forward and backward through the entries in the address book.

Attributes

The address book view will display the contents of an address book part. We could add an instance of the address book part, but for reasons of reuse, we decide that the specific instance of the address book will be provided at run time and add the variable \texttt{addressBook} instead (see Figure 18 on page 43). This variable is also added to the interface as an attribute. Information related to each person will be provided by the attribute \texttt{current person}.

Note: For an explanation about variables, instances and attributes see 5.3.2, “Nonvisual Parts: Variables and Instances” on page 61 and 5.3.3, “Attributes” on page 62.
### 4.3 Building the Application

In building the application we assume that all the nonvisual parts (Address, Person and AddressBook) exist in the environment and give some guidance on how to build the visual parts.

If the nonvisual parts do not exist you need to build them. For a description, see:

- 6.4.1, "Creating a Nonvisual Primitive Part" on page 79 for Address and Person.
- 6.4.2, "Defining the Interface for AddressBook" on page 79 for AddressBook.

**Note:** The description of the examples is based on the VisualAge Team. You may encounter minor differences if you are using the VisualAge entry version.

Figure 20 shows the Composition Editor and illustrates some of its elements as well as terms we will be using.

![Composition Editor: AddressView Create Visual Part](image)

*Figure 20. Composition Editor: AddressView Create Visual Part*
Here are a few tips to help you create the view and arrange subparts on the surface:

- You can place parts from the palette onto the editor surface by selecting the appropriate category, selecting the desired part from the parts palette, moving the pointer to the surface and clicking mouse button 1.
- You can select a subpart by clicking mouse button 1 when the pointer is on the desired subpart.
- You can select multiple subparts by moving the pointer on subparts and clicking Ctrl + mouse button 1.
- You can move selected subpart(s) by dragging it (them) with mouse button 2.
- You can resize any subpart by selecting it with mouse button 1 and then using mouse button 1 to drag the corner of the part.
- You can change the text portion of a subpart (for example, label) by using Alt + mouse button 1, typing the new text, and either clicking elsewhere with the mouse or by pressing Shift + Enter.
- A context menu for an object can be displayed by clicking mouse button 2 with the pointer over the object.
- Composition Editor provides many functions to help you arrange your visuals. You may explore them using either the icons in the tool bar, or various choices in the Tools pull-down menu.

**Making Connections:** To make a connection, do the following:

- Place the pointer over the source component.
- Click Alt + mouse button 2 to display the list of features.
- Select the feature you want to connect with mouse button 1.
- Drag the mouse over the target component.
- Click mouse button 1 to select the desired feature from the list that is displayed.

**Note:** What is described above applies to the OS/2* platform and equivalent functions are available in all other platforms. However, the usage of the mouse may differ among different platforms with respect to standard usage of the mouse in each platform.
4.3.1 Creating and Opening an Application

The first step is to create the application AddressBookAppl in which we can create our parts. This is achieved by:

- Choosing Launch from the Visual Tools pull-down menu in the System Transcript window to open the AbtApplication Manager window.
- Choosing Applications-Create from the Applications pull-down menu.
- Entering the name AAAddressBookAppl when prompted.

Once the above steps are completed, you can click mouse button 1 twice over the name of the application just created to browse the application itself.

4.3.2 AddressView Part

Let us start with the building of the AddressView that will be reused in the AddressBookView.

4.3.2.1 Creating AddressView and Adding Subparts

To create AddressView, select Create Visual Part from the Applications pull-down menu in the Application Browser just opened, and enter the name AAAddressView in the New Part Request window.

The application builder creates a class named AddressView and opens on the Composition Editor that contains a shell default window (see Figure 20 on page 48). The default window has a frame and it is not suitable for views that will be embedded in other views as in our case. Therefore we need to replace it with a window without a frame.

**Using a Form:** In order to replace the default window with a form:

1. Position the pointer over the default window.
2. Press mouse button 2 to invoke a context menu for the default window.
3. Select Delete from the context menu.
4. Use mouse button 1 to select a form from the category Canvas in the palette.
5. Move the pointer to the free-form surface and click mouse button 1 to place the window.

**Adding an Attribute for Address:** Before constructing the visual aspects of the AddressView, you can now (though you could also do this later) add the variable for the nonvisual subpart Address that this part needs.

One way to add a variable is by selecting Add variable from the Options pull-down menu. A prompt appears asking for the name of the variable and its class. In this example, we called the variable address. The class that serves
as the address model is **AAAddress**, so enter that as the class name of the variable. This results in an icon, named **address**, being added to the surface.

The variable **address** can now be added to the interface by moving the pointer on it, clicking **mouse button 2**, and selecting **Add to interface** in the context menu.

**Building the View:** You can begin building the address view by placing the necessary text-entry fields and labels selected from the parts palette, and arranging them in the layout proposed in Figure 19 on page 44.

**Changing the Settings of an Entry Field:** Because the **zip code** attribute is a type of number it has to be converted when it passes from the text-entry field to the Address part and vice versa. To achieve this, do the following:

1. Click **mouse button 1** twice with the pointer on the text-entry field for **zip code** to open its settings.
2. Set the **Field type** to **Number** in the Data Type page of the Settings notebook. This will match the type that is specified for the **zip code** attribute in the **Address** part.

**4.3.2.2 Making Connections**

Although all the necessary subparts have been added in the AddressView part, you still need to indicate how the model relates to the text-entry fields. Specifically, you need to determine which text-entry field will be used to display which attribute in the model. This linkage is established by making connections, namely attribute-to-attribute.

To illustrate this, let’s start with the **street** attribute:

1. Place the pointer over the **address** variable and press **Alt + mouse button 2** to bring up a connection context menu.
2. Select **street** from the context menu, by clicking **mouse button 1**.
3. Move your pointer to the text-entry field for street and press **mouse button 1** to bring up a connection context menu.
4. Select **object** from the menu to complete the attribute-to-attribute connection.

To complete the address view, repeat the steps above to make attribute-to-attribute connections for all the other attributes in **address**.
4.3.2.3 Saving the AddressView Part and Adding It to the Palette
You have now completed the AddressView, so you need to save it by choosing Save from the File pull-down menu.

If you want to be able to reuse the AddressView later on by selecting it from the palette instead of entering its name, simply promote it by:

• Choosing Promote part from the File pull-down menu.
• Specifying in which category you want it to be added.

4.3.2.4 A Shortcut: Quick Form
Building the AddressView as described in the previous sections is useful to become familiar with functions of the application builder. However, there is an alternative and quicker way to build the AddressView part. You start with a form and the address attribute. Then you select Quick form for the attribute self of the address and place within the form the view that application builder prepares for you. You organize the visual subparts and change the labels to match your view.

In this way, the application builder, not only returns the default views, but it also sets the necessary connections and takes care of the type of fields as necessary.

4.3.3 AddressBookView Part
The final visual part to be created is the AddressBookView.

4.3.3.1 Creating AddressBookView and Adding Subparts
After building the AddressView part, the task of composing the view for AddressBookView is straightforward. Figure 18 on page 43 should be used as reference. Let us briefly list the steps:

• Create the new view class.
• Enter Address Book in the title bar text on the default window.
• Place into the default window labels and text-entry fields for name, work and home phone. Place also two AddressView parts.
• Place two push buttons and label them Next and Previous.

Adding an Attribute for AddressBook: Since you want the view to display and flip through the contents of the address book, you need an AddressBook nonvisual. Based on the consideration made earlier, you should add a variable named addressBook, specifying AAAddressBook as type (class). The AddressBook part is able to scroll backward and forward and can return the
current person. However, in order to access the person’s information, we need another variable that will hold instances of Person at run time. One way to add this variable is by tearing it off from addressBook. To achieve this:

- Activate the context menu on addressBook.
- Select Tear off attribute.
- Select the attribute current person from the list.
- Place the variable on the free-form surface.

The application builder builds and maintains the proper connections between the variable and the attribute within addressBook.

### 4.3.3.2 Making Connections

You can now make all the connections that are needed:

- **Name**, home phone number, work phone number, to the attribute object of the respective text-entry fields
- home address of current person to address of the first AddressView subpart
- work address of current person to address of the second AddressView subpart
- clicked of the Next push button to get next person of addressBook
- clicked of the Previous push button to get previous person of addressBook

When you have finished making the connections, you can save the part and close.

### 4.3.4 Testing

To test the view of your application, click on the Test icon in the tool bar. However, this will not test the functioning of the part because the part does not know how to instantiate the attribute addressBook. The attribute addressBook is a variable that is instantiated at run time.

In order to completely test the part you may “simulate” the run time by building a part that makes use of AddressBookView and specifies the instance of addressBook to be used. To create it, proceed as follows:

1. Create a new view called TestAddressBookView.
2. Delete the default window.
3. Add the part AddressBookView by selecting the option Add part from the Options pull-down menu.
4. Similarly, add the part **AddressBook** by selecting the option *Add part* from the Options pull-down menu.

5. Connect the attribute **addressBook** of the part **AddressBookView** to the attribute **self** of the part **AddressBook**.

6. Click on the test icon in the tool bar and test the functioning of the part.
Chapter 5. Visual Tools: Assembling

As described in 2.7, “Roles in Application Development” on page 21 there are three different uses of VisualAge. This chapter addresses some aspects of the upper level of use, assembly. It is not our intention to describe the details of how to use the visual tools. We want, instead, to give you some insight that may help in the part assembly phases.

5.1 Terminology and Concepts

Before proceeding, it is worthwhile to introduce and clarify some terms and concepts that are used in the following sections. Because VisualAge provides object-oriented visual programming tools, some of the terms derive from object-oriented environments.

5.1.1 Objects, Classes, and Instances

- As said earlier (see 2.6, “Object-Oriented Technology” on page 20), an object is a self-contained software entity that comprises attributes and behaviors (a set of operations). Attributes and behaviors allow the object to accomplish its responsibilities. Contents and values of attributes change during the life of an object.

  Objects are highly modular, because the functioning of an object does not depend on internal details of other objects.

- Class refers to the description of objects that are alike and have common characteristics. A class provides the abstract definition of the static and dynamic structure of objects, through the specification of their attributes and behaviors.

  Classes are used as templates to create objects. They are organized in a hierarchy tree, that reflects how classes inherit characteristics. This means that the attributes and behaviors defined and implemented in a class are automatically available, without additional coding, to the classes that are below it in the tree. If we point to a class in the hierarchy, any class above it in the hierarchy is called superclass, while any class below it in the hierarchy is called subclass. Whenever a new class is defined, it is created as a subclass of a selected class.

- Instance refers to one of the objects described by a class. It is a concrete realization of a class template, with contents assigned to its attributes.
Generally, “class” is used whenever the focus is on the abstract structure of objects. “Instance” is used when it is necessary to emphasize the particular occurrence. “Object” is used to mean either a class or an instance. We will adhere to this common usage of terms, using object for both classes and instances, with further specifications whenever confusion may arise.

Objects are the underlying software representation of parts. The use of the terms as described above maps to the common meaning of the word part: we may use the term part to refer to its external specifications as described in a catalog. We also use the same term to refer to either its more concrete implementation, or to that specific component that the dealer is providing to us.

5.1.2 Composite Parts

Parts may be composed of other parts. We have mentioned this aspect when we introduced nested visuals (see “Nested Visuals” on page 30). Figure 21 on page 57 shows an example: the AddressView part is “made of” multiple text-entry fields and labels.

Nesting is not the only possible way to compose parts. Composite parts can also be made of components that are not nested. For example, consider a part that lists the names of people contained in an address book, and, when a name in the list is selected, the part is also able to show the page of the address book with detailed information on the person. Clearly, this part is composed of:

- Two separated visual parts:
  1. A list box to show the list of names.
  2. An AddressBookView to show the page of the address book.
- One nonvisual part, AddressBook.

When we want to emphasize the aggregation aspects of a part, we refer to it as **composite part**. The parts within a composite part are called **subparts** (see Figure 21 on page 57). Subparts are parts on their own, and we use the term subpart only to distinguish between a part and its components.

Any kind of part, not just visuals, can be aggregated or made of other objects or parts. For instance, the nonvisual part Address is made of different objects: three objects of class String to hold information for city, street and state, and one object of class Number for zip code (see Table 1 on page 46). Similarly, a Person part is made of objects of class String for name and tele-
phone numbers and two Address. These elements are externalized as its attributes.

The aggregation relationship allows an object to send messages to its component objects whenever this becomes necessary to accomplish what the object is requested to do. The object may require information from its components, or it may ask them to perform some tasks. This way of relating objects makes changes easier to implement. For instance, Address may be changed to add country information without affecting parts that reuse it.

The application builder extends the concept of composition to include how subparts within a composite part interact, and provides a means to specify and build those interactions. The relationships of part-to-subparts is thus extended to include also the relationship of subparts-to-subparts within a part.
5.2 The Process of Assembly

It should be clear by now that the process of implementing an application scenario consists of selecting existing parts, most likely (but not necessarily) from the palette, and establishing connections among them.

Often, the assembly process is done in stages in which intermediate composite parts are edited and constructed.

The Composition Editor allows you to:

- Visually build the static aggregation of a part.
- Visually define and specify the dynamics between subparts by making connections.

To perform these tasks, you need to be familiar with the parts that are available and their interfaces, as well as with the Composition Editor.

There is no rigid sequence you have to follow or respect; rather, the aggregation and the making of connections usually will be performed iteratively. And they will be further intermixed with testing as well. However, for clarity in explanation, we describe the two activities as being distinct. Figure 22 on page 59 shows the iterative process of constructing intermediate parts necessary for an application.
For each intermediate part the iterations are among the tasks described below:

- Search for existing reusable part.
- Add the part to the Composition Editor free-form surface.
- Customize the part by setting defaults and default values.
- Make connections.
- Test.

If the part does not exist, it will have to be defined and/or fabricated (see Chapter 6, “Visual Tools: Using the Part Interface Editor” on page 75).

5.3 Building Composite Parts

In this section we assume that all the parts necessary to build a composite part are available or can be easily defined by customizing generic parts. This means that new parts that are edited are parts that can be composed by reusing existing visual and nonvisual parts.
The first step in assembling a composite part is to edit it. If the part already exists and you need to update it, you select the option *Browse Class* in the *Applications* pull-down menu. Unless otherwise specified:

- **New visual parts** are created as a subclass of *AbtAppBlldrView*. A visual part should always be a subclass (either direct or indirect) of the class *AbtAppBlldrView*; thus it can inherit the visual characteristics and the editing protocol.

- **New nonvisual parts** are created as a subclass of *AbtAppBlldrPart*. A nonvisual part that is to be composed with the application builder, should always be a subclass (direct or indirect) of the class *AbtAppBlldrPart*; thus it can inherit the editing protocol.

Whenever you see the need for a new composite part, you decide how the view of this part should look (if it is a visual part), which are its responsibilities, and what should be the name of the part. You should give parts meaningful names that correspond to the semantic of the real-world object you are representing. Examples are, *AddressView, Address, AddressBookView, AddressBook, CustomerView, OrderView, Order, etc.*

When you create a new visual part, VisualAge opens on the Composition Editor; you can start building the part view by selecting the necessary elements from the parts palette and placing them onto the free-form surface. Depending on your design the visuals will be either nested or non-nested, or a combination of both. The visual defined as **primary part** is the visual that will be presented to the end user at run time.

In general, the visuals themselves are not enough to provide the required services. A visual part usually needs some nonvisual subparts, such as subparts that can provide business logic and business-related information. The part may also need other parts that access a database, remote or local programs, etc. Visuals and nonvisuals will be connected to specify how they cooperate to provide the part behavior. For example, when you build the *AddressView* part shown in Figure 21 on page 57, the visual defined there can display and accept the fields of an address, but it needs a model object, that can hold and return the values of the text-entry fields. You use the part *Address* as model object and name it *address*.

Most likely a resulting visual part will be composed of multiple nested and non-nested visuals, and multiple nonvisual subparts. Do not lose focus. The parts that you see on the surface are the subparts that compose your part, while the part you are building is the one being edited (see Figure 21 on page 57). The part you are editing is the *AddressView* part, that is composed of:
• Visual subparts to construct the view that the AddressView will present when reused.
• The nonvisual subpart address.

Even though you may include multiple subparts, your part will be a single unit in the system and from the perspective of parts that will be using it. As an analogy, let us consider a tool that can make transparent the external wall of a house and show you what is inside. When you look from the outside, the house is still a single unit. The Composition Editor goes further: it not only shows the component subparts, but it also lets you add, modify, and delete them.

5.3.1 Setting Characteristics of Visuals

When you place visuals into the free-form surface, you can set some of the characteristics by selecting the Open settings option in the context menu. Aside from common properties, such as position on the screen, size, style, etc., the properties you can set depend on the kind of visual you are using. For visuals that support display and entry of fields (text-entry field, list box, combo box, etc.), an interesting attribute that can be set is the type of object that must be supported. You may also customize conversion and editing options for the object. Once the object is set, the application builder automatically converts and edits the object from the type of object declared and the displayable format supported by the text-entry field. This avoids definitions of intermediate connections that are only needed to perform the conversion, but are not related to the logic of your application.

5.3.2 Nonvisual Parts: Variables and Instances

There are two kinds of nonvisual parts you can add when you construct a part:

1. Variables. Variables identify parts that will hold instances of objects at run time. Examples are Address, represented by the variable identifier address, and AddressBook, represented by the variable identifier addressBook.

2. Instances. Examples are parts obtained by customizing a generic part, such as a generic DLL of a generic query and nonvisual parts you select from the parts palette or add using the option Add part.

Once a variable has been added to the part, it represents the run-time object. Thus, variables and instances are treated in the same way from the perspective of making connections. However, there are some differences
when you add the part by placing it onto the free-form surface, when you test the part, and when a part has to be reused.

5.3.3 Attributes

While all three elements of the public interface of parts (attributes, actions, events) play an important role when you establish connections among subparts, attributes are also important during the aggregation phase. Understanding when to specify a nonvisual subpart as an attribute is important in building composite parts, because attributes are necessary to make connections.

Attributes are the logical properties of parts; these properties are known to other parts and can be returned or set upon request. Attributes correspond to objects of some class whose content may change during the life of a part, due to external interactions. Attributes are a general concept and they are defined in both nonvisual and visual parts.

Let us consider some examples.

**Nonvisual:** Address is a rather simple nonvisual part. The Address part has various attributes defined, such as street, city, state and zip code (see Table 1 on page 46). Being attributes, they can be accessed by other parts. For example, a part that needs to display the contents of the attributes of Address will be able to request them, as well as be able to set new values based on the user input. In this way, for instance, the AddressView visual subpart shown in Figure 21 on page 57 can access the attributes of the Address.

**Visual:** Consider the attribute of a text-entry field. You see attributes that are related to its visual properties, such as size, position in the window, etc. that can be customized by using the Open settings option. You also see object, that is, the kind of object that must be displayed. In fact, this is its most important attribute when you make connections.

5.3.3.1 Attribute Definition while Composing Parts

Subparts in a composite part may represent logical properties of the part that need to be accessed from the outside.

For instance, the AddressView part shown in Figure 21 on page 57 is composed of:

- The AddressView visual
- The nonvisual **address** that represents an Address business object.
address is an attribute of the AddressView part that is analogous to the
object attribute in the text-entry field.

To understand why address is defined as an attribute, let us analyze which
properties of the AddressView we need to access whenever we want to
reuse the AddressView part to build a new part. It is clear that we need the
AddressView visual, in order to display and accept the address information.
Furthermore, we also need to access the attribute address, in order to
connect it to the subpart that holds the address information. In the example
shown in Figure 23 on page 64 the attribute address of the AddressView
subpart may be connected to the attribute address of the current person (of
class Person). Thus the two parts AddressView and Person will be able to
point to the same part Address (see also Figure 18 on page 43).

The application builder provides you with a means to declare subparts as
attributes when you assemble a part, as described in 5.3.4, “Criteria for
Defining Attributes.” Only application-specific attributes must be declared in
a visual part, because the application builder provides the attributes of
visuals that are related to their visual properties, such as size, style, etc.

5.3.4 Criteria for Defining Attributes

When you add a variable subpart to your part, you may declare it as being
an attribute of the part or not. If you specify it as attribute, its name is added
to the public interface of the part you are building. Figure 23 on page 64
shows the AddressBookView, that contains two nonvisuals labelled
addressBook and current person of addressBook. The former is an attribute,
while the latter is not. Why the difference?

Let us use the house analogy once more and focus on some of its compo-
nents. People need to know that a telephone is in the house to request its
number in order to reach the inhabitants. We would declare the telephone
as being an attribute of the house. On the other hand, we would not declare
the air conditioner or the heater as attributes, because no one from the
outside needs to know about them.

Attributes are those subparts (the telephone) that must be known to the
outside. They can be seen as public variables. There are subparts instead,
(the heater), that are used only internally. They are necessary to the part to
perform its services, as the attributes are, but their existence is not made
public. They can be seen as private variables.

Each time you declare a nonvisual subpart to be an attribute, it becomes an
element of the public interface of the part you are building (as you may see
by editing its interface by clicking on the edit interface icon). Consequently, when the part is reused, the attribute will be known to other subparts and connections to it can be established.

Whenever the visual is not declared as an attribute, it can be used for connection within the part, but its existence will not be shown when the part is reused.

5.3.5 Tearing Off

Let us consider the addressBook (of class AddressBook) shown in Figure 23. Its attribute is current person (of class Person) as listed in Table 1 on page 46. When we want to display the page of the address book we need to know which is the current person, but we also need to access Person’s attributes, such as name, telephone number, address, etc.

In these cases, the application builder provides a handy solution that allows you to easily access subparts (and their attributes) that are attributes of
parts. The technique is called **tearing off attributes**. For instance, the attribute current person can be torn off, thus making current person a subpart whose attributes can be directly accessed. The application builder keeps the proper connections between the original part (AddressBook) and the attribute that is torn off. This technique makes the components of a part accessible without additional coding.

### 5.3.5.1 Use of Tearing Off Attributes

The technique of tearing off attributes is extremely useful for accessing attributes of parts that are wrappers of external resources, such as external functions or databases. Examples are the record of a program, the result table of a query, etc.

This technique can also be comfortably used when the attribute that is torn off is a rather stable part that, most likely, will not be subject to changes and the part from which the attribute is torn off will not have to use a different class for this attribute.

However, overuse of tearing off attributes leads to less reusable code and its usage should be properly evaluated, because it breaks the encapsulation of objects and may impose limitations to the freedom of changing the internal implementation of a part.

### 5.4 Connections

Let us continue with the house analogy and imagine that our house tool is also able to make the internal walls and floors transparent. In this way, we can see water pipes, telephone and power lines, TV cables, etc. They can be considered somehow equivalent to the connections in the Composition Editor. Once again, the Composition Editor not only allows you to see them, but lets you add, change, and delete connections.

A part comes to life when we make connections among its subparts. Connections cause subparts to interact and provide the desired results. But, what do we connect? And why? And when?

### 5.4.1 Actions, Events and Scripts

Before proceeding, it is worth giving some more information on actions and events, that together with attributes, are used to make connections.
5.4.1.1 Actions
Actions are the behaviors of the parts, the services or operations it may be requested to perform. Let us make a couple of examples. For instance, an action is go to first entry in the AddressBook; an action could also be an open or a close in a visual part, or undo and redo operations.

5.4.1.2 Events
The part interface includes events that provide a notification mechanism to signal that something has happened to the part, aside from the changing of an attribute. Whenever something happens, the part needs an independent event defined, that it can post. As a consequence, all the parts connected to this event will be notified.

Events are rather common in visual parts. Let us make an example of an event also for nonvisual parts, and assume that we are defining the part Person, to which we assign the attribute day of birth. If we want Person to signal its birthday, we define the event birthday in its interface. On the birthday, the part would signal the event, and all the parts connected to it would be notified.

5.4.1.3 Scripts
Events and attributes of a part or of a composing subpart can be connected to scripts. These connections are used whenever some internal processing must be performed. Examples are:

- Some logic is necessary when building a view of a part, but this logic is not a common responsibility of that part. An example could be a view that would display the sorted list of the names of people in the AddressBook. If the AddressBook does not need to be sorted and no other views need to see the address book sorted, we would rather use a connection to a script within the new visual part we are building than add a method to the nonvisual part.
- Internal logic related to an option.
- Preprocessing of an attribute before displaying it, that goes beyond the conversion and editing provided by the application builder.

5.4.2 Advantages of Attributes in Making Connections
The support for attributes, that VisualAge provides, simplifies the task of making connections among parts. When you make connections you are only concerned with the logic of your application. If attributes were not supported, you would have to handle directly further connections related to the getting and setting of a part’s properties, as well as the events that would cause the get and set requests. Instead, whenever you connect two attributes, the
application builder knows **if and when** it is necessary to request the setting
or the getting of the object and automatically provides the necessary support.

5.4.3 Accessing the Part Interface

There are two cases in which you need to access the features of the public
interface of parts:

1. To make connections.
2. To edit the interface in order to update/add/delete features.

5.4.3.1 Listing Features

To list and access the elements of the interface of a subpart, you move the
pointer to the desired subpart, activate the context menu, and choose the
**Connect** option in the context menu on the selected subpart.

If you would rather list the interface of the part you are building, you must
activate the context menu in an open area of the free-form surface.

5.4.3.2 Editing the Part Interface

To edit the interface of the part you are building, you must click on the inter-
face icon (on the bottom right of the editor window) that activates the Part
Interface Editor (see Figure 20 on page 48).

If you need to apply changes to the interface of a subpart, you must first
choose the **Edit part** option in the context menu, then click on the interface
icon.

5.4.4 Making Connections

To explain which kind of connections can be reasonably made, let us use the
plug and socket metaphor that is shown in Figure 24 on page 68. Plugs and
sockets can be connected; to make a connection, we connect **from** a plug to
a socket. They can be connected only if their interfaces match.

In VisualAge the **from** feature is called source, while the **to** feature is called
target.

Figure 24 on page 68 shows that attributes can be either plugs or sockets,
while actions are sockets and events are plugs. The available connections
are:

- Attribute-to-attribute
- Event-to-action
- Attribute-to-action
Aside from the attribute-to-attribute connection, in all the other connections it is possible to pass parameters when making connections; the selection of the parameter that has to be passed can be done graphically. Whenever a parameter is expected, the connection is incomplete and it is represented by a dotted line. Connecting the connection line to an attribute of a subpart, for instance, will cause that attribute to be passed as parameter.

Connections are used to establish links between parts. Be aware that visual and nonvisual subparts are parts on their own and have the full spectrum of the public interface, made of attributes, actions and events. As shown in Figure 25 on page 69, the resulting spectrum of possible connections is:

- Attribute-to-attribute between attributes of the same part.
- Attribute-to-attribute between two parts.
- Event-to-action between events and actions of the same part.
- Event-to-action between two parts.
- Attribute-to-action between attributes and actions of the same part.
5.4.4.1 Attribute-to-Attribute

An attribute of a nonvisual part, such as the street of the part Address, can be connected to the attribute of a view, such as an entry field in the address book example. This will make the view able to display and set the value of the attribute. If the attribute changes, the view is notified; thus it can request the updated information.

When we connect attributes we establish a strong relationship among them. They are tied together. It is common to connect attributes of nonvisual subparts to attributes of visuals. However, you may also connect attributes of two nonvisuals, or attributes of the same part.

The part that owns the attribute is responsible for returning and setting the content of the attribute itself. The application builder keeps connected parts synchronized and updated on the state of the attributes.
Attribute-to-attribute is the only kind of connection in which the direction in which the connection is made may be important. Whenever two connected attributes both have an initial content, the content of the source attribute prevails over the other.

### 5.4.4.2 Event-to-Action

Events relate to something that happens to the part: the user has clicked a button or a check box, or he has selected an item in a list box. He may have selected an option from a context window. Something went wrong with the remote connection, or today is that person’s birthday. Whenever an event happens, some action must take place. Events are plugs: something must be activated due to the occurrence of an event. Actions are the sockets we most likely need. We make event-to-action connections. Events can occur in any part, and the required action may be in any part.

Examples are many. We connect the clicked event of the Next push button in the AddressBookView to the get next person action in the Address. We may connect a selected item event in a list box to the open action of a window that would display the details data, and so on.

### 5.4.4.3 Event-to-Attribute

An event can also be connected to an attribute, when we want to set an attribute instead of triggering an action. In this case, whenever the event is signaled, the attribute is set to a value that is passed as parameter.

### 5.4.4.4 Attribute-to-Action

An attribute can be connected to an action. Whenever the attribute changes, the action is triggered. This is similar to the event-to-action connection, in which the event is that the attribute has changed.

### 5.4.4.5 Connections to Scripts

There are some cases that cannot be resolved with actions in parts or straight connections to attributes and some behavior is needed that is specific to the part that is being built. In these cases we need to make connections from an attribute or an event of the part itself or of any of its subparts to some code implemented in the part (see Figure 26 on page 71). From the perspective of making connections, all we need is the name of the code (method) implemented within the part. Often, when you connect to scripts you will have to enter the code itself (see Chapter 7, “Visual Tools: Customization” on page 87).
5.5 Some Examples

In this section we describe some examples that show how you can easily implement different scenarios by reusing existing parts. The examples include:

- Building a new view for an existing nonvisual part
- Combining two composite parts in a new part

5.5.1 A New View for the AddressBook

This first example shows how to build a new part that presents a different view of AddressBook to the end user.

Description: We want to display the list of the names of people in the address book as shown in Figure 27 on page 72. This can be achieved by simply constructing a new part.

Parts: To construct this new part we need the following parts:

- A visual part that can list collection of elements and provide scrolling functions, such as the list box visual part provided by VisualAge.
An AddressBook part that contains the collection of people.

**Construction:** The steps to construct the part are:

1. Select the option *Create Visual Part* and, in the New Part Request window, specify *AddressBookListView* as class name.

   The application builder opens on the Composition Editor, with a default window.

2. As we have mentioned, the visual part that is most suitable to list collections of objects is the list box. Select a list box from the parts palette and place it on the free-form surface inside the default window.

3. Add the variable *addressBook* specifying AddressBook as class for the attribute. We want the nonvisual AddressBookView to be an attribute, because we are going to reuse the part we are building in the next example; thus choose the option *Add to interface* from the context menu.

4. Make the following connections:
   
   a. Attribute *people* of *addressBook* to attribute *items* of list box.

   b. Attribute *current person* of *addressBook* to *selectedItem* of list box.

5. We want the list box to list the names of each person in the *addressBook*. For this, select the option *Open settings* from the context menu and enter *name* as Attribute name in the first page of the Settings notebook.

6. Save the part.

7. Promote the part for further reuse and close.
Test: The attribute `addressBook` in the `AddressBookListView` is a variable that is instantiated at runtime. In order to test the part `AddressBookListView`, build a part to test it as described in 4.3.4, “Testing” on page 53.

Note: You may notice that the attribute `people` is not one of the attributes of `AddressBook` listed in Table 1 on page 46. In fact, to complete this example the attribute `people` must be added to the `AddressBook` part and the attribute `current person` must be updated (see 6.4.3, “Adding an Attribute to `AddressBook`” on page 80). Furthermore, some simple customized code must also be added (see 7.3.1, “Adding Code to an Action Selector in `AddressBook`” on page 102).

### 5.5.2 Combining the Original View of `AddressBook` and the List of Names

Besides showing the reuse of existing parts this example gives some insight about the usage of attributes. It is for this example, for instance, that we have defined the `addressBook` as a variable attribute in `AddressBookView` and in `AddressBookListView`, rather than using instances.

**Description:** We want to build a part that can show two windows. The first one lists the names of people in the address book, while the second one presents the address book itself. Whenever a name is selected in the list, the address book must position on the person selected. Furthermore, if the address book is scrolled either using the push buttons or one of the menu options, the current item in the list must be the name of the current person of the address book.

**Parts:** To implement this new part we need the following parts:

- The `AddressBookListView` we have just constructed.
- The `AddressBookView` built in the tutorial application.
- The `AddressBook` that contains the collection of people.

**Construction:** The steps to construct the part are:

1. Select the option *Create Visual Part* and specify `AddressBookCombinedView` as class name in the New Part Request window.

   The application builder opens on the Composition Editor, with a default window in it. In this case we delete the default window, by selecting the *Delete* option from the context menu.
2. The two visuals we need are already available. They are: AddressBookListView and AddressBookView. Select them from the parts palette (or use the Add part option) and place them on the Composition Editor surface. The window that contains the list of names is the window we want to be displayed first when the part is activated at run time. To obtain this, define AddressBookListView as the primary window for this part by selecting the option Become primary view from the context menu.

3. When the window that displays the list of names opens, we want to open AddressBookView. To achieve this, connect:
   - Event openedWidget of AddressBookListView to action openWidget of AddressBookView.

4. The part AddressBookCombinedView also needs the nonvisual AddressBook that represents the part that contains the information about people. Add the subpart AddressBook using the Add part option.

5. AddressBookView and AddressBookListView both present a view of the address book. Furthermore addressBook is an attribute of both. The attribute holds the instance of address book at run time. The list view and the address book view will be able to display information about the same people only if the attribute addressBook of the two parts point to the same instance. This instance is the AddressBook subpart of the part AddressBookCombinedView you are building.

   The requests described above are easily satisfied by making the following connections:
   a. Attribute addressBook of AddressBookListView to attribute self of the nonvisual subpart AddressBook.
   b. Attribute addressBook of AddressBookView to attribute self of the nonvisual subpart AddressBook.

6. Test the view, using the option Test to check that the two windows open as desired. The test will also tell you if there is no primary view defined in the part. Because the subpart AddressBook is an instance, you can also test the functioning of the part.

7. Save the part.
Chapter 6. Visual Tools: Using the Part Interface Editor

In some cases, the need may arise to use the Part Interface Editor to edit the interface of a part either to update or define it. This is a common task for the parts providers, whenever they build primitive parts. However, during the assembly phase you also may need to add or make changes to attributes, actions or events. You may also need to use a part that does not exist yet; thus, you have to create it and define its interface. In these cases, the Part Interface Editor can effectively be used as a tool to define part specifications to be passed to people who will fabricate the part.

The usage of the Part Interface Editor may be more common for nonvisual parts, but it may also be necessary for simple visuals as you will see in 6.4, “Examples” on page 78. To perform these tasks, you need to be familiar with the VisualAge’s Part Interface Editor.

Once the interface is either updated or defined, you can use the new elements to make the connection, even though the code is not available yet. Sometimes, this may help you find elements of the interface that were initially overlooked.

6.1 Opening the Part Interface Editor

Whenever you edit a part VisualAge can recognize if it is a visual or not and opens on a different editor as shown in the following table:

<table>
<thead>
<tr>
<th>Editor</th>
<th>Kind of Part</th>
</tr>
</thead>
<tbody>
<tr>
<td>Composition Editor</td>
<td>Visual and nonvisual parts that are built with the Composition Editor.</td>
</tr>
<tr>
<td>Public Interface Editor</td>
<td>Primitive visual parts, primitive non-visual parts, any part that was not built with Composition Editor, or classes without a part interface defined.</td>
</tr>
</tbody>
</table>

If you need to update the interface of a visual part, you click on the Part Interface Editor icon in the Composition Editor window (see Figure 20 on page 48).
6.2 Terminology and Concepts

At this point, we need to introduce some new terminology and clarify some concepts.

6.2.1 Public Interface of Parts

As described earlier (see 3.2.2, “Part Interface Editor” on page 32), VisualAge introduces the concept of public interface of parts, that consists of features of the part that are publicly exposed and are used to make connections. Attributes are the logical properties of a part, while actions are the operation that the part must be able to perform.

VisualAge also formalizes the concept of event, that is commonly overlooked, by letting you specify events that are associated with something that happens to the part and triggers some action.

We would like to emphasize that parts in the palette already have their part interface defined. It is also worth restating that, when you build visuals, the application builder provides and builds the interface for you that is necessary to perform their responsibilities as visuals, such as display themselves, accept input, scroll etc.

6.2.1.1 Names of Attributes, Actions, and Events

When you need to access elements of the part interface by making connections, you do it by using their names. For example, to connect to an attribute, you make the connection to its name. Internally VisualAge, uses the get and set selectors as appropriate. The selectors and their code (method) must be in the class. This approach allows, on the one hand, the use of names that don’t have to respect a programming language syntax and are more natural to a non-programmer. On the other hand, it allows the customization of the names of the attributes without touching the Smalltalk code. This could be useful, for instance, to translate the names in a different national language, without changing the names of the methods within the class.

The above considerations about the names of attributes apply as well to the names of the other elements of the interface: actions and events.
6.2.2 Implementation Concepts

Selectors are the names of the methods that implement the features defined in the interface. The term selector indicates that a selection is performed from the methods of an object. There are two major kinds of selectors:

1. Set and get selectors, that will set and return attributes.
2. Action selectors, that correspond to some processing.

In general, the definition of an object is made through the formal specification of its interface.

Instead of using directly the names of selectors to specify the public interface of a part, VisualAge distinguishes between the two categories of selectors described above, by introducing attributes, for which get and set selectors can be specified, and actions that have action selectors associated with them.

6.3 Adding Elements to the Interface

Let us start with an example and imagine that you want to extend the AddressView part by adding country information. You can easily edit it and add to it new label and a new text-entry field, and save the part. As soon as this is done, the visual part AddressView is modified, and these changes will also be reflected in all the parts that use AddressView.

However, to store country information you also need a country attribute in the part Address. To do this, you edit the part, that will open on the Part Interface Editor (see Figure 28 on page 80). In the attribute page you add the country attribute and you list the get and set selectors that are available in the Smalltalk implementation of the part. If selectors for setting and getting country information are already available, you select them as appropriate; otherwise, you click on the Defaults push button and, before saving, you request the editor to generate the code for the new selectors. In some cases you may need to edit the selectors with the Script Editor to add some custom code.

Get and set selectors, and the change event symbol are optional. For example, you need the set selector if other parts can request the part to set the attribute; otherwise, it is not required.

You may proceed similarly to add new actions. Whenever you cannot find an appropriate existing selector that implements a new action, you will either have to use the Script Editor to enter the body of the code or request part fabricators to implement it.
Adding events in nonvisual parts is less common. In any case, whenever a new event is defined, customized code that uses the event symbol to signal when the event happens is required.

6.3.1 Defining Primitive Parts

If you need a primitive part that does not exist in your environment, you may need to create it by editing the new part and defining its interface. Most commonly, this occurs for nonvisual parts on which we focus in this section.

Two different situations may occur:

1. Sometimes a class with the necessary characteristics exists already in the environment, but does not have its part interface specified. In this case, you edit the new part and specify the desired class type. You can define its interface by associating attributes and actions with the proper selectors that exist in the class.

2. If a class that meets your needs does not exist, the editor creates a new class in the environment associated with the new part.

In this way you can reuse existing parts and classes also by subclassing. The classes you will be adding are classes related to the problem space, and most likely they will be subclasses of either AbtPart or some existing business model parts. This means that you are not required to know the numerous classes that exist in the environment, but only those related to your business models.

The next step is to define attributes, actions and events, most likely, using defaults for the selector names.

In both cases, if simple code is necessary, you may add it using the Script Editor. For more complex behaviors, the part you have defined and its interface will be used as the specification of the part by the parts fabricators who will implement it.

6.4 Examples

This section contains examples related to updating the interface of a part and the definition of the interface of new parts. The examples include:

- Creating a nonvisual primitive part and defining its interface.
- Defining the part interface to an existing nonvisual class.
- Adding and updating attributes of an existing nonvisual part.
- Adding events to a visual part.
6.4.1 Creating a Nonvisual Primitive Part

This section describes how to define the nonvisual part Address. In a similar way you can also create the nonvisual part Person. The Address part and its interface can be created and defined using the Part Interface Editor. To do so, select Create Nonvisual Class from the Applications pull-down menu in the Application Browser for the AddressBookAppl application, and enter the name AAAddress in the New Part Request window. Because this is not a composite part, click on the Inherits from push button and replace AbtAppBldrPart with AbtPart. The application builder creates the Address class and opens on the Part Interface Editor to let you define the part interface. This part only has attributes that need to be defined. Be sure to be in the Attributes page and then define the attribute street as follows:

- Enter street as attribute name.
- Press the Defaults button.
- Change the Instance variable class to String.
- Press the Add push button.

The above steps should be repeated for all the attributes described in 4.2.1, “Nonvisual Parts” on page 44. At the end, the attribute page should look like the page shown in Figure 28 on page 80. To conclude:

- Choose Generate code-Selectors from the File pull-down menu.
- Press the Generate all push button when prompted for which code you want to generate.

Once you have finished, save the part and close.

6.4.2 Defining the Interface for AddressBook

This example shows how to define the interface for a class that exists in your environment. If you do not have the class AAAddressBook in your system, please refer to Appendix A, “Visual Tool AddressBook Class” on page 111.

In the Application Browser for the AddressBookAppl application, select AAAddressBook in the list and then select Browse Class.

We need to define the attribute current person and the two actions get next person and get previous person. To do so:

- In the Attributes page:
  - Enter current person as Attribute name.
  - Press the Defaults push button.
  - Change the Instance variable class to AAPerson.
  - Press the Add push button.
In the Actions page:

- Enter **get next person** as **Action name**.
- Click on the scroll button to list the **Selector** names.
- Select **getNextPerson** from the drop-down list.
- Press the **Add** push button.

Repeat for **get previous person**.

- Save the part and close.

### 6.4.3 Adding an Attribute to AddressBook

This section describes how to add the new attribute **people** to AddressBook and how to update the **current person** attribute, as required in the example described in 5.5.1, “A New View for the AddressBook” on page 71.

As before, in the Application Browser for the AddressBookAppl application, select **AAAddressBook** in the list and then select **Browse Class**. The application builder opens on the Public Interface Editor for the part AddressBook. After the notebook is opened:
• Switch to the Attributes page and add the new attribute `people`. You only have to specify the get selector `people` because the list box only needs to get the `people` attribute connected to the `items` of the list box itself. Finally, the class type should be set to `OrderedCollection`. Click on the `Add` push button.

• The list box `selectedItem` is connected to the attribute `current person`. Both the get and the set selector associated with the attribute `current person` must be defined and implemented. The get selector is necessary to retrieve information for each list item. Furthermore, whenever a new item is selected in the list, the list box will require the `addressBook` to set the new `current person`. Thus, the set selector must also be specified.

To add the set selector to the attribute `current person`, choose the `current person` attribute and click on the `Defaults` push button. Click on the `Update` push button.

• Before saving the part, select the option `Generate code` from the option menu, choosing `Generate selectors`.

• When prompted you select to generate the code for the `currentPerson:` selector.

The `currentPerson:` selector must be customized. How to do this is described in 7.3.1, “Adding Code to an Action Selector in AddressBook” on page 102.

### 6.4.4 Numeric Key Pad

We want to build a numeric key pad that could be used, for example, to build a calculator. The scope of a numeric key pad goes beyond its usage in a calculator; thus, we want to build it independently from its possible usage. The numeric key pad must have buttons for the numbers from 0 to 9, and a button for the decimal point. Its responsibilities are:

• Pass the number associated with the numeric button that is pressed.
• Signal when the decimal button is pressed.
• Prevent people from pressing the decimal key twice for an entry.
• Enable the decimal button for new entries.

To describe the initial processes of designing the numeric key pad we make use once more of the visual notes technique. We also want to represent some additional characteristics; therefore, we introduce the usage of arrows to represent further elements of the part with the following differentiations:

• Solid lines for behaviors (actions and code hooks)
• Dashed lines for attributes
The arrowheads for actions, attributes and events are outside the “border” of the part, while the arrowheads of code hooks are inside the border. To draw a visual note as illustrated in Figure 29, you start from the most evident and clear element of the part you are designing. Based on the part’s responsibilities you start adding subparts, behaviors and events. Further design considerations will make you decide which are the elements of the public interface of the part itself. Taking a step forward you will also decide if some elements require the creation of a new intermediate part to be reused.

In our design, we started drawing the NumericKeyPad part with the view composed of all the keys for the numbers and the decimal key. After this, we saw the need for a variable, keynumber, in which to store the number associated with the numeric key that was pressed. The variable keynumber is an attribute so that parts that will use the NumericKeyPad can access it. The class type for this variable must be able to store a number without any other special feature. We have selected Float.

In order to enable and disable the decimal key, the part NumericKeyPad needs two behaviors defined, enableDecimalKey and disableDecimalKey. The enableDecimalKey is clearly an action because the NumericKeyPad does
not know when a new entry begins. We have decided to define also the `disableDecimalKey` as an action to allow for the intermixing of entries from the NumericKeyPad and from a keyboard.

The various numeric keys have a special characteristic, in that they have a number associated with them. Furthermore, it is natural to expect that when one of these keys is clicked it is able to pass its number. The above considerations made us decide to create a new part, the NumericKey that passes its number when clicked, instead of assembling the view for the numericKeyPad from generic push buttons.

**Parts:** The NumericKeyPad part can be constructed from existing parts included in VisualAge and a new NumericKey part. Other parts needed are: a form, a push button, and a variable.

**Construction:** We will not describe in detail all the steps necessary to compose the parts, because the process of building them does not differ from what has already been described.

**NumericKey**

![NumericKey: Event with Parameter](image)
Among the various ways to build the NumericKey, we have selected the following:

- Edit the new part *(Create Visual Part)* **AANumericKey**.
- Delete the default window.
- Place a push button on the free-form surface, size it and set its label to \( n \).
- Switch to the Part Interface Editor by clicking on the proper icon on the bottom right of the Composition Editor window.
  - In the Attributes page add the attribute **number** and specify Float as Class type.
  - In the Events page, add the event **clicked** and specify **number** as parameter. (See Figure 30 on page 83.) After the event is defined, parts that will reuse the NumericKey will be able to connect to its event. (7.3.3, “Numeric Key” on page 104 describes the code hook necessary to signal this event.)
  - Select **Generate Code-Selectors** from the **File** pull-down menu. Generate all the code.
- Save the part.
- Promote it.
- Close.

**NumericKeyPad**: To compose the NumericKeyPad, follow these steps:

- Edit the new part **AANumericKeyPad**.
- Delete the default window from the Composition Editor free-form surface, and add a form.
- Place into the window a push button for the decimal key, size it and set its label.
- Place into the form NumericKey parts for the numbers and set *(Open settings)* the number and label of each of them to the appropriate value.
- Organize all the keys in a form suitable for a numeric key pad as shown in Figure 29 on page 82.
- You may also change the name of the parts and call them, one, two, three, etc., decimal.
- Add a variable named **keynumber** and specify Float as Class type. Add it to the interface.
- For each NumericKey, connect the event **clicked** to the **self** attribute of **keynumber**.

The next step is to define an event to the part interface that specifies the symbols used to signal that the decimal key has been pressed. To do so, switch to the Part Interface Editor for the part, select the Events page and enter the event **decimalKeyClicked** as shown in the following table.
The last step is to define the two actions to enable and disable the decimal key. To do so, select the Actions page and for each action enter its name and press the Defaults button. (See Figure 31 as reference.) As before, you must now add the code that signals the event. You also have to add the code for the two actions. This is described in 7.3.4, “Numeric Key Pad” on page 105.

<table>
<thead>
<tr>
<th>Event Name</th>
<th>Event Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>decimalKeyClicked</td>
<td>decimalKeyClicked</td>
</tr>
</tbody>
</table>

**Figure 31.** NumericKeyPad: Actions
6.4.5 Notes on Selecting Names

Whenever you add new parts or elements to a part interface you should use names that provide a useful vocabulary of interactions, and that provide consistency in attributes and operations across multiple parts without redundancies.

Whenever you add a part, its name should reflect what the objects are in the real world. For example, names of nonvisual parts are Address, AddressBook, Order, etc.

Similarly, the names you assign to attributes, actions and events should reflect what they are and what they do, independently from the part in which they are defined. For example, “city” and “street” are clear names for attributes, while “city of address book” and “street of address book” are not. A similar criteria should be followed for action names and code hooks. For instance, “go to first entry,” “about,” “update,” etc. are names that reflect the operation that must be performed. While “go to first entry of address book,” “about address book,” and “update customer” should be avoided.

The second set of choices relates the attributes and actions to a specific part. This is unnecessary because it is already clear to which part you are defining them. Furthermore, it also has negative effects, such as redundancy in the vocabulary, and poses strong limits on the usage and exploitation of polymorphism, which is an important characteristic of object-oriented programming. That is, the interpretation of a message depends on the object that receives the message and the same message can be interpreted in different ways by different objects.
Chapter 7. Visual Tools: Customization

This chapter relates to the customization of parts realized by using scripts to add or modify code either for code hooks or for actions. The writing of scripts is performed by using the Script Editor.

This chapter opens with an overview of the major elements and the syntax of the Smalltalk language that are used to write scripts. The focus is on the language, not on the overall Smalltalk system, knowledge of which is not necessary at this stage.

7.1 A Quick Walk through the Script Language

In Smalltalk everything is an object. To achieve a result, objects interact by sending messages. Objects and messages are used to implement the entire programming environment. A message is a request to an object to perform an operation. When an object receives a message, it determines and selects the appropriate operation and executes it. The implementation of the operation is called a method. The object to which the message is sent is called a receiver.

<table>
<thead>
<tr>
<th>Receiver</th>
<th>Message</th>
<th>Result</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>address</td>
<td>city</td>
<td>aString</td>
<td>The message city (selector with no arguments) is sent to address. The result is the instance of String object (aString) that holds city information.</td>
</tr>
<tr>
<td>7</td>
<td>* 2</td>
<td>14</td>
<td>The message * 2 (selector * with one argument ‘2’) is sent to the receiver 7. The result is the object 14.</td>
</tr>
</tbody>
</table>

A message is composed of:

- A selector, that specifies the name of the operation to perform. The term selector indicates that a selection will be performed from the allowed operations of an object.
- Arguments are objects, passed along with a selector, that are needed to perform the operation.
The response to a message is always an object, that signals the end of the execution and returns the proper answer.

You may notice that the emphasis is different from the emphasis that is common in traditional languages. Here the focus is on the receiver, that knows which operation to use to perform what is requested. In a traditional language, the operator is the key element. For instance, the operator * would be applied to the two “operands” 7 and 2 in the example 7 * 2.

It may become clearer now what we meant when we said that objects are self-contained software entities that are characterized by their state and behavior.

An object can be seen as an autonomous computing component that has private memory in which it stores its state, and an operation set. Computation and information about an object state can be requested by sending messages to it. The particular operation set depends on the type of entity the object represents. For instance, Number objects have mathematical operations, while Collection objects have operations to add, remove, enumerate elements, etc. Objects that represent your application entities have operations to perform business logic.

7.1.1 Variables

An object uses variables to reference its state. Each variable has an identifier that is unique within the object. All variables contain pointers to objects in memory. Unlike the usual concept of a pointer, the variable itself “stands” for the object. That is, you use the variable name in writing your statements and the resulting effect is as if you had sent the message directly to the object.

Note: We will be using the name of variables to refer to the instance of the objects to which the variables point.

Variables are assigned with code of the form:

```plaintext
city := aString.
number := enteredNumber * 10.
```

One thing to remember is that the variable does not contain a copy of the object; it contains a pointer to the object. Thus an assignment like `result := number`, does not involve any copy from `number` to `result`, but the variable `result` points to the same object as the variable `number`. 
7.1.1.1 Types of Variables
There are various types of variables in Smalltalk. Those that are likely to be used in writing scripts are:

- Instance variables
  These variables are defined for every object in a class. They assume specific contents for a specific instance. Their life lasts as long as the life of the instance.
- Temporary variables
  These variables are defined within a method and are lost when the method ends. They are placed between two vertical bars (||). For example:

<table>
<thead>
<tr>
<th>number:</th>
</tr>
</thead>
<tbody>
<tr>
<td>number := enteredNumber * 10.</td>
</tr>
</tbody>
</table>

7.1.1.2 Pseudo-Variables
Pseudo-variables point to instances of commonly used objects. For example, self and super point to the receiver. Whenever super is used, the lookup for a method starts in the superclass of the receiver. The pseudo-variable nil, instead, is used for objects that are undefined. We will mention some more of them in this chapter.

7.1.2 Practicing
While you read this section, you may like to practice by entering and executing the sample expressions as they are described. Smalltalk lets you execute any fragment of code and can return the result. A way to do this is:

- Type the fragment of code in the Transcript window.
- Select it, by clicking the mouse at the beginning of the code and dragging forward and down till the end of it.
- Activate the context menu by clicking on button two of the mouse.
- Choose the option Display from the menu. The code selected is executed and the result returned. The option Execute, instead, executes the code without showing the result.

The technique of executing fragments of code is commonly used even by expert Smalltalk programmers to quickly verify some new code they are writing!
Note: Executing the sample expressions that contain variables would cause errors, because the statements to define the variables are omitted. However, most of the examples that contain variables refer to selectors that exist in the parts implemented for the tutorial application. You may open the part, switch to the Script Editor, and look at the complete code.

7.1.3 Messages

There are three types of messages: unary, binary and keyword. There are rules of precedence when messages in an expression are evaluated. Parentheses can be used to enforce a desired sequence of execution. However, the understanding of the rules of precedence avoids the use of unnecessary parentheses that make the code more difficult to read.

7.1.3.1 Unary Messages

Unary messages have a selector and zero arguments. The message pattern is the selector itself. Examples are:

<table>
<thead>
<tr>
<th>Table 3. Unary Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Receiver</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 4. Unary Expressions: Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unary Expression</td>
</tr>
<tr>
<td>‘Anna’ size</td>
</tr>
<tr>
<td>3 factorial</td>
</tr>
<tr>
<td>addressBook goToFirstEntry</td>
</tr>
</tbody>
</table>

When multiple unary messages are sent one after the other, the precedence rule is from left to right. The rule maintains the emphasis on the receiver and guarantees that there is always a receiver to which a message is sent. For instance, the expression receiver message1 message2 is executed as follows:
1. receiver message1 returns an object that is the receiver of the second message.
2. receiver message2 returns the final object.

It should be clear that message2 is sent to the result object of the first step, not the original receiver. For instance, the expression address city street would give an error, because the result of address city is a string that does not understand the message street. Instead, the expression address city size is correct, because string understands the message size.

### 7.1.3.2 Binary Messages

<table>
<thead>
<tr>
<th>Receiver</th>
<th>Selector</th>
<th>Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>Special characters: +, =, ≠, &gt;, &lt;, etc.</td>
<td>argument</td>
<td></td>
</tr>
</tbody>
</table>

Binary messages are messages composed of a selector and an argument in which the selector is a special character, such as +, =, ≠, >, <, etc. The message pattern is selector argument. Examples are:

<table>
<thead>
<tr>
<th>Binary Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 + 2</td>
<td>The message + with argument 2 is sent to the number 7. The object returned is 9.</td>
</tr>
<tr>
<td>'Anna' = 'Martin'</td>
<td>The message = with argument 'Martin' is sent to the string 'Anna'. The result is an instance of the object False.</td>
</tr>
</tbody>
</table>

Once more, when multiple binary messages are coded in an expression, the rule of precedence is from left to right, that guarantees we always have an object to which the next binary message is sent.

This rule applies also to arithmetic operations that are typical binary messages. This implies that the execution of arithmetic operations does not conform to the common rules that assign priorities to the various operators. Parentheses are useful, in this case, to enforce a desired sequence of evaluation.
7.1.3.3 Keyword Messages

Keyword messages are the last type of message, in which the selector contains one or more keywords each one with an argument. Each keyword ends with a colon (:).

The message pattern of a keyword message is: receiver keyword1: argument1 keyword2: argument2 keyword3: argument3. Examples are:

<table>
<thead>
<tr>
<th>Receiver</th>
<th>Selector</th>
<th>Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>keyword1:keyword2:keyword3:</td>
<td>argument1 argument2 argument3</td>
<td></td>
</tr>
</tbody>
</table>

Table 7. Keyword Expressions

Smalltalk assumes that all the keywords that follow a receiver constitute its selector. If this is not the case, parentheses must be used. For example, if you code the expression #(1,’Martin’,3,’Anna’) at: 2 at: 2, the message at: at: is sent to the array and would return an error. (#(1,’Martin’,3,’Anna’) at: 2) at: 2 would cause the first message at: to be sent to the array and the second message at: to be sent to ‘Martin’ that would answer ‘a’, that is the second letter in Martin.

7.1.3.4 Evaluating Expressions

We have mentioned the rules that are applied to the various types of messages. But what happens when an expression contains different types of messages? Each expression is evaluated in an order that adheres to the basic concept of going through steps that resolve receivers, keeping in line with the basic format of interaction among objects: send a message to a receiver (receiver message):

1. Messages in parentheses are first.
2. All the unary messages, from left to right, are second.
3. All the binary messages, from left to right, are third.
4. Keyword messages are last.

### 7.1.3.5 Sequence of Expressions

When multiple expressions must be coded and evaluated in sequence, a period (.) is used at the end of each expression. If the period is omitted, Smalltalk concatenates all of the expressions into a single expression which it resolves, based on the kind of messages it supports and the precedence rules.

An expression preceded by the ^ symbol is called **return expression**. After the expression is evaluated, the processing ends and the result of the expression is returned. For example:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>^people at: currentIndex</td>
<td>Suppose the variable <em>currentIndex</em> points to the current index within the addressBook, while the variable <em>people</em> points to the collection of persons in the addressBook. The message at:currentIndex is sent to people and the current person instance is returned.</td>
</tr>
</tbody>
</table>

### 7.1.3.6 Cascade

When multiple messages must be sent to the same receiver they can be written as a sequence of expressions. For example:

```plaintext
address city: aString. address street: aString. address zipCode: aNumber.
```

However, a more concise way is to use **cascaded messages**. Cascaded messages are sent to the same receiver and are separated by semicolons (;). The above example becomes:

```plaintext
address city: aString; street: aString; zipCode: aNumber.
```

### 7.1.3.7 Results

The result of expressions are objects, as described in the examples in Table 4 on page 90, Table 6 on page 91, and Table 8 on page 92. There are some cases that are worth pointing out:
• When a receiver is requested to perform an action that does not return an object, the receiver itself is returned. An example is `addressBook goToFirstEntry` described in Table 4 on page 90.

• An object can always be requested to return itself by sending it the message `yourself`.

The following example shows the effect of the message `yourself`.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>#(1 2 5 6) copy at: 2 put: 7.</td>
<td>The message <code>at:put:</code> with arguments 2 and 7 is sent to a read-write copy of the array <code>#(1 2 5 6)</code>. The second element of the array is set to the number 7. The result is the number 7, that is, the second element of the array.</td>
</tr>
</tbody>
</table>
| #(1 2 5 6) copy at: 2 put: 7; yourself.  | The message `at:put:` with arguments 2 and 7 is sent to a read-write copy of the array `#(1 2 5 6)`. The second element of the array is set to 7. The message `yourself` is sent to the array; the result, in this case, is the array `#(1 7 5 6)`.

### 7.1.4 Methods

**Method** is the name that is given to the implementation of an operation. When a message is sent to an object, the message selector is the unique identifier of a method in that object. When the selector in the message matches a method name of the object, the method is invoked and the function associated with it is performed.

The body of the method is a sequence of statements whose evaluation results in an object that is returned as the value of the method. Each statement is composed of expressions that contain the name of objects, selectors, and arguments as required.

Methods have the following form:

```
message pattern
  "comment"
body
```

For example, the following is the method in the AddressBook part that returns the attribute `current person`:
There are two kinds of methods:

1. Class methods. Typically class methods create new instances of a class.
2. Instance methods. These methods implement most of the operations of an object. They are performed by instances of a class.

### 7.1.5 Conditional and Iterative Control Structures

Control structures are implemented using objects and messages among objects. Control structures make use of **blocks** that are instances of standard Smalltalk classes. A block object is represented by square brackets.

#### 7.1.5.1 Blocks

Blocks allow the writing of advanced and customized control structures for almost any object. For the scope of writing scripts we focus on the usage of blocks in standard control structures, such as conditional and iterative structures. In this perspective, a block can be seen as a sequence of one or more expressions enclosed in square brackets, such as:

\[
\text{[expression1. expression2. expression3. ]}
\]

All the expressions in a block are evaluated and the result of the last expression in the block is returned, unless there is an explicit **return expression** somewhere.

#### 7.1.5.2 Conditional Structures

Conditional structures are handled using the two pseudo-variables **true** and **false** that point to the two corresponding boolean objects that are instances of the boolean classes True and False.

The **true** and **false** objects understand both a true/false protocol and respond accordingly. The simplest selectors are: **ifTrue:[ ]** and **ifFalse:[]**. The code in the block is executed depending on which of the two objects is receiving the message.
Who returns the objects true and false? Any binary message that compares two objects, for instance, such as variable1 < variable2, or variable1 >= variable2, and so on. The result of each one of the mentioned expressions is either a true or a false.

Here is an example:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>true ifTrue: [&quot;yes&quot;]</td>
<td>The message ifTrue: with parameter [&quot;yes&quot;] is sent to the true object. The expression in the block is executed and the answer is 'yes'.</td>
</tr>
<tr>
<td>true ifFalse: [&quot;yes&quot;]</td>
<td>The message ifFalse: with parameter [&quot;yes&quot;] is sent to the true object. The expression in the block is not executed and the answer is nil.</td>
</tr>
<tr>
<td>false ifTrue: [&quot;yes&quot;]</td>
<td>The message ifTrue: with parameter [&quot;yes&quot;] is sent to the false object. The expression in the block is not executed and the answer is nil.</td>
</tr>
<tr>
<td>false ifFalse: [&quot;yes&quot;]</td>
<td>The message ifFalse: with parameter [&quot;yes&quot;] is sent to the false object. The expression in the block is executed and the answer is 'yes'.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(number 2) = 0 ifTrue: [&quot;The number is even&quot;]</td>
<td>The message 2 is sent to number; the result is the remainder of the division. The message = 0 is sent to the remainder of the division. The result is either the boolean true or the boolean false; the message ifTrue: with the block as argument is sent to it. When the result is the boolean true, the block is executed and the string 'The number is even' is returned. Instead, when the result is the boolean false, the block is not executed.</td>
</tr>
</tbody>
</table>

There are some other selectors that true and false understand, such as ifTrue:ifFalse:. Here is an example that is in the body of getPreviousPerson method in the addressBook part:
Table 10. Conditional Structures: ifTrue:ifFalse: Message

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
</table>
| (currentIndex = 1) ifTrue: [currentIndex := (people size)] ifFalse: [currentIndex := (currentIndex - 1)] | currentIndex is a variable that contains the current page number of the addressBook. (currentIndex = 1) returns either the boolean true or the boolean false.  
When the answer is true, the message ifTrue:ifFalse: with the two blocks as arguments is sent to true. As result, the block that is argument of ifTrue: is executed.  
The effect is that the addressBook will position at its end.  
When the answer is false instead, the block argument of the ifFalse: is executed. |

7.1.5.3 Iterative Structures

Iterative structures include conditional and deterministic repetitions.

Conditional Iteration: Conditional repetitions are obtained by sending one of the following messages to a block:

1. whileTrue and whileFalse.
2. whileTrue: [code] and whileFalse: [code].

The following table points out the difference between the two whileTrue kinds of messages.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[code] whileTrue</td>
<td>The message whileTrue is sent repetitively to the block. Each time the message is sent, the block is evaluated by executing the code in the block. The message is sent until the evaluation is false.</td>
</tr>
<tr>
<td>[code] whileTrue: [code.]</td>
<td>The message is sent repetitively to the receiver block. The receiver block and the block argument of whileTrue: are evaluated until the evaluation of the receiver block is false.</td>
</tr>
</tbody>
</table>

The usage of the whileFalse kinds of messages are similar.
Deterministic Repetition: The following table shows some of the common expressions used to obtain deterministic repetition.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n timesRepeat: [code]</td>
<td>The message timesRepeat: is sent to the number n; the argument block is evaluated n times.</td>
</tr>
<tr>
<td>n to: m do: [code]</td>
<td>The message to:do: is sent to the number n. The argument block is evaluated multiple times, and each time the number n is incremented by one, until the number m is reached. The message to:by:do: is a variation, in which the increment between the number n and the number m is the argument of the keyword by:.</td>
</tr>
<tr>
<td>object do: [:variable</td>
<td>code ]</td>
</tr>
</tbody>
</table>

The following example shows how the do: message can be used to evaluate the sum of the square of the first 5 even numbers.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sum := 0.</td>
<td>The message do: is sent to the array. The array uses the :number as temporary variable to point to its elements (one at a time) and executes the code repetitively for each element.</td>
</tr>
<tr>
<td>#(2 4 6 8 10) do: [:number</td>
<td>sum := sum + (number * number)]</td>
</tr>
</tbody>
</table>

The message do: is not limited to iterations over fixed ranges of integers. It can be implemented in any class that we want to iterate over one or multiple of its operations.

7.1.6 Creating Instances of Parts from Code

The code described in this section gives an example of the code that may be used to create instances of parts from Smalltalk code. It may also be useful to test parts that contain variable attributes.
After a new instance of the part AddressBookListView is created (1), the keyword message `valueOfAttributeNamed:ifAbsent:put:` with arguments is sent to it (2).

Based on the Smalltalk precedence rules, the first expression that is executed is `AddressBook new`. This will create an instance of AddressBook. The method `new` invokes the method `initialize` that initializes people’s information.

When the keyword message is executed, the attribute `addressBook` is assigned to the instance of `AddressBook`. If the attribute does not exist (`ifAbsent:`) for some reason (typically errors in typing), the attribute itself is assigned to `nil`.

The last cascade message sent to the part is `openWidget` (3), which will cause the part to open, presenting whichever primary part was defined in it, a list box in this case.

**Note:** If the primary window is a form, instead of the message `openWidget`, the message `openInShellView` should be used.

### 7.2 Notes on Using the Script Editor

The Script Editor provides the editing facilities to enter the script attached to an event or an attribute when you make connections to code hooks. It is also used to enter methods related to actions or attribute selectors.
Figure 32 depicts the Script Editor. It shows you and lets you operate on the internals of a part: its variables, and methods. You can request to list either class or instance elements and, in both cases, you can request to access either public or private methods. Usually, public methods are methods that implement actions, while private methods are commonly used for scripts connected to events and attributes. You can either update an existing method or add a new method to the part.

Along the left side of the Script Editor there is a tool bar with tools that provides an aid when you enter new methods: you can ask the Script Editor to insert the appropriate code for selected attributes you need to refer to or use in your method. You can also ask the Script Editor to assist you in writing common Smalltalk expressions such as comparisons, iterations and control statements. To access an attribute in a subpart, you select the Attribute tool. The Attribute values window lists all the subparts and for each subpart you can select the desired attribute from the list that is shown. After the selection is made, you can ask the Script Editor to paste the code for either a set or get of the attribute. Similarly, using the Action tool you can be assisted in writing expressions that request a subpart to perform an action.
The appropriate code, as shown below, is inserted at the current position of the cursor in the top-right window where the method is edited.

<table>
<thead>
<tr>
<th>Element</th>
<th>Opt.</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attribute</td>
<td>Get</td>
<td>(self partAttributeValue: #(subpartName attributeName))</td>
<td>The result of the execution of this code returns the attribute specified. If you select the self attribute, the subpart itself is returned. After this code you can enter any message that this type of object understands. The messages it understands are the instance messages of the class type (and its superclasses) of the part.</td>
</tr>
<tr>
<td>Attribute</td>
<td>Set</td>
<td>(self partAttributeValue: #(subpartName attributeName) put: &lt; your expression here &gt;)</td>
<td>You replace &lt; your expression here &gt; with the necessary code. The result of the execution of this code sets the attribute specified. If you select the self attribute, the subpart itself is set.</td>
</tr>
</tbody>
</table>

**Note:** Scripts for code hooks can also be entered in the code hooks settings view (see Figure 34 on page 104) that is opened when you add a code hook or when you open its setting.

### 7.3 Writing Scripts

Examples are the best way to become familiar with the Script Editor. This section contains some examples of scripts.

To add scripts, you open the Script Editor for the part you are editing by clicking on the proper icon in the Composition Editor window. Remember that, whenever you make connections to scripts, the connection is between an event or attribute of the part itself or of any of the subparts and code in the part you are editing.
7.3.1 Adding Code to an Action Selector in AddressBook

In this example we customize the set selector `currentPerson:` of the attribute `current person` in the AddressBook, as required by the example 5.5.1, "A New View for the AddressBook" on page 71.

Each time a new name is selected in the list, the selected object changes. Because the `selectedItem` attribute is connected to the attribute `current person` of the AddressBook, the list box informs the AddressBook about the selected object. The responsibility of AddressBook is to set the `current person` of `addressBook`, that means set the index of the `addressBook` to point to the person selected. Thus the code in the selector `currentPerson:` must set the `currentIndex` of the `addressBook` to the selected object.

To do what has been described:

- Edit the part `AddressBook`.
- Switch to the Script Editor.
- Be sure that `instance` and `public` are selected.
- In the list of methods, click on `currentPerson`.
- In the method editing window enter the code as described below.
- Select `Save` from the context window.

```small
    currentPerson: aAAPerson
    "Save the value of current person."
    aAAPerson notNil ifTrue: [
      currentIndex := people indexOf: aAAPerson
    ].
    self signalEvent: ("current person" asSymbol).
```

The code is quite straightforward: if the object `aAAPerson` is not `nil`, it is used as an index to position into the `people` variable. The result is assigned to the variable `currentIndex`. The fact that the current person is now different is signaled in the last expression that uses the `current person` changed event symbol.

7.3.2 A Sorted List of People’s Names

This example is similar to the one described in 5.5.1, "A New View for the AddressBook" on page 71. The major difference is that we want the list of people to be sorted by name, as shown in Figure 33 on page 103. You may build a new part, or you may modify the existing `AddressBookListView` part. In the former case, you edit a new part and you go through the steps described in the mentioned examples skipping the step that connects `items` of the list box to `people` of the `addressBook`. 
Instead of the attribute-to-attribute connection, this time you must add an attribute code hook.

Let us start entering the code using the Script Editor (see Figure 32 on page 100). To open the Script Editor click on the Script Editor icon shown on the bottom-right of the Composition Editor. After the Script Editor is opened, be sure that instance and private are selected. In the pull-down Methods menu select New Method Template. In the method editing window enter the following code, which sorts the items in the list box based on the names of people:

```plaintext
getSortedListOfPeople
^{(self partAttributeValue: 
   #(addressBook people))}
ifNotNil: [:people | people asSortedCollection: [:new :old | new name <= old name]]
```

There are other examples that can be constructed with similar code. You may want to add a Sort menu bar option and define various choices for sorting. In this case you would use events (the option is selected) to script connections, in which the script would be similar to the one described above.

To add the attribute to script connection:

- Position the pointer on the list box and click mouse button 2 to open the context menu and select Attribute-script connection.
- A window similar to the window in Figure 34 on page 104 is displayed.
- Be sure that private is selected.
- Select items in the Attribute name entry field.
- Select getSortedListOfPeople in the methods list.
7.3.3 Numeric Key

This section shows how to add an event to script connection. The NumericKey needs a script to signal when it has been pressed and pass at the same time the number associated with it. The code that performs the request is:

```plaintext
clicked
"signal clicked passing the attribute number"
self signalEvent: #clicked with: self number
```

After entering the above code with the Script Editor, return to the Composition Editor:

- Position the cursor over the push button, click `mouse button 2`, and select `Event-to-script connection`.
- Specify `clicked` as Event name (you can use the drop-down list box).
- Select `clicked` as method name.
- Click `mouse button 1` on the `Make` push button.
7.3.4 Numeric Key Pad

Add the event to script connection for the `decimalKeyClicked` as explained for the NumericKey (see 7.3.3, “Numeric Key” on page 104). In this case the method you should enter is:

```plaintext
decimalKeyClicked
self disableDecimalKey.
self signalEvent: #decimalKeyClicked
```

We see that the method to disable the decimal key is executed before signaling the decimal key has been pressed. To enter the code for the two actions, switch to the Script Editor. The two methods are already there because we selected to generate the code when we defined them in the Part Interface Editor. However, we must customize the code. Let us describe how to disable the decimal key. To do so (see Figure 35 as reference):

- Position the cursor on the first line below the comment that the Visual Tool has provided.
- To access the decimal key, use the Attributes tool, select **decimal** (if you have renamed the parts when building the NumericKeyPad) in the Attribute values window and **enabled** in the attributes list.

- Press **mouse button 2** to access the context menu and choose **set value**.

- In the line of code that is written for the method, replace `< your expression here >` with **false**.

- Save the method.
Chapter 8. VisualAge Supporting Components: Overview

This chapter contains an overview of the framework that VisualAge provides for application development. The chapter is intended to be an introduction for people who will fabricate parts that make use of the VisualAge components. It can also be read by those people who want to have a better understanding of the VisualAge components that support the generic visual parts for external functions and database access.

For a description of cooperative processing issues in object-oriented environments and a preliminary design of VisualAge cooperative processing components see Cooperative Processing in an Object-Oriented Environment.

The framework itself as well as its tools and components are used by VisualAge to implement some of the generic nonvisual parts. The same tools and components can also be used directly by your applications; some of the components, organized in classes and subsystems, may be refined by your applications and extended, if necessary, to provide additional support.

The chapter includes:

• Application support to structure applications
• Tools to interface COBOL and C programs
• Client/server support over multiple network protocols

8.1 Tools to Interface COBOL and C Programs

VisualAge provides interactive tools and libraries for defining interfaces to remote and local COBOL and C programs. They are the underlying implementation for the generic visual tools DLL part. A model builder facility in the application builder provides a means to create and customize a part that models a DLL and its records.

Whenever information has to be exchanged between an application written in an object-oriented language and an application written using a third generation language (3GL), such as COBOL or C, there are rather different interface structures to match: objects in the object-oriented side, C or COBOL flat record structures on the 3GL side. To allow information exchange between them, a mapping between the two different structures must be done.
3GL languages use the concept of data types, fields, subrecords and records. The way records can be defined varies from language to language. In any case a record has a layout, that is, a sequence of subrecords and/or elementary items, which have a specific data type and a specific length.

8.1.1 COBOL and C Parsers
The interface aids support the mapping of objects to and from fields in record structures. They simplify the exchange of information between objects and 3GL programs by providing application objects and parts with a means to access records and fields by using their names. All the mentioned services are transparent to the programmer; that is, the programmer does not become involved with the underlying code.

The parsers read C header files and COBOL copy files and parse them to create the necessary objects that describe the layout of the record, handle 3GL types, handle the definition of constants, manage the memory areas that store the values of the fields, etc.

8.1.2 Remote Programs (Foreign Structures)
When the 3GL program runs on a system with a different architecture, the encoding scheme may be different. In fact the internal physical representation of a data type can be quite different in dissimilar systems. For character data, the representation may also depend on the country and the code page used. In these situations a conversion is needed.

The interface aids provide support for automatic conversion of ASCII to EBCDIC and vice versa, binary number and packed decimal. Once more, conversion services are transparent to the programmer.

8.2 Relational Database
VisualAge provides a set of classes that facilitate accessing the OS/2 Extended Services Database Manager. It provides functions to start and stop the DB manager, create and drop DBs, connect and disconnect DBs as well as access DBs for select, insert, delete and update operations. Among the classes, those of most interest to the application programmers are the classes that represent queries and query results.

The DB support is the underlying implementation of the generic DB parts available in the visual tools.
8.3 Client/Server

VisualAge provides support for the three client/server areas described in 2.1, “Client/Server” on page 11:

- Access to remote databases
- New application with access to transactions and programs in network
- Facelift

The access to remote databases is transparently provided by the relational database support. The two other kinds of client/server support are described below.

8.3.1 Access to Transactions and Programs in a Network

This component is the underlying implementation of the generic remote function parts provided by the visual tools.

VisualAge provides application developers with an innovative and simple interface for the client portion of a client/server application whose servers are remote transactions and programs. The interface is expressed in terms of the information that has to be exchanged and is completely independent from any semantic that is usually associated with both communication and networking.

The interface is provided by classes called Dialogs that abstract the information exchange. What Dialogs understand is a simple protocol that consists of a message that simply requests to perform the exchange (message doit); what they accept and return are records. However, the support is high function, allowing the exchange of a single record (similar to a procedure call) as well as the exchange of multiple records within a single request. Dialogs provide an interface that is both communication and protocol independent.

VisualAge provides Dialogs for various protocols, such as APPC, NetBIOS, TCP/IP, and ECI. How sophisticated the Dialog can be depends on the support provided by the protocol and the communication interface.

The communication components have a layered structure for dialogs, system and interface, with classes that provide support at the various levels.

- The Dialog layer implements the dialogs and provides application objects with a protocol-independent interface to access remote transactions.
- The system layer provides a complete object-oriented implementation of the communication protocol and interface.
• The interface layer accesses the operating system network interface.

This layering enhances the extendibility of the components, simplifying the possible additions of either new dialogs or new protocols.

8.3.2 Facelift: EHLLAPI Support

VisualAge provides classes that interface the EHLLAPI terminal support and abstract concepts such as screen, screen fields, host applications, etc. By using this support a client portion of an application with an advanced GUI can be implemented to access existing 3270 and 5250 applications, without changes to the host application.

In the client, the business objects can easily be kept separated from the details of the 3270 support and the host program. If at a later stage, the host application is renewed and changed to make use of a more advanced and effective communication protocol, the client portion or the application can use the classes for the new protocol without changes to the business logic or to the user interface. This is achieved through the design of the communication components and the nature of object-oriented technology. The facelift support of VisualAge provides a powerful means to move toward client/server computing by stages.

This component is the underlying implementation of the generic 3270 parts provided by the visual tools.
Appendix A. Visual Tool AddressBook Class

This appendix contains the code for the AddressBook class used in the address book sample application (see Chapter 4, "A Sample Application" on page 41).

PxPart subclass: #AAAddressBook
  instanceVariableNames: 'people currentIndex'
  classVariableNames: '
  poolDictionaries: '' !
AAAddressBook methods !
currentPerson
  "This method name matches the default selector generated
  by the Interface definition view for the current person attribute"
  ^ (people at: currentIndex)!
currentPerson: anAAPerson
  "Save the value of current person."
  self signalEvent: ('current person' asSymbol).!
getNextPerson
  "This method name matches the default selector generated
  by the Interface definition view for the get next person action"
  (currentIndex = people size)
  ifTrue: [currentIndex := 1]
  ifFalse: [currentIndex := (currentIndex + 1)].
  "Signal the change event as defined in the current person attribute
to indicate that the value of the current person has changed"
  self signalEvent: ('current person' asSymbol).!
getPreviousPerson
  "This method name matches the default selector generated
  by the Interface definition view for the get previous person action"
  (currentIndex = 1)
  ifTrue: [currentIndex := (people size)]
  ifFalse: [currentIndex := (currentIndex - 1)].
  "Signal the change event as defined in the current person attribute
to indicate that the value of the current person has changed"
  self signalEvent: ('current person' asSymbol).!
goToFirstEntry
  "This method name matches the default selector generated
  by the Interface definition view for the go to first entry action"
  currentIndex := 1.
  "Signal the change event as defined in the current person attribute
to indicate that the value of the current person has changed"
self signalEvent: ('current person' asSymbol)!

goToLastEntry

“This method name matches the default selector generated by the Interface definition view for the go to last entry action”

currentIndex := (people size).

“Signal the change event as defined in the current person attribute to indicate that the value of the current person has changed”
self signalEvent: ('current person' asSymbol)!

initialize

“This method will initialize the address book with sample people and set the initial currentIndex to the first entry”

people := ((OrderedCollection new)
  add: self personCleopatra;
  add: self personNapoleonBonaparte;
  add: self personGeorgeWashington;
  add: self personJoanRivers;
  add: self personAlbertEinstein;
  yourself).

currentIndex := 1.!

people
“Return the value of people.”
^people!

people: anOrderedCollection
“Save the value of people.”

people := anOrderedCollection.
self signalEvent: #people.!

personAlbertEinstein

“This method will create a fictitious sample person”

^(AAPerson new
  name: 'Albert Einstein';
  homeAddress: (AAAddress new
    street: '607 Relativity Way';
    city: 'San Diego';
    state: 'CA';
    zipCode: 48338;
    yourself);
  homePhoneNumber: '555-7908';
  workAddress: (AAAddress new
    street: '4242 Newtons Lane';
    city: 'San Diego';
    state: 'CA';
    zipCode: 48522;
    yourself);
  workPhoneNumber: '555-5030';
  yourself)!

personCleopatra

“This method will create a fictitious sample person”

^(AAPerson new

112 VisualAge: Concepts and Features
name: 'Cleopatra';
homeAddress: (AAAddress new
    street: '5432 Mark Anthony Avenue';
    city: 'Egypt';
    state: 'VA';
    zipCode: 23474;
yourself);
homePhoneNumber: '555-4542';
workAddress: (AAAddress new
    street: '343 Pharoah Place';
    city: 'Nile Creek';
    state: 'VA';
    zipCode: 23474;
yourself);
workPhoneNumber: '555-8423';
yourself);

personGeorgeWashington

"This method will create a fictitious sample person"

^(AAPerson new
    name: 'George Washington';
    homeAddress: (AAAddress new
        street: '434 Betsy Ross Road';
        city: 'Marthas Vineyard';
        state: 'MD';
        zipCode: 32345;
        yourself);
    homePhoneNumber: '555-3453';
    workAddress: (AAAddress new
        street: '2343 Cold Court';
        city: 'Valley Forge';
        state: 'MD';
        zipCode: 32345;
        yourself);
    workPhoneNumber: '555-7737';
    yourself);

personJoanRivers

"This method will create a fictitious sample person"

^(AAPerson new
    name: 'Joan Rivers';
    homeAddress: (AAAddress new
        street: '3443 Arsenio Avenue';
        city: 'Grand Canyon';
        state: 'AZ';
        zipCode: 34177;
        yourself);
    homePhoneNumber: '555-2475';
    workAddress: (AAAddress new
        street: '32734 Can We Talk Trail';
        city: 'Hollywood';
        state: 'CA';
        zipCode: 34454;
        yourself);
    workPhoneNumber: '555-6563';
    yourself);

personNapoleanBonaparte

"This method will create a fictitious sample person"

^(AAPerson new
name: 'Napoleon Bonaparte';
homeAddress: (AAAddress new
    street: '42762 Bastille Blvd';
    city: 'Paris';
    state: 'AL';
    zipCode: 56550;
yourself);
homePhoneNumber: '555-5342';
workAddress: (AAAddress new
    street: '3443 Waterloo Way';
    city: 'Leningrad';
    state: 'AL';
    zipCode: 56554;
yourself);
workPhoneNumber: '555-8734';
yourself)! 

# List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>APPC</td>
<td>Advanced Program-to-Program Communication</td>
</tr>
<tr>
<td>ASCII</td>
<td>American Standard Code for Information Interchange</td>
</tr>
<tr>
<td>CICS</td>
<td>American Standard Code for Information Interchange</td>
</tr>
<tr>
<td>combo</td>
<td>combination box</td>
</tr>
<tr>
<td>CUA</td>
<td>American Standard Code for Information Interchange</td>
</tr>
<tr>
<td>DB</td>
<td>database</td>
</tr>
<tr>
<td>DDE</td>
<td>dynamic data exchange</td>
</tr>
<tr>
<td>DLL</td>
<td>dynamic link library</td>
</tr>
<tr>
<td>EBCDIC</td>
<td>Extended Binary-Coded Decimal Interchange Code</td>
</tr>
<tr>
<td>ECI</td>
<td>external call interface</td>
</tr>
<tr>
<td>EHLLAPI</td>
<td>emulator high-level language application programming interface</td>
</tr>
<tr>
<td>GUI</td>
<td>graphical user interface</td>
</tr>
<tr>
<td>IDP</td>
<td>iterative development process</td>
</tr>
<tr>
<td>ITSC</td>
<td>International Technical Support Center</td>
</tr>
<tr>
<td>LOB</td>
<td>line of business</td>
</tr>
<tr>
<td>NetBIOS</td>
<td>Network Basic Input/Output Subsystem</td>
</tr>
<tr>
<td>OLTP</td>
<td>online transaction processing</td>
</tr>
<tr>
<td>SQL</td>
<td>Structured Query Language</td>
</tr>
<tr>
<td>TCP/IP</td>
<td>Transmission Control Protocol/Internet Protocol</td>
</tr>
<tr>
<td>UI</td>
<td>user interface</td>
</tr>
<tr>
<td>3GL</td>
<td>third-generation language</td>
</tr>
</tbody>
</table>
Index

Numerics
2-D graphical 25
  2-D graphical editor 25

A
actions 27, 45, 75
  actions 27, 45, 75
Adding
  attribute 50
  constructing 50
  subpart 50
Adding an Attribute
  attribute 80
adding code 102
  adding code 102
adding elements 77
  adding elements 77
adding attributes 78
  adding events 78
address 44
  address 44
address book 41, 45, 64
  address book 41, 45, 64
addressbook 45
addresses 41
creating applications 41
  models 42
  object-oriented programming 41
address view 47, 51
  address view 47, 51
  address 47
  address view 47, 51
  instances 47
  interface 47
addressbook 71, 79
  addressbook 71, 79
addressbookappl application 79, 80
  addressbookappl application 79, 80
  attribute 79
addressview 47, 52, 62
  addressview 47, 62
advanced graphical user interface 17
  advanced graphical user interface 17
application 25, 29
  application 25, 29
application developers 9
  application developers 9
application development 8, 11, 13, 107
  application development 8, 13, 107
  iterative approach 13
Application Development Themes
  application development 11
application model 37
  application model 37
Application Scenario 25
  application scenario 25
  application scenarios 1
  computing scenarios 1
  information system 1
  personal workstations 1
  typical scenario 2
application scenarios 1
application support 107
  application support 107
  client/server 107
  object-oriented language 107
application-driven 12
  application-driven 12
application-specific attributes 63
  application-specific attributes 63
assembling 55
  assembling 55
attribute 31, 50, 80
  attribute 31
  attributes 62
  connections 62
  subparts 62
attribute connected 81
  attribute connected 81
  get selector 81
attribute connected (continued)
set selector 81
attribute-to-attribute connection 103
attributes 26, 31, 45, 46, 62, 75
attributes 26, 31, 45, 46, 75
editor 75
visual 75

B
build visuals 76
attribute 76
build visuals 76
get 77
non-programmer 76
selector 77
set 77
building composite parts 59
building composite parts 59
inherited 60
nested 60
non-nested 60
real-world object 60
subclass 60
building the application 48
building the application 48
subpart 49
business applications 5
business applications 5
business environment 2
business environment 2
dynamic 2
organization 2
business logic 3
business logic 3
business models 78
business models 78
business objects 9
business objects 9

C
change event 33
change event 33

Changing
attribute 51
ten field 51
changing environment 2
class methods 95
class methods 95
Client/Server 109
access 109
APPC 109
client/server 109
generic perspective 11
client/server application development tool 5
client/server component 26
client/server computing 5, 11
client/server computing 5, 11
code hooks 87, 101
code hooks 87, 101
collection objects 88
collection objects 88
instance variables 89
pseudo-variables 89
temporary variables 89
variables 88
common user 6
common user access 6
communication support 6
APPC 6
communication support 6
ECI 6
EHLLAPI 6
netbios 6
TCP/IP 6
company 1
analysis 1
changing environment 1
cOMPANy 1
decision making 1
individual 1
processing 1
rapidly changing 1
reshaping american business 1
sophisticated 1
company (continued)
workplace 2000 1
compose parts 29
compose parts 29
composite parts 56
component 57
composite part 56
composite parts 56
nesting 56
subparts 57
conceptual model 37
conceptual model 37
concrete realization 55
concrete realization 55
instance 56
objects 56
conditional structures 95
conditional structures 95
configuration management 17
configuration management 17
connected attributes 70
attribute 70
connected 70
connected attributes 70
connections 70
connections 30, 65
actions 65
attribute-to-attribute 51
connections 30, 65
events 65
making connections 51
notification 66
scripts 65
text-entry fields 51
Construction 83
construction 83
construction from parts 18
construction from parts 18
context menu 51
connections 51
context menu 51
text-entry field 51
conventional programming 3
conventional programming languages 3

cooperative processing 107
cooperative processing 107
Creating
primitive part 79
creating instances 98
creating instances 98
smalltalk code 98
Criteria
addressbookview 63
attribute 63
connections 64
defining attributes 63
subparts 64
CUA notebook 33
CUA notebook 33
customize conversion 61
customize conversion 61
customize parts 25
customize parts 25
customized processing 33
customized processing 33

D
database access 13, 17
database access 13
database manager 13
database manager 13
database queries 6
database queries 6
decision support 12, 13
decision support applications 12
decision support tools 13
deferred updates 39
deferred 39
deferred updates 39
intermediate agents 39
defining attributes 63
description 71
addressbook 73
collections 72
construct 71
description 71
objects 72
test 73
description (continued)

teaching part 71
deterministic repetition 98
deterministic repetition 98
development methodologies 20
application development 21
common behaviors 21
development methodologies 20
interchangeable components 20
object-oriented technology 21
visual programming 21
development paradigm 18
development paradigm 18
distributed databases 5
distributed databases 5
dynamic reconfigurations 2
dynamic reconfigurations 2

E
editing facilities 99
connected 100
ingesting facilities 99
events 100
editing options 61
editing options 61
EHLAPI support 110
EHLAPI support 110
enterprise networking 5
enterprise networking 5
multimedia 5
pen-based systems 5
entry field 51
evaluating expressions 92
cascade 93
evaluating expressions 92
expressions 93
methods 94
results 93
event code hook 104
event code hook 104
events 27, 75
events 27, 75
Examples
some examples 71

F
fabricate parts 107
fabricate parts 107
financial transactions 13
financial transactions 13
flexible application 5
flexible application 5
form processing 36
form processing 36

G
generate code-selectors 79
generate code-selectors 79
generate selectors 81
generate selectors 81
generic parts 37
generic parts 37
generic perspective 11
generic query 36
generic query 36
generic transaction 36
generic transaction 36
graphical representations 15
graphical representations 15
visualization 15
graphical user 6, 11
graphical user interface 6, 11
graphical user interface 20

H
heterogeneous environment 20
heterogeneous environment 20
heterogeneous servers 3
heterogeneous servers 3
Numeric
adding subparts 82
arrowheads 82
arrows 81
behaviors 82
decimal key 81
events 82
numeric button 81
numeric key pad 81
numerickeypad 82
numeric key 104
numeric key 104
numeric key pad 81, 105
numerickey 83
numerickey 83

O
object-oriented 5
  graphical user interface 20
  object-oriented 5
object-oriented application 7
  object-oriented application 7
object-oriented development 6
  object-oriented development 6
remote 6
transactions 6
object-oriented environments 107
  object-oriented environments 107
object-oriented interface 16
  object-oriented interface 16
object-oriented skills 9
  object-oriented skills 9
object-oriented technology 9
  object-oriented technology 9
objects mapping 7
  objects mapping 7
original view 73
  original view 73
parts 73

P
packaging 17
  packaging 17
  part assembly 55
    attributes 55
    objects 55
    part assembly 55
    terminology 55
    visual programming 55
    part interface 67
      editing 67
      part interface 67
      part interface editor 80
      part interface editor 80
    part-to-subparts 57
      application scenario 58
      assembly process 58
      part-to-subparts 57
      visually build 58
      visually define 58
parts 25, 83
  construction 25
  parts 25, 83
performance 18
  performance 18
person 44, 45
  person 44, 45
person model 44
  person model 44
personal 1
  interesting phenomenon 1
  metamorphosis 1
  personal 1
power tools 5
  database access 17
  power tools 5
practicing 89
  practicing 89
predefined parts 18
  predefined parts 18
primitive part 78, 79
  primitive part 78
primitive parts 28
  primitive parts 28
production development 25
  production development 25
professional programming 17
  application development 17
Supporting
  supporting components 107
  supporting components 107

T
  table control 36
  table control 36
  team programming 17
  team programming 17
  tearing off 65
  tearing off 65
  testing 53
  testing 53
  text-entry field 51, 62, 77
  addressview 62
  set selectors 77
  smalltalk 77
  subparts 62
  text-entry field 51, 62, 77
  text-entry fields 53, 60
  connections 53
  text-entry fields 53, 60
  the information system department 4
  traditional languages 88
  operator 88
  traditional languages 88
  transaction processing 5
  transaction processing 5
  transcript window 89
  context menu 89
  transcript window 89
  transparent access 2, 3
  transparent access 2, 3
  types of parts 28
  types of parts 28

U
  undo and redo 40
  undo and redo 40
  user interface 30
  user interface 30
  user-friendly interface 12
  user-friendly interface 12
  user-oriented applications 5
  user-oriented applications 5

V
  variable attributes 98
  addressbooklistview 98
  attribute 99
  variable attributes 98
  various protocols 109
  ECI 109
  netbios 109
  TCP/IP 109
  various protocols 109
  view separation 39
  business logic 39
  model 39
  view 39
  view separation 39
  visual 7
  visual programming 7
  visual language 34
  visual language 34
  visual notes 42
  address book 42
  address view 43
  design processes 42
  visual notes 42
  visual part 75
  terminology 76
  visual part 75
  visual parts 28, 31, 32, 66
  addressbook 66
  visual parts 28, 31, 66
  visual programming 5, 15, 25
  graphical 15
  icons 15
  metaphors 15
  technology 5
  visual programming 5, 15
  visual subparts 61
  visual subparts 61
  Visual Tools 50
  addressview part 50
  visual programming 25
Visual Tools (continued)
visual tools  50
visuals  30

W
workstation application  3
  interfaces  3
  intuitive  3
  user-centered  3
  workstation application  3
workstations  3
  workstations  3
writing scripts  101
  writing scripts  101
Printed in U.S.A.