Series: Monografias em Ciência da Computação,
No. 17/91

# CONSTRUCTING COMPOSITE INTERACTIVE DOCUMENTS FROM INTERACTIVE COMPONENTS

D.D Cowan
Roberto Ierusalimschy
T.M. Stepien

Departamento de Informática

**In charge of publications:**

Rosane Teles Lins Castilho
Assessoria de Biblioteca, Documentação e Informação
PUC Rio - Departamento de Informática
Rua Marquês de São Vicente, 225 - Gávea
22453 - Rio de Janeiro, RJ
Brasil

Tel.:(021)529-9386    Telex:31078    Fax:(021)511-5645
E-mail:rosane@inf.puc-rio.br

**Abstract:**

Integration of interactive applications is usually done by cut-and-paste methods, importing documents from one application into another. This paper proposes a new concept, called "hole", to achieve this integration, through reuse of components of interactive applications. A hole is a part of a document which is managed by an external program. Each hole has an associated script, that coordinates the behavior of the hole and its connected application program. The paper also discusses how programs should be structured to facilitate this integration, and shows some examples of hole-scripts to solve some typical integration problems. A prototype of a text editor supporting holes, written in Smalltalk, is currently running in our laboratory.

**Keywords:**

Programming environments, script languages, windowing systems, interactive applications.

**Resumo:**

Usualmente a integração de programas interativos é feita por métodos "cut-and-paste", importando em uma aplicação documentos gerados por outros programas. Este artigo propõe um novo conceito, chamado "hole" (buraco), para alcançar esta integração através da reutilização de componentes de programas interativos. Um buraco é uma parte de um documento que é gerenciado por um programa externo. A cada buraco está associado um roteiro que coordena o comportamento do buraco e o programa a ele conectado. O artigo também discute como programas devem ser estrururados para facilitar esta integração e mostra exemplos de roteiros para resolver alguns problemas típicos em integração de programas interativos. Um protótipo de um editor de textos com suporte a buracos, escrito em Smalltalk, está funcionando em nosso laboratório.

**Palavras-chave:**

Ambientes de programação, linguagens de controle, sistemas de janelas.

# Constructing Composite Interactive Documents from Interactive Components

D.D. Cowan R. Ierusalimschy T.M. Stepien *

August 30, 1991

**Abstract**

Integration of interactive applications is usually done by cut-and-paste methods, importing documents from one application into another. This paper proposes a new concept, called *hole*, to achieve this integration, through reuse of components of interactive applications. A hole is a part of a document which is managed by an external program. Each hole has an associated script, that coordinates the behavior of the hole and its connected application program.

The paper also discusses how programs should be structured to facilitate this integration, and shows some examples of hole-scripts to solve some typical integration problems. A prototype of a text editor supporting holes, written in Smalltalk, is currently running in our laboratory.

Categories and Subject Descriptors: D.2.2 [**Software**]: Software Engineering – *Tools and Techniques*; D.2.6 [**Software**]: Software Engineering – *Programming Environments*

General Terms: Programming, Interactive Applications

Additional Key Words and Phrases: Script Languages, Windowing Systems

## 1 Introduction

The current generation of documents such as magazines, technical articles, and corporate annual reports are often composed from a combination of text, pictures, drawings, database reports, spreadsheets and charts using various software packages in which the components of a document are glued together by copy-and-paste methods. Although these packages have made it possible for many non-experts to produce quality documents, there are still many gaps in computer-based document technology.

For example, consider a report which is being produced weekly with a spreadsheet reflecting sales figures and some charts comparing figures with the same period in the previous year. Typically the report is produced using a word processor with the spreadsheet and chart copied and pasted

*D.D. Cowan, R. Ierusalimschy and T.M. Stepien are with the Computer Science Department and Computer Systems Group, all at the University of Waterloo, Waterloo Ontario, Canada. R. Ierusalimschy holds a postdoctoral fellowship from the CNPq (the Brazilian Research Council) and is currently on leave from the Departmento de Informática, Pontifícia Universidade Católica do Rio de Janeiro, Brazil.

from the original using a clipboard. When the report is prepared the following week the new figures are placed in the spreadsheet and then the copy and paste procedure is repeated. If the copy-and-paste step is forgotten[1], then the report is circulated with last week's figures, a very embarrassing situation. The problem, of course, is that the various components which form the report are static copies of the original data and graphics and have lost their relationship to the source package and corresponding data files.

One solution to this dilemma would be to establish a link between the main application and the other packages. This link could invoke the other applications but there would still be some problems with the efficiency of frequent copy-and-paste operations. Moreover, to make a minor modification to the imported document one must go through the appropriate package, as the image of the existing window application inside the document is passive.

In this paper, we propose a concept called *hole* to deal with these problems. A hole is an area in the document that is managed by an external program. So, in our example, one could open a hole in the document where the spreadsheet or chart is to appear, then connect the hole to the appropriate application package. All output from the package will appear in the hole and will also be in the printed document; also, all input performed in the hole will be directed to the attached package. When the report is saved and recalled, the links to the data and the software package will be retained. Thus the spreadsheet and charts in the report will change as the data is updated. In this paper we call such a document a *composite interactive document*; similar types of documents are described in [Hel91].

Composite interactive documents may be used in other applications since a hole in a document may be connected to many different facilities including animation software, and video and sound systems containing video-discs and CD players. Thus the document on the computer could be dynamic and a printed version of the document would represent its state at a given moment in time.

In Section 2 we introduce some application examples to motivate the need for appropriate models and programming technology, and to describe many of the possible interactions which can occur between components of composite interactive documents. Based on these examples we propose in Section 3 and 4 a general model for application software which should allow construction of composite interactive documents. Some existing packages can also be made to work within this framework. In Section 5 we demonstrate tools and techniques for composite interactive documents by producing partial solutions to some of the examples presented in Section 2. Implementations of interactive applications using the model described in Sections 3 and 4 and the tools and techniques in Section 5 exist as prototypes in our research laboratory.

## 2 Motivation by Example

In this section we introduce four examples of composite interactive documents in order to motivate the need for both an appropriate model of interactive applications, and constructive programming tools and techniques. These examples further illustrate our view of the form of a composite interactive document and the dynamic relationships which are likely to exist between the various

[1] A situation which can so easily happen in the rush to produce the report each week.

components.

All the components of composite interactive documents used in the examples are supposed to retain their connection to the entire document even when scrolling or saving for future access. If a document is capable of being scrolled then each component inside this document should stay fixed in position relative to the document's contents. When a composite interactive document is saved or closed its components are saved and can be opened again. Printing a composite interactive document produces a static replica of the document and its components at the time of printing.

## 2.1 Example 1

Consider a simple composite interactive document which requires a drawing in the middle of some text. We open a window of a draw program inside the text area and proceed to produce a drawing using the usual draw commands. The window containing the text document and the text document are called the *enclosing window* and the *enclosing document* respectively, while similarly the drawing window and the drawing are called the *enclosed window* and *enclosed document.* Of course the definition applies recursively; an enclosed document can become an enclosing document, since it also can contain an enclosed window. The enclosed window can be manipulated in the usual way within the enclosing document, that is it can be opened, closed, moved and resized.

In this example the enclosed document is using the user interface of the draw package. The enclosing and enclosed documents are independent, they do *not* need to communicate anything about their contents to each other; the drawing does not affect the text and the text does not affect the drawing. However the enclosing document and the enclosed window must communicate, as both can be affected by commands to open, close, resize, move or print the enclosed window.

## 2.2 Example 2

Consider a document where a table representing the result of an SQL database query is to be placed in an enclosed window in a location within the enclosing document. We open an SQL query window on the screen (not in the enclosing document) in which we type the query. We open an enclosed window for the result in the document.

This example requires that both an input and output window be opened for the SQL query where the output window is an enclosed window. The output from the SQL query must be redirected to the enclosed window.

## 2.3 Example 3

In this example the enclosing document is a spreadsheet with one cell containing an SQL database query which will produce a single numerical result. When this number is computed by an SQL server and placed in the spreadsheet data structure, there could be a recalculation of the contents of several other cells in the spreadsheet. The cell containing the query is both the enclosed window and enclosed document.

When the cell with the query is recognized as containing a foreign element it is passed to the appropriate application for computation. Upon completion of the calculation the results of the query are returned to the spreadsheet data structure. The interaction is strictly between the SQL server and the spreadsheet data structure and program; the SQL server does not link to the

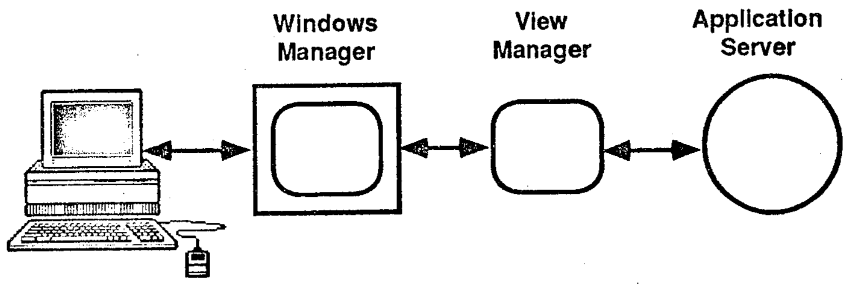

Figure 1: A Model for Composite Interactive Documents

spreadsheet user interface. The solution to this problem is already solved in several commercial software systems.

### 2.4 Example 4

We are typing a document with a chart representing the result of an SQL database query. We open an SQL query window on the screen (not in the document) in which we type the query, and connect the result to the chart package. An enclosed window is opened in the document for the chart. This example is a more complex version of Example 2.

## 3 A Model to Support Composite Interactive Documents

The examples described in the previous section are composed of a number of interactive applications or components working together to produce a composite interactive document. In order to produce a composite interactive document using interactive components it is necessary to have access to not only the internal functions of the application (operations over its abstract data types), but also its user interface. In this section we propose how an interactive component should be structured to provide this access.

In this model, a single interactive component is factored into three elements called the *windows manager*, the *view manager*, and the *application server;* Figure 1 illustrates these modules and the communication paths between them. Usually the interactive component is composed of the view manager and the application server, while the windows manager is part of the underlying system. One windows manager serves many composite interactive documents and each window can have a view manager and an application server.

The model we propose here is similar to the one adopted by Smalltalk/V Windows 3.0 [Dig91][2]. However, the rationale behind our design is somewhat different. The main goal of the Smalltalk model is to allow reuse from an internal point of view, using inheritance to build new applications, while our goal is external reuse, viewing the modules as closed components.

[2]In fact the Smalltalk model has one more module, called Windows, that interfaces the Smalltalk environment with the windows manager.

## 3.1 The Application Server

The application server is the element of an interactive application which manipulates the underlying data structures to perform a specific task. It can be viewed as an abstract data type encompassing all internal data manipulated by the program. The application server should be cleanly separated from the part of the interactive component which interacts with the user by accepting input and presenting output.

A well structured interactive application should export all the functionality of its application server for external access. This functionality can be provided by means of a single client-server model, with the server accepting requests in some predefined syntax.

For example, a spreadsheet server should supply such operations as opening and closing a file, modifying or returning the value of a cell, and recalculation of the values of cells. The server should not include operations related to the presentation of a document such as changing the width of a column or formatting the display of numbers.

## 3.2 View Manager

The view manager manages the interaction between the user and the application server. It is responsible for the presentation of the application data in a human-readable form, and the translation of user commands provided through pointing devices and the keyboard into commands to the application server.

The more interactive and user-oriented a program, the more important is the view manager in relation to the server. For some programs, in fact, the server component is almost empty because almost all the functionality is in the view manager. A typical example is the PaintBrush in Microsoft Windows [Pet90], which is basically an interface between the user and the graphical routines of the underlying system.

Unlike the application server, a view manager does not have a direct way to export its services to other programs, because these services are user-oriented. However, one can reuse a view manager as a separate component as long as it remains user-oriented, in other words the view manager maintains its connection to a window. The key to reuse is to structure the view manager so that it can open and operate its window within another window, instead of always opening an independent window in which to run. Operating a window within another one is not difficult to achieve, but some programs can lose some functionality. For example, pull-down menus may be located in the top window and may not be accessible to the user, when the view manager operates inside an external enclosing window. These problems can be solved by appropriate user-interface design, such as the use of context-sensitive pop-up menus.

## 3.3 Windows Manager

The windows manager provides basic window facilities, and is usually part of the underlying system. Examples are Microsoft Windows 3.0 [Pet90] and the X Server (plus Xlib) in the X Window System [Nye90]. In our model the windows manager must provide a subwindow facility, where each subwindow may have independent user interaction. Such interaction independence implies that the subwindow receives its own messages from the pointing devices and the keyboard, and creates its contents without interference from the enclosing window. Both Windows 3.0 and the X Window System have this feature.

### 3.4 Communication Paths

The arrows connecting the interactive components in Figure 1 represent communication paths. All input from the computer system is first sent to the windows manager which determines whether to act on the input or pass it on to the view manager or application server.

Opening, closing, resizing, or moving a window is handled by the windows manager, but actions such as resizing could cause messages to be passed to the view manager which would modify the contents of the window. As soon as the pointing device moves inside a window then many of the messages produced by actions of the input devices are passed to the view manager, since these actions are causing manipulation of material directly related to the application. Actions in the window may cause changes to the application data structures as well as the contents of the window, and so there should be a connection from the view manager to the application server.

## 4 The Hole Concept - A Window Structuring Method

The model presented in the previous section indicates that there may be substantial communication between the windows manager, the view manager and the application server. This communication is associated with each enclosed window since an enclosed window represents an instance of an interactive application. In order to coordinate the communication and subsequent actions to be performed there should be a program attached to the window which is created dynamically as the components for a composite interactive document are being acquired. The program will be written in an event-driven scripting language to be consistent with the event-driven nature of interactive applications. The window plus its attached script is called a *hole*, since it is part of a composite interactive document which must be "filled" from another interactive application. The script associated with a hole is called a *hole-script.*

The main program, which owns the enclosing document and is managing the entire composite interactive document, must be able to support the concept of a hole, but should not be responsible for supporting all the possible links between each hole and its accompanying interactive components. Supporting all possible links directly from the main program would require that the application programmer have advance knowledge of all types of interactions so that the appropriate links could be included in every application program which could support composite interactive documents. Such a requirement is almost impossible to satisfy. However the link that is included must be flexible enough to allow the connection with many different types of programs, and extensible so as to support new programs and interactions. The solution is the use of a general-purpose script language, then the main program only needs to support one kind of link, a standard one with a hole-script. The hole-script then interacts with its own interactive components.

The main program must create the hole and an accompanying template hole-script, open and close the hole in the proper place inside the enclosing document, and be able to move (mainly for scrolling) and resize the hole. Other kinds of information, when needed, can be communicated between the main program and various hole-scripts using the conventional message-passing channels supplied in window systems. The template hole-script which is generated initially to accompany a hole can be modified through either a user dialogue or by editing. The template contains a program which will initially invoke the dialogue or an editor for writing script programs.

After creating the hole and its accompanying skeletal hole-script, communication between the main program and the hole-script can be composed of the following message extensions (and if necessary the usual message-passing mechanism):

- open (parentWindow, initialSize) - After receiving this message, a hole-script must establish a connection with the external program, and create the sub-window to contain the information for the hole. This sub-window can be created directly by the script, or the script can instruct the external program to do it. The parameter "parentWindow" indicates the enclosing window of this sub-window, and initialSize is a structure containing the initial dimensions of the sub-window and its position.

- move (newPosition) - This message indicates that the sub-window must be shown in the newPosition inside the parentWindow. We keep this message independent of resize in order to have an efficient implementation of this operation (this efficiency is needed to allow holes to scroll inside a window along with the document).

- resize (newSize) - This message indicates that the hole must be resized, usually because the parentWindow has been resized.

- close () - This message indicates that the main program is quitting, or closing the current document, so the script can close its connection with the external program(s) and destroy the hole sub-window (again, this can be done by the script itself or by the external program which created that window).

The interaction between a hole-script and an external program does not necessarily have a regular structure. In fact, sometimes a single hole-script can be connected with more than one program, as in Section 2.4[3].

For simple cases, as in Section 2.1, the hole-script can be created semi-automatically, by an appropriate tool that asks the user for the relevant parameters, such as position and size of the hole (this can be supplied in a direct way, using a pointing device), and the external program to manipulate the sub-window's contents. In these situations, the user does not even need to know the script language. Producing tools to create scripts for more advanced applications such as in Section 2.4 for a user with minimal or no knowledge of the script language is a challenging problem.

## 5 Application of the Hole Concept to the Examples

In this section we use holes to provide outline implementations for the examples presented in Section 2. We use an object-oriented style for the script language, but without a commitment to a specific language. More important than the language itself are the available libraries for window management and inter-program communication, and we indicate some of the requirements for such libraries.

[3]This interaction is based on a message-passing facility, expanded with the features shown in Section 3.

## 5.1 Example 1

The first example has a trivial solution if we assume that the Draw program follows the guidelines of the previous sections. More specifically, we want the Draw package to be able, upon request, to open its window inside another one[4]. With this option, the script code could be as follows:

```
Var window: WindowHandle ;
    drawProgram:  Link ;
    documentName: String ;

Procedure Open (parent: WindowHandle; size: Rectangle) ;
  drawProgram := StartChannel ("DrawProgram") ;
  drawProgram.Open (documentName) ;
  window := drawProgram.OpenWindow (parent) ;
  window.Size (size) ;
End.

Procedure Move (newPosition: Point) ;
  window.Move (newPosition) ;
End.

Procedure Resize (newSize: Rectangle) ;
  window.Resize (newSize) ;
End.

Procedure Close () ;
  window.Close () ;
  drawProgram.CloseChannel () ;
End.
```

The StartChannel command is supposed to open a communication channel between the script and the named program ("DrawProgram"). Then we command the draw program to open a document, and then open a window inside the specified window. The document name is stored in a string initialized when the script is created.

## 5.2 Example 2

This example is also simple, but this time the script must create its own window to show the results. It also can create a warm-link[5] from the SQL-server (if this feature is supported) to update the results shown in the window automatically.

[4] If the package does not support this option the script can open it in the standard way, and then attach the draw window to the appropriate parent.

[5] A warm-link notifies a named program that a change has occurred, but it does not institute the change.

```
Var window: WindowHandle ;
    SQLProgram: Link ;
    query: String ;

Procedure Open (parent: WindowHandle; size: Rectangle) ;
  SQLProgram := StartChannel ("SQLServer") ;
  window := TextWindow.new (parent) ;
  window.Size (size) ;
  window.Show () ;
  SQLProgram.WarmLink (query, UpDate) ;
  self.UpDate () ;
End.

Procedure UpDate () ;
  Var result: Table;
  result := SQLProgram.Query (query) ;
  window.Erase () ;
  window.Print (result) ;
End.
```

In this example, the SQL program is used as a conventional server, and so the script is responsible for creating the hole; this task is performed by the *new* operation acting on the object TextWindow which is supplied from an appropriate library. The WarmLink operation asks the SQL program to call the script operation UpDate every time the result of the query changes so that the screen can be changed. The query itself is stored as a string, initialized on script creation, although it could just as easily have been typed in a prompt window. The UpDate procedure asks the server to evaluate the query, and shows the result in the appropriate window.

The procedures to move, resize, and close are similar to the previous example.

## 5.3 Example 3

This example is different in that the hole does not need a window, although a hole-script and a phantom window could still be used. In fact, there is no need for the hole mechanism to solve this problem, but the example is worth examining to show the versatility of the mechanism.

```
Var SQLProgram: Link ;
    SelfProgram: Link ;
    query: String ;
    cellPosition: Point ;

Procedure Open (parent: WindowHandle; size: Rectangle) ;
  SQLProgram := StartChannel ("SQLServer") ;
  SelfProgram := StartChannel (parent.Owner()) ;
  SQLProgram.WarmLink (query, UpDate) ;
  self.UpDate () ;
End.

Procedure UpDate () ;
  Var result: Number ;
  result := SQLProgram.Query (query) ;
  SelfProgram.Store (result, cellPosition) ;
End.
```

The procedure Open must start two channels; one with the SQL server and other with the spreadsheet program, which is the owner of the parent window. Then it calls UpDate, which is also called by the warm-link whenever needed, to put the query result into the appropriate cell. Again, the variables query and cellPosition are defined when the script is created.

### 5.4 Example 4

This example is a mix of the previous ones. The script must open channels with both the SQL server and the Chart program, and maintain the communication between them.

```
Var window: WindowHandle ;
    ChartProgram: Link ;
    SQLprogram: Link ;
    query: String ;

Procedure Open (parent: WindowHandle; size: Rectangle) ;
  SQLProgram := StartChannel ("SQLServer") ;
  ChartProgram := StartChannel ("DrawProgram") ;
  window := ChartProgram.OpenWindow (parent) ;
  window.Size (size) ;
  self.UpDate () ;
End.

Procedure UpDate () ;
  Var result: Table;
  result := SQLProgram.Query (query) ;
  ChartProgram.DrawChart (result) ;
End.
```

## 6 Conclusions

Constructing composite interactive documents from interactive components is currently possible in restricted ways using existing commercially available software tools. However there are still many gaps in computer-based document technology.

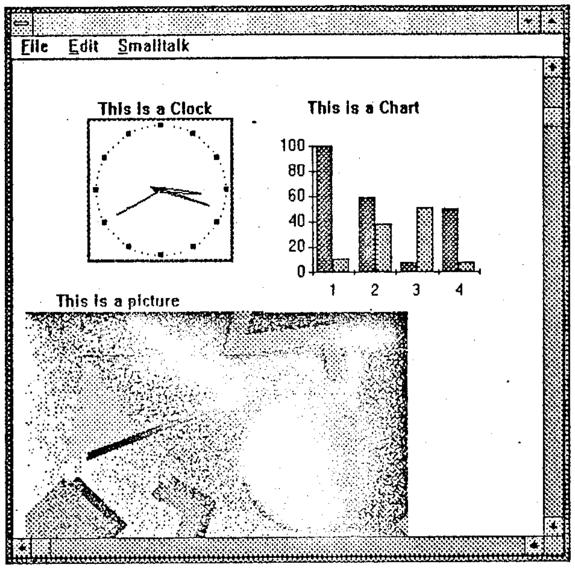

Figure 2: A Sample Composite Interactive Documents

Many systems attempt to define standards for inter-application communications. These include Microsoft OLE [Hel91], Hewlett Packard NewWave [Pet89], Sun Tooltalk [Ull91] and Apple Computer's Apple events [Inc91]. Each of these systems provides the underlying technology to enable the construction of some kind of composite interactive documents from interactive components. Each of these systems requires that applications be specifically written to use the particular vendor's package.

In this paper, we describe a software architecture for building interactive applications which can be configured together to build composite interactive documents. The separation of the Application Server from the View Manager and Windows Manager creates a software environment which permits reuse and reconfiguration in the future. This software design model is independent of the specific windowing system.

The concept of holes and their associated hole-script describe a flexible interface between interactive applications. Holes define a region within the document where the enclosed application displays output. The enclosing document communicates through the hole-script to the external application to perform operations such as move, resize and printing. The hole-script defines a uniform exposed programming interface which may be tailored for specific composite interactive documents.

Prototype systems which demonstrate the software architecture for interactive applications and the concept of holes and hole-scripts have been implemented and tested. The prototype was implemented in Smalltalk [GR83] using the Smalltalk/V Windows system. The hole-scripting language is currently Smalltalk. We intend to use a more common scripting language such as REXX [Cow90, Gig91] in future versions.

A sample composite interactive document created with our prototype is illustrated in Figure 2. In this example the enclosing window is a text editor. There are three enclosed windows; a clock, a chart and a bitmap picture. The clock is an unmodified version of the standard Microsoft Windows 3.0 application which has been opened in the document. The clock application operates in real time with the clock linked dynamically to a Microsoft Excel [Cor91] spreadsheet. The Microsoft Excel spreadsheet receives the input data from the clock and updates the bar chart which appears in the document. The bitmap picture is a hole controlled by Microsoft Paintbrush and can be modified directly in the document. The entire composite interactive document operates as a unit and can be scrolled, opened and closed.

The use of application linking and holes presents problems if any of the linked components are deleted or moved, particularly over a network. Problems of this type need to be investigated in the context of new types of file and database systems.

## References

[Cor91] Microsoft Corporation. *Microsoft Excel User's Guide Version 3.0.* Microsoft Corporation, 1991.

[Cow90] M. F. Cowlishaw. *The REXX Language: A Practical Approach to Programming.* Prentice-Hall, 2nd edition, 1990.

[Dig91] Digitalk. *Smalltalk/V Windows - Tutorial and Programming Handbook.* Digitalk, 1991.

[Gig91] Eric Giguère. *Amiga Programmer's Guide to ARexx.* Commodore-Amiga, Inc., 1991.

[GR83] Adele Goldberg and D. Robson. *SmallTalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[Hel91] Martin Heller. Future Documents. *Byte*, 16(5):126–135, May 1991.

[Inc91] Apple Computer Inc. *Inside Macintosh, Volume VI.* Apple Computer Inc., 1991.

[Nye90] Adrian Nye. *Xlib Reference Manual.* O'Reilly & Associates, 1990.

[Pet89] Charles Petzold. First Looks. *PC Magazine*, 8(17):33–35, October 1989.

[Pet90] Charles Petzold. *Programming Windows.* Microsoft Press, 2nd edition, 1990.

[Ull91] Ellen Ullman. Sun Aims to Get Unix Applications Talking to Each Other. *Byte*, 16(9):30–31, September 1991.