SUPPORT FOR OPEN TOOLS AND SYSTEMS

Luís Carriço, Pedro Antunes, Nuno Guimarães,
Paula Pereira, Maria Moreno

IST/INESC,
R. Alves Redol, 9, 6º., 1000 Lisboa, Portugal

e-mail: {lmc,paa,nmg}@inesc.pt

Abstract: The construction of interactive tools for current computing environments relies upon a generic set of characteristics that include an architecture for interaction, a programming methodology based on the object oriented paradigm, and support for rapid prototyping and experimental programming. This paper describes foundational components that support the construction of these tools. Two different tools, an application builder and an authoring system, are presented as examples built on those components.

1 Introduction

Interactiveness is a pervasive characteristic of current tools and applications. Multiple devices and media, heterogeneous interaction environments, and more demanding user requirements, increase the complexity of the tools, the cost of their construction and maintenance, the difficulty in keeping openness[1] and a satisfactory degree of reusability. It is therefore essential to base our work, as tools developers, in a clear framework where the requirements can be functionally partitioned, and addressed separately by complementary developments.

This paper presents a framework that addresses the construction of interactive applications and tools in a three axes referential: application architecture, object oriented programming and rapid prototyping. According to this approach, we considered the following development and research directions:

- The definition of a comprehensive architecture handling all the requirements that may be raised in the development of interactive tools and systems. These requirements are related with both the internal structure of the user interface/application, concerning active components and communication between components, and the use of external services like graphical systems, storage systems, communication facilities, and handling of multimedia information.

- The intensive use of object oriented programming methodologies and techniques, including both the facilities provided by the languages themselves, and the notions that derive from current and future results of the work on object oriented analysis and design.

- The creation of a support for interpretation and

[1] We understand openness as the ability to interconnect software modules from different vendors, in heterogeneous platforms, with a significant degree of customisation and tailorability.

G. Heeg, B. Magnusson, and B. Meyer, Eds. Proceedings of the Seventh International Conference on Technology of Object-Oriented Languages and Systems, TOOLS Europe '92. Dortmund, Germany: Prentice Hall, Englewood Cliffs, 1992, pp. 301-310. (ISBN: 0-13-917436-2).

rapid prototyping, that underlies any interactive tool.

In this paper, we describe what we consider to be the essential support for interactive tools, in an open environment. The next section proposes an extensible architecture that covers most aspects that tools and applications are concerned with. The following section describes a runtime system for C++ [Ellis 90] that supports our interactive programming needs. Then a section is dedicated to tools that were built using these two main developments. The first tool is the INGRID application builder. We just describe how the facilities of the support components have been used. More detailed description of the tool is given in [Carriço 90,Guimaraes 91a,Guimaraes 91b]. The second tool is a hypertext system [Conklin 87], that mimics part of the HyperCard [Harvey 88] functionality in the Unix[2]/X[3] environment. The most interesting aspect of this tool, HypIngrid, is that it evolved from the INGRID tool with a minimal effort, thus proving the concept that the underlying support for interactive programming and application construction is generic and reusable. We finish with an overview of future directions, as well as a set of conclusions.

2 Architecture for Interaction

The definition of an architecture for interactive applications should provide the structural guidelines for its construction according to the following premises:

- a clear separation between the computational and interactive parts of an application (*dialogue independence* as defined in [Hartson 89]);
- a well defined, and complete, functional partition within the interactive component;
- a methodological approach to the composition of those functional parts, for organising simple components into higher level abstractions.

[2]Unix is a trademark of Unix Systems Laboratories

[3]X Window System is a trademark of the Massachussets Institute of Technology

Naturally, the use of an object-oriented approach contributes to the clarity of the above issues, providing the basic structuring mechanism for the design of the application. Moreover, considering the effort already spent in existing toolkits and libraries, the architecture should promote an easy integration of those elements, in its structural and behavioural aspects.

Functional partition

According to the above principles, we defined the 4D architecture. This architecture classifies the objects composing an interactive application into four possible categories as shown in fig. 1:

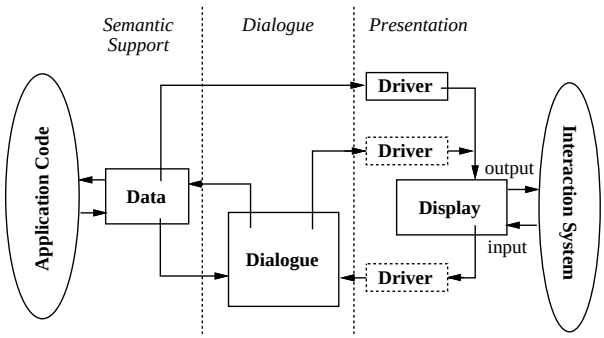

Figure 1: The 4D functional architecture

Display: This category is the interface to the interaction system, usually a graphical system or a set of multimedia devices, and corresponds to the presentation component of the UI (according to the models commonly accepted [Myers 89]). One of the roles of Display objects is to integrate existing UI components, such as *Xt* widgets [Young 89,Young 90], adapting them to the global 4D architecture.

Data: The main role of these objects is to provide the interface between the interactive and computational parts of the application. In general this category represents abstract data types that range from simple ones (like Integers, Floats or Lists), to full database access, interprocess communication, or even types addressing distribution [Antunes 91]. Data ob-

jects are another locus for integration of existing class libraries (like libc++, libg++ or NIHCL [Gorlen 90]).

Dialogue: Dialogue objects define the control of the interaction. As they integrate with the other components, they propose the definition of event driven dialogues. However, other dialogue models can be easily supported.

Driver: Elements of this category manage the transfer of information between objects. Their role is basically to perform data conversion, optimising the integration of the other components. Examples of drivers are objects that convert toolkit specific data (e.g. Xt values) to or from abstract data (e.g. integers dialogue tokens, strings, collections ...).

Models like PAC [Coutaz 87,Coutaz 90] and MVC [Goldberg 83a,Dodani 89] define a similar global functional partition for interactive applications. However, MVC forces a separation between user input (*Controller*) and display output (*View*) not imposed in 4D. On the other hand, neither model considers the role of drivers as an independent component of the application.

Dynamics of the architecture

The 4D architecture introduces a fifth concept addressing the dynamics of the application. It defines a single communication paradigm between components:

Link: A selective communication channel that carries messages between objects. Each *link* is characterised by a *sender* object, an *event identification* on that sender, a *receiver* and an *action* to be performed on the receiver.

When an event is triggered in an object, its links (for which the object is the sender) are scanned and, for those associated to that event, the corresponding actions are executed. The execution of an action corresponds to a message sent to the receiver. On the reception of a message, the new object may trigger its own events, repeating the process. On display objects, events are usually triggered as a consequence of a user action. On data objects, events may be triggered by application specific code or by the interaction component, but always through messages.

The link, specially through its action component, is therefore a programmable entity that has proved to increase the degree of flexibility and reusability of the different components of the application. In fact, since actions may specify any message, including or not the sender's internal data, each component can work autonomously, without explicit knowledge of the other objects.

The 4D architecture defines a set of rules for object interconnection restricting the establishment of links, as shown in fig. 1. Basically, Data and Display objects are not to be linked together, and in their links, the action specification is disabled. This standardises the connections made out of these objects, which concentrates the definition of the specific behaviour of each interface in the Dialogue and Driver components.

The definition of restrictions for component communication is also present in the related models referred above (PAC and MVC). Nevertheless, the existence of a specific intercommunication component, defining a selective and programmable communication channel is hardly found. Even MVC, which defines the Model component as an active value, allowing objects depending on it to be notified when a change occurs, uses a quite rigid protocol, where the receptor is responsible to decode all the possible notifications.

Methodology for composition

In the process of construction of interactive applications, a methodology for organising components is a main requirement. In fact, as interfaces become more complex and domain specific, the existence of higher level abstractions, composed of simpler elements, eases the programmer's task.

Taking advantage of the adopted OO approach and the definition of the above functional architecture, the 4D architecture defines a methodology for object composition. This methodology is special-

ly applicable when objects from different categories (sub-parts) are gathered in a composite object (encapsulator) which manages their behaviour, filtering the incoming messages and the definition of outgoing links. Sub-parts, in this case, are usually not visible outside the encapsulator.

Apart from the abstract principles that guide object classification into each 4D category, encapsulation must obey the following rules:

1. Links emerging directly from the sub-parts to objects outside the encapsulator follow the restrictions applied to both objects (sub-part and encapsulator).

2. Incoming links can never specify actions directly in sub-part objects. Instead, the encapsulator object may extend its interface and delegate some of the incoming messages to its components.

3. Display objects can only be composed into an encapsulator display object. Since only display objects can communicate with the interaction system, it makes no sense to include objects of this category in others.

The definition of a methodology for composition includes the 4D architecture within the so called hierarchical models like PAC (and unlike MVC). It differs from PAC since in this one compositions are restricted to the Control (Dialogue in 4D) component, whereas in our architecture composition is actually centered in (but nor restricted to) the Display component (Presentation in PAC).

3 Run-time support

The implementation of the above architecture requires a run-time support for message exchange between objects. In fact, the concept of link, and specially its action part, can be easily mapped into a general run-time invocation mechanism. The use of 4D objects in interactive applications, and in tools supporting interactive programming, stresses this requirement. We identified the following services, to be provided at run-time:

- a support for interpretation allowing creation, customisation and identification of objects. These mechanisms provide a basis for experimental programming [Sheil 86] and rapid prototyping. Also, they should offer the flexibility required for the implementation of the link concept.

- a mechanism for object storage and retrieval that supports application saving and recovering throughout the programming process.

- the ability to provide information about the structure of objects and, in general, their programming interface. This information can be used for guidance purposes in interactive programming tools, and may also provide support for the other services.

The language level support

The choice of a language like C++ hardly matches the above requirements. However, we chose it for the sake of the openness and portability it provides. According to this perspective, ICE[4], a run-time support for interpretation and storage/retrieval in C++, has been developed. Its main services are:

Object identification: enables the association of user readable names to objects.

Object creation: provides a primitive for object instantiation, independently of the class it belongs to. Since it bases its behaviour in class constructors, the semantics of C++ object creation are kept.

Message invocation: allows the invocation of member functions through a message like mechanism, using a common generic primitive, maintaining the characteristics of the host language. Namely, it provides the polymorphic constructions of C++ (e.g. member overloading) performing type checking at run-time before method execution.

[4] support for **I**nteractive **C++** **E**nvironments

Object storage/retrieval: allows object passivation and activation, supporting multiple external representations, including C++ itself.

Except for the first service, that was implemented as a set of fast access hash tables (referred to as *name-service*), all the other are based in the existence of *type-objects*. These objects are an extension of the Smalltalk's *class-object* concept [Goldberg 83b], since they describe both classes and C++ primitive types. Their basic role is to provide run-time type information, uniform to all C++ types, enabling an easy implementation of type-identification and type-checking mechanisms.

Class specific type-objects in particular, offer access to objects (*method-objects*) describing constructors and member functions of each class. In cooperation with these, type-objects implement the algorithms for method lookup and provide the services for object creation and message invocation. Performance in these services is not actually a bottleneck. In fact, a "pre-compilation" option at run-time can be performed, leaving a degradation of about 4 times between a message invocation and an C++ member function call.

Information about instance structure is also available, and used in the automation of the storage and retrieval mechanism. This mechanism, however, is implemented in dedicated objects, named *io-objects*, that, based on the instance description obtained from type-objects, save and retrieve instances according to the syntax and storage medium they specify. This way different io-objects can save/retrieve the same instance using different syntaxes and storage sources/sinks (e.g. file, pipe, local memory, ...).

Finally, ICE provides a common interface to most of the above services, as shown in figure 2. This interface can be inherited by other classes deriving from the *IObject* class, or simply through typed-references, represented by *IOID*. This last form will allow easy integration of existing code for which class declarations can not be changed.

Besides the described services, ICE includes a parser for C++ definitions, that generates code for the static instantiation of type-objects and aggregated method-objects. This simplifies the integration of classes in ICE, and avoids the need for the availability of class definition code.

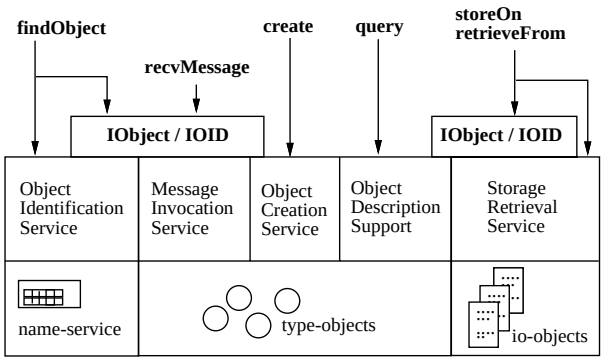

Figure 2: The ICE services

Architecture specific support

Once the interpretation capabilities and storage/retrieval mechanisms were provided, a basic toolkit was developed, establishing the generic functionality defined by the architecture. On this base, more specialised classes were then implemented (mainly integrating existing toolkits), providing the components needed for the construction of interactive applications.

The fundamental behaviour of 4D components are defined on the *4DObj* class. It provides the mechanisms to establish links, verify and maintain lists of links, trigger events, select them and executing the associated actions. The validation of the actions specified for each link, and its execution, rely on the ICE message invocation mechanism provided by the *IObject* base class.

Structural composition of the 4D architecture is obtained by defining four classes, derived from *DObj*: *DataObj*, *DriverObj*, *DisplayObj* and *DialogueObj*. An object belongs to a specific category if its class inherits from one of these classes. Basically, they constrain the establishment of links between components as defined by the architecture.

4 Tools for Application Construction

The 4D architecture and the ICE run-time provided the support for the construction of two tools. The first tool, **INGRID**[5], is an interactive tool for user interface construction, allowing rapid prototyping and incremental development. The second tool, **HypIngrid**, is an open hypertext authoring system that allows the creation of hypertext applications, with a functionality very similar to HyperCard.

Both tools are characterised by its interactive or interpretative nature, as well as by the requirements for representation/interaction components, data objects with persistence attributes, dialogue definition and encapsulation. Overall, they constitute an assessment of the underlying concepts and functionality.

4.1 INGRID

The INGRID tool is composed by a set of subtools that address the construction of the several interface components and their interconnection. The top-level component is the **Interface Organizer**. It allows global access to the interface objects, manages the creation of links, and provides the interface to global functions like save/retrieve, C++ code generation, on-line help, etc. Display objects can be created and parameterised with the **Display Editor** (fig. 3). The editor is divided in two areas: palette and working space. These areas are the interface to the instantiation and customisation operations. Object parameterisation can also be performed through class specific *inspectors*. The instantiation of Data,Driver and Dialogue objects is done through very simple editors, that basically associate a user readable name to each created object.

INGRID and the 4D architecture

Ingrid is organised according to the model defined in the 4D architecture. The definition of four categories, each one grouping objects with similar functionality leads to the existence of four separated editors, specially conceived according to the functional role of each component.

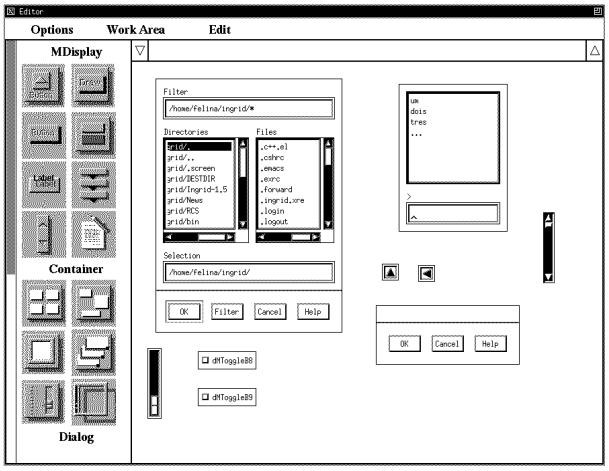

Figure 3: The INGRID Display Editor

The adoption of the 4D model for user interface construction offers several advantages:

- A user interface can be defined by direct manipulation of the available 4D objects. New objects can be easily added and parameterised. Links between the several objects can be established so that they can communicate with each other, according to their functional role.

  The great advantage of the interactive behaviour of 4D objects is that it allows to develop and test the interface without having to compile it. Links between the objects can be set, changed or deleted at run-time, changing the application behaviour.

- Composition allows the definition of new classes that organise simple components and can be easily inserted in the INGRID tool.

- The Store and Retrieve functionality allows the tool to save and recover the UI at any point of its definition. Each object in the 4D toolkit knows how to save and retrieve itself, including links and references that it may have to other objects.

[5] **IN**teractive **GR**aphical **I**nterface **D**esigner

INGRID and ICE

Besides the specific functionality of 4D objects, the ICE facilities were intensively used by INGRID. With the facilities described above, INGRID is able to:

- Know the names of all the classes available in the environment and instantiate new objects by simply giving the class name.

  This allows a new class to be easily added to the tool. As an example, the integration of the Motif widget set was done simply by generating a new Display class for each widget. The new Display classes where generated with auxiliary tools and, once they were added to the ICE environment, they could be used transparently by INGRID.

- Know the names of all the methods defined by a particular class. INGRID uses this run-time information to build specific class *inspectors* for object parameterisation and to send messages to any UI object.

  With this mechanism, the definition of the class *inspectors* is performed with great flexibility and independently of the class details, assuring extensibility of the tool.

- Access any instance of a particular class in the environment. INGRID uses this facility to establish links between UI objects, identify an object's type, check type conformance and conversion, and provide help about instance data and member functions.

4.2 HyperCard on Unix

Once INGRID was available, we observed that it could be easily modified to produce an hypertext authoring system, similar to HyperCard. In fact, the visual characteristics of HyperCard were provided by the INGRID Display editor, the Data objects of the 4D architecture are the components that encapsulate the introduction of persistence, and the HyperCard scripting language could be handled as a dialogue mechanism.

Functionality

HypIngrid supports the stack/card model. The basic node object is the *card*, and a collection of cards is called a *stack*. A stack contains a *background* and several cards. A card may contain *text*, *bitmaps*, *buttons*, and *fields*. Buttons may have links to other stacks/cards or scripts associated to them. A HyperTalk-like programming language was developed in order to write scripts that are associated to buttons. An example of a card created with HypIngrid is shown in fig. 4.

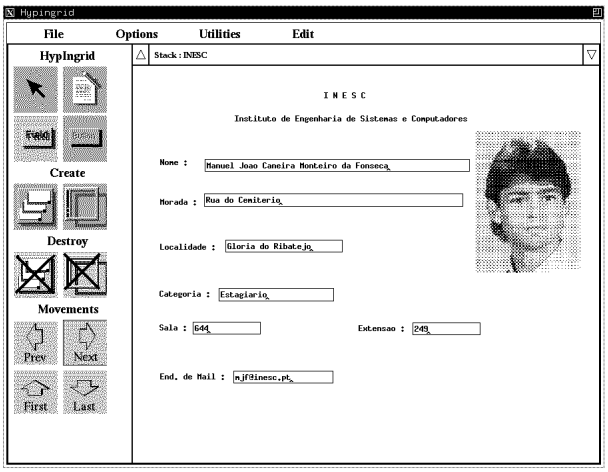

Figure 4: An example of a **HypIngrid** card

The system provides a set of operations on these objects, namely, interactive creation of texts, buttons and fields on a card, creation and destruction of cards and stacks, and also operations of movement. Menu options allow to export/import a stack, edit objects, perform search and sort operations, among others.

Internal Architecture

The architecture of HypIngrid includes three components: storage, management, and representation of information:

- Information is stored using the SOHO[6] system.
- The management system defines HypIngrid abstractions and makes the interface to the rep-

[6] Storage Of Hypermedia Objects

resentation system. It also includes the parsing and interpretation of the scripting language[7].

- The representation system is responsible for the manipulation of the graphical objects defined by HypIngrid.

The development of this system explored the functionality of existing components: The SOHO storage system, the 4D Toolkit, and the INGRID interface builder (fig. 5).

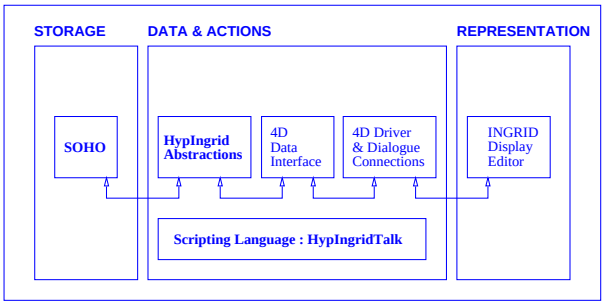

Figure 5: The **HypIngrid** architecture

HypIngrid and SOHO The SOHO system is based on the HAM model [Campbell 88] and its object oriented interface offers a set of hypertext objects: graph, context, node, and link. All of these objects may have attributes attached to them. Internally, SOHO is implemented using the *sdbm* library. HypIngrid defined the *Stack, Background, Card, Field and Button* abstractions using these storage facilities.

The HypIngrid Management Component The HypIngrid abstractions were encapsulated into 4D Data objects, to be incorporated in the overall application architecture.

This component is also responsible for the parsing and execution of HypIngridTalk, the scripting language that is a subset of HyperTalk.

One important aspect to consider is the storage and retrieval of components like Cards. These operations are rather transparent, and rely upon the ICE and 4D functionality. In fact, Cards are saved using the ICE external representation, together with Xt resource files, in SOHO nodes.

The HypIngrid Visual Interface The visual interface of HypIngrid evolved from the INGRID Display Editor. INGRID provides a fairly high degree of independence between its subtools. The adopted strategy was therefore to reuse the Display Editor in HypIngrid.

All the direct manipulation facilities offered by the Display Editor apply now to the creation and manipulation of HypIngrid objects. Some facilities where added for stack import/export operations, script editing, and expedite hypertext linking.

The overall result highlighted several advantages of the approach. First, the openness of the system allows the integration of other tools, like the audio device of the Unix workstation, dedicated editors, for text, drawings or images, or other specialised processes.

Second, the graphical interface is also based on generalised tools (Xt). This provides *look and feel* compatibility with other applications. On the other hand, it is conceivable to extend the HyperCard concepts with other objects like, for example, a Motif *toggle* to be used as a pin while navigating in the hypertext document.

5 Conclusions and Future Directions

The fundamental conclusions we draw from the work described in this paper are the following:

- Interactive applications and tools require a comprehensive and open architecture to support functional partition and integration of available components.

  The comprehensiveness of the architecture facilitates the programming process by providing a uniform object model for all the components, and standardised interconnection mechanisms.

  We believe that the 4D architecture is a good step in this direction.

[7] Referred to as *HypIngridTalk*.

- The construction of interactive tools and systems requires support for interpretation and rapid prototyping. This can be provided by the language and programming environment (Smalltalk [Goldberg 83b], Objective C [Cox 86]) or added through a run-time support system like ICE. Although implying an extra effort, it proved successful to design and implement such a run-time, given that we were able to keep openness and easy integration with external components.

- The facility of evolving the INGRID tool to a hypertext/authoring system shows that there is a common denominator in these family of tools, which is implemented by the joint cooperation of the architecture and the run-time. This fact also suggests that further functionality of the run-time and architecture should be made generic and reusable across different tools and applications.

The future directions can be easily extrapolated from the above conclusions. The promising directions are the extension of the architecture and the extension of the run-time. Our goals are to extend them in the following way:

- The purpose of the run-time is to support interactive programming and rapid-prototyping. Generic support for knowledge acquisition and manipulation is a requirement for adaptive and more intelligent user interfaces [Sullivan 91], which makes it an important extension .

- The architecture can be extended to support distribution. Promising experiences have been made in this direction, [Antunes 91] opening the way to a smooth transition to CSCW [Greif 88] applications.

Acknowledgements

This work was supported partially by the Commission of the European Communities, under the *Comandos* Esprit Project, and partially by JNICT, the Portuguese National Board for Research.

References

[Antunes 91] P. Antunes, N. Guimar aes, and R. Nunes. Extending the User Interface to the Multiuser Environment. European Conference on Computer Supported Collaborative Work, CSCW Developers Workshop, Amsterdam, September 91.

[Campbell 88] B. Campbell and J. Goodman. HAM: A General Purpose Hypertext Abstract Machine. *Communications of ACM*, 31(7):856–861, July 1988.

[Carriço 90] L. Carriço, N. Guimaraes, and P. Antunes. INGRID : A Graphical Tool for User Interface Construction. In *Proceedings of the EUUG Spring Conference, Munich,*, April 1990.

[Conklin 87] J. Conklin. Hypertext: An Introduction and Survey. *IEEE Computer*, pages 17–41, September 1987.

[Coutaz 87] J. Coutaz. PAC : an Implementation Model for Dialog Design. In *Proceedings of INTERACT'87*, pages 431–436, September 1987.

[Coutaz 90] J. Coutaz. Architecture Models for Interactive Software: Failures and Trends. In G. Cockton, editor, *Engineering for Human-Computer Interaction*, pages 137–153. Elsevier Science Publishers B.V, North-Holland, 1990.

[Cox 86] B.J. Cox. *Object-Oriented Programming - An Evolutionary Approach.* Addison-Wesley, 1986.

[Dodani 89] M. Dodani, C. Hughes, and J. Moshell. Separation of powers. *BYTE*, pages 255–262, March 1989.

[Ellis 90] M. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual.* Addison Wesley, 1990.

[Goldberg 83a] A. Goldberg. *Smalltalk-80: The Interactive Programming Environment.* Addison-Wesley, 1983.

[Goldberg 83b] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its implementation.* Addison-Wesley, 1983.

[Gorlen 90] K. Gorlen, S. Orlow, and P. Plexico. *Data Abstraction and Object-Oriented Programming in C++*. John Wiley & Sons, 1990.

[Greif 88] I. Greif. *Computer Supported Collaborative Work :A Book of Readings*. Morgan Kaufman Publishers, 1988.

[Guimaraes 91a] N. Guimaraes. INGRID: Interactive Graphical Interface Designer. Tutorial presented at the 5th Annual X Technical Conference, Boston, January 1991.

[Guimaraes 91b] N. Guimaraes, L. Carriço, and P. Antunes. INGRID : An Object Oriented Interface Builder. In *Proceedings of the TOOLS'91 Conference, Santa Barbara, California*, July 1991.

[Hartson 89] H.Rex Hartson and Deborah Hix. Human-Computer Interface Development:Concepts and Systems for its Management. *ACM Computing Surveys*, 21(1), March 1989.

[Harvey 88] G. Harvey. *Understanding Hypercard*. Alameda, CA : SYBEX Inc., 1988.

[Myers 89] Brad Myers. User Interface Tools: Introduction and Survey. *IEEE Software*, pages 15–23, January 1989.

[Sheil 86] B. A. Sheil. Power Tools for Programmers. In David R. Barstow, Howard E. Shrobe, and Erik Sandewall, editors, *Interactive Programming Environments*, chapter 2, pages 19–30. McGraw-Hill, 1986.

[Sullivan 91] J.W. Sullivan and S.W. Tyler. *Intelligent User Interfaces*. ACM Press, 1991.

[Young 89] D.A. Young. *X Window Systems Programming and Applications with Xt*. Prentice Hall, 1989.

[Young 90] D. Young. *OSF/Motif Reference Guide*. Prentice-Hall, 1990.