Extending the User Interface to the Multiuser Environment

Pedro Antunes, Nuno Guimarães, Ricardo Nunes
IST/INESC, Lisbon, Portugal
e-mail: {nmg,paa,ron}@inesc.pt

1 Introduction

This paper describes our experience in extending a set of tools, designed for user interface construction, to support multiuser execution. The enhancement of the tools was based on an existing user interface architecture. The goal was to keep the programming paradigms as close as possible to the ones of the available environment. The strategy was to enhance a particular category of user interface components through specialization mechanisms (*inheritance*). Two alternatives for this specialization have been evaluated, *shared* and *replicated* objects.

The next section gives a brief outline of the initial system, the single user environment. The enhancement strategy and mechanisms are described next, followed by a discussion and comparison.

2 The single-user environment

Our environment is based on a user interface (UI) architecture, a toolkit (*4D*), and a tool for interactive user interface construction called *Ingrid* [1,2,3]. The object oriented approach is adopted throughout the toolkit and tools, and the implementation uses *C++* [4].

The 4D architecture The *4D* architecture provides structural guidelines for the construction of interactive applications. Globally, this architecture applies a clear separation between the computational and interactive parts of an application, and divides the interactive part (the UI) in several functional categories:

- *Display* objects. This category is the interface to the representation system, usually a graphical system, and corresponds to the presentation component of the user interface (according to UI models commonly accepted [5]). *Display* objects are meant to be implemented with existing UI components, such as *Xt* [6] widgets. Their role is more to encapsulate these components and adapt them to the global architecture than to perform the input/output operations directly.

- *Data* objects. These objects provide the interface between the interactive and computational parts. They provide general purpose data types and structures, and behave as *active values* [7].

- *Dialog* objects define the syntactic structure of the interaction. The *Dialog* category groups dialog control objects. The *4D* architecture does not impose a particular dialog expression mechanism and allows multiple implementations of dialog control: dialog languages, dialog cells, etc.

- *Driver* objects manage the transfer of information between *Data* and *Display* objects. Their role is basically to perform *data conversion*. These objects optimize the integration of available *Data* and *Display* objects.

The communication between objects is performed through a mechanism designated by *link*: a permanent communication channel that carries messages between objects. The main advantage of this connection mechanism results from the fact that an object sending a message does not require information about the receivers (like their address or name, for example) but it just uses the outgoing link. This mechanism increases the degree of flexibility and reusability by providing an autonomous behavior for each object in the toolkit. This feature is even more important when considered in the context of an interactive tool, where objects are often created "on the fly", and later related with others. The creation of links between objects follows the discipline illustrated by the directions of the arrows in Figure 1.

The INGRID tool The construction of an UI is a process of creation and composition of objects belonging to the above four categories and available in the toolkit, and creation of *links* between them. We designed and implemented the *INGRID* tool based on the *4D* architecture to perform this function.

The core of *INGRID* is composed by the *4D* toolkit and a run time support system for interactive programming, called *ICE - Interactive C++ Environment*. This run time system provides a set of services that includes type information, object identification, message communication and object storage and retrieval.

The *INGRID* tool itself can be described as a set of editors, one for each object category, and another for the global UI. Direct manipulation is broadly used.

3 Multiuser extension

Given the *4D* architecture, the *Data* objects were elected as the locus for multiuser extension. The reasons for this decision were:

- *Data* objects handle multiuser communication at the semantic level of the interactive process, minimizing the complexity of interactions and also net-

Papers from the European Conference on Computer Supported Cooperative Work, ECSCW '91, Developers Workshop, held in Amsterdam, September 1991, ACM Special Interest Group on Office Information Systems (SIGOIS) Bulletin, vol. 13, no. 1, April, pp. 5-7, 1992.

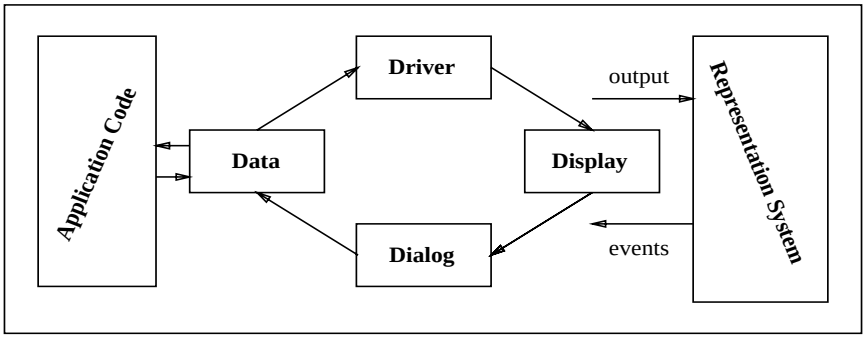

Figure 1: The 4D architecture

work bandwidth requirements. Communication at the lexical level, e.g. *Display* objects, implies event distribution and synchronization and therefore, much higher throughput needs. This decision excludes the "WYSIWIS" approach.

- Communication through *Data* objects allows the use of heterogeneous UI like multimedia devices, or alphanumeric terminals.

- Communication through *Data* objects allows different applications to interoperate. In fact, there is no requirement that the cooperating users be executing the same application. The unit of communication is the object, and not the application.

Once *Data* objects were chosen for multiuser extension, two alternatives were considered:

- Shared *Data* Objects. This solution is based on the notion that only one *Data* is viewed by the multiuser applications. This object communicates with the *Driver* and *Dialog* objects through the standard *4D* protocols. It is illustrated in figure 2(a.1). The requirements for its implementation consider the availability of support for global, distributed and shared objects. Figure 2(a.2) shows other alternative, where the *Data* objects (*PData*) are *proxies* [8] of a shared object.

- Replicated *Data* Objects. *Data* objects may, given an underlying communication system, act as *replicates*. *Replicates* means, in this context, that every invocation performed in one object will occur in all of the associated objects.

At this experimental stage we were not concerned with synchronization issues or locking mechanisms. Our goal is to make semantic actions performed by one user visible to all cooperating users. We believe that the full consideration of the synchronization mechanisms will have impact on the protocol between *Data* and *Display*, for instance, to express locking conditions and provide visual feedback.

4 Shared and replicated approaches

To experiment the possible architectures, we extended the *4D* toolkit with a set of specialized *Data* classes. The *shared solution* used a programming platform supporting distributed objects, designated by *Comandos* [9,10,11]. The *Comandos* platform provides an extended form of the *C++* language that allows creation, access and manipulation of distributed and persistent objects. Objects have global and unique references, and can be invoked transparently along networks. An object becomes persistent if registered in a global Name Manager, or referenced by another persistent object.
The *replicated solution* was based on a *reliable multicast* protocol (*AMp*) [12] that provides support for *group* communication. A primitive *remote invocation protocol*, including object and method identification and data transfer (between homogeneous machines), was implemented to allow experimentation. The use of higher level services, as the ones provided by *Isis* [13], is a possible solution for future implementations.
The next subsections describe the extention of the *4D* toolkit with both services and the applications built with the two approaches.

A shared hypermedia graph The first application is a primitive hypermedia browser that provides access to nodes of a graph, each one containing different types of information (e.g. text, images). Users can add or remove nodes, modify their contents, add or remove links. To allow multiuser access, *Data* objects like *Graph*, *Node* or *Link* have to be global.

Comandos class	*4D Data*	*4D Display*
Graph	PDGraph	DGraphView
Node	PDNode	DNodeView
Link	PDLink	DLinkView

The characteristics of the classes shown above are the following:

- The *Comandos* classes implement the hypertext functionality, i.e. a *Graph* stores a set of references to *Nodes*, a *Node* contains data and a set of references to *Links*, and a *Link* contains a reference to a *Node*. This classes include basic functions like

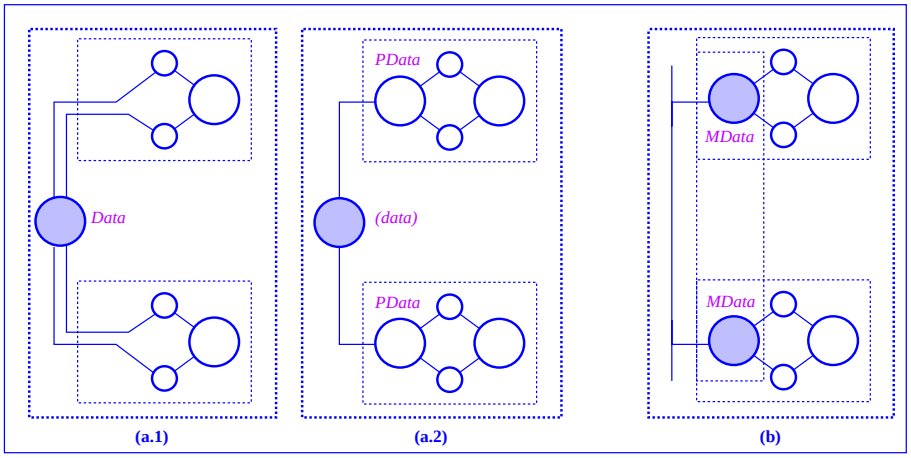

Figure 2: Architecture alternatives

AddNode, *RemoveNode*, *AddLink*, *RemoveLink* or *FollowLink*.

The *Graph* has a name registered in the *Comandos* Name Manager in order to become persistent and shareable. The *nodes* and *links* are also persistent since they are referenced by the *Graph*.

- Each *PD-* object owns a reference to the corresponding *Comandos* object. It reproduces the interface of the shared object but locally to the application. This is the solution illustrated by figure 2(a.2).

  The reason for this solution is that *Comandos* objects do not need to be *4D* objects, i.e. to understand the *4D* links and protocols. On the first hand, this allows the *Comandos* objects to be used by any other application, not exclusively a *4D* one. On the other hand, this avoids having to extend the *Comandos* functionality through the whole *4D* environment. Global, persistent and shared objects require run time support that can be avoided for strictly local and volatile objects.

  This architecture assumes that *Comandos* objects are *active*, just as *4D Data* objects are. When a *Node* is added or deleted, the *Graph* object has to send *update* messages to all his *PDGraph proxies*.

- The *4D Display* objects present the *Graph* as a list of *Node* names, the *Links* as a list of names and the *Node* contents as text or bitmaps.

The *4D* Colab tools The goal of this experiment was to build a simplified emulation of some of the Colab tools [14] with the extended *4D* toolkit. The two target applications were the *Noter*, a tool for posting messages to a cooperating group of users, and the *Denoter*, a tool that provides globally visible *pointers* in the screens of the users.

Replicated *4D Data*	*4D Display*
MDInteger	DTextView
MDString	DTextView

The *4D* toolkit basic types *DInteger* and *DString* where specialized to allow replication (*MD-* objects shown above). The mechanism is the following:

- At initialization time, every application joins a *multicast* group, a service provided by the *AMp* protocol.

- Every invocation on a *MD-* object is packed in a message containing [*object_name, method_name, parameters*] that is sent to the group. Replicates belonging to the group have the same name. This is similar to the Colab *broadcast method* mechanism.

- When a message is received by an application, the underlying *ICE* run time system does the name-to-address translation and invokes the corresponding methods of MD- classes. The changes propagate to the *Display* objects according to the *4D* protocols.

The *Noter* application is simply a *MDString* with auxiliary *private* objects for message composition and message dispatching. The *Denoter* uses three *MDInteger* objects to maintain the *x-*, *y-* and *pointer* values. The *pointer* is also replicated to be seen in the context of each user.

In our system, invocations are broadcasted, instead of state. This fact introduces the requirement for an initialization protocol. At this stage, there is no such protocol. Socially, this means that the meeting is restarted whenever someone enters the group. The introduction of this protocol is an obvious requirement.

5 Open issues

We now focus on the issues raised by the two solutions, comparing them when appropriate, and describing some relevant observations.

Group Management The group management facilities offered by the *multicast* protocol seem to be a very adequate paradigm for our kind of applications.

The applications execute transparently to the number of participating members.

The initialization protocol is at the application level and has to be handled as such. However, an adequate run time support, together with the *ICE* facilities, may hide most of the initialization procedures. Messages to individual users are useful for initialization purposes but also for higher level protocols that, for example, support the introduction of a meeting moderator.

The possibility of having an application belonging to several groups may prove interesting to allow "multi-conferencing" in the same application. This idea can be exploited to partition conferencing contexts according to any adequate criteria like, for example, simultaneous restricted and public conferences.

Synchronization Synchronization services have to be provided and used in both platforms. Replicated objects require specific mechanisms for synchronization, like distributed locks. Shared objects, given their centralized nature, ease the implementation of locking mechanisms.

Dependability According to the *Comandos* architecture, the *Data* object is, at a given instant, mapped in the context of one application and accessed remotely by the others. The failure of the "wrong" machine may therefore affect all the participants. With replicated objects, given the fault tolerant nature of the *AMp* protocol, the users and applications are unaware of the failure of an external machine. The group looses a member but can recover that member later.

Object Design By object design we understand the object interfaces and class hierarchies that have to be defined and implemented. In the *shared* solution, shared objects have to be *active*, as there is a *client-server* relation between *Comandos* and *4D* objects. This nature imposes a protocol to shared objects. From the *4D* applications viewpoint, this is not a problem, but it may result in the creation of global objects that are not usable by other applications, which seems to be a wrong principle in a global object oriented environment.

Persistence Persistence is an important feature if the state of a conference needs to be kept across different sessions. This is definitely an advantage of the *Comandos* solution. Persistence and sharing avoid the need for initialization protocols.

Communication While a multiuser session is proceeding, there should be no major difference in the network load generated by each one of both solutions. The first one generates one remote invocation and n update indications. The second one broadcasts the invocation to n group members. The differences depend on the communication protocols used in both systems, which we are not concerned with here.

The second solution requires, however, the transfer of the objects' state at initialization time. For large objects, this operation may prove too costly. A large *Graph* object in a replicated version, may produce intolerable overheads at initialization or group reconfiguration time.

Application Construction Both solutions integrate well with the *4D* toolkit and *INGRID* construction environment. Both *PD-* or *MD-* classes have the same interface as the *non-shared* or *non-replicated* ones. This means that the application can be built and tested in single-user mode, and later shifted into a multiuser application by replacing the conventional *Data* classes by shared or replicated classes.

The commonality between the *PD-* or *MD-* classes and the remaining *4D* classes allows their integration in the *INGRID* tool which can thus be extended to support construction of multiuser applications.

References

[1] 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.

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

[3] 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.

[4] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1985.

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

[6] J. McCormack, P. Asente, and R. Swick. *X Toolkit Intrinsics - C Language Interface*, 1988.

[7] P.A. Szekely and B.A. Myers. A User Interface Toolkit based on Graphical Objects and Constraints. In *Proceedings of OOPSLA'88 Conference*, pages 36–45, September 1988.

[8] M. Shapiro. Structure and Encapsulation in Distributed Systems: the Proxy Principle. In *Proceedings 6th Intl. Conf. on Distributed Computing Systems ,IEEE Cambridge , Mass.(USA)*, pages 198–204, May 1986.

[9] J.A. Marques and et al. Implementing the COMANDOS architecture. In *Proceedings of the Esprit'88 Conference*, pages 1140–1158. North Holland, November 1988.

[10] P. Guedes and J.A. Marques. Operating System Support for an Object-Oriented Environment. In *Proceedings of the 2nd IEEE Workshop on Workstation Operating Systems, Asilomar*, September 89.

[11] J.A. Marques and P. Guedes. Extending the Operating System to Support an Object-Oriented Environment. In *Proceedings of OOPSLA'89, ACM, New Orleans*, October 89.

[12] P. Verissimo, L. Rodrigues, and M. Baptista. AMp: A Highly Parallel Atomic Multicast Protocol. In *Proceedings of SIGCOMM'89 Symposium, Austin*, September 1989.

[13] K. Birman, T. Joseph, and F. Schmuck. Isis - a distributed programming environment, user's guide and

reference manual. Technical report, The ISIS Project, Dept. of Computer Science, Cornell University, Ithaca, March 1988.

[14] G. Foster. *Collaborative Systems and Multiuser Interfaces*. PhD thesis, University of California at Berkeley, 1986.