INGRID: A Graphical Tool for User Interface Construction

L. Carriço, N. Guimarães, P. Antunes

INESC, Rua Alves Redol, 9, 6D, 1000 Lisboa, Portugal
e-mail: lmc@inesc.inesc.pt, nmg@inesc.inesc.pt, paa@sabrina.inesc.pt

ABSTRACT

INGRID is an interactive tool for user interface construction. The tool enforces a specific user interface model that considers both the functional composition of the user interface elements and an object-oriented approach as the fundamental design and development methodology.

The implementation of INGRID highlights three main components: a run-time support system for interactive programming (in C++), a toolkit that defines the abstract interfaces to the several components of the UI allowing integration of multiple graphical toolkits, and the user interface of INGRID itself.

1. Introduction

As user interfaces (UIs) become more sophisticated and easy to use, they also become harder to create. It is clear that the construction of quality user interfaces requires the existence of an iterative refinement cycle of prototyping and validation. Therefore, interactive and easy to use tools for UI construction must be provided. Also, the definition of an adequate architectural model along with a powerful interactive support, are essential, both to guarantee the coherence of an UI design and optimize the iterative refinement process.

This paper presents an overview of a graphical, object-oriented, UI editor (INGRID - INteractive GRaphical Interface Designer). User interfaces built with INGRID are based on a proposed architectural model (4D). The construction process is supported by an environment for interactive programming (ICE).

The first two sections of this paper describe the models adopted for the UI and its construction with our editor. Complete application and interface models are fundamental to ease the task of UI design and construction. The models must be adapted to the requirements of direct manipulation, multi-thread dialogue control and semantic feedback, while keeping a comprehensive interface organisation.

The editor, described in section 3, includes multiple facilities requested by developers such as: graphical construction of interfaces with minimal coding effort; reduced iterative refinement cycle, with on-line testing, verification and validation of prototypes; direct manipulation of resources and attributes; guidance during the construction phase, with help and browse.

The next sections of the paper are dedicated to implementation aspects. Section 4 focuses on the ICE run-time support that offers the required platform for interactive programming. ICE supports most of the interactive features of the editor: on-line creation, customisation and inspection of interface entities, and dynamic support for object inter-communication.

Section 5 describes the 4D toolkit and the way it conforms to the proposed model and supports the editor functionality. The toolkit aims to provide transparent integration of existing and ‘‘standard’’ graphical toolkits allowing multiple options of functionality and ‘‘look and feel’’.

Proceedings of the EUUG '90 Conference. Munich, Germany, 1990.

1.1. Background

INGRID is a tool that results from the evolution of the IMAGES UIMS [123] developed within the SOMIW ESPRIT project [4] and is a component of a global object-oriented Application Development Environment under development at INESC [5]. IMAGES provided a valuable experience in the subject of UI creation and gave important guidelines about the developers and user expectations from interface construction tools.

The fundamental conclusions drawn from the work with IMAGES were:

- UI development environments must support ‘‘standard’’ user interface toolkits to achieve a significant degree of quality, user acceptance and avoid replication of existing work.
- The use of specification languages (the IMAGES approach) does not represent a major breakthrough in the process of UI construction, which reinforces user preference for interactive tools.

2. The Models

INGRID assumes two fundamental models in the process of building a UI: a generic model for the whole application and a specific model for the user interface itself.

As currently accepted by users and developers of UIs and UI Development Systems, the generic model considers that an application is separated between an interface component and a computational component (dialogue independence, as defined in [6]). An object-oriented approach to this model leads to the definition of both components as abstract data types. Their interfaces are highly dependent on the communication needs between them.

2.1. The Interface Model

The interface model is fully dedicated to functional aspects within the interface component of an application.

Some preliminary discussions over INGRID implementation carried up the idea that, if good functionality was envisioned, it was fundamental to provide a model with a consistent view of a UI along its design, development, and execution phases.

The object-oriented technology has proliferated in the UI area and has already shown to be helpful for UI developers in the design and construction process. Some systems reflect this approach and provide a flat object-oriented structure: MacApp [7], ET++ [8], or InterViews [9]. The main drawback that emerges from this structure is that it gives few guidelines for the design of a UI [10].

Other systems, e.g. GWUIMS [11] or IMAGES, rely on classical UI models like Foley [12], Seeheim [13], or the reference model [14] and apply object-orientation specifically to identify UI components and their relationships. However, the strictly layered structure of the underlying UI models imposes communication overheads that disqualifies them to support fast interactions [15].

Other proposals address the two needs - design methodology and functional separation in a UI - by providing the same separation of the layered UI models, presentation, dialogue control and semantic support, still organizing this layers in a nested structure. MVC [16], PAC [17], and the 4D model are included in this last group.

2.1.1. Components

The 4D model defines four UI components, each one grouping objects with the same functionality. These components are:

Display - The graphical presentation (input and output) of the UI
It groups graphical objects: button, menu, scrollbar and composite Display objects. A composite Display object has a collection of objects also organised in accordance with the four elements of the model. This composition originates a recursive model [18].

Data - Application abstract information
This component provides common usage data structures (int, file, list, ...) with active values [19], i.e. values that automatically propagate their changes. Some Data objects also represent semantic actions of the application.

The primary reason for the existence of Data objects is to provide a clear interface between UI and computational parts while including semantic information on the UI side, in order to be able to support semantic feedback [20].

The existence of Data and Display objects enforces the separation between data and view. The support to different view types that use the same data increases the degree of flexibility of a UI.

Dialogue - The syntactic structure of the human-computer interaction
It groups dialogue control objects. Multiple implementations of dialogue control are feasible: dialogue languages, dialogue cells, etc.

Driver - Conversion of Data information in the format accepted by Display
Is composed by objects that perform data conversion (such as integer to a string).

Several Drivers using the same Data object allow multiple views of its contents.

Drivers introduce also one extra degree of flexibility by reducing the needs of new Data and Display objects. As an example, a ‘‘drawing’’ Display may be driven to represent a table, replacing the need for a ‘‘table’’ Display. In this example, one Driver should have to be coded instead of a Display, a major difference since the Display would be much more hard to code.

Figure 1: The 4D Model Figure 2: An Example

The names of the four elements, Display, Data, Dialogue, and Driver led to the ‘‘4D’’ designation. Figure 1 illustrates the 4D model.

2.1.2. Links

Communication between objects is modeled in 4D through the concept of link. A link can be viewed as a permanent communication channel. One important difference between using links or method invocations is that an object, when sending a message, does not require information about the receiving objects, but only the outgoing links (the same difference of having a permanent communication channel or a datagram communication facility). This increases the degree of flexibility and reusability by providing well defined and autonomous behaviour for objects.

Components and links define how user events are handled, how feedback is provided, and which actions are executed. Figure 2 shows an example of this conduct. Events associated with the scrollbar are delivered to the Bar Display. The Bar notices mouse selections on the bar position and sends a message to the Dialogue with the new position. The Dialogue, in turn, is programmed to invoke the Integer Data object to update its value. Since Data objects have active values they invoke automatically linked Drivers to report changes in their values. Finally, Drivers convert the new value and update both the Label and the Bar.

3. INGRID

INGRID is designed according to the following main objectives:

Interactive Construction of the Interface - The process of creating an interface includes selection of the appropriate UI entities (category, component, class), its instantiation, parameterisation of its attributes (e.g. colour, text) and definition of their relations with other objects (links).

Although these operations can be available through an underlying language description - general purpose or UI specific - we believe that they should be executed interactively. For each selected action the UI programmer should be immediately aware of the change that was performed, either on the interface aspect or on its behaviour. Naturally, INGRID should also provide mechanisms to verify, and validate that behaviour, enforcing the incremental process of UI development.

Direct Manipulation of Entities - Experience has shown that direct manipulation of UI entities is better accepted by developers and drastically reduces the construction effort.

If direct manipulation is a good approach to some problems raised in UI construction, namely when the displayable part of the interface is being handled, it is sometimes difficult to apply to generic purpose costumisation of objects, for example the attributes of an abstract data type. In these cases, solutions must be adopted that keep in mind the graphical aspect of INGRID. A hierarchical chain of menus, providing adequate guidance mechanisms, should be preferred to the direct coding approach.

Model enforcement - The importance of a good model for UIs has been stressed in the last section. INGRID adopts the proposed 4D model to describe the interface it presents to the designer, and enforces it through all the construction process of new interfaces.

Multi-thread Dialogue - Multi-thread dialogues, understood as the ability for the user to interrupt an action, execute another and return to the first, has been used successfully in the process of human-computer interaction. INGRID should offer it to the UI designer.

Binding to the Computational Component - INGRID is responsible for the creation of the interface component of the application. However, mechanisms must be provided to bind it to the computational part. Those mechanisms should emphasise the separation between application components, enforcing the fundamental concept of dialogue independence. They should also offer a flexible front-end that supports connections to multiple programming languages without requiring language specific knowledge from the UI programmer. Besides C++ and C, bindings to the computational part should also be available for languages like Pascal.

Help and guidance - The flexibility provided by interactive programming, with graphical direct manipulation and multi-thread capabilities, may present to the designer a very large set of options for which decisions are required. INGRID should provide guidance capabilities through the construction process, either by pointing the most adequate solution, or defining a structured approach to the interface organisation coherent with the model. On-line help facilities fall in the scope of these needs.

3.1. INGRID Components

INGRID has five main components which closely follow the adopted model (see Figure 3):

Figure 3: INGRID components

- **Display sub-editor -** This editor handles objects with display properties, like position, size, text and bitmaps for which the direct manipulation paradigm seems most appropriate. The editor is divided in three main areas: *palette, canvas,* and *attribute-programmer*.

  The palette is an area where 4D Display classes are represented and can be picked to create the UI. Picking such objects is equivalent to create a new instance that can be positioned in the working canvas. Several palettes can be available representing categories of related Display classes.

  The canvas is the board where the programmer defines the UI external aspect. Multiple canvases can be used to give the notion of depth in the UI behaviour.

  The attribute area is a general purpose programmer composed by a set of menus that allow the definition of attributes that cannot be customised within the canvases (e.g. fonts and colours).

- **Data sub-editor -** Data objects represent abstract structures not visible to the programmer. For these, the Data sub-editor provides an identification mechanism, which associates a name to each object. Later, these objects can be selected from a menu of created instances and parameterised through an attribute-programmer similar to the one available in the Display sub-editor. Instantiation can be performed from a collection of menus that refer to the 4D toolkit Data classes.

- **Driver sub-editor -** The nature of driver objects makes difficult the adoption of a direct manipulation paradigm for their parameterisation. In this case are also available menus for instantiation of Driver classes, attribute-programmers and an identification service similar to the one of Data objects. Composition of cascade Driver objects can be edited with a specific graph oriented editor.

- **Dialogue sub-editor -** The dialogue sub-editor depends on the dialogue control model that is made available by the underlying toolkit. Our approach defines an object-oriented event driven dialogue for which an event or a conjunction of events arriving to an object may trigger a predefined action. This kind of dialogue can be represented as a directed graph, suitable for interactive programming using direct manipulation techniques. The dialogue sub-editor, besides the graph editor, provides an attribute-programmer and an action-programmer which is used to associate tokens and arguments to graph arrows.

- **Interface organiser -** While the above sub-editors provide the definition of each component of the 4D model, the interface organiser allows global access to the interface entities. It is the front-end of INGRID, and will allow the establishment of links between the model components, the composition of objects into composite Displays, access to a generic attribute-programmer and global functions for store/retrieve, testing, etc.

- **Application connector -** The application connector is a specialised attribute-programmer that provides transparent access to the computational component of the application. That component is viewed as a set of Data objects, that defines the necessary translations between interface requests and calls to the computational component. The connector also provides a name service that can be used by the computational component to invoke interface entities.

3.2. Underlying structure

INGRID is build according to the layers shown in Figure 4. The background layer of INGRID is the ICE run-time support that provides the interpretation capabilities needed to achieve a fast interactive design of the UI.

Figure 4: INGRID structure

The next layer, is composed by a set of toolkits which follows the 4D model concepts. Those toolkits provide a library of objects which implement the several components of the interface. When the UI designer picks an object from the Display editor palette, it actually instantiates, through the ICE support, an object belonging to a class available in one of the toolkits.

4. ICE - Support for Interactive Programming

ICE has been developed as a run-time support to provide interpretation features within C++ [21]. Essentially, it makes available mechanisms for:

- **Type information** with type identification, conformance and conversion testing, and knowledge over instance data and member functions.
- **Communication by message** based on the available type information, provides a standardised paradigm of communication between objects. Dynamic binding is an immediate consequence of this communication paradigm.
- **Object storage and retrieval** again supported by the type interface description, enables transparent storage/retrieval facilities.
- **Object identification** enables the assignment of human readable names to objects, which enforces a coherent mechanism of interaction between the user and the programming entities. Being a global service it also provides the functionality of a run-time instance management table.
- **Interface uniformity** offers an abstract handling of objects, essential to ease the development of flexible programming environments, i.e. if all objects are accessible through a common interface, other types can be easily integrated without code modification.
- **Common use facilities** like error handling, debugging, ...

4.1. Implementation

The ICE functionality is implemented by a library of C++ classes, which can be classified under three main categories: type classes, name service, and common interfacing.

Type classes: The basic concept of ICE is the notion of *type-object*. It is an object that fully describes a C++ type. A type-object can be viewed as an extension of the Smalltalk's *class-object*, to all C++ types: classes, basic types (char, int, float, ...), pointer types and function types. The introduction of this broader notion, enables an uniform, coherent, access to the heterogeneity of C++ types. Type-related mechanisms like storage/retrieval and communication by message are available both to user defined classes, and C++ basic types.

The type classes closely follow the C++ type model:

- **IType** is an abstract class that defines the generic interface of type-objects. It declares methods for type checking, member function execution through message passing, instance creation, storage, and retrieval.
- **IBasicType** implements the interface for C++ basic types (e.g. operators +, -, ...).
- **IFuncType** provides type checking capabilities for functions.
- **IPointer** defines the generic behaviour of pointers.
- **IClass** provides the functionality associated with class and struct types: defines member function invocation by message with type checking, member overloading, and default arguments; considers inheritance both in conformance tests and message sending; uses constructors for object instantiation; uses instance data information for automatic storage and retrieval facilities; provides access to members according to their protection attribute (public, protected or private).

To have ICE complete functionality, an user-defined class must have at run-time its type-object. Such an object must contain very exhaustive information for which even the availability of suitable macros (as in OOPS and ET++) require a large effort from the class programmer.

ICE includes a parser for C++ definitions, that automatically generates code for the corresponding type-object.

Name Service: The name service is an instance of INameService class which provides a dictionary for translation of user readable names into object pointers and vice-versa. INGRID heavily uses this service to perform actions on objects specified by name.

Common Interface: ICE offers two different approaches to achieve common interfacing to objects. One provides a complete integration (similar to the one used in Smalltalk [22], OOPS [23] and ET++ [8]), by specifying a common base for all classes, named IObject. The other allows the integration of types that do not derive from that class. It is accomplished by means of an encapsulating object that enables the access to the corresponding type-object and offers an interface similar to IObject.

5. The 4D Toolkit

The 4D toolkit provides a set of classes (see Figure 5) upon which INGRID creates and customises a UI. The toolkit must be rich enough in order to satisfy most of the needs of INGRID as well as developers and users.

With the purpose to support the 4D model and its components and links, all objects have a base class, DObj, that defines a common behaviour to establish, verify and maintain lists of links, using links to send and receive messages. Links rely on the ICE message passing mechanism provided by the IObject base class.

Four classes are derived from DObj: DataObj, DriverObj, DisplayObj and DialogueObj. An object belongs to a specific component if its class inherits from one of these.

Figure 5: Toolkit Classes

Display objects have access to the Window System. The consensus around the X Window System [24] made available several graphical toolkits (Xt [25], Andrew [26], InterViews). The current implementation of the 4D toolkit uses Xt (and several widget sets: Athena, Motif, Open Look). This provides 4D with features like system-independence, portability, network transparency, multiple options of functionality and ''look and feel''. Furthermore, and since these toolkits are often difficult to use, an independent high-level access to its entities is available.

One class, DisplayXt, defines the interface to the Xtoolkit intrinsics. Another class, DisplayRoot, is the root class in the display hierarchy. The other Display classes bind to the specified widget sets (Label, Command, ...) and are derived from DisplayXt. Tools are available to parse widget declaration files and automatically produce these classes.

A specialised Data class, Proxy, is dedicated to interface with the computational part. A Proxy defines a set of semantic actions of an application. This object uses the ICE run-time support to dynamically create its interface.

Essential for interactive construction of the UI is the capability of doing a snapshot of its run-time structure. This save/retrieve facility has two concerns, one related with the run-time structure within the 4D model, mainly the created objects and their links; the other related with graphical parameterisation of Display objects that escape from the modeled objects to the specific widgets. In the first case, the ICE run-time information is sufficient to produce a snapshot; in the second case, a database with graphical information must be generated, using directions from the Window System.

6. Conclusion

The final goal of INGRID is the creation of a flexible interactive environment for user interface creation, in the context of an object-oriented framework. The target applications are the small and medium sized applications typically found in the engineering and office environments. We believe that the requirements defined for INGRID provide a solid ground to achieve this goal:

- The interactive behaviour provides a definite advantage in the construction of medium scale user interfaces. Actually, automatic mechanisms such as a specification language may have to be considered in larger scale applications, that is however out of the scope of this work.
- The support for the integration of graphical toolkits provided by the 4D toolkit is essential. This integration assures conformance with ''standard'' behaviour and avoids the effort involved in the development of a graphical toolkit.

  The 4D toolkit has been used independently from the editor. Its use did not reduce significantly the amount of code but provided a homogeneous structure to the several components of the interface, added functionality, greater independence from the graphical toolkit and the advantages of the C++ language.

  The issue of toolkit independence needs further validation by including access to Andrew or InterViews. This integration will provide a clearer definition of an adequate interface for the

Display components and help to filter out any bias introduced by Xt.

- The development of a run-time support system in C++ provides a fairly open platform for integration of other software systems, either the graphical toolkits mentioned above or other available libraries.

  The run-time support system has been fully tested and approved by the implementation of INGRID itself. As a general purpose interactive/interpreted environment, its use can be considered in applications that require these characteristics and usually rely on object-oriented languages or environments such as Clos or Smalltalk.

The INGRID components have reached a stage of reasonable maturity and a first interface for INGRID has been built. INGRID will now evolve according to user requirements. From our viewpoint, one of the advantages of an interactive tool is the capability to reach a larger number of users, at least for experimentation purposes. This is definitely the major drawback of a linguistic tool.

We expect that user demands will concentrate on current limitations, namely higher direct manipulation capabilities, performance and compatibility and integration with other widget sets/toolkits.

References

1. L. P. Simoes and J. A. Marques, ‘‘IMAGES - An Object Oriented UIMS,’’ in *Human-Computer Interaction - INTERACT’87*, pp. 751-756, North-Holland, 1987.
2. J. A. Marques, L. P. Simoes, and N. Guimaraes, ‘‘A Uims and Integrated Environment for the Somi Workstation,’’ in *Proc. of the ESPRIT’88 Conf.*, pp. 1001-1019, Brussels, November 1988.
3. L. Simoes, ‘‘IMAGES - An approach to an Object Oriented UIMS,’’ in *Proc. of the Autumn 1988 EUUG Conf.*, pp. 143-157, Portugal, October 1988.
4. ‘‘Secure Open Multimedia Integrated Workstation,’’ SOMIW Esprit 367 - Technical Annex, 1985.
5. J. A. Marques and P. Guedes, ‘‘Extending the Operating System to Support an Object Oriented environment,’’ in *OOPSLA’89 Conf. Proc.*, pp. 113-122, New Orleans, October 1989.
6. H. R. Hartson and D. Hix, ‘‘Human-Computer Interface Development: Concepts and Systems for its Management,’’ *ACM Computing Surveys*, vol. 21, no. 1, March 1989.
7. K. J. Schmucker, ‘‘MacApp: An Application Framework,’’ *Byte*, vol. 11, no. 8, pp. 189-193, August 1986.
8. E. Gamma, A. Weinand, and R. Marty, ‘‘ET++ - An Object-Oriented Application Framework in C++,’’ in *Proc. of the Autumn 1988 EUUG Conf.*, pp. 159-173, Portugal, October 1988.
9. M. A. Linton, P. R. Calder, and J. M. Vlissides, ‘‘The Design and Implementation of InterViews,’’ in *Proc. of the USENIX C++ Workshop*, New Mexico, November 1987.
10. R. Hartson, ‘‘User-Interface Management Control and Communication,’’ *IEEE Software*, pp. 62-70, January 1989.
11. J. L. Sibert, W. D. Hurley, and T. W. Bleser, ‘‘An Object-Oriented User-Interface Management System,’’ *Computer Graphics*, pp. 259-268, August 1986.
12. J. Foley and A. Van Dam, *Fundamentals of Interactive Computer Graphics*, pp. 220-222, Addison-Wesley, 1982.
13. M. Green, ‘‘Report on Dialogue Specification Tools,’’ in *User Interface Management Systems*, ed. Gunter E.Pfaff, pp. 9-20, Springer Verlag, 1985.
14. K. Lantz, ‘‘Reference Models, Window Systems and Concurrency,’’ *Computer Graphics*, pp. 87-97, April 1987.
15. B. A. Myers, ‘‘User-Interface Tools: Introduction and Survey,’’ *IEEE Software*, pp. 15-23, January 1989.
16. A. Goldberg, *Smalltalk-80: The Interactive Programming Environment*, Addison-Wesley, 1983.

17. J. Coutaz, ‘‘The Construction of User Interfaces and the Object Paradigm,’’ in *ECOOP’87, European Conf. on Object-Oriented Programming*, pp. 121-130, Paris, June 1987.

18. J. Coutaz, ‘‘Architecture Models for Interactive Software,’’ in *ECOOP’89, Proc. of the Third European Conf. on Object-Oriented Programming*, pp. 383-399, Nottingham, July 1989.

19. P. A. Szekely and B. A. Myers, ‘‘A User Interface Toolkit Based on Graphical Objects and Constraints,’’ in *OOPSLA’88 Conf. Proc.*, pp. 36-45, San Diego, September 1988.

20. J. R. Dance, ‘‘The Run-time Structure of UIMS-Supported Applications,’’ *Computer Graphics*, pp. 97-101, April 1987.

21. B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, 1986.

22. A. Goldberg and D. Robson, *Smalltalk-80: The Language and its implementation*, Addison-Wesley, 1983.

23. K. E. Gorlen, ‘‘An Object-Oriented Class Library for C++ Programs,’’ in *Proc. of the USENIX C++ Workshop*, New Mexico, November 1987.

24. J. Gettys, R. Scheifler, and R. Newman, *Xlib - C Language X Interface, X Window System X11R3*, 1988.

25. J. McCormack, P. Asente, and R. Swick, *Xtoolkit Intrinsics - C Language Interface, X Window System X11R3*, 1988.

26. C. Neuwirth and A. Ogura, *The Andrew System Programmer’s Guide to the Andrew Toolkit*, ITC - Carnegie-Mellon University, January 1988.