Adaptable Dialogue Controls in User Interfaces

Holger Vogelsang
Institute for Microcomputers and Automation, University of Karlsruhe
Haid-und-Neu-Str. 7, 76131 Karlsruhe, Germany
E-mail: vogelsang@ira.uka.de

Abstract

This paper describes an approach for a platform- and implementation-independent design of user interfaces using the UIMS idea. It is a result of a detailed examination of object-oriented techniques for program specification and implementation. This analysis leads to a description of the requirements for human-computer interaction from the software-developers point of view. On the other hand, the final user of the whole system has a different view of this system. He needs metaphors of his own world to fulfill his tasks. It's the job of the user interface designer to bring these views together. The approach, described in this paper, helps bringing both kinds of developers together, using a well defined interface with minimal communication overhead.

Keywords: *graphical user interface, behavior model, dynamic model, interpreter, user interface management system*

1. Overview

One of the most important results in the separation of gui and application is the creation of two different working areas: The user interface designer and the application developer. Both of them have special skills and knowledges, the communication between them is done using a well defined interface. To bring both together, firstly, we have defined a basic system for symbolization and manipulation of structured information. This system is used as a hardware independent platform [4],[5],[6]. On the other hand, we wanted to realize the whole system as a UIMS (user interface management system, [21]), which normaly leads to a large communication overhead between UIMS itself and the application: A solution in the construction of such a system is to give the UIMS as much independence as possible. One of the best ways to solve this problem is to allow the UIMS to handle most parts of the dialogue control itself. For this reason, we have introduced two models in our system *Fluids* to define the user interface [5]. The *static model* describes the user interface structure, using the design and placement of their components [6]. The idea of symbolic information visualization is consequently used: *Menus* are aggregate symbols, composed of buttons, which are symbols too. *Picture* as the basic class is a container for a set of symbols without any internal relation. From this class are more specific classes derived: *Menu*, *Mask*, *Table.* Additional basic classes are not necessary, because classes with other semantic like *Hierarchical Graphs* can be constructed using the *dynamic model.* The complete static model is discussed in other papers ([4],[5],[6]), so that we put our focus on the dynamic or behavior model.

2. Requirements Analysis

Different kinds of people have different views of an application:

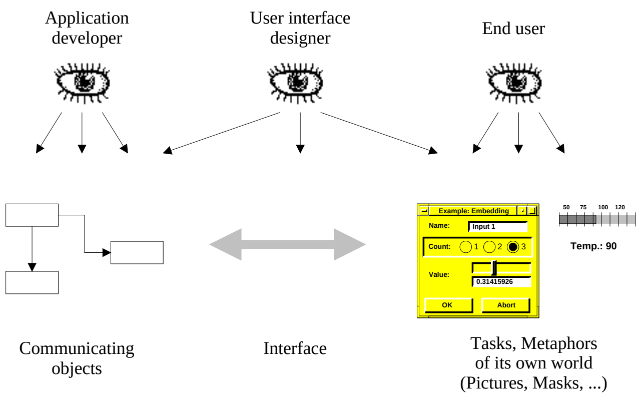

Figure 1. Views of an application

The application developers model consists of objects and relations between objects, the end user see metaphors of its own world. Finally, the user interface designer has to bring both groups together, building the bridge between them. Our system "Fluids" should help him to solve this task. It must be a powerful tool to map communicating application objects and relations to user interface components. The user interface designer is a specialist concerning user guidance questions. He is familiar with the applied object-oriented modeling technique. His main task is the modeling of the look and control of the dialogues, independent of the application desgin. The result must be an user interface service, build by the user interface designer and a functional core service, build by an application developer. Both systems communicate and coordinate their work to fulfill the overall requirements analysis. This approach is based on the separation of application core functionality and user interface, using the UIMS idea. There are several publications available, which try to solve this problem [9]. Before we start to discuss our approach, we want to give a short overview over other ideas to solve the dialogue control task [12].

2.1. Application coded

This is the classical solution, which was a standard for many years in the construction of interactive applications. And it *is* the commonly used approach in many commercial tools nowadays. The dialogue control is hard coded in the application, mixed with the core function code. As a result, the dialogue control is spread over the whole application, making support and modifications a very hard job. Microsoft Foundation Classes (MFC) uses this approach.

2.2. Schematic notations

Schematic notations are these notations, which use a graphical specification for dialogue control.

- **State transition diagrams**
  State diagrams are a very old concept, which was applied for many different areas in computer science in the recent years. It is used in dialogue control to specify states in user work flow and the changes between different tasks. A state changed is initiated by user events or application interventions. The major problem of state diagrams are the huge number of states in complex systems.

- **Hierarchical state transition diagrams**
  Hierarchical state diagrams are the next evaluation step of simple state diagrams. The idea consist of the introduction of hierarchical states. This makes it possible to examine only these events and states at a hierarchy level, which are required at this level to describe the changes.

- **Concurrent dialogues**
  The next step to reduce the number of states is the introduction of so called concurrent dialogues. In addition to simple or hierarchical state transition diagrams, concurrent dialogues allow the specification of parallel dialogues or parallel state changes, more than one state can be active at a given point of time. A dialogue is split into many sub-dialogues, state changes are defined on sub-dialogues. Concurrent dialogues are a often solution for systems with help functionality to avoid the description of state changes due to access to the help function.

- **Function flow diagrams**
  Function flow diagrams are not often used in dialogue control specification, because one of their hardest disadvantage is the missing of asynchronous events.

2.3. Textual notations

Simple textual notations operate without any (mostly helpful) graphical diagrams.

- **Context free grammars**
  Context free grammars are used in dialogue control specification for many years. Especially the *Backus-Naur Form* (BNF) with numerous extensions can be found in many different fields of computer science. But in the last years there were some disadvantages of BNF for direct-manipulative systems discovered: Grammars are not easy do understand by humans, which is a real problem for large dialogues.

- **Event-based techniques**
  The trend in textual notations is the relinquishment of BNF to give event-based techniques the advantages due to demands of direct-manipulative dialogues. In such models input devices generate events, which are processes in a first-in first-out manner. The event handlers can be expressed in a high-level language or any other notation. This technique allows the introduction of parallel dialogues, which is — considering the controllability of a dialogue — often required. A disadvantage of event-based techniques is the nonexistence of control flow, directly visible in state transition diagrams.

- **Formal techniques, CSP**
  We will not discuss any formal techniques like *Communicating Sequential Processes* (CSP) here, because they are mainly used to prove the correctness of a dialogue. This is not part of this paper.

2.4. Object-oriented techniques

Although object-oriented modeling techniques are used in application development for many years, their introduction in dialogue control is relative new. In this paper we want to present one of the most important representative of this class.

- **Jacob**
  Jacob presents a specification language for direct-manipulative user interfaces, which is based on an object-oriented approach [18]. He treads user interfaces as a set of interacting objects, which behaviors are firstly described individually. Objects with a similar behavior are aggregated in classes. They own a set of variables (size, position on screen, . . .) together with methods to access and modify these variables. New classes can be derived from existing classes. Ja-

cob applies state transition diagrams, which access methods, to describe the behavior.

3. Conception

End users of modern systems must be able to adapt the user interface or parts of the functionality for their own requirements or preferences. Some of todays applications like Microsoft Word or Borland Paradox contain an interpreted or precompiled language to allow such adaptions. These languages are build for very special tasks like database access or word processing. We want to present a general solution firstly for dialogue control and secondly with some extensions for general application dependent tasks. The conception consists of two steps: Firstly, we present the idea of equal partners. This means, a core application and a user interface application communicate using the UIMS system. Secondly, we show the realization of the user interface application, using the interpreted language. The overall structure looks like the diagram, shown in picture 2.

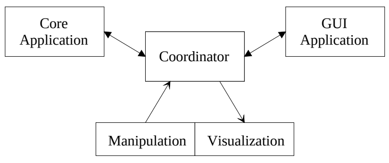

Figure 2. System of equal partners

Keeping picture 1 in mind, we have to define the visualization of application objects first. This can easily be done using a slider as an example. The attributes of the slider symbol are divided into two parts: The *static* part is configured (f.e. off-line) and not changed by the application in most cases. The *dynamic* part can be changed by the end-user or the application and works as the communication attribute between application and user.

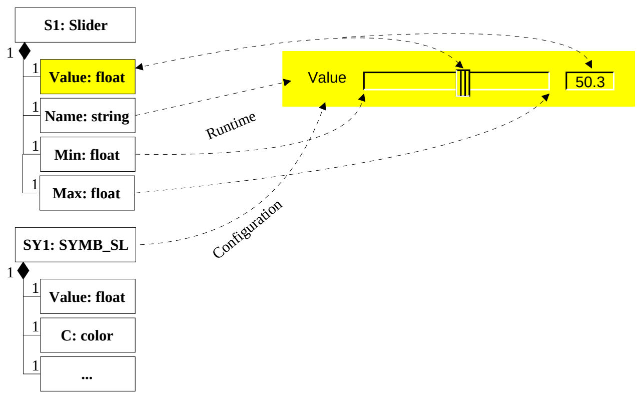

Figure 3. Visualization of application objects

But we put our focus on event handling and user interaction, as shown in picture 4. Each user event causes a state change at least in the man-machine interface. Picture 4 shows the interaction idea, found in nearly any kind of user interface system. We extend this idea in the next section to the principle of equal partners.

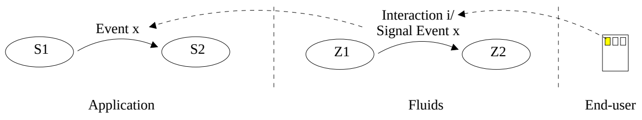

Figure 4. Interaction in conventional system

In addition to this picture, the principle of interaction in Fluids is the change of a control flow either in the core application or in the behavior of the user. Both are equal partners, where the UIMS is used as the guidance to coordinate the actions. Picture 5 contains the basic structure. As we can see, the result is a (nearly) symmetric diagram. Both partners pass their orders and queries as events to the symbol management. On the other hand, input are transmitted back as events too. The applications (core or gui) interpret the events and act as configured.

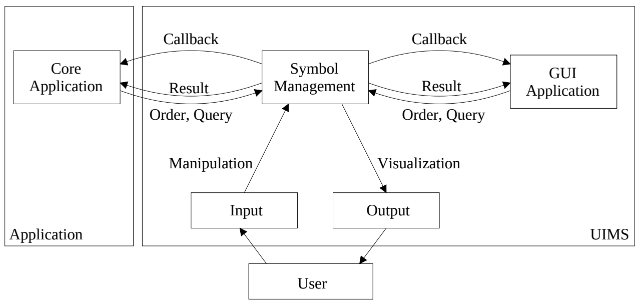

Figure 5. Interaction in Fluids

Picture 6 shows the simplified system structure of the behavior model using UML notation[2].

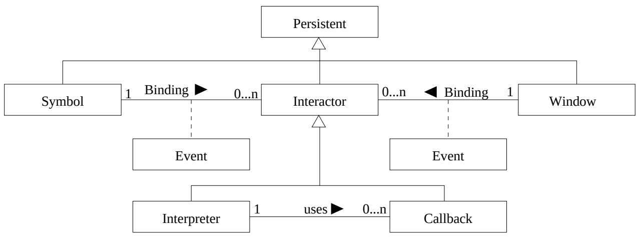

Figure 6. Behavior model

The user causes events by symbol or window manipulation. If an interactor object is bound to this symbol and the binding condition concerning the event type is true, the specified interactor is called. This function is either coded in a Pascal-like syntax and executed by a built-in interpreter, or a precompiled callback function of the application or man-machine service Fluids. The interpreter normally “knows” a basic set of built-functions, which are now extended by the methods defined on the user interface components. Since every application needs some kind of communication with its user interface, applications can register callback-methods to their own objects to allow asynchronous event notification. These callback-methods are available in the interpreter as ordinary functions. The class-hierarchy makes a distinction between callback- and interpreter interactors to allow an uninterpreted fast call of time-critical callbacks. Special application functionality, which could be required for text editors, is added using this technique. Picture 7 shows the scheme:

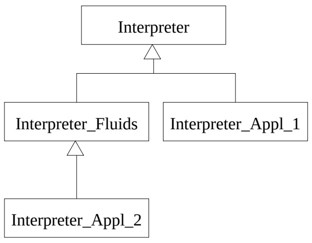

Figure 7. Interpreter usage

The basic interpreter has only a core functionality. Derived interpreter classes add more functions for special tasks. The user interface management service builds its own extended interpreter, while special application-dependent interpreters are either derived from the user interface interpreter, if this functionality must be available to the end user too, or directly from the basic interpreter. Because this work is based on the distributed environment *CORBA*, callback-functions are not limited to platform boundaries. For this reason, the interpreter is also able to call remote functions or methods, if their callbacks are registered.

The interpreter and its Pascal-like language are not discussed in detail in this paper, because it is based on the *LUA* interpreter of the the *TeCGraf-Grupo de Tecnologia em Computacao Grafica* in Rio de Janeiro[13]. This interpreter is freely available for commercial and non-commercial applications[16]. More information is also available in the LUA-manual[17].

4. Example

The largest application build with this tool is a simulator for an automated guided vehicle system (AGVS), created for a well-known German company in the car industry. Picture 8 shows a screen-shot of this application.

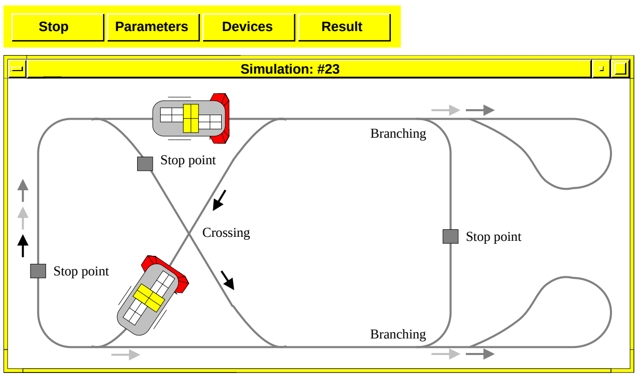

Figure 8. Simulator for an automated guided vehicle system

During normal operation, the positions of all transport units are displayed on the screen. But during the configuration of a new plant or course, the map of this course is constructed using an interactive editor, written (mostly) in the interpreter code. To test and simulate the behavior of this course together with the transport units without the need of expensive tests, a simulator is needed. This simulator tool is based on Fluids, most of the functionality is expressed in interpreter code.

5. Conclusions

The usefulness of the above described approach was shown in some example applications at our institute. The most important features are discussed in this section.

- **Adaptable dialogue control**
  The dialogue control, which means the control flow and behavior of the user interface, is adaptable to user preferences during runtime. As shown in Picture 6, all objects of the user interface are kept persistent in a database. This offers an easy to handle mechanism for user-dependent graphical user interfaces.

- **Extensibility**
  LUA offers mechanisms to register callback functions and to modify or access all interpreter variables and functions. It is created as a library, which can be accessed from the host implementation. The untyped language offers a very flexible way to implement communication with the host language: **Tables**, defined as associative arrays, allow the handling of host-specific data-structures. LUA also has built-in functions, which handle so called fallbacks. These functions are called in special error situations (access to non-existing table indices, call to undefined functions, floating point errors, . . .). Fallbacks can be used to implement a kind of object-oriented extension to LUA.

- **Unique interpreter**
  One of the most important advantages of the described approach is the availability of a unique interpreter in all kinds of applications. The interpreters are scalable, which means, that the application can define, which functionality is available to the end-user.

- **Interpreted code**
  The interpreter is able process either ASCII-Strings as code or precompiled P-Code. This allows runtime modifications together with fast code execution using the built-in compiler.

6. Annotation

This paper is based on research done at the Institute for Microcomputers and Automation in Karlsruhe.

References

[1] Len Bass, Joelle Coutaz, “Developing Software for the User Interface”, *Addison-Wesley Publishing Company*, 1990

[2] Grady Booch, James Rumbaugh, “Unified Method for Object-Oriented Development”, *Document Set 0.8*, Rational Software Corporation, 1995

[3] Grady Booch, Ivar Jacobson, James Rumbaugh, “The Unified Method Language for Object-Oriented Development”, *Document Set Version 0.91 Addendum UML Update*, Rational Software Corporation, 1996

[4] Uwe Brinkschulte, Marios Siormanolakis, Holger Vogelsang, “Man Machine Service”, *in: proceedings of the IAR-Workshop on Knowledge Engineering and Object Oriented Automation, KEOOA’95*, Strasbourg, France, 1995

[5] Uwe Brinkschulte, Marios Siormanolakis, Holger Vogelsang, “Graphical User Interfaces for Heterogeneous Distributed Systems”, *in: proceedings of EI’96*, San Jose, USA, 1996

[6] Uwe Brinkschulte, Marios Siormanolakis, Holger Vogelsang, “Visualization and Manipulation of Structured Information”, *in: proceedings of Visual’96*, February 1996, Melbourne, Australia

[7] Uwe Brinkschulte, Holger Vogelsang, “Eine objektorientierte Schnittstelle für ein Echtzeitprozeßdatenhaltungssystem”, *in: Proceedings of Echtzeit’96*, Karlsruhe, Germany, 1996

[8] Hans-Jörg Bullinger, Klaus-Peter Fähnrich, Christian Janssen, “Ein Beschreibungskonzept für Dialogabläufe bei graphischen Benutzungsschnittstellen”, *Informatik Forschung und Entwicklung 11: page 84–93*, Springer, 1996

[9] David T. Clarke, Geoff P. Crum, “Dialogue Specification and Control: A Review of Models and Techniques”, *Information and Software Technologie, Volume 9/36*, 1994

[10] Ralf Danzer, “An Object-Oriented Architecture for User Interface Management in Distributed Applications”, *University of Kaiserslautern, Fachbereich Informatik*, 1992

[11] Rich McDaniel, Brad A. Myers, “Amulet’s Dynamic and Flexible Prototype-Instance Object and Constraint System in C++”, *Carnegie Mellon University, Human-Computer Interaction Institute Technical Report CMU-HCII-95-104*, July 1995

[12] Alan Dix, “Human-computer interaction”, *Prentice Hall*, 1993

[13] Luiz Henrique de Figueiredo, Roberto Ierosalimschy, Waldemar Celes, “Lua: an Extensible Embedded Language”, *Dr. Dobb’s Journal*, December 1996

[14] D. Galara, B. Iung, G. Morel, F. Russo, “Intelligent actuation and measurement: system based modeling in priam”, *in: proceedings of the 2nd IFAC Workshop on Computer Software Structures*, August 94, Lund, Sweden

[15] Oliver Hammerschmidt, Holger Vogelsang, “Design of Distributed Real-Time Systems in Process Control Applications”, *in: proceedings of I-CIMPRO’96*, Eindhoven, The Netherlands, 1996

[16] Roberto Ierosalimschy, Luiz Henrique de Figueiredo, Waldemar Celes, “Lua: an extensible extension language”, *Software Practice and Experience*, 19xx

[17] Roberto Ierosalimschy, Luiz Henrique de Figueiredo, Waldemar Celes, “Reference Manual of the Programming Language Lua 2.5”, 1996

[18] R. Jacob, “A Specification Language for Direct-Manipulation User Interfaces”, *ACM Transactions on Computer Graphics, Vol. 4, No. 4, Page 283–317*

[19] OMG, “IDL C++ Language Mapping Specification”, *Object Management Group (OMG), Technical Paper 94-09-14*, 1994

[20] OMG, “The Common Object Request Broker: Architecture and Specification — Revision 2.0”, *Object Management Group (OMG), Technical Paper 95-07-20*, 1995

[21] Günther E. Pfaff (ed.), “User Interface Management Systems”, *proceedings of the Workshop on User Interface Management Systems in Seeheim*, Eurographics (The European Association for Computer Graphics), 1985, Seeheim, Germany

[22] C. Pereira, Th. Rathke, “Objektorientierte Entwicklung von Echtzeitsystemen in der Automatisierungstechnik”, *in: proceedings of the 39th Int. Wissensch. Kolloquium*, Sep. 94, Illmenau, Germany