Requirements for UIMSs

Joëlle Coutaz
Laboratoire de Génie Informatique (IMAG-Campus)
BP 53 X, 38041 Grenoble Cedex, France
e-mail: joelle@imag.imag.fr

Len Bass
Software Engineering Institute, Carnegie-Mellon University
5000 Forbes Av., Pittsburgh 15213 PA, USA
e-mail: ljb@sei.cmu.edu

Submitted to the Esprit/Eurographics Workshop on User Interface Management Systems and Environments, Lisbon, 4-6 June 1990.

The material of this paper is based on chapter 5 of a forthcoming book copyrighted to Addison Wesley.

Abstract

The notion of UIMS covers different levels of services ranging from ad-hoc dialogue controllers to user interface generators. In this article, the term "UIMS" is employed in a generic way. We first propose an interpretation of the Seeheim model in which we stress separation of concerns, high band width and parallelism between the basic components of a UIMS. For each of the fundamental components we then identify precise requirements. These requirements aim at satisfying cognitive psychology principles such as the opportunistic behavior of the end user, and at supporting software engineering principles such as reusability and iterative refinement.

1. Introduction

In the literature, the notion of UIMS covers different levels of services. This paper proposes the following definition: A UIMS is primarily a run time kernel which controls the interaction between a computer system and a user. When such a kernel is reusable, as in MacApp [Schmucker 86], Aïda [Ilog 89] or ApEx [Coutaz 87a], it becomes an application framework. When it is used as the run time support of an automatically generated user interface, it is part of a user interface generator as in Serpent [Bass 88], Cousin [Hayes 85], ADM [Schulert 85], UIDE [Foley 88], or Mickey [Olsen 89]. User interface generators produce a user interface from a high-level specification. Sometimes, the specification is performed through an interactive editor which avoids learning a new language and which instantly shows the result of the specification [Hullot 86, Karsenty 87, Cardelli 87]. In some cases, the construction of a user interface involves a combination of interactive specification and traditional programming activities as in Interface Builder [Webster 89].

In this article, the term "UIMS" is employed in a generic way to cover the three classes of services: dialogue manager, application skeletons and user interface generators. All of these tools rely on the same driving design principle: the separation of the user interface from the functional core. All of them have been devised to support rapid prototyping and iterative refinement.

In software engineering, the effectiveness of rapid prototyping and iterative refinement relies on reusability and extensibility. In turn, these two features have a strong impact on software architecture. The next section presents a basic framework upon which separation, reusability and extensibility may be elaborated. This framework shows a number of components for which functional requirements can be identified. Such requirements will be analyzed in sections 3 and 4: requirements on dialogue control, requirements on the functional core, and requirements on the presentation. Section 5 discusses issues related to general services such as cut-and-paste and undo-redo facilities.

2. The Overall Framework

The Seeheim model has set the foundation for the separation between the functional core (the application) and the presentation (the user interface) [Pfaff 85]. Unfortunately, the ambiguity of the term "separation" has led to a number of wrong interpretations. One of them is the linguistic mapping of the three components identified in the Seeheim model with semantics, syntax and lexical analysis: the application implements the semantics, while the dialogue controller and the presentation component take care of syntactic and lexical issues. This sequential pipe-line model of the interaction, based on compiler technology, is convenient for automatic generation of simple user interfaces, but it does not support interleaving inherent to cooperative dialogues [Coutaz 89].

In real life dialogues, utterances are interleaved between partners and may overlap over time. The dialogue is either directed by a leader or distributed among the partners, depending on the characteristics of the task and of the participants. In general, the style of the interaction switches between these two extremes. A similar schema should apply to computer human interaction.

In computer human interaction, participants may be identified at various levels of abstraction. As shown in figure 1, one first perspective defines the end user and the application system as two participants. When considering the application system per se, the functional core, the dialogue controller and the presentation components are cooperative participants which make it possible the interaction with the end user. When refining one step further, the presentation component may be viewed as a set of agents, also called presentation objects, which act as participants for implementing the user interface [Coutaz 87b].

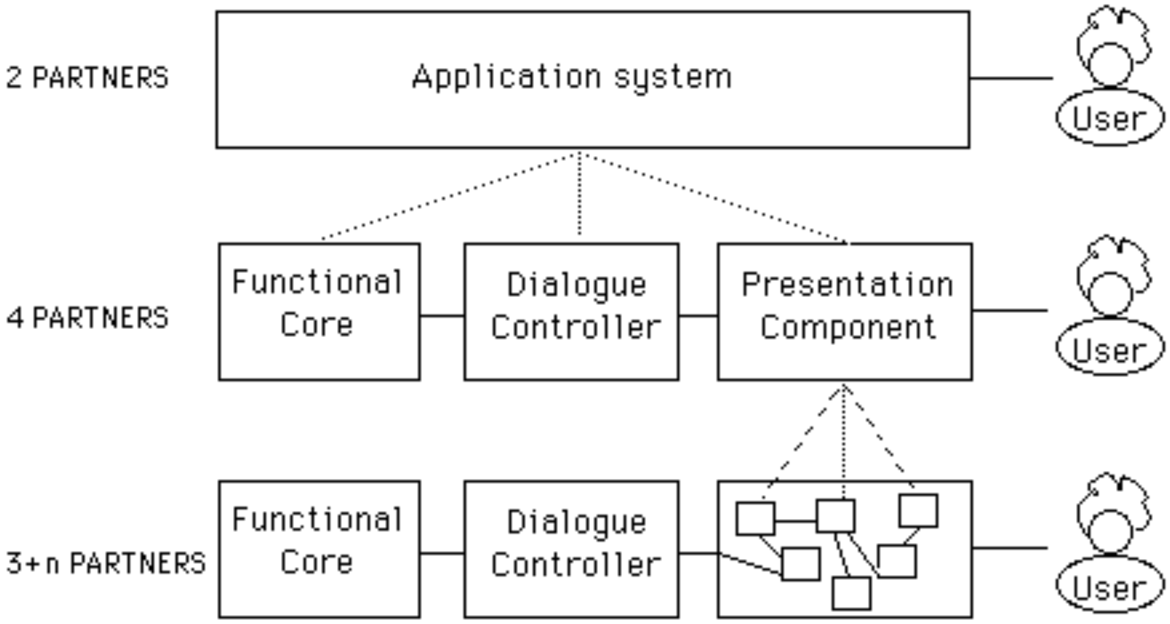

Figure 1: *The levels of partners in an application system.*

The successive refinement of participants into subparticipants indicates that the interaction is distributed at various levels of abstraction among multiple concurrent activities. The sequential pipe-line stream of computation suggested by the linguistic interpretation of the Seeheim model, is now replaced by a set of cooperating agents acting at various levels of abstractions. At the highest level, the application implements "domain dependent" concepts in a "presentation independent" way. At the other end of the spectrum, "toolkit dependent" agents present domain concepts in a "domain independent" way.

"Independence" does not mean lack of communication nor absence of mutual knowledge. Actually, independence refers to the notion of reusability. In order to be reusable, a software component should not make any direct reference to any other component. Communication is then performed with a combination of dynamic linkage and indirection. In the particular case of an interactive system, the dialogue controller is in charge of the indirect communication between the functional core and the presentation agents. It is not a syntax analyzer as advocated by the linguistic approach. It is the necessary mediator between two computational worlds. As such, it must be able to provide two categories of services: control and translation.

- Conceptually, the functional core and the presentation objects are animated by concurrent activities. Each one maintains the cooperation between its own internal activities according to its own rules; But the overall functioning of the computer system requires the adjustment of the relationships between the functional core and the presentation. The central position of the dialogue controller makes it the natural arbitrator for such a task.

- The functional core models the solution to a problem at a high level of abstraction and communicates with the outside world using its own formalism. The presentation models its relationships with the outside world in a formalism adequate to a particular type of interaction. It includes general purpose toolkit objects which may not be tailored to the particular needs of the functional core. More generally, the functional core and the presentation define two perspectives of identical concepts (that of the task domain), and, for doing so, they use different data types, different formalisms. Thus, a translation between the two formalisms must be performed somewhere. If independence between the internal modelling of the domain concepts and their presentation to the end-user must be preserved, then the translation must be performed outside of the functional core and of the presentation. The central position of the dialogue controller makes it the natural candidate for such a task.

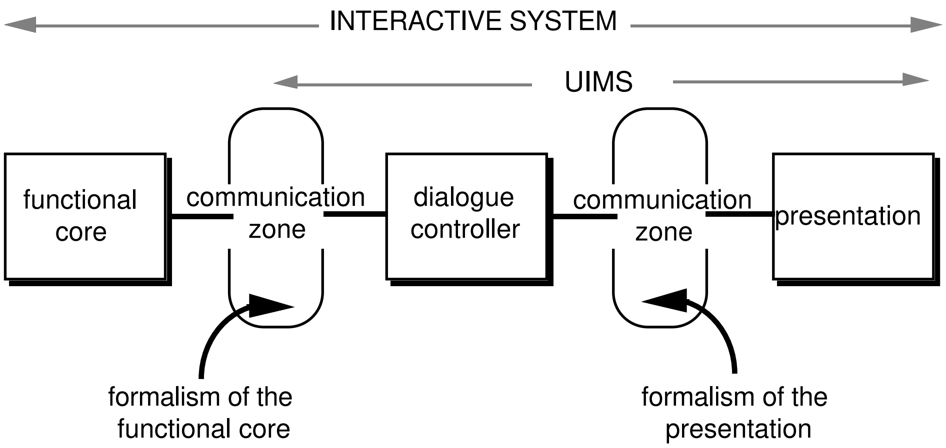

***Figure 2**: The overall architecture of an interactive system.*

Figure 2 illustrates the principles of the transfer for the particular case of UIMSs. The dialogue controller is surrounded by two communication zones. The first one uses the formalism of the functional core whereas the second one uses that of the presentation. The controller performs the necessary transformations between the two. Generally speaking, translation between formalisms refers to the notion of type conversion. When the functional core and the presentation are expressed in different programming languages, an additional transfer mechanism is needed to switch between implementation languages. Since type conversion theory is not sufficiently advanced yet, ad-hoc technics may be applied. In particular, automatic type conversion is replaced with indirection and ad-hoc computations. Indirection defines the relationship between the types handled in the functional core and that of the presentation objects; ad-hoc computations perform the necessary transformation between source values and destination values.

As shown in Figure 3, the transfer between formalisms may be performed through two indirections on each side of the controller. The drawback is the increasing cost of the transfer. The benefit is the existence of a reusable kernel independent from the formalisms used in the functional core and in the toolkit.

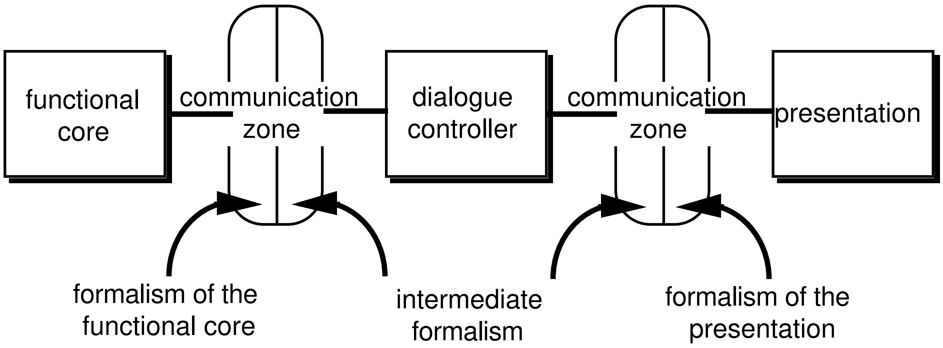

Figure 3*: Transfer between formalisms through double indirection.*

The framework presented in figure 2 and the embedded parallelism described above, identify three strategic locations for the definition of general requirements for UIMSs: requirements for the control of parallel activities and requirements for the two communication zones. In addition to the requirements specific to particular components, we need to investigate the reusability of "universal services" such as cut-and-paste and undo-redo facilities.

3. Dialogue Control: Requirement for Interleaving at Multiple levels of Abstraction

The successive refinement shown Figure 1 indicates that the overall control of a dialogue is determined by the policies defined at each level of abstraction. Let's consider the desirable policies at the three levels identified so far : between the end user and the application system, between the three main components of the application system and inside the presentation component.

3.1. Policy between the end user and the application system

Dialogue between an end user and an application system may be either user driven or system driven. The choice between these two styles depends on the nature of the application system as well as on the model designers have elaborated about the end user. The distinction between the styles lies in the number of semantically valid inputs the end user might possess at any point in time. If this number is always 1, then the end user is lead from subtask to subtask without any decision power at the task level. If the number is greater than 1, the end user is able to dynamically update his task plan in a more opportunistic way. He may also initiate and accomplish several subtasks in parallel. Since a task (or a subtask) involves domain dependent computation, such parallelism requires modelling multiple threads of dialogue in the functional core.

A common case of multiple threads of dialogue occurs in systems which provide extensive verifications and suggestions about the task domain. When a semantic error occurs, the functional core may reject the end user's action and require an immediate correction, or it may accept the end user's action and maintain a model of the error. In the first case, the end user has to enter a planning phase to correct the error and restart the subtask from scratch. There is no such notion as "a pending task". In the second case, the functional core accepts partially accomplished subtasks by remembering semantic errors. By doing so, it maintains multiple threads of task accomplishment.

Clearly, a functional core which has only one thread of input is easier to implement but it imposes a task decomposition on the end user which may not be "natural" and sometimes unnecessarily constraining. The collaboration between an application system and the end user is thus constrained at the semantic level by the functional core. UIMSs are of no help for such purpose but they should be ready to support multithread functional cores.

3.2. Policy between the functional core and the dialogue controller

Deciding which component of the application system controls the interaction may be approximated by considering the following classification: external, internal, or mixed control [Tanner 83]. The control is internal when it resides in the functional core, that is: the interaction is driven by computational considerations, and the UIMS is modelled as a presentation server. The control is external when it is maintained in the UIMS: the interaction is driven by the end user's input events and the functional core is viewed as a semantic server. The control is mixed when it is alternatively handled by the functional core and the UIMS.

In general, mixed control is most appropriate. Mixed control allows the functional core and the components of the UIMS to be equal participants. The server model, implicit in internal and external control, biases the balance of the interaction. With such models, the interaction is driven either by the functional core or by the end user. Such systematic behavior is in contradiction with reality. For example, during the specification of a mission for a mobile robot, the interaction is typically user driven. At this stage, the presentation component is the main controller of the application system: it acquires the events which express the end user's actions and the functional core is invoked for checking the feasibility of the mission. On the other hand, during mission execution, the functional core (i.e. the robot) is typically in charge of the interaction. Although, at this stage, the end user is essentially an observer, he is allowed to act on the progress of the mission. The consequence for the application system is that although the functional core performs relevant computations and sends messages to the UIMS to express the current state of the robot, the presentation component must listen to and process the end user's inputs. Another example where the non-mixed model of control breaks down can be found in the Macintosh Toolbox.

The Macintosh Toolbox imposes an external control of the interaction. A number of primitives, such as DragGrayRgn and MenuSelect, keep control of the interaction until the mouse button is released, tracking the mouse and performing lexical feedback as needed [Rose 86]. Although this level of service is convenient for the programmer, the CPU is entirely dedicated to the end user. In order to take other events into account, such as reading characters from a serial line, the Toolbox provides hooks which allow a particular routine to be called repeatedly while the mouse button is down. Clearly, this is an ad-hoc solution, not a general policy. Another example where mixed control is required concerns direct manipulation user interfaces.

In direct manipulation user interfaces, the form of an input expression may evolve in parallel with the production of an output expression in order to provide immediate and meaningful feedback. In particular, during the specification of a command, inputs and outputs may be interleaved. Rubber banding is a common example. Other examples include the Macintosh desktop Finder. In Finder, moving an icon across the screen is a partial input expression until the mouse button is released. The end user's action denotes the command name (i.e. copy) and a file name (i.e. the file being pointed at) but the destination recipient is missing. Although the command is partially specified, the system is able to provide the end user with two complementary forms of immediate feedback: a lexical feedback (an icon outline following the mouse cursor), and a semantic feedback (the reverse video of icons representing valid recipients). As discussed earlier in Section 2, sequential models which imply complete input expressions before invoking any semantic actions make these exchanges impossible.

In summary, the internal and external locations of control are special constrained cases of a more general view where the components of an application system are equal participants. In addition, the internal-external approaches suggest a sequential model of computation which may be convenient for simple application systems but which make it hard to support high bandwidth communication between the UIMS and the functional core.

3.3. Policy inside the presentation component

The presentation component, which expresses the state of the functional core, should be able to map the multithread facilities (if any) supported in the functional core. To do so, multiple interaction objects should be simultaneously available to the end user and manipulable in any order. In addition to reflecting the internal multithread structure of the functional core, the presentation component may also support interleaving between presentation objects. Interleaving inside the presentation component allows the end user to act on a presentation object, for example specify a couple of fields in a form, then manipulate other presentation objects, and come back later to the previous one. Clearly, this facility supports opportunistic planning.

In summary, interleaving supports dialogue flexibility at several levels of granularity. At the task level, that is between the functional core and the end user, several threads of reasoning may be carried out in parallel. At the command level, that is between the components of an application system and the end user, immediate and domain dependent informative feedback may be provided. At the action level, that is between the presentation objects and the end user, focus of attention can switch from presentation object to presentation object at the user's will.

3. Requirements on the Protocol with the Functional Core

A protocol for transferring information between two communicating entities requires the definition of a common data format which can be understood by both partners. This format embeds a level of abstraction as well as a meaning to signal or provoke state changes. This section is concerned with these two aspects, level of abstraction and expression of state changes, for the particular case of the functional core communicating with the dialogue controller. This protocol is sometimes called the Application Programming Interface (API).

After a presentation of the general principles underlying the definition of an API, we discuss the pros and cons of the declarative and procedural descriptions for communicating data between the functional core and the dialogue controller.

3.1. API: General Principles

The nature of an API is driven by the capabilities of the computing technology as well as by principles derived from cognitive psychology. Immediate and informative feedback is such a driving principle. It requires the API to handle a communication at a high level of abstraction and at a high bandwidth.

3.1.1. A High Level API

The principle of separation between the functional core and the user interface is motivated by software engineering considerations: modularity provides the basis for iterative refinement. In turn, iterative refinement of the user interface requires that the functional core be "presentation independent", that is: data exchanged with the dialogue controller should not contain any reference to the presentation objects. This requirement is satisfied only if the API protocol is high level.

A high level API means that data exchanged between the functional core and the UIMS are expressed at the level of abstraction defined by the functional core. For example, a functional core which uses a real number to model the notion of temperature, should communicate with the UIMS in terms of real numbers; it should not have any knowledge about the presentation of the real value to the end user.

A high level API is one principle on which computer scientists agree. Linking high level data structures and data types to presentation objects relies on using the dialogue controller as the mechanism for indirection. On the other hand, expressing the dynamics at a high bandwidth and at a high level of abstraction is still a topic for research.

3.1.2. A High Bandwidth API

The separation advocated so far stresses the difference in the formalisms and the levels of abstraction used in the functional core and in the UIMS. It does not mean a lack of communication at run time! As mentioned earlier, interleaving is an unavoidable phenomenon which concerns any component of the application system, including the API.

The API should allow the functional core to express state changes meaningful to the end user that is, state changes of the semantic concepts relevant to the current task. For example, the state may include a boolean to indicate that the notion of temperature is not currently modifiable by the end user. Such a condition may be determined by the functional core on the basis of domain dependent information. The UIMS is then in charge of translating the unavailability of the concept into the presentation. For example, the user might not be able to move the thermometer, which presents the notion of temperature, towards a source of heat.

3.2. The Declarative/Procedural Description of the API

The expression of the high-level dynamics can be performed according to three techniques: the declarative model (as in Mickey [Olsen 89]), the procedural paradigm (as in ApEx [Coutaz 87a]), or as a mixture of both (as in Serpent [Bass 88]). Generally speaking, a declarative expression specifies "the what" (i.e. the desired result), whereas a procedural expression describes "the how" (i.e. the computation leading to the result).

3.2.1. Declarative and procedural API: Definitions

In the area of UIMSs, a declaration allows the designer of the functional core to define the data structures and the functions which can be referenced at run time by the UIMS. It also includes the expression of the dynamic semantics, such as preconditions and postconditions, for access control to the functions and the data structures of the functional core.

The procedural approach defines the API as an abstract machine whose instruction set allows the functional core to dynamically express changes relevant to the external world. Since the API is high level, the operands of such instructions are data structures of the functional core. For example, this instruction set would include operations such as:

- "write data" and "read data" to respectively signal that the data structure "data" (e.g. temperature) has a new value, and needs a new value,
- "invalidate data" to indicate that "data" should not be modified or invoked until a "validate data" is issued,
- "create data", "destroy data" to express the creation of an instance of a concept with the initial value "data", or to signal its destruction, etc.

The declarative and the procedural descriptions share the same level of abstraction and thus, guarantee the same degree of mutual independence between the functional core and the UIMS. However, neither of them is fully satisfactory.

3.2.2. Drawbacks of Procedural API

Although it is high level, a procedural API does not facilitate the modification of the behavior of the functional core. This behavior, with regard to the dialogue controller, is explicitly expressed in the code of the functional core by calling API instructions. On the other hand, the declarative approach, in essence, implies that the specification be performed outside the code of the functional core, making these adjustments easier.

A procedural API may require more exchanges between the functional core and the UIMS than the declarative approach. At run time, the declarative specifications are linked to the UIMS. By doing so, the UIMS is able, in some circumstances, to perform semantic checking without involving calls to the functional core.

A procedural API does not lead to the automatic generation of user interfaces. In UIMSs such as Mickey [Olsen 89], MacIda UIMS [Petoud 89], and ITS [Bennett 89], default presentation objects can be automatically elaborated from the definition of the data structures exported by the functional core. For example, a record structure may be presented by the UIMS as a dialogue box, an enumerated type may be linked to a list of radio buttons, and a function may be mapped to a menu item.

3.2.3. Drawbacks of Declarative API

A declarative API may prevent the designer from accessing details that would be otherwise useful to express highly dynamic events. This is a general problem common to specification languages: either they are too high-level (they may hide details that one would like to control), or they are too low-level (they require too much effort to express an action). The class of events which poses an interesting problem is the creation and destruction of objects. For example, an error detected by the functional core results in the creation of a semantic object which fully and symbolically describes the reasons of the failure. The job of the UIMS is to create an object specialized in the presentation of errors. In this example, the difficulty is to statically specify the conditions which will provoke, at run time, a failure ascribable to the functional core only. In the extreme, why not to describe the entire functional core within an API specification? This question leads to the next problem.

At run time, the declarative model may duplicate semantic checking and therefore, may reduce system performance. Any functional core needs to maintain its own state according to its own representation system. This mechanism is domain specific and thus cannot rely on the general purpose API specification. For example, the usual implementation for a shared buffer verifies that no two processes are simultaneously delivering information into the buffer. In order to show the end user that the buffer is currently unavailable for deposit, the UIMS must execute the preconditions provided by the specification. By doing so, the user is always informed of the semantically valid actions. On the other hand, his actions, which are necessarily permitted, are checked redundantly by the functional core.

Considering the respective pros and cons of the two techniques, we believe that high quality semantic feedback is better served with the procedural approach but that configuration management is, by far, better supported by a declarative language. To conclude, a mixture of the two approaches appears to be the most adequate compromise. The difficulty is to identify what should go into the declarative part.

4. Requirements on the Protocol with the Presentation

The presentation of an application system should clearly convey the semantics and the state of the functional core, and should be able to support multiple views. These two aspects are discussed in this section.

4.1. Conveying the semantics and the state of the functional core

"Appropriate" presentation objects are selected at design stage for conveying the semantics and the state of the functional core. At implementation stage, we are concerned with the expression, in a graphical or other form, of the design decisions. Output facilities provided by graphics packages rely on rendition attributes such as color and size, and on the definition and the maintenance of constraints such as alignments.

The problem for a UIMS is to be able to control such attributes and relationships along two directions: first, for expressing the state of the functional core to the end user; second for interpreting the actions of the end user in terms of the functional core. Thus, the behavior of presentation objects should be controlled on the basis of data coming from two opposite sources: the functional core and the end user. Because of this double origin, it is tempting for the implementer to wire data from the functional core into the presentation objects and conversely, reference rendition attributes in the functional core. This type of explicit dependence makes it difficult to perform and manage configuration changes.

Tools for the declarative expression of relationships between data of the functional core and behavior of presentation objects alleviate the problem of explicit dependence. A specification, which is interpreted or transformed by a special purpose abstract machine, is not embedded in the code of the functional core nor is it in presentation objects. It sits between. Thus, the specification of the relationships between the functional core and the presentation objects may be easily modified. Serpent [Bass 88] and Garnet [Myers 89] both include a language for elaborating such specifications.

4.2. Supporting Task Accomplishment with Multiple Views

The notion of multiple views allows the end user to observe and manipulate a concept through several perspectives according to his current focus of attention. The problem for the UIMS is to guarantee the consistency between the perspectives. For example, if the end user of a text editor erases a sentence from a document, the same sentence, visible in another window, should also be erased (unless this is not desired). In any case, the focus of attention of the end user is a unique conceptual object (a particular sentence) whereas the UIMS has to deal with at least two presentation objects (i.e. one per perspective).

In summary, a UIMS should include mechanisms for expressing the dependence of the behavior of presentation objects on data produced by the functional core and the end user. This requirement, which is in conflict with the principle of independence between the user interface and the functional core, may be satisfied through special purpose declarative tools. A UIMS should also include a mechanism for supporting consistency between multiple views.

5. Requirements for "Universal" Services

Experiments from cognitive psychology have demonstrated the usefulness of general services such as Cut-and-Paste, Undo-Redo and Help. These facilities involve the functional core as well as the UIMS in manners which we now discuss.

5.1. Cut-and-Paste

Cutting, copying, and pasting information may happen either inside a single use of an application system, or between multiple usages of the same application system or with different application systems. Windowing systems define a format (or a set of formats) for data transfer. For example the Macintosh toolbox supports a wide range of formats since it allows application programmers to define their own protocol. However, the cut-and-paste facilities provided by windowing systems do not address the difficult aspects of the transfer of data.

The difficult portion is the interpretation of the incoming data by the recipient. In most current implementations, the recipient is the functional core. The problem with this approach is that the level of abstraction of the data transferred through the "cut-and-paste" buffer generally does not match the level of abstraction of the data structures used for computation in the functional core. In particular, the formalism used to express the content of the "cut-and-paste" buffer relies heavily on layout considerations. As a result, the functional core must maintain a translation process to convert from a layout specification to a functional specification. This responsibility should lie, instead, with the dialogue controller which acts as the mediator between the functional core and the presentation.

Delivering data at a level of abstraction usable by the functional core is an important step towards presentation independence. However, the new incoming data must be checked by the functional core for semantic validity. One important and yet unsolved problem in the area of structure editing is a general mechanism for type checking and type recasting. For example, in a structured document editor, the end user may cut a section from one document and paste it back into a second document at a location where only subsections are expected. Clearly, the functional core should be able to recast the incoming section as a subsection.

In summary, cut-and-paste facilities involve three levels of abstraction:

- the presentation component which gets data through the cut-and-paste service,
- the dialogue controller for the translation process between the format used in the transfer protocol and the functional core internal data structures, and
- the functional core for type checking and type recasting.

5.2. Undo-Redo

Undo and redo facilities, which respectively undo and redo the effect of "the last command", both rely on the existence of a history. A history is a mechanism for remembering the most recent commands. In this discussion, a command should be interpreted as an end user action which results in a change in the application system state. As shown earlier, the application system includes a number of participants. Its state is therefore defined as the union of the states of the various participants: the semantic partner (i.e. the functional core) and the syntactic partners (i.e the presentation objects and the dialogue controller).

Undoing a semantic action relies on the services of the functional core. Undoing a syntactic action (such as resizing, scrolling or moving a window) relies on the services of the presentation objects. The semantics of undo-redo relies on the maximum size of the stack of saved commands maintained by the history. The stack size may be limited to 1 (as in the Macintosh environment) or parameterized (as in the Unix Shell). The breadth of undo-redo depends on the extent of the history mechanism.

The history may be viewed as a global service, it may be distributed across the interactive system, or it may be available as a combination of both. In the first case, the "last command" is the absolute last action of the end user, whereas in the distributed basis, the undoable command is relative to the current presentation object.

The Macintosh implementation of MS-Word is an example of a mixture of global and local histories: semantic commands are global to the application system whereas syntactic commands are local to interaction objects. MS-Word allows the end user to open several documents simultaneously. The last undoable semantic command is made available as an item in the Edit pop-up menu. If, for example, the user "pastes" some information in the first document, then the menu item makes it explicit the nature of the undoable function: "undo paste". Now, if the end user switches to a second document, the undoable command is still "undo paste" although the paste action does not concern the current document. On the other hand, windows, which are interaction objects, remember their previous size. As a result, undoing a window size modification is possible at the window level.

In summary, undo-redo facilities involve three levels of abstraction:

- the presentation component for maintaining local histories and reverse functions specific to presentation objects,
- the dialogue controller to maintain a global history or a local history for commands with semantic side effect,

- the functional core for implementing the reverse functions.

5.3. Help

Like history, help facilities may be available globally or locally. In addition, help is very much concerned with effectiveness: it has to provide the end user with a useful support. This goal is hard to achieve for three main reasons:

- it requires the definition of a model about the functioning of the application system and a model about the end user. Typically the system model is useful for providing procedural knowledge: given a question such as "how can I do such and such thing?", the help system would be able to suggest a plan of actions. The model about the end user is useful for devising the form of the answer, in particular for determining the level of details to be provided.

- it requires access to the state of the application system, which, by definition is distributed across multiple participants. State knowledge is helpful for elaborating answers relevant to the current situation.

- it involves the definition of a query language. Should it be a natural (spoken or written) language ? Should it be yet another language to learn? Etc.

In summary, full fledged help for effective problem solving is desirable but requires the definition of a query language, the development of an automatic planner based on a knowledge base. However, simple static help may be provided at minimal cost by assigning help information to every presentation object in the application system.

6. Summary

In summary, requirements for UIMSs boil down to the following topics:

• stress parallelism in order to support the cooperation of agents acting at various levels of abstraction. In particular, the functional core and the presentation are animated by concurrent activities. They should be viewed as equal participants (and not as servers which biaised the balance).

• consider the dialogue controller as a mediator between two worlds and not as a syntactic analyzer. Basically, this mediator includes translation and linkage mechanisms between different perspectives of identical concepts. If we consider implementation issues, surround the controller with two communication zones: one for the formalism used by the functional core and one for the presentation portion.

• define a protocol to access the communication zones.

- Considering the communication zone of the functional core (the API), the dialogue controller may be implemented as a combination of two abstract machines. The first one supports a "procedural API". It interprets simple requests such as read, write, create, invalidate, which are performed at run time by the functional core. The second machine is a constraint system which checks that "declarations" expressed previous run time are maintained permanently at run time.

- Considering the communication with the presentation portion, the dialogue controller may be viewed as an abstract machine which is able to interpret the expression of dependence between concepts from the functional core and abstractions of the presentation objects.

• include general purpose services: cut-and-paste, undo-redo, history, help. As stressed in the paper, general purpose services involve the three basic levels of abstraction: the presentation component, the dialogue controller and the functional core. One difficulty is the lack of theoretical results. Another difficulty is to convince designers and implementers of functional cores to provide such services.

In addition to the fundamental requirements presented in this paper, we need to envision a more general view of the development process of user interfaces. In particular, if the UIMS is part of a User Interface Development Environment (UIDE), then we need to consider additional run time UIMS features. In particular, we need to integrate into the UIMS an observer which registers the behavior of the end user both at the task level and the action level. Such facilities would automatically infer the difficulties encountered by the end user and support the iterative process of user interface refinement.

7. Bibliography

[Bass 88]
L.J. Bass, E. Hardy, K. Hoyt, R. Little, R. Seacord: The Serpent run time architecture and dialogue model; Carnegie Mellon Technical Report, CMU/SEI-88-TR-6, January, 1988.

[Bennett 89]
Bennett, W.E., Boies, S. J., Gould, J. D., Green, S. L., Wiecha, C. F. "Transformations on a Dialog Tree: Rule Based Mapping of Content to Style", Proceedings ACM SIGGRAPH Symposium on User Interface Software and Technology, Williamsburg, Virginia, Nov, 1989, pp 67-75.

[Cardelli 87]
L. Cardelli: Building User Interfaces by Direct Manipulation; Digital Systems Research Center, Technical Report, 22, October, 1987.

[Coutaz 87a]
J. Coutaz: The Construction of User Interface and the Object Oriented Paradigm; ECOOP'87, The European Conference on Object Oriented Programming, Paris, June, 1987, pp. 135-144.

[Coutaz 87b]
J. Coutaz: PAC, an Implemention Model for Dialog Design; Interact'87, Stuttgart, September, 1987, pp. 431-436.

[Coutaz 89]
J. Coutaz: UIMS, Promises, Failures and Trends; HCI'89, People and Computers V, A. Sutcliffe, L. Macauley eds., Cambridge University Press, 1989, pp. 71-84.

[Foley 88]
J.D. Foley, W.C. Kim, S. Kovacevic, K. Murray: The User Interface Design Environment, Technical Report GWU-IIST-88-04, Department of Electrical Engineering and Computer Science, The George Washington University, Washington D.C. 20052, January, 1988.

[Hayes 85]
P.J. Hayes, P. Szekely, R. Lerner: Design Alternatives for User Interface Management Systems Based on Experience with Cousin; Proceedings of the CHI'85 Conference, The Association for Computing Machinery Publ., April, 1985, pp. 169-175.

[Hullot 86]
J.M. Hullot: SOS Interface, un Générateur d'Interfaces Homme-Machine, Actes des Journées Afcet-Informatique sur les Langages Orientés Objet, Bigre+Globule, 48, Publ. IRISA, Campus de Beaulieu, 35042 Rennes, janvier, 1986, pp. 69-78.

[Ilog 89]
Ilog: Aïda, Environnement de développement d'applications; Manuel de référence Aïda, Version 1.32, Copyright Ilog, mars, 1989.

[Karsenty 87]
S. Karsenty: Graffiti: un outil interactif et graphique pour la construction d'interfaces homme-machine adaptables; Thèse de doctorat de 3ème cycle Informatique, Université de Paris-Sud, Centre d'Orsay, décembre, 1987.

[Myers 89]
B.A. Myers: Encapsulating Interactive Behaviors; CHI'89 Conference Proceedings, Special Issue of the SIGCHI Bulletin, ACM Press, 1989, pp. 319-324.

[Olsen 89]
D. R. Olsen: A Programming Language Basis for User Interface Management; CHI'89 Conference Proceedings, special issue of the SIGCHI Bulletin, ACM Press, pp.171-176.

[Petoud 89]
I. Petoud, Y. Pigneur: An automatic Visual Approach for User Inetrface Design; Proceedings of the second working conference of IFIP WG 2.7 on Engineering for Human-Computer Interaction, Napa Valley, 21-25 August, 1989.

[Pfaff 85]
User Interface Management Systems; G. E. Pfaff ed., Eurographics Seminars, Springer-Verlag, 1985.

[Rose 86]
C. Rose et al.: Inside Macintosh; Addison Wesley Publ., 1986.

[Schmucker 86]
K. Schmucker: MacApp: An Application Framework; Byte 11(8), 1986, pp. 189-193.

[Schulert 85]
A.J. Schulert, G.T. Rogers, J.A. Hamilton: ADM - A Dialog Manager; Proceedings of the CHI'85 Conference, The Association for Computing Machinery Publ., April 1985, pp. 177-183.

[Tanner 83]
P. Tanner, W. Buxton: Some Issues in Future User Interface Management Systems (UIMS) Development. IFIP Working Group 5.2 Workshop on User Interface Management, Seeheim, West Germany, November, 1983.

[Webster 89]
B. F. Webster: The NeXT Book; Addison Wesley, 1989.