28

THE MYTH OF THE SEPARABLE DIALOGUE: SOFTWARE ENGINEERING VS. USER MODELS

Hans Wegener

GMD SET/SKS, Schloß Birlinghoven, D-53754 St. Augustin, Germany
Phone: +49-2241-14-2840, Internet: hans.wegener@gmd.de

KEYWORDS: User Interface Design, Dialogue, Reusability, User Model, Architecture Model, Factorisation, Refinement, UIMS.

ABSTRACT: Conceptual models of user interfaces and object-oriented models of interactive systems appear to be incompatible as regards the concept of dialogue. A case study reveals that the structure of components identified according to these two perspectives cannot be brought into proper correspondence. Hence, traditional UIMSs must be considered unable to provide separable, reusable abstract data types for the dialogue. Conclusions are drawn concerning the architecture of UIMS systems.

When designing interactive applications, developers traditionally make use of *user interface architectures* in order to identify the technical components to be separated. On the other hand, we have *conceptual models* (Norman, 1988) that were designed to help users to express expert knowledge about the application domain. To what extent do these two perspectives conform?

As reference, I chose the IFIP-Model for user interfaces (Dzida, 1987), which focuses on the user's conceptual model and claims to offer the advantage of a natural correspondence between user concepts and technical artifacts. It furthermore highlights the role of the dialogue. Four aspects of interactive applications can be identified: input/output, dialogue, tool and organizational environment (see Figure 1). *Application-independent concepts* (Dzida 1987) can serve as fragmentary components for building user interfaces. In the form of abstract data types they can be employed in the development of user interface tools. A survey yields the following result:

- *Input/Output:* push buttons, menus, icons, text fields and the like can be found in almost any application. We have a concept of widgets.
- *Dialogue:* Nothing to be found, yet.
- *Tool:* generic objects such as stacks, arrays, graphs or streams are application-independent concepts. In addition to these, Dzida identifies generic tools like cut, copy, move or paste (Dzida 1987).

Widgets have successfully been turned into data types, as well as generic objects. Generic tools can be realized through *aspect classes* (Budde et al., 1992). It would certainly prove beneficial to add application-independent dialogue concepts to the list, because application development could be shortened. However, we have to adhere to software reuse principles.

Several architecture (or implementation) models have already been proposed. Among them is the MVC-Paradigm (Goldberg, 1990), the PAC-Model (Coutaz, 1987) and the model suggested by Jacobson (Jacobson et al., 1992). The three components of the former do not directly correspond to the IFIP aspects. For example, the view in MVC only corresponds to the output, whereas the control contains input and dialogue aspects. A similar statement holds for the PAC-Model. As regards the Jacobson-Model, we find that interface objects implement aspects of input, output and dialogue. Entity objects implement dialogue and tool aspects. Jacobson adds a third kind of objects, the controls, which can't be attributed clearly to any of those of the IFIP-Model; he does so in order to avoid too high a complexity in the application architecture and does not structure these objects in any way.

DESIGNING REUSABLE SOFTWARE

The three basic techniques for designing reusable software (not to be confused with reuse principles) are the *use*, the *factorisation* and the *refinement* of data types. In object-oriented technology, this is

K. Nordby et al. (eds.), *Human-Computer Interaction*

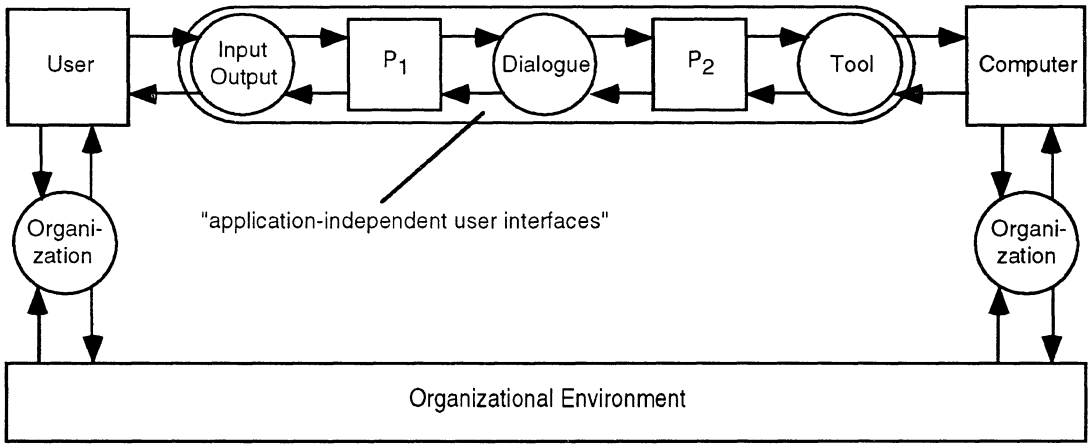

Figure 1: The IFIP WG 6.5 model of user interfaces. The input/output aspect defines rules for translating the user activities into symbolic signals (e.g. keyboard shortcuts, graphical appearance). The dialogue aspect is concerned with the sequence of activities and how the system responds regarding feedback or use of the available tools (e.g. that the user first selects an object and afterwards the operation to be performed on that object). The tool aspect, at last, defines how the tools behave and how they can be accessed (e.g. deleting an object causes it to be marked for later deletion, but not to be deleted immediately). User and computer are also integrated into the organizational environment, but that is of no relevance here. The boxes labeled P_1 and P_2 signify activities, i.e. processing, circles signify means of processing, arrows mark an information flow.

realized employing *classes*, *superclassing* and *subclassing*. A small example will demonstrate how this is usually done.

Imagine an application for drawing and manipulating graphical objects. We would declare classes `Object`, `Rectangle` and `Circle` in C++ as follows:

```
class Object {
  int x, y;
  virtual draw ();
};

class Rectangle: Object {
  int width, height;
  draw ();
};

class Circle: Object {
  int radius;
  draw ();
};
```

The class `Rectangle` uses the classes `Object` and `int` and composes them to form a new data type; `Circle` does so likewise The method `draw` has to be implemented for `Rectangle` and `Circle`, so it makes good sense to factorise it by moving it into the common superclass `Object`. The actual drawing algorithms will be refined in the two subclasses, but we now have a unique form of access for all `draw` methods. Any other class that uses instances of `Object` and desires to draw them, simply calls `draw` and the appropriate method will be executed.

A software-engineering principle now tells us to factorise as many features (attributes or methods) as possible, thereby moving them upwards in the class hierarchy. In this way, the most abstract, i.e. most reusable design is found.

Equipped with this knowledge, we may try to design an example application according to the IFIP-Model and according to software engineering principles.

AN APPLICATION SHOWCASE

Imagine a simple calendar application which schedules your appointments. If you enter a new appointment, the date and time must be provided. The application checks whether this date is still free and returns an error message if not. A schematic user interface can be seen in figure 2. According to the IFIP-Model, we have three interface descriptions:

- *Input/Output:* Defines the graphical appearance of the output in figure 2, a description of how mouse interaction behaves (e.g. that the user signals confirmation by clicking on the Enter

button), as well as how keyboard input is handled.

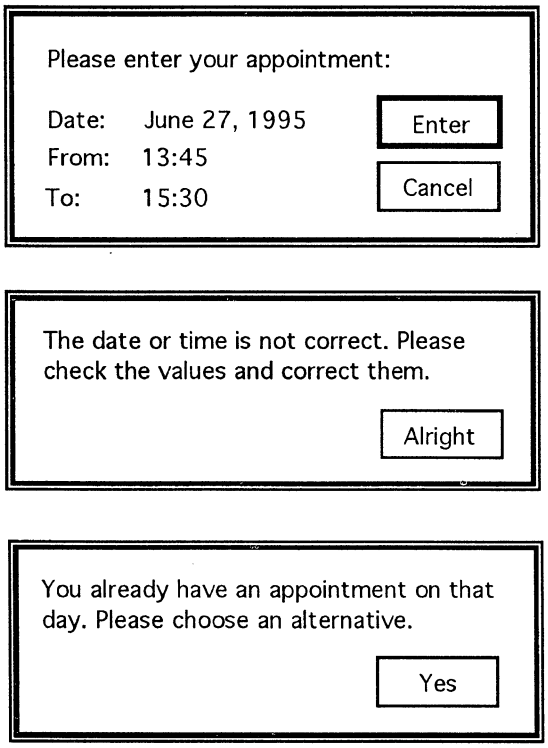

Figure 2: Graphical presentation of the example application. On top, we see the central input form for the user. If error messages must be presented, the two boxes below will be displayed.

- *Dialogue:* As described in figure 3; the user must enter the appointment date and time, after which a formal check for correctness is done (e.g. that the user doesn't enter a date like February 30). Only if the specified date is free (i.e. there's no other appointment at that time) will the appointment be registered. Otherwise, another error message is given.
- *Tool:* Registers the appointment in the database.

As can be seen, there is a clear separation between the three aspects. How would the same application be constructed according to architecture models, like the one suggested by Jacobson?

- *Interface:* We would declare a simple class `CalendarInterface` that describes the graphical appearance of the widgets in figure 2 and handles the interaction concerning filling in date and time or dismissing the error messages. Particularly, this class owns a method `ok` that is called when the user clicks on the Enter button.
- *Control:* When returning from interaction, the control part checks the formal validity of date and time. Given that, it continues to check whether the appointment can be made (using method `isFree`, see below). If any of the above assumptions are wrong, error messages are output. Otherwise, the appointment is registered using `register`. The class for the controller simply reimplements the `ok` method of the interface class and uses the entity classes.
- *Entity:* A class hierarchy for the appointment database, providing a method `register` (in the class implementing the database) to register an appointment and `isFree` to check whether the desired date is still free.

When returning from interaction the interface widget provides, for example, an unformatted date. It is interesting to discover now that the formal check can be done independently of the semantic check, so we could redefine the interface component as follows: check the date formally and call the controller method `ok` only if it is correct, otherwise display the appropriate error message. In this way, we factorise the controller behaviour of formally checking the date and time, and move it into the interface using refinement.

The rest of the dialogue aspect (according to the IFIP-Model) remaining in the controller is the semantic check. However, we could equally well move the semantic check into the method `register` that stores the appointment. *This is just a matter of design decision.* The semantics of this newly defined method would be to check for correctness (using `isFree` and probably prompt an error message) and then store the appointment. Again, we use factorisation and refinement to accomplish that.

This case study reveals that the dialogue aspects we have identified from the user perspective have (due to factorisation and refinement) been moved into parts of the architecture that were originally dedicated to input/output or tool aspects. Actually, we find that the controller semantics has become void.

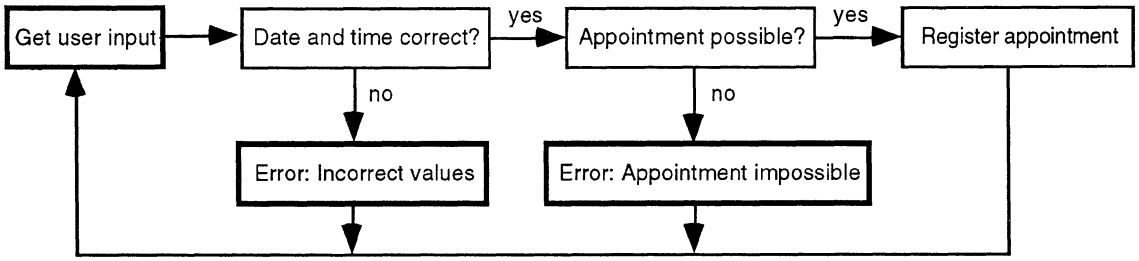

Figure 3: Dialogue structure of the example application. For each of the box-shaped dialogue steps involving user interaction (thick borders), we find the according input/output aspects in figure 2.

CONCLUSIONS

As has been demonstrated, user models and software-engineering principles produce different results. In particular, the architecture of our technically motivated design scatters parts of the code responsible for handling dialogue aspects across application components originally not dedicated to it. This is in contradiction to the user-oriented design, which models and evaluates the dialogue separably. We are therefore faced with the question what conclusions can be drawn with regard to application domain modeling from the user perspective (Dzida, 1984).

Technical architectures and conceptual user interface models are sometimes *not compatible*, due to reuse considerations. It is questionable whether it makes sense to design applications according to user models like the IFIP-Model. This has particularly been attempted with the Seeheim-Model, the leading model for UIMS design (Green, 1985, Olsen, 1992). From the software engineering perspective we may doubt that there can ever be reusable components that separate the dialogue as in the user model. But since every UIMS employs some kind of dialogue model (Green, 1986), we must ask whether current UIMSs can produce reusable dialogue components.

If we take a look at interactive systems designed according to architectures invented with reuse in mind, like the MVC-Paradigm (Goldberg, 1990) or the PAC-Model (Coutaz, 1987), we discover that they don't have any dialogue concept at all that could be compared with the IFIP-Model. Therefore, we must strive for intelligent design decisions that keep the user perspective in contact with the structure of technical architectures.

REFERENCES

Budde, R., Christ-Neumann, M.-L. and Sylla, K.-H. Tools and Materials: An Analysis and Design Metaphor. In Heeg, G., Magnusson, B. and Meyer, B., editors, TOOLS '92 Conference Proceedings, pages 135-146, New York, 1992. Prentice-Hall.

Coutaz, J. PAC, an Object Oriented Model for Dialog Design. In B. Shackel, editor, INTERACT '87 Conference Proceedings, pages 431-436, North-Holland, 1987. Elsevier Science Publishers.

Dzida, W. and Valder, W. Application Domain Modeling By Knowledge Engineering Techniques. In Shackel, B., editor, INTERACT '84 Conference Proceedings, pages 481-488, North-Holland, 1984, Elsevier Science Publishers.

Dzida, W. On Tools and Interfaces. In Frese, M., Ulich, E. and Dzida, W., editors, Psychological Issues of Human Computer Interaction in the Work Place, pages 339-355, North-Holland, 1987. Elsevier Science Publishers.

Goldberg, A. Information Models, Views, and Controllers. Dr. Dobb's Journal, pages 54-61 and 106-107, July 1990.

Green, M. Report on Dialogue Specification Tools. In Pfaff, G. E., editor, User Interface Management Systems, pages 9-20, Berlin, 1985. Springer.

Green, M. A Survey of Three Dialogue Models. ACM Transactions on Graphics, 5(3):244-275, July 1986.

Jacobson, I., Christerson, M., Jonsson, P. and Övergard, G. Object Oriented Software Engineering. Addison-Wesley, Reading, 4th edition, 1992.

Norman, D. A. The Psychology of Everyday Things. Basic Books, New York, 1988.

Olsen, D. R. User Interface Management Systems. Morgan Kaufmann Publishers, San Mateo, 1992.