Computer Support by Knowledge Enhancement, Constraints and Methodology

de Greef, H.P.

Citation for published version (APA):
Amsterdam: UvA

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: http://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.

UvA-DARE is a service provided by the library of the University of Amsterdam (http://dare.uva.nl)
Chapter 7

Cognitive Support: Designing Aiding to Supplement Human Knowledge

with Mark Neerincx

Modern information technology brings about new opportunities to enhance the effectiveness of joint human-computer systems. Our research aims at improving the involvement of the users by an aiding computer facility which compensates their cognitive deficiencies. To aid the users, the computer system takes the initiative to present task knowledge that is context-specific and procedural. A design method was developed for systems that provides this type of aiding. This chapter discusses users’ needs for support and presents the method exemplified with the design of an aiding interface for the statistical program HOMALS.

The method has been applied to design a conventional plain interface and an aiding interface for the statistical program HOMALS. Both designs have been implemented as prototypes. In the experiment, the performance of users working with the aiding interface is compared with the performance of users working with the plain interface. As a consequence of the addition of the aiding function, users with minor HOMALS expertise prove to perform their tasks better and to learn more.

Introduction

The development of advanced—knowledge-based—information technology has brought new possibilities to arrange task performances. The scope of these possibilities is rather wide, but the corresponding profits are not always thoroughly investigated. Major efforts have been invested to investigate possible utilities of information technology with a focus on either the technology or on the human. However, software systems and humans each perform sub tasks which are parts of a more general task. A single focus on either the technology or the human insufficiently addresses the cooperative aspects of the joint human-system task performance.

The research approach in this chapter tries to overcome this shortcoming. First, the research addresses the highly interactive cooperative involvement of both the human and the technology. The objective is, in correspondence with “classical’ human-computer interaction (HCI) research (Norman & Draper, 1986), to improve human involvement. Second, the design of cooperation is integrated into current system design techniques to make it really applicable. Current software engineering techniques are extended to incorporate HCI-principles (Barnard & Harrison; 1989; Sutcliffe, 1991; Johnson, 1992). This chapter presents the result of this approach: an explicit design method for a user interface which aids the users’ task performance.

The next section proposes to establish one type of cooperative problem solving by designing cognitive support, as an alternative to the technological tradition of attempting complete automation and as replacement or adjunct to personnel selection and training. Section 7.2 presents our approach to integrate human-computer interaction into model-based design of cognitive support systems. Section 7.3 centres around a specific type of support function, called aiding, which supplements human knowledge. In section 7.3 some important problems of humans in applying knowledge and their corresponding needs for aiding are discussed. The section ends with requirements for aiding functions. Section 7.4 presents a design method for aiding functions which meet these requirements. Section 7.5 exemplifies this method with a design of aiding for the statistical program HOMALS. Section 7.6 evaluates the resulting HOMALS-aiding and section 7.7 presents the conclusions.

7.1 Cooperative Problem Solving

Researchers have sought to improve human-machine task performance with different approaches. Two “classical” approaches can be distinguished: (i) the technological approach present in traditional automation projects and (ii) the user-centered approach present in human-computer interaction research. We will first discuss the main shortcomings of these two approaches. Then, we propose as alternative a combination of both approaches: the design of a support function to establish cooperative problem solving.
Technology-Centered System Design.

The aim of the technology-driven approach, present in traditional expert system design projects, was to replace the human by the machine or—at least—to let some problem-solving tasks be performed by the machine (e.g., MYCIN, Buchanan & Shortliffe, 1984). Explicit or implicit assumptions of first expert system developments were that the machine can perform tasks, originally done by humans, better (e.g., less errors), cheaper (e.g., less personnel, less training, less salary), with more reliability (e.g., less dependent on personal expertise), etc. Human's role in such systems was reduced, mainly consisting of providing the current parameter values required for the system's problem solving. Such a data-entry role comprises a kind of master-slave relation between computer and human: the human has to be docile and answer the questions of the system enabling the system to solve the problem.

The technology-driven approach has some serious shortcomings. A computer master may force an inflexible task execution which is not suited for irregular—and maybe unforeseen—situations. Possible intelligent contributions of humans will be overlooked when technology options are investigated. These contributions might be needed in irregular, and especially in unforeseen situations (Wickens, 1992). Further, new problems arise with the development of knowledge-based systems, such as problems of acceptance, in particular for knowledgeable users (Shortliffe, 1989). In practice, knowledge-based systems are difficult to realize. It might be that the designers, in their focus on automation of work, oversee other, more simple solutions which can be beneficial. For example, to support ambulance dispatchers, a simple display of patient data as external memory aid has a beneficial effect that is at least as large as that of a complex knowledge-based diagnosis system (Post, 1996).

User-Centered System Design.

Partly as a reaction to the failures of the technology-driven approach, research focusing on the human task performance sought to develop usability and learn-ability principles of software systems. This research centered around the communication between users and computers, especially the interaction with the user interface which, among other things, should be simple and consistent. The computer was mainly viewed as a tool with the human as master and the system as slave obeying the commands of the user.

This view is, for example, present in Norman's (1986) distinction of two gulfs between user and system. The gulf of execution refers to the problems users have in transferring their goals into commands the system "understands" and can execute, that is, in specifying the correct actions. The gulf of evaluation refers to the problems users have in understanding system behaviour such as the interpretation of error messages (e.g., "fatal system error").

In the gulfs, a further distinction can be made: the semantic and articulatory distance (Hutchins et al., 1986). Semantic directness concerns the relation of what the user wants to say (i.e., user's goals) to the meaning of an expression in the communication. Articulatory directness has to do with the relationships between the physical form of an expression and it's meaning. Notice that the semantic and articulatory distances center around the communication between user and software system.
Human-computer interaction research has provided principles and techniques to establish a consistent and simple communication. The execution and evaluation gulf are small if there is a direct mapping of the sub task the user wants the system to accomplish (e.g., the calculation of the mean for a data-set) on a system operation (e.g., the command “mean data-set”). The goals of the user should be present in the interaction language, so that actions to control system functions are straightforward and consistent. Moran (1983) describes how to establish a direct and consistent mapping of the external task space on the internal task space. In particular, the communication is simple if a fine task-action mapping is established for DM- (Direct Manipulation) or WIMP-interfaces (Windows, Icons, Mouse and Pull-down/popup menus). Learning how to control such interfaces is easy, because the icons, menus and windows provide information about the actions which can be done with the system (Hutchins et al., 1986; Shneiderman, 1987; Coats & Vlaeminke, 1987). For example, a menu-item “mean data-set” informs the user that he or she can accomplish this specific goal with the system—the semantic meaning is rather obvious for the user—and with minor experience the user even knows how to accomplish this goal (i.e., the articulation: clicking on this menu-item with the mouse).

The traditional focus on the computer user in HCI-research had some serious shortcomings. The user is not always able to play the master role (i.e., may not know which goals have to be performed or when they have to be performed) and may not profit from the computer as much as possible. It is often—implicitly—assumed that the users are well capable of doing their primary tasks, that is, users are thought to know which goals have to be performed when. System designers might be tended to pass these problems on the selection and training of personnel, while selection and training will not always lead to optimal task performance and can be very costly. An alternative or adjunct may be to design a support function which supplements human deficient capabilities.

Design of Cognitive Support.

A combination of the technology- and user-centered approaches can help to overcome the shortcomings of each. Aiming at such a combined design approach, it was recently proposed to integrate HCI-research—or its results—into the current practice of software engineering (e.g., Barnard & Harrison; 1989; Sutcliffe, 1991; Johnson, 1992; deGreef et al., 1993). A major problem is the so-called formality gap, that is, the leap from the informal, real-world requirements and constraints to a formal and structured development process (Dix et al., 1993). In our view, this gap can be bridged and substantial improvements of the joint human-machine task performance can be established by analyzing human needs for support and harmonising methods of system design to these needs. General needs determine the specific type of support (requirements) and corresponding design techniques, whereas analyses of specific needs (i.e., specific population or situations) have to be incorporated in the design method.

Our specific interest is in problem solving and the possibility information technology provides to improve the problem solving. We use the term cooperative problem solving, to emphasize that we do not aim at the design of software systems which take over all human problem solving. The purpose is to design cognitive support which enhances and amplifies human cognition or, more specifically, to improve human in-
volvement by designing system functions which supplement human knowledge and capacities. The effect of cognitive support should be a better joint human-machine task performance. A wide range of support functions may contribute to such result, from "simple" help functions (e.g., Elkerton, 1988) to "intelligent" critics (e.g., Silverman, 1991; Fischer et al., 1993).

A spelling checker is a well-known example of a support facility. This facility is added to a so-called base system, the editor. Users doing their editing job often have some deficient knowledge with respect to the primary task of writing a document: they may not exactly know how to spell a word. With current spelling checkers the users can ask whether the spelling of a word is correct and, if not, what a possible correct spelling is. Thus, the support facility can supplement the knowledge the user is lacking. Further, the spelling checker can be used to detect spelling errors in a complete document.

Table 7.1 sums up the three approaches towards system design. The technology-centered approach seems not to be applicable to design support for writing, because writing cannot be completely automated. A user-centered approach, which comprises only the user-system communication leaving general task knowledge out of consideration, would also not result in a support function for writing. The brief example of the spelling checker shows that an analysis of the general or primary task and the cooperative involvement of the human and computer can result in a fine support facility. The question is what the general requirements of cognitive support are.

Table 7.1: The design of cognitive support compared to the “classical” technology- and user-centered approaches to system design.

<table>
<thead>
<tr>
<th>Approach</th>
<th>Focus</th>
<th>Impact on design</th>
</tr>
</thead>
<tbody>
<tr>
<td>Technology-Centered</td>
<td>primary task</td>
<td>SE models and methods</td>
</tr>
<tr>
<td>User-Centered</td>
<td>interaction tasks</td>
<td>HCI-principles</td>
</tr>
<tr>
<td>Cognitive Support</td>
<td>joint execution</td>
<td>SE models and methods</td>
</tr>
<tr>
<td></td>
<td>of primary task</td>
<td>incorporating HCI-principles</td>
</tr>
</tbody>
</table>

Trade-off. Next to possible benefits of cognitive support, the control and learning of a support facility can have negative effects on the overall effectiveness. Such a facility may enhance the performance of important sub tasks, but can in turn lead to inferior performance of less critical, yet important sub tasks (Adelman et al., 1993). Computer aids can be designed and used that allow the operator to selectively off-load tasks to automation. However, the burdens associated with managing the automation by the operator can sometimes outweigh the potential benefits of the support facility (Kirlik, 1993). Facilities providing help information can complicate the system, because the control of these facilities invoke more functions that must be used and learned, and more information to process during task performance. For example, for the statistical program SPSS/PC the costs of learning to use the help facility prove to outweigh the benefits of the knowledge presentation (Neerinckx & deGreef, 1993).

Thus, a support facility may be harmful! There is a trade-off between the potential benefits, such as off-loading and knowledge provision, and the costs of learning and using the support facility (Rouse, 1988). A good trade-off is arrived when the benefits
are great and the costs small, imposing three requirements for systems providing
cognitive support.

First, the cognitive support function should be directed at major limitations of
human task performance. The potential benefits of support and the chance of a
positive trade-off are large when it meets general needs or immediate difficulties of
the human task performer.

The next two requirements center around minimising the costs of using and learn­
ing the software system. Human task performers try to perform their tasks and
will not invest much in the use and learning of a software system, even though this
would eventually lead to better, more efficient task performance; this is known as
the production paradox (Carroll, 1987). They might also have insufficient capacities
—possibly temporarily— to use the system. Therefore, as a second requirement, the
cognitive support function should be well-integrated into the task execution. A sep­
parate support facility is not attractive, because it imposes extra interactions such as
data-entry and information search. It should be a well-integrated add-on to a base
system that in itself is considered useful for —possible imaginary— people who are
very well capable of doing their share of the work. As the base system has benefits in
itself, this increases the likelihood of a positive trade-off. Moreover, integration allows
the support function to obtain data from the base system rather than from the user,
and it allows the support function to act rather than recommend actions to the user.
A stand-alone system can only provide advice; it will not be able to perform tasks
and, consequently, may not meet some important general needs or difficulties of the
human task executor.

The third requirement states that the user interface of the overall system —the
contingent base system and support functions— should be easy to learn and use.
The use of a software system for the execution of tasks brings about extra tasks: the
interactions with the system. To minimise the costs of these interaction tasks, the
human-system dialogue should be simple and minimal. Figure 7.1 summarises the
three requirements for cognitive support.

1. The cognitive support function should be directed at major deficiencies of
human task performance.

2. The support function should be well-integrated into the human task exe­
cution.

3. The user interface of the overall system —base system and support
functions— should be easy to learn and use.

Figure 7.1: Requirements for cognitive support.

Due to a lack of empirical research on the possibilities and effects of cooperative
problem solving, it is still rather unclear how to design cognitive support and what
form of support is best under which conditions. The next section will provide a frame­
work for model-based design of highly interactive systems incorporating evolutionary
design of a cognitive support function. Then Section 7.3 will discuss the requirements for one type of support, called aiding: supplementing human knowledge.

7.2 Model-based Design of Human-Computer Systems

The above provides three requirements for cognitive support, but it does not tell how to build a system that provides such support. Below we integrate widely used model-based software engineering (SE) methods with contributions from the field of human-computer interaction (HCI). We first look at the state of the art in software engineering: a model for the design process and models or modelling languages for representing requirements, task knowledge and the artifact to be designed.

The model-based SE method is directed at identification of the tasks for the computer system and at further elaboration of the design of the human-computer system. These SE methods are powerful, but not enough, as design of an interactive computer system implies the design of a human-computer system, and human characteristics need to be taken into account. Therefore, we extend the SE approach to improve the practice of human-computer system design.

Model-based Software Engineering.

A software development process model is a prescriptive model for system development. The simplest example is a Life Cycle model, viewing system development as a sequence of stages: analysis, design, implementation and maintenance. There are more sophisticated software development process models (e.g., Boehm, 1988) that emphasize that design is iterative or even evolutionary, and that acknowledge the utility of prototypes to drive and guide the development process.

A modern view on software engineering and knowledge acquisition is that of a modelling activity (e.g., Yourdon, 1989; Rumbaugh et al., 1991; Schreiber et al., 1993). Each stage in the development process produces intermediate or mediating models, with the implemented system —"the code"— as the final model. In a model-based design elaboration of the Life Cycle Model, the analysis stage produces a conceptual model of the prospective system. This analysis model acts as a specification of requirements and the design model represents the prospective software system, that is, the "artifact". In the implementation stage this model is transformed to the final and most detailed model, i.e., the code in the implementation language (Hayward et al., 1987). This view on system development is summarized in figure 7.2.

In model-based design, methodological support of system development is provided firstly by the model for system development, with the Life Cycle Model as an example. Further support can be given by providing "modelling languages", possibly as a graphical diagramming technique, which can be used to draft analysis and design models. Modelling languages provide a notation for the products of analysis, design and implementation stages, providing a standard and allowing computer storage of models. Thirdly, the transformations from analysis model to design model and to the implementation can be supported by tools and even partly automated.

Analysis Model. A large model, such as the analysis model, can be modularized in sub models, each having its own modelling language, and each providing a projection
or perspective of the complete model. It is now widely accepted that there are at least four of such perspectives in system analysis: Functional, Data, Behavioural and Social (e.g., Curtis et al, 1992). It is assumed that in developing a new application the analysis model can be established by constructing the smaller models for each perspective, in different modelling languages, provided that they are “made consistent with each other” (Yourdon, 1989) meaning that the connections between various models through shared referents are made completely explicit.

Table 7.2 summarises the four perspectives with corresponding modelling languages taken from structured analysis (Yourdon, 1989). Viewing a task as a process, the functional perspective represents what process elements are being performed and what flows of information objects are relevant to these process elements. As a modelling language, (hierarchical) data flow diagrams can be used (see, for an example, Figure 2.7, P. 20). It should be noticed that a functional model only represents types of data that are input or output to subtasks or process elements. The behavioural perspective represents when process elements are performed, that is, the control structure. One may think that the behavioural perspective is not needed since the functional model has the behavioral requirements implicitly built into it, but in fact many different control regimes could govern the execution of the process elements present in the functional model (see, e.g., Wallace et al., 1987). The data perspective represents the vocabulary and structural properties for information objects. Examples of functional, data, and behavioural models will be provided in section 7.5.

The modelling languages used in structured analysis methods provide a good ex-
Table 7.2: Modelling languages for the four perspectives of the analysis model.

<table>
<thead>
<tr>
<th>Perspective</th>
<th>Topics</th>
<th>Modelling Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functional</td>
<td>Task Decomposition,</td>
<td>Hierarchical Data-Flow Diagram</td>
</tr>
<tr>
<td></td>
<td>Information flow</td>
<td>(functional model)</td>
</tr>
<tr>
<td>Data</td>
<td>Objects, Relations,</td>
<td>Entity Relationship Diagram</td>
</tr>
<tr>
<td></td>
<td>Attributes</td>
<td>(data model)</td>
</tr>
<tr>
<td>Behavioural</td>
<td>Control structure,</td>
<td>State Transition Diagram</td>
</tr>
<tr>
<td></td>
<td>Events</td>
<td>or Process Language</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(behavioural model)</td>
</tr>
<tr>
<td>Social</td>
<td>Task Allocation</td>
<td>Partitioning of functional or behavioural model</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(user-system partition)</td>
</tr>
</tbody>
</table>

ample, but are by no means considered to be the ultimate modelling languages. We view those for the first three perspectives as particularly useful to support a detailed task analysis, resulting in a formalized, semi-formal task model.

The social perspective represents the allocation or distribution of process elements among “agents”, that is, humans and computers. Focusing on human-computer interaction, that is, the interaction of one human and one computer, the social perspective only involves two agents. The social perspective can be modelled by allocating tasks in the model created in the first three perspectives, and it can be attractively visualized in the style of a role-activity diagram (Holt et al., 1983; Curtis, 1992). Figure 7.3 shows an example. It should be noticed that this diagram represents a behavioral perspective, showing two coordinated threads of execution.

Figure 7.3: A role-activity diagram for modelling the social perspective.
Integration of modelling languages from structured analysis with human-computer interaction design has been suggested by several authors (e.g., Sutcliffe, 1991; Johnson, 1992), but until recently the data perspective tended to be overlooked. Data models were also absent in early versions of structured analysis, but are included in Yourdon's version of 1989 and they seem to gain more and more recognition, but now under the flag of "object-orientation" (e.g., Rumbaugh et al., 1992).

**Interface Design Model.** In the design stage of the Life Cycle Model, the analysis model is transformed in a design model. Focusing on human-computer interaction, only a subset of the design model need to be considered: the interface design model. Technological development has yielded a number of user-interface paradigms (e.g., command-line or WIMP —Windows, Icons, Mouse, Pull-down/pop-up menus) and modelling languages and software tools are available for the design of such user interface. A simple model for the interface design process comprises selection of a user-interface paradigm, selection of design elements available in the paradigm (e.g., windows, buttons, text fields), establishing a mapping from elements of the analysis model to instances of design elements selected, and, last but not least, a specification of the behaviour of the system as a function of user-generated events (e.g., mouse clicks). Different paradigms may provide different sets of design elements, but any paradigm needs to address the specification of behaviour. With command-line interfaces, with a single-threaded dialogue, state-transition diagrams can be used to model behaviour. Modern windowing systems are conceptualized as possibly many dialogues in parallel—the user has many opportunities to generate events. For the behavioural model of such interface an event-handler language (Green, 1986) is recommended.

The behavioral aspect of the analysis model differs from the behavioral model in interface design where it refers to process elements in interaction tasks. For example, a change on the screen display, the user clicks on a particular button, or a system procedure generates a certain event. In the analysis model the behavioural model refers to a control structure over the process elements in the task analysis.

There is no reason why the behavioral modelling language of task and dialogue cannot be the same. And indeed, event-handlers and different threads of execution in parallel are appropriate in the task analysis of real-world tasks (e.g., process control). A generally useful "event handler language" for modelling behaviour is Hoare's (1978, 1985) language for communicating sequential processes (CSP).

**Support of the Analysis—Design Transformation.** Model-based design not only provides modelling languages for analysis and design models, it also aims to support the transformation from analysis model to design model. Full automation of the transformation is perhaps an unreachable aim, but support and automation of parts of the design process is within reach (e.g., Kim & Foley, 1993; Johnson et al., 1993).

To move in the direction of full automation, or, to maximize support of the transformation, the analysis model must contain sufficient information to drive and guide the transformation to a design model. The detailed task analysis provided by the three perspectives provides a background to derive the consequences of the task allocation for the human-computer dialogue. For example, a data flow between tasks allocated to system and user becomes a interaction task between system and user. The task allocation in the functional model, provided that shared referents with the other two perspectives are explicit, propagates to the data model and the behavioural model. In the data model, the task allocation thus determines with which parts of
the data model the user is involved with—very important information for the user interface design. In the behavioral model, the task allocation may determine the need for user and system to exchange real-world events. In general, the more complete the task model is about interdependencies among subtasks, the more complete the requirements for the human-computer dialogue that can be derived. We thus assume dialogue structure follows task structure and allocation.

In three projects we have used such analysis models in the design of the human-computer system: a system for cognitive support for statistical analysis (described in detail in section 7.5), a system for cognitive support of railway traffic control (Neerincx, 1995) and a system for ambulance dispatching (Post, 1996). In each of these, the transformation from analysis model to an interface design model for the WIMP paradigm appeared straightforward. This implies that such analysis model almost completely determines the user interface design, and that the SE modelling languages can be used as high-level specification language for user interfaces. This does not mean that a user interface specified in such manner is well-suited to the user. The SE approach needs to be integrated with insights from HCI.

In this section the emphasis is on models for analysis and design, but two important HCI issues can be briefly mentioned: job design and user task analysis. From a human-factors point of view, task allocation has a large influence on the users’ job design. To optimize usability, performance, safety and the like, cognitive characteristics of the human have to be taken into account. An important factor is human mental workload, but assessment of workload is difficult, because total workload is not a simple sum of workload of process elements, and different types of loading components have to be taken into account (e.g., Rasmussen, 1986; Wickens, 1992). The consequence is that user behaviour and usability cannot be very well predicted, certainly not from the analysis model. They can only be assessed after implementation of a prototype. Fortunately, the distance from analysis model to running interface prototype has become small with modern software tools.

User task analysis is important in interface design, as it helps to identify the information needs of the user. Historically, SE approaches have emphasised quick identification of the tasks for the system and further detailed analysis of these tasks and design of system to perform these tasks. However, the SE modelling languages presented above are also applicable to analyze tasks allocated to the user (cf., Shepherd, 1993).

Section 7.3 addresses HCI aspects of cognitive support more extensively.

Modelling Cognitive Support.

The model-based design approach outlined above is suited for the design of master-slave systems and systems with a fixed task structure and task allocation. When there is a need for dynamic allocation (Rouse, 1988), or when system and user both contribute to the execution of a task (joint function allocation; e.g., Clegg et al., 1989), it is not immediately obvious how this can be modelled in the analysis model and how this fits in a model-based design process.

With cognitive support in which the system presents context-specific task knowledge to the user (i.e., aiding), both user and system are involved in the same task. For model-based design of such cognitive support function, the joint execution process needs to be explicitly modelled.
Even in the simplest master-slave system there is a joint execution process, that is, a main loop in which the user gives a command and the system subsequently executes the command. Because this is so simple, modeling the joint execution process is no great concern in the design of master-slave systems. In system development practice, the process of joint execution is not explicitly modelled in the analysis and it is only being defined in the interface design model, using an explicit model describing user interface behaviour as a function of user-generated events.

The SE modelling techniques may also be used to explicitly model joint execution processes. Figure 7.4 shows the analysis model for a master-slave system.

![Figure 7.4: A model for the joint execution process in a master-slave system.](image)

The examples above show the utility of SE modelling techniques in modelling of joint execution processes and support functions. After discussing model-based software engineering methods and our view on extending these methods to design cognitive support, we now turn to the modelling process.

**Modelling and Testing of the Human-Computer System.**

The above mostly summarizes a widely known approach to modelling of information systems and data base systems. Modelling languages that can be used to create an analysis model can be very profitable, but it is still the question how to proceed to create a model for a future application.

Prospective computer users are the primary source of information concerning all aspects of both current and future tasks (Phillips *et al.*, 1988). In software engineering methodologies the collection of task information from these users is somewhat neglected. It has had much more attention in cognitive psychology, knowledge acquisition and HCI research. HCI literature discusses Task Analysis as a method for analyzing and modelling the user's task. It is recommended to use representative task scenarios (*e.g.*, Johnson, 1992) or criterion tasks and even if it is not possible to
create model, it is recommended to use the scenarios to see how the use of the system would fit in the execution of these tasks.

To obtain information from future users two techniques can be distinguished: interview and experimental dialogues (Hickman et al., 1989). Interviewing is a rather traditional technique to acquire information. However, it is difficult to obtain accurate information about the behaviour or strategy of future users who only describe their tasks in general. In experimental dialogue techniques users actually perform specific tasks. These techniques seem more adequate to predict dialogue patterns than interviewing. Such a dialogue technique would be most focussed and effective if a prototype interface, possibly a mock-up, is used. With a modern graphical user interface toolkit it is not difficult to implement a mock-up with which a number of task scenarios can realistically simulated, perhaps with the help of humans to fill-in system functions not yet implemented (Wizard of Oz simulation).

In the experimental dialogue, a prospective user of the future system is asked to perform some representative tasks with this mock-up or prototype interface. Verbal and behavioural protocols are collected to model the static and dynamic information needs of these users. During the interaction with the prototype interface the relevant information is obtained by recording the system events that the users notice and that precede user actions (e.g., giving a command to open a window).

Experimental dialogue can serve different functions in system development. To identify problems and difficulties prospective users may have, they could use the prototype interface to solve a number of task scenarios. To obtain the expert knowledge, experts would use the prototype interface to solve a number of task scenarios.

Above, empirical techniques for modelling the user's task were briefly described (for a more extensive overview, see Johnson, 1992). However, human problem solving behaviour cannot be predicted completely for future systems and problems. Further, in creating the models of user's task performance errors can be made. Therefore, the usability of a software system has always to be tested with a user performing tasks. The experimental dialogue technique can also be used for this purpose. In this way, design is an iterative process of generating a model, testing it, generating a better model, etc. (i.e., a generate-and-test procedure).

Generate and Test of Cognitive Support

The software engineering techniques can be used to draft and systematically generate alternative models, the empirical techniques support the drafting and testing of models. We now turn to the method or procedure for efficiently designing cognitive support that meets the requirements listed in Figure 7.1.

The central assumption of this chapter is that stand-alone cognitive support is not feasible because of the trade-off between the benefits and costs of a support facility (see section 7). To design cognitive support, two systems have to be designed: the base system and the cognitive support function. The base system comprises a set of tasks or functions that —under control of the user— can be delegated to the machine, whereas the support function participates in the human task performance, because the human is not able to do it completely on its own.

As a consequence of the requirement that cognitive support must be directed at major difficulties and needs of the prospective user, the design of the cognitive support function must be based on an analysis of what these needs or difficulties
are. Experimental dialogue techniques seem promising qua efficiency and validity. To apply these, a mock-up prototype of the base system is needed. Even if other techniques are employed to assess the needs of prospective users, and if there appears a need for support, the support function must be designed as an integrated add-on to the base system. Therefore, the global structure of a method for design of cognitive support is a sequence of three steps as shown in Figure 7.5.

Given that the human has sufficient knowledge and capacities to perform his or her tasks, it is well within the scope of current technology to design a base system that is easy to learn and use for experts. With a task-action mapping designed to be as direct and consistent as possible, and with a WIMP interface that provides the user without system experience with task action mapping knowledge, the resulting system will in general be easy to learn and use for experts. However, as will be discussed in more detail in the next section, even experts might have problems in using the base system and, consequently, may need cognitive support.

1. Design a base system using SE and HCI model based design techniques and test whether it fits expert performance.

2. Analyze the needs and difficulties of prospective users (e.g., with experimental dialogue techniques centred around a mock-up prototype of the base system).

3. Design cognitive support, add it to the base system, and test the support.

Figure 7.5: The framework for design of cognitive support.

7.3 Supplementing Human Knowledge

The remainder of this chapter will elaborate on one type of cognitive support, which we call aiding. The basic idea of aiding is simple and straightforward: present the knowledge that the human is lacking. Various kinds of facilities have been proposed to apply this idea in practice, but for most inventions it is rather unclear what the effects of aiding are on the task performance. Furthermore, it is often unclear what type of human knowledge deficiencies the aiding is supposed to supplement.

Fisher (1991) distinguishes two dimensions characterizing the knowledge of computer users. The expertise dimension refers to the primary task that has to be done (e.g., writing a letter). The extremes on this dimension are naive and expert. The experience dimension refers to the interaction knowledge, that is, the experience with a specific system (e.g., a specific editor). The poles of this dimension are novice and experienced. Thus, there are two types of knowledge: task knowledge and interaction knowledge. Our conception of aiding centers around compensating task knowledge deficiencies and minimizing the interaction knowledge required for computer use.

Human knowledge about tasks can be very different. Knowledge about facts or things is called declarative knowledge; knowledge about how to perform various cog-
nitive activities is called procedural knowledge (Anderson, 1990). Our conception of aiding is specifically directed at supplementing the procedural knowledge humans are lacking. Hereby, the aiding should be directed at major, "general" needs and problems of the human task performer and the system providing aiding should be to use and learn as easy as possible. What are the needs for aiding and the constraints for its use?

**Human Needs for Aiding**

Task performers might profit from knowledge presentation if it compensates some important deficiencies in expertise and is harmonised to their limited capacities to apply current knowledge.

**Human Expertise.** Rasmussen (1983) and Reason (1990) distinguish two levels of problem-solving behaviour: knowledge-based and rule-based behaviour. If the expertise is small for solving a specific problem, than no procedures have been developed and retained in memory. To solve this problem a procedure has to be planned at the knowledge-based level. With repeated practice or training knowledge develops about how to solve the problem. Now, task performance is at the rule-based level, that is, based on procedures consisting of "if condition then action" rules. Production systems incorporate domain-general and domain-specific rules. One way to develop expertise in a problem-solving domain involves creating rules specific to that domain to replace domain-general rules.

The GOMS-analysis of human task performance is directed at modelling rule-based expert behaviour (Goals, Operators, Methods and Selection rules; Card *et al.*, 1983). A GOMS-model proves to correspond well with the use and development of task knowledge in humans (Elkerton Palmiter, 1991; Gugerty *et al.*, 1991; Kieras & Polson, 1985). It may be viewed as a model of human expertise prescribing task procedures for specific conditions.

**Expertise Deficiencies.** Mostly, task performers' knowledge has some vacancies; even the expertise of human experts can show deficits. They are lacking foreknowledge about parts of the task and have to acquire this knowledge from manuals, colleagues or the computer. According to the production paradox, these persons will try to perform their task almost immediately and will insufficiently try to gather the knowledge required for successful task execution (Carroll & Rosson, 1987; see introduction of this chapter). In other words, they start to perform tasks, while their expertise is deficient.

Furthermore, the knowledge —present in the long term memory— is not always used properly or available. First, memory contains information that we cannot recall easily. It is a well-known fact that many things can be recognised which cannot be recalled. We may fail to retrieve the information while consciously trying, or we may simply forget to apply the relevant knowledge (slip of memory, Rasmussen, 1982).

Second, current knowledge can interfere with solving a problem correctly. If a tool or problem solving operator was applied in a specific situation before, then alternative applications to new situations may be overlooked (functional fixedness, Anderson, 1990). On the other hand, knowledge can be transferred to new situations wrongly, such as the transfer of knowledge about the function of the space bar on a typing machine to the function of the space bar on a text editor (assimilation paradox, Carroll &
Rosson, 1987; similarity matching, Anderson, 1990). In general, there is a bias to use knowledge already often applied. In frequency gambling, conflicts between partially matched knowledge structures are resolved in favour of the more frequently employed items (Anderson, 1990); in the application of “strong-but-wrong rules”, procedures often applied successfully can be incorrectly used in new situations (Reason, 1990); and preferences for certain problem-solving operators in solving a problem occur when some knowledge structures — procedures or declarative information — become more available at the expense of others (set effects, Anderson, 1990). In correspondence with these biasing effects of expertise, a critical source of difficulty in automated workplaces for process control was found in the operator’s ability to modify the adherence to pre-specified procedures if this modification was necessary in changing and uncertain environmental conditions. Pre-programmed procedures prove to be brittle. Too often the “if” part is just assumed, and the “y” is done. A careful evaluation of whether condition “x” actually exists is not performed (Wickens, 1992).

*In summary,* there is a variety of expertise deficiencies which are due to lack of expertise, difficulties in applying expertise or applying “wrong expertise”. Correct expert task performance (expertise) can often be modelled in the form of procedural task knowledge. However, even so-called experts do not always act according to such a model. In practice, the expert who has complete knowledge of all tasks showing always perfect knowledge application does not exist. Below, we will briefly go further into one specific cause of problems with knowledge application: the limited capacities of human information processing.

**Limited Capacities.** Human information processing capacities are limited. Above we discussed the distinction between knowledge- and rule-based behaviour. The first is relatively inefficient: it costs much mental effort and takes a lot of time. Task performers are inclined to act on the efficient rule-based level, if possible. Sometimes, the processing capacities of the task performer are too limited to solve complex problems perfectly, especially for problems requiring knowledge-based behaviour. In such situations of “overload”, the task performers will change their problem-solving strategy. Sperandio (1971, 1978) studied how air traffic controllers modify their operative working methods when task load increases. Operating procedures generating lower levels of load were more employed as work demand increased. In these procedures, the controllers operated selectively on subsets of useful information and ignored other information. According to Larichev and Moshkovich (1988), people change their decision strategy when task complexity, and therefore task load, increases. As a consequence, task performers may not incorporate all relevant factors in the decision process in a balanced way. They will apply simplified strategies, possibly resulting in “simplification for the sake of consistency” or the “local experience phenomenon” (Baron, 1988a, 1988b; Madni, 1988; Triggs, 1988). In general, under stress there is a resistance to consider alternative hypotheses; this is known as cognitive tunnel vision (Wickens, 1992).

*Summarising,* human capacities are limited. When humans are overloaded, they employ “default” problem-solving strategies which may not be optimal and focus or narrow their attention noticing less of their environment. Efficient task performance can often be modelled in the form of procedural task knowledge.
Use of an Aiding Facility

Above, a number of important problems —non-optimal knowledge applications— of human task performers was discussed. However, the use of computers itself brings about new problems and these problems have to be minimised. Fortunately, solutions are available for a number of such interaction problems.

In human-computer task performance, two levels of task performance can be distinguished. In the last section about human task performance, the primary task was discussed. Knowledge about the primary task concerns the above-described expertise dimension. But in joint human-computer task performance extra tasks are present, the interaction tasks: the user has to provide the computer with information and the computer has to present information to the user. The knowledge of interaction tasks refers to the experience dimension. A consistent mapping of the primary task on the interaction tasks (i.e., task-action mapping) established for WIMP-interfaces assures that the system is easy-to-use (i.e., requires little experience to use).

Software systems can have a lot of "bells and whistles". New releases of software consist of more and more functions. Consequently, systems seem to require more and more learning about its possibilities. In contrast, a system with a minimal interface provides a minimal set of functions. This set is derived from the goals the user wants to accomplish with the system and is just sufficient to achieve these goals. The use of a minimal interface requires relatively little knowledge (Carroll & Rosson, 1987).

In general, users are unwilling to invest much in interaction. In knowledge-based systems, for example, they may not use the system only because data entry costs too much (Sassen, 1993). For the same reason, current help facilities may not be used. Most systems have help facilities that resemble manuals. To use these facilities, users have to retrieve the information needed at a certain moment and, after that, they have to use this information for their task execution (cf. elkerton, 1991). These activities are troublesome; moreover, according to the production paradox the users will hardly use "screen-manuals". It would be better to integrate the help into the task execution of the user.

In summary, computer use can be a major problem itself and may hinder the task performance completely. In the design of computer applications such problems should be prevented first, only then aiding can be really beneficial. We are aiming at aiding systems which require minimal experience. Systems comprising a good task-action mapping and minimal interaction are easy to use and learn.

Requirements of Aiding

The addition of an aiding facility to a software system should be done with great care. The effects on the number of correct executions, errors and efficiency (execution time, mental effort) in the task performance have to be taken into consideration. Two design principles are directed to an optimal trade-off between the benefits and costs of aiding. First, the aiding should be harmonised to the task performers’ needs, that is, when they need aiding and what kind of aiding they need. Second, the extra work caused by the aiding facility, the extra demands of using and learning to use the aiding facility, should be minimised.

Harmonising to Task Performers’ Needs. The need for aiding is present for problem-solving tasks in which the persons have minor expertise. They may sel-
dom do these tasks or have to deal with rarely occurring situations. For these tasks there is a need (1) to compensate the knowledge the human task performer is lacking, (2) to promote efficient task strategies, and (3) to help the human task performer to retrieve the relevant knowledge and possibly discard irrelevant knowledge.

The aiding should be compatible with task performers’ knowledge and comprise an efficient task strategy. Therefore, the aiding facility provides procedural knowledge corresponding to the rule-based behaviour of expert task performers. The procedure is complete for the task concerned, so that the vacancies in persons’ (procedural) memory are compensated completely and these persons do not have to deduce a missing part of the procedure at the knowledge-based level. To help the user to retrieve the relevant knowledge only the procedure (goals) for task execution in the current context is presented. The availability of irrelevant knowledge can inhibit problem solving and, therefore, the aiding is minimal: it provides only context-specific information at the moment it is needed (this information is complete). Thus, to harmonize the aiding to important human needs or difficulties the aiding facility should take the initiative to provide context-specific, procedural task knowledge at the right time. The information should be minimal, but comprise complete routines.

*Minimising Extra Work.* Problem solving for which aiding is needed should not be burdened with a complex interaction or information search. To minimise the interaction, aiding is part of a minimal WIMP-interface. Minimal refers to the combined aiding and base system. The interface is minimal in Carroll’s sense: the base system provides just sufficient functionality to do the tasks well and the user interface comprises a simple and consistent interaction language to control these functions. Consequently, the experience required for successful task execution is minimal, that is, the user interface of the combined system is easy to use and easy to learn (minimal interaction). In the last section, it was briefly described how to design such a user interface.

Figure 7.1 presented three requirements which apply for cognitive support in general. This section discussed some major limitations of human task performance. Based on these limitations, the requirements for cognitive support can be extended with specific requirements for an aiding facility (see Figure 7.6).

1. The aiding facility should take the initiative to provide information at the right time.
2. The information should consist of context-specific, procedural task knowledge; it should be minimal, but comprise complete routines.
3. The user-aiding communication should be a well-integrated part of the user-system dialogue encompassing of a minimal interface.

Figure 7.6: Requirements for an aiding facility.
7.4 A Design Method for Aiding Functions

Section 7.2 presented a framework for design of cognitive support and introduced software engineering concepts: the life cycle model, modelling, models in-between, and model-based user interface design. Section 7.3 provided specific requirements for aiding harmonized to major limitations of the human task performer and discussed how to design user interfaces that are easy to learn and use. This section will provide the integration into a comprehensive design method for a base system and the aiding function which is to support users who lack ready knowledge or who have difficulties in using their knowledge. Following section 7.2, the method encompasses of three consecutive stages.

I. Designing the Plain Interface.

The first stage consists of the design of what we call the “plain interface” for the base system. This plain interface is a simple, minimal user interface for the base system. A mock-up implementation of the plain interface is to be used in the next stage to provide a context to assess the availability of task knowledge of prospective users. In the third stage it is used again to provide a context for acquiring task knowledge from experts. For this purpose, the plain interface aims to be optimal regarding ease of learning. The plain interface may not be the most optimal regarding efficiency or other criteria. For example, it does not provide shortcuts for often used sequences of operations. The interface requires minimal system experience to accomplish the tasks successfully and is optimal for novices in Fisher’s (1991) terminology (see section 7.3).

The method of designing the plain interface is founded on model-based software engineering methods (Yourdon, 1989; Rumbaugh et al., 1991), task analysis methods from ergonomics or human factors (e.g., Drury et al., 1987), and methods from HCI (Moran, 1981, 1983; Johnson, 1992). It is structured as a generate and test process. Generation is decomposed in two steps: the analysis and design steps from the life cycle model (section 7.2). The analysis comprises task analysis and an allocation of parts to the user or to the system, thus providing a definition of the roles of user and system in Johnson’s (1992) sense. This analysis uses modelling techniques from software engineering to create a model of the task and the roles. In this task analysis, criterion tasks (Moran, 1981) or task scenarios (Carroll, 1989; Johnson, 1992) are used as example problems — task instances — for which the task analysis is to provide a general model. The analysis results in an abstract specification of the required user-system interaction.

To complete the generate step, the analysis model of the plain interface is transformed into an interface design model. This can be done in a straightforward way using model-based interface design techniques. For a complete application, the tasks or functions of the system need to be designed as well, but this is beyond the design of the human-computer system and not addressed here, but see Rumbaugh et al. (1991) for methods for general information system design and Schreiber et al. (1993) for knowledge-based system design.

Next in the design process, the design of the plain interface is tested. This is to be regarded as quality control to make sure that there are no serious difficulties and discrepancies between how the plain interface can be used and what is required for expert task execution. It may take a few generate and test cycles before a satisfactory
result is achieved. To test the plain interface, it is implemented as a mock-up or Wizard of Oz prototype. This is not a full implementation, but rather a simulation that only needs to function realistically for a set of criterion tasks. Various software tools exist that can be used to create a running interface and system functionality can be simulated by a human at another terminal (Wizard of Oz). Often the solutions of criterion tasks can be prepared in advance and included in the software. For the actual testing, experts are simply asked to try the prototype to solve the criterion tasks.

When experts are easily identifiable and can be expected not to differ very much it only takes a small number of experts to identify the bugs in the plain interface. These trials are intended as quality control of the plain interface. Even though design of such an interface is well within the scope of modern methods and technology, errors or omissions may have occurred, and experts are regarded as good judges and subjects to assess whether the plain interface is suited to, and fits well in their tasks (we do not want to suggest that experts never need support, see section 7.3). Care must be taken that the set of criterion tasks is representative and provides a reasonably complete coverage of the task domain.

II. User Analysis.

In the second stage of the design method, user analysis, the prospective users are investigated. Among prospective users, the level of expertise may vary considerably. To investigate if there is a need for aiding or cognitive support among prospective users, these users are simply asked to try the prototype of the plain interface to solve the criterion tasks. This provides information about the problems and difficulties they may have and helps to identify in which parts of the task they appear to lack knowledge. It may of course turn out that there is no need for aiding or other cognitive support. Individual differences in expertise between prospective users may be large and at least a small sample (e.g. 10) is needed to get some indication of performance problems of the average user.

III. Designing the Aiding Function.

The third stage of the design method is concerned with the design of the aiding function for the task parts in which prospective users lack knowledge. The design process has the same structuring as in the design of the plain interface. First the analysis and interface design models are generated, then a prototype is implemented and tested. Not much is known about the analysis model for cognitive support. We assume that it should describe the behaviour of the system in the process of joint task execution. For aiding functions that present knowledge to the user, the analysis model should also contain the expert task knowledge to be presented by the aiding function.

To acquire the expert knowledge, experts are again asked to solve criterion tasks using the prototype of the plain interface. The purpose now is not quality control of the plain interface, rather it is to acquire the expert knowledge for those tasks for which prospective users lack knowledge. The use of the prototype to walk through the criterion tasks helps to generate and debug the expert model (it is a specific type of experimental dialogue technique, see section 7.2). The reader may wonder why
I. Designing the Plain Interface

— Generate a design of the Plain Interface.

1. Generate the analysis model of the Plain Interface
2. Generate the interface design model of the Plain Interface.

— Test the design of the Plain Interface.

3. Implement a mock-up prototype of the Plain Interface.
4. Test whether the Plain Interface fits expert task execution by having experts use the prototype to perform criterion tasks.

II. User Analysis

5. Test the performance of prospective users by having them use the Plain Interface prototype to perform criterion tasks.

III. Designing the Aiding Function

— Generate the design of the Aiding Function.

6. Generate the analysis model of the Aiding Function.
   • Generate an expert model (procedural task knowledge) for those parts of the task in which prospective users need aiding.
   • Generate a model of the joint execution process.

7. Generate an Interface Design Model of the Aiding Function.

— Test the design of the Aiding Function.

8. Implement a mock-up prototype of the Aiding Function as an extension of the Plain Interface.
9. Test the Aiding Function by having prospective users perform criterion tasks using the mock-up prototype of Plain Interface + Aiding Function.

Figure 7.7: The method for the design of aiding.

This knowledge acquisition is not done the first time experts try the plain interface prototype. The reason is that knowledge acquisition is much more work for both experts and system designer, compared to just quality control of the plain interface, and only after the user analysis the parts of the task are known for which it is actually necessary to spend the effort of acquiring the knowledge and create an expert task model.

After acquiring and modelling the knowledge, the presentation and structuring of the knowledge and the interaction with the user have to be designed to complete the design of the aiding interface consisting of the plain interface and the aiding function. The aiding function can be prototyped and, subsequently, tested by prospective users trying to accomplish criterion tasks. This is the quality control of the aiding and comparisons with plain interface performance data show whether prospective users can benefit from the aiding.
Figure 7.7 summarizes the structure of the method. If a base system already exists, and if problems users have are known, one might consider to move directly to the design of aiding, but if the existing system has a difficult and hard to use user interface, we think it is worthwhile to start with the re-design of the user interface. The potential gain of this may be large, it may even be larger than the gain of aiding.

The above provides a proposal of a method based on software engineering practice and psychological requirements. It is the starting point of an investigation in which two questions are central: (1) is the method operational and usable, and (2) can non-experts indeed benefit from such aiding? The next section applies the method in a software engineering project in which new user interfaces are developed for an existing statistical analysis program called HOMALS. The interpretation of the different HOMALS statistics is a specialist expert task and for this an aiding function has been developed. Experience in this case study provides information regarding the usability of the method. To investigate the utility of aiding thus designed, the final test of the aiding function is performed as a full statistical experiment, comparing performance and learning in a plain interface condition with a condition comprising a plain interface plus an aiding function. Two such experiments have been conducted, one with users without any HOMALS knowledge and one with users with some, but little knowledge. These are reported in section 7.6.

7.5 The Design of HOMALS Aiding

The proposed method for design of aiding is investigated by applying it to design an aiding application (this section) and by formally testing the application in a comparative statistical experiment (the subsequent section). For an application, we start from an existing program for statistical data analysis called HOMALS. Reasons to choose HOMALS are threefold. First, the experience and expertise of the users of the existing system varies considerably (even though the system has a user interface that is hard to learn and to use), so we expect that there are users who may profit from aiding. Second, HOMALS analysis is a difficult problem solving task and it is a non-trivial problem to provide effective support. Finally, experts and users are available and easily accessible.

This section follows the three stages of the design method for cognitive support outlined in the previous section. In the first step, the Life Cycle Model is traversed to design a plain interface for HOMALS experts. The existing HOMALS system is here regarded as a base system, but with an old fashioned user interface that makes it hard to use for inexperienced users. In this first step the user interface is redesigned to provide a plain interface that is easy to learn and use. In the second step—as expected—it appeared that prospective users lack HOMALS expertise and in the third step the Life Cycle Model is traversed again to design an aiding function that provides users with context specific knowledge to help them perform the HOMALS analysis.

Below, we provide a brief introduction to HOMALS-analysis and then we walk through the steps of the design method for aiding functions.
The existing HOMALS system is based on a statistical model with the same name (Gifi, 1990). This statistical model can be used to analyze a data set with nominal (categorical) variables. An example of such a data set are answers to a multiple choice questionnaire for a sample of respondents, numerically encoded in a computer file. In a HOMALS model such data are mapped to a space with one, two, or more dimensions. This is rather similar to principal components analysis or factor analysis, but suited for variables with a nominal scale.

To use the existing HOMALS system the user has to specify a simple data model for the data set, the number of dimensions for the statistical model and the data set itself. A data set is a rectangular table, a matrix, with a row for each individual or object in the sample and a column for each variable. The numerically encoded value in row (i) and column (j) represents the i-th object’s value on the j-th variable. A simple data model suffices to describe such data set: the name of the data file and the names of the variables in their column order. The existing HOMALS system also demands a specification of the number of values —categories— each variable has. Henceforward we will use the term meta data for this simple data model, to avoid confusion with the compound representation of all the data involved in the HOMALS analysis task (i.e., meta data, data set, model specification and five statistics). After specifying the meta data, the system can compute the five HOMALS statistics by fitting the model against the data and present these to the user in the form of tables or plots.

![Object Scores](image)

This data point is an "outlier".

Figure 7.8: Example of the Object Scores statistic with an outlier.

The interpretation of statistics such as provided by a HOMALS-type of analysis is a difficult expert task (Visser & Slooff, 1991) in which the expert interprets statistics and may decide to change the data set or the statistical model. Figure 7.8 provides an example of the "object scores" statistic. This is a mapping from the objects in the sample to the dimensions of the HOMALS model for two dimensions. Each point
in the plot represents an individual object's values on the two dimensions. Such a scatterplot may show that objects are extreme on one or two dimensions and thus far removed from the cloud of points. Because such “outliers” may have an unduly large influence on the estimates of model parameters, it is customary to remove these objects from the data set and re-compute the model parameters and other statistics. This episode with an outlier in the plot of object scores and the removal of the object from the data set is only an example of what may happen in the interpretation task. More about HOMALS as a statistical method can be found in Gifi (1990).

We now turn to the case study in which the method is applied to design an aiding interface for HOMALS.

**Step I: Designing the Plain Interface**

In the first step of the design method, “designing the plain interface”, the Life Cycle Model is followed to develop a base system suited to experts and with a user interface that is easy to learn and use. The Life Cycle Model (analysis, design, implementation) is used to structure the design of the plain interface. In this HOMALS case study, there already is an existing system with a functionality that is correct for HOMALS experts, but with a user interface that is difficult to learn and use, even for HOMALS experts who know exactly what they want to do. Although the functionality of the existing system is appropriate for a base system, the user interface has to be redesigned to provide a user interface that is easy to learn and use.

**Analysis**

Instead of analysis from scratch, we abstract from the existing system to construct an analysis model that is modularized using the four perspectives mentioned in section 7.2. This analysis model then serves as specification and starting point for the design a new user interface that is easy to learn and use.

*Functional and Social Perspective.* Figure 7.9 shows a functional model, as a data flow diagram, of the HOMALS analysis task, with a user-system partition to specify the roles of user and the existing HOMALS system. Abstracting this model from interviews, the existing system and documentation is not difficult.

This diagram specifies that there are five different HOMALS statistics and that during HOMALS analysis the user can use three operations to modify the data set: delete object, delete variable and re-code variable, for example, to pool categories. The user may also use two operations to change a model specification (i.e., delete and add dimension).

*Data Perspective.* Figure 7.9 shows eight information objects crossing the user-system partition: metadata, dataset, model specification, marginal frequencies, eigen values, discrimination values, object scores, and category quantifications. These information objects are global and not well defined. The functional modelling could be pursued in more detail, by further hierarchical decomposition of the information objects, and it is possible to arrive at a complete coverage of what the information objects encompass of, but this approach does not take account of overlap or interrelationships between information objects.

A better approach is to devise a central data model in which each object and data element is represented only once, and in which interrelationships among objects...
are made explicit. In the context of data bases data modelling techniques have been
developed for this purpose, for example entity-relationship modelling (Chen, 1976)
for data base design and object-oriented modelling for information system design
(Rumbaugh et al., 1992). This technology is not suited to all types of data (one
would, for example, not store video images in this manner), but is suited for symbolic
representation of many domains such as a data base of products, suppliers, customers
and personnel. This approach applies equally well to knowledge-based systems
(cf., Schreiber et al., 1993) and work on ontologies (Gruber, 1992) can be regarded as a
further development, extending the data model or ontology with logical or functional
expressions.

Such a central data model for the entire system may be the focal point of integra­
tion between the design of the user interface and system design. The data model, or a
subset of it, represents the data that crosses the user-system partition, and comprises
all the information objects in the functional model. Thus, for the HOMALS-example,
a data model has to be established encompassing of all eight information objects
crossing the user-system partition in Figure 7.9.

User Views: the link between functional and data model. The central data model
provides a framework and language for data storage that meets technical desiderata.
For user interface design, it has to be established which parts of the data belong
together from the users’ point of view and should be transferred concurrently. To
some extend this is already specified in the functional model: the information objects
crossing the user-system partition in the functional model.

User views can serve to provide the explicit linkage between the functional model
and the data model. Each information object in the functional data flow diagram can
be defined as a user view of the data model.

Figure 7.9: A functional model of the HOMALS system with a user-system partition
to represent the roles of user and system.
relationship diagramming technique. A user view can be specified as a subset or partition of the entity relationship diagram. Figure 7.10 presents two user views in this way, for the meta data and the data set. These two user views refer to the corresponding information objects distinguished in Figure 7.9. The user view “Meta Data” consists of the types of data needed for a HOMALS-analysis: a data set (with filename) consisting of objects (e.g., persons) with a variable which can have several categories. The user view “Data Set” consists of the data of a specific HOMALS-analysis: a data set (e.g., answers of several persons to a questionnaire) which has variables (e.g., sexe) which in turn have categories (e.g., male or female).

Figure 7.10: Data model using Entity-Relationship Diagramming, with two user view outlines.

A user view as defined above is incomplete, that is, the user view outline as described above only isolates part of the data storage system, and cannot select from the data stored under this part of the data model (i.e., the system cannot find the user view). Therefore we propose to complete the definition of a user view with a specification as a query that takes one or more data elements as parameter. For example, in the HOMALS context, a filename of a Data Set as parameter for the user view “Meta Data”, allows the storage system to find the Meta Data belonging to that Data Set. There is no diagramming technique for this, but a query language like SQL or Prolog can be used to define parameterized user views.

Behavioural Perspective. The analysis model so far provides a static description. The model specifies which user views are required, but it does not tell us anything about which user views are required when. It specifies which tasks there are, and who will execute which tasks, but it does not specify which task will be activated when and who will activate its execution. At this point one could strive to model the human-
computer interaction completely incorporating users' computer use. Such a model would show all dependencies or interactions among different subtasks, and without it, one may overlook some of these in the design of the user interface.

However, modelling the users' task completely is very difficult and may even be impossible in practice (e.g., for writing). For HOMALS, for example, such a model should provide the decision making and the control structure for the task of interpreting the HOMALS output and, when necessary, doing data transformations and re-computing the statistics. Unlike with the functional and data model, the existing HOMALS system and documentation provides no information regarding the behavioural model and to "extract" a model from interviewing experts appears difficult. Experts are reluctant to state hard general rules. It is possible to formulate some production rules or state transition diagram fragments, but this is not complete and without insightful structure. It is possible to wait, and it is better to wait for two reasons. Firstly, one may not be sure that lack of expertise is a bottle-neck with the intended users, and this can be investigated in stage two of the design method with more precision and efficiency using a prototype of the plain interface. Secondly, if lack of expertise is indeed a bottle-neck, experimental dialogues with experts using the plain interface prototype are a more efficient technique to model the expert knowledge required for the user's task. Below, in discussing step three of the design method, "designing the aiding function", we return to modelling the user's task using a state transition diagram.

We take the position that for the design of a plain interface it is not strictly necessary to have a complete and detailed expert model of the user's task and that possible errors will become apparent in usability testing of the plain interface prototype. Still, some behavioural aspects of the base system have to be modelled. The functional model can be extended with initiative markers for the information flows —user views— between user and system (de Greef & Breuker, 1992). In this way the user-system dialogue is specified minimally.

Interface Design Model

In the above, the four perspectives and modelling languages from Structured Analysis have been used to draft an analysis model. It can serve as a rather abstract specification of the human-computer interaction, independent of a particular type of user interface design style. The transformation from the analysis model for the HOMALS base system to a WIMP interface is quite straightforward.

- **Interface Elements.** We have used a small subset of design elements from the huge selection of "frames and widgets" provided by an available commercial graphical user interface management system (TeleUse).

- **Presentation of User Views.** The User Views in the analysis model can be presented to the user in many ways. The Interface design model specifies the type of presentation selected. In consultation with experts it was decided that two statistics would be presented in tabular format and the three remaining statistics as two-dimensional scatterplots (e.g., Figure 7.8 shows the scatterplot presentation of object scores).

- **Interface Behaviour.** TeleUse provides an event handler language. User actions (e.g., clicking a particular button) are called events (such events in user-system
interaction are at a different plane than events in the expert task). The event handler language of the Teleuse system provides rules to specify the actions to be triggered by user initiated events. For example, when the button with label "delete object" is clicked, the procedure delete-object has to be called.

The plain interface for HOMALS is extremely simple. It has a menu to select a statistic—one statistic being shown at a time—and a menu to activate data modification tasks (remove variable, re-code variable, remove outlier, add dimension and remove dimension). Each statistic has a presentation window. Figure 7.8 shows what the user interface looks like during inspection of the Object Scores statistic.

This plain interface provides exactly the same functionality as the existing HOMALS system, but now with a user interface that is suited to expert users without system experience. Both systems are base systems founded on the master-slave assumption: the user is a competent expert and the system is slave under full control of the master.

The design of the plain interface for HOMALS shows the utility of the software engineering techniques. They can be used for a detailed task analysis and interface specification and they promote consistency and simplicity of the user interface.

Implementing and Testing the Plain Interface

In the HOMALS case study, the plain interface as specified above was implemented as a mock-up prototype using TeleUse on VAX/VMS and some C-code to present plots and tables. Experts, who were familiar with the existing HOMALS system, but were not involved in our design project, find the new interface a large improvement. The mock-up prototype can read in and show files with statistics computed by the existing HOMALS system, but cannot actually perform data transformations. The number of dimensions of the HOMALS statistical model is fixed at two dimensions.

Step II: User Analysis

In this second step of the design method, N prospective users try M representative tasks using the mock-up of the plain interface.

The HOMALS plain interface was tested with 10 users of the existing HOMALS system with various levels of expertise. These were asked to analyze five task scenarios using the mock-up. The task scenarios were derived from published analyses performed with the existing HOMALS system. Compared to expert solutions for these scenarios, task performance left much to be desired. Although the users inspected various statistics and derived some conclusions, their interpretations of the statistics and their data transformations proved to be incomplete. Because the knowledge deficiencies centered around diverse aspects of the HOMALS-analysis task and since this task is rather small, the design of an aiding function for the complete HOMALS-analysis task was warranted.

Step III: Designing the Aiding Function

The requirements listed in section 7.3 state that the aiding function is to present context-specific procedural task knowledge to the user. We may distinguish two aspects in the design of aiding: (i) the acquisition and modelling of the procedural
expert knowledge to be presented by the aiding function and (ii) the design of the aiding function that presents this knowledge in a context-specific way.

**Analysis**

*The Expert Model.* In section 7.2 it was argued that having experts perform tasks using a mock-up of the plain interface might be the most efficient procedure, compared with interviewing, to acquire operational procedural task knowledge. In the above it was suggested that state transition diagramming might be a suitable modelling language. This modelling technique and method was used to create an expert model as shown in figure 7.11. It is a behavioural model that prescribes an efficient procedure for the HOMALS analysis.

This prescriptive model is based on 10 subjects. Half are HOMALS theoreticians and half are users of the existing HOMALS system. They were interviewed and they were asked to work with the plain interface prototype to solve several task scenarios. Especially this experimental dialogue provided the most information for the expert task model. The model as shown here in figure 7.11 was not used in interviews. It served the knowledge engineer to record the information obtained.

The expert model has three aspects:

- An ideal ordering for inspecting the five statistics.

- For each statistic the conditions that give rise to transformations of the data set or of the HOMALS model.

- For each condition a verbal explanation that helps the non-expert user to establish whether the condition holds.

The expert model as a state transition diagram is a comprehensive for the HOMALS analysis. If we want to provide a detailed specification of the ideal expert behaviour, the state transition diagram is not suitable. Because the re-computation of the HOMALS statistic takes some time, it is better to identify as many needed transformations as possible before re-computing the statistics. This behaviour cannot be expressed in a state transition diagram, but the desired control structure can easily be expressed in any procedural programming language. What is needed is a data structure (a list or stack) to collect the desired transformations. First as many transformations as possible are collected, then they are applied and the statistics re-computed. Nevertheless, the state transition diagram provided a useful intermediate model that proved a great help in the acquisition of the expert knowledge. This expert model provides the basis for an aiding function added to the plain interface.

*The Aiding Function.* With the expert model, we have a complete prescriptive model for HOMALS analysis, but we have not yet specified how the aiding function will operate. Assuming the expert knowledge comprises an ideal ordering of five subtasks (one for each statistic) and that the expert knowledge for each subtask can be represented as a set of condition-action rules, Figure 7.12 shows a model of the joint execution process and the role of the aiding function therein. This model is based on a task analysis at a meta level. The use of expert task knowledge itself is viewed as a task and this meta level task is decomposed and allocated to user or system. With this approach we have an explicit model of joint function fulfillment by user and system.

161
Inspection of Five Statistics

- **c1**: `skewed_distribution(V)`.
- **c2**: `missing_values(V)`.
- **c3**: `not(c1) and not(c2)`.
- **c4**: `number_of_dimensions = 2` and `dimensions = [D1, D2]` and `high_eigenvalue(D1)` and `low_eigenvalue(D2)`.
- **c5**: `number_of_dimensions = 2` and `dimensions = [D1, D2]` and `high_eigenvalue(D1)` and `high_eigenvalue(D2)`.
- **c6**: `number_of_dimensions = 2` and `not(c4) and not(c5)`.
- **c7**: `number_of_dimensions = 2` and `dimensions = [D1, D2]` and `low_cat_quant(V,D1)` and `low_cat_quant(V,D2)` and `not(crucial(V))`.

(etcetera)

Figure 7.11: The expert model for the HOMALS analysis task.
This approach can be adapted to different methods of representing expert knowledge, and it provides a general solution to what Clegg et al. (1989) call the problem of joint function allocation.

**Interface Design, Implementation and Testing**

Using state of the art interface technology, the aiding function might use an additional window to present the user with context-specific task knowledge. Figure 7.13 shows the HOMALS aiding interface during the task "inspect object scores". The window Object Scores is inherited from the plain interface. The window Aiding Object Scores belongs to the aiding function; it shows operations for transformation of the statistical model or data set (buttons right), the conditions that trigger transformations (buttons left), and a button to continue the analysis. A click on a left-hand button causes the presentation of a text window that clarifies the condition.

The aiding interface, like the plain interface prototype, was implemented as mock-up (in Telellse). In this mock-up, transformations of the data set were not implemented, that is, clicking on the task buttons in the action window does not result in the actual execution of the task. The user is asked to continue the HOMALS analysis as if the system has executed this task. After a justified removal of an object (an "outlier"), the mock-up presents adjusted statistical data.

The last step of the method is the evaluation of the aiding function. In this investigation of aiding we use the most powerful evaluation method: a complete statistical experiment to study the effects of aiding on performance and learning. We performed two of such experiments, one with users with no knowledge of HOMALS whatsoever, and one with users with some but little knowledge of HOMALS analysis. These are reported in the next section.

### 7.6 Evaluation of the HOMALS Aiding

Karat (1988) distinguishes several methodologies for the evaluation of software. The most complete usability test is an analysis of the task performance of prospective users (Desurvire et al., 1992; Karat et al., 1992). In our view, help systems should be evaluated in an experiment in which the performance of users working with an aiding interface is compared with the performance of users working with a minimal interface. This kind of comparison could even be performed during system design using prototypes or mock-ups.

In two experiments, the performance of users working with the aiding HOMALS interface is compared with the performance of users working with the minimal HOMALS interface. Our main objective is to study whether the above-described type of aiding can be beneficial for users performing a statistical task. The conditions in the first experiment—which is a kind of pilot-experiment— are not optimal for this objective. However, together with the second experiment, it provides interesting results and, therefore, it is included in this chapter. For both experiments, we expect that the task performance of users working with the aiding interface is better than the task performance of users working with the minimal interface.
Figure 7.12: The joint execution behaviour of the aiding interface.
Experimental Method.

In general, the method of the two experiments was similar. Twenty students participated in each experiment, 10 in the minimal interface condition and 10 in the aiding interface condition.

In a short introduction, the student was shown the types of information the interface presents and the mouse movements to control the information presentation. This information concerns a real data-set and the corresponding HOMALS-statistics. The student was then required to analyze five data-sets successively. Every student received the data-sets in the same order. For each data-set, a form was provided with information about the research question, the objects, and the variables. The data-set had to be modified and a questionnaire had to be filled in with conclusions about the data. The student had to use a mock-up interface and was asked to continue the HOMALS analysis as if the system had executed this task.

7.6.1 Experiment I

The participants in the first experiment had followed a short introductory statistical course a year ago. A part of this course centred on HOMALS. We assumed that these participants had some, but incomplete knowledge of HOMALS. However, in the introduction of the experiment, they proved to know hardly anything about HOMALS and most participants did not even know for what purpose or surveys HOMALS can be used. Assuming that some statistical knowledge would be necessary, we decided to provide some minimal information of this type in the introduction.

Procedure. The student participated in an experimental session of about 60 minutes. Each data-set had to be analyzed within 10 minutes.

Results. For each subject, the overall performance was the sum of their performance on the data modifications (correct minus incorrect modifications) and conclusions (correct minus incorrect conclusions). In the minimal and aiding condition the mean overall performances were, respectively, 6.4 and 7.6 (F(1,18) = 0.69, p = 0.42).
There was no difference between the performance of students working with the mini­
mal interface and students working with the aiding interface: performance levels were 
very low in both conditions.

Discussion. At first sight, the results are rather disappointing; we did not find 
beneficial effects of aiding.\textsuperscript{1} However, an examination of the experiment reveals three 
circumstances which may have caused this result. The first one is that the users 
proved to have hardly any knowledge about homals. It might be that aiding is not effective for users without any task knowledge, 
but is effective for users with minor task knowledge. Second, the data sets in the 
experiment required few data modifications and warranted few conclusions. Conse­
quently, the performance measure could be too coarse-grained to convey a beneficial 
effect of aiding. Third, the expert model on which the aiding facility was based proved 
to have some redundancies, for example, it was possible to do a data modification 
(remove outlier) at different steps in the analysis. Thus, the help information was not 
minimal, that is, not completely in correspondence with the second requirement of 
aiding presented in Figure 7.6. To test whether the aiding can have a positive effect, 
we conducted a second experiment in which these three circumstances were changed 
(see below).

7.6.2 Experiment II

In the general introduction, it was postulated that procedural task knowledge can be 
processed relatively easily. Sweller (1988) maintains that “... conventional problem 
solving in the form of means-ends analysis requires a relatively large amount of cog­
nitive processing capacity which is consequently unavailable for schema acquisition”. 
Procedural task knowledge may be learned more easily. To test how far computer 
users learn from aiding, we added a knowledge test to the second experiment.

The three above-described experimental circumstances which may explain the re­
sult of the first experiment were changed as follows. First, the subjects had more 
knowledge of HOMALS in the second experiment: they had just finished an introduc­
tory statistical course which dealt with, among other things, HOMALS. Second, we 
used data sets requiring more data modifications and warranting more conclusions 
than in the first experiment, as to obtain a finer grained performance measure. For 
one task, for example, after a justified data modification (removal of an “outlier”), 
the mock-up presents adjusted statistical data for which more conclusions must be 
drawn. Third, the redundancy in the aiding interface was removed. We expect that 
for these experimental circumstances, the task performance and learning of users 
working with the aiding interface is better. Performance measures are the number of 
data modifications performed correctly and the number of correct conclusions in the 
questionnaire.

Procedure. The student participated in an experimental session of about 90 min­
utes. Each data-set had to be analyzed within 14 minutes. After the five HOMALS 
analyses, the student was required to describe on paper, for each HOMALS statistic, 
when data modifications are necessary and what kind of conclusions can be drawn. 
This knowledge test had to be completed within 10 minutes.

\textsuperscript{1} We also did not find harmful effects like the SPSS/PC help system (Neerincx & de Greef, 1993a).
Table 7.3: The results of the second HOMALS-experiment.

<table>
<thead>
<tr>
<th>Interface</th>
<th>Correct data modifications</th>
<th>Correct conclusions</th>
<th>Knowledge</th>
</tr>
</thead>
<tbody>
<tr>
<td>minimal</td>
<td>5.6</td>
<td>11.4</td>
<td>6.5</td>
</tr>
<tr>
<td>aiding</td>
<td>9.5</td>
<td>15.7</td>
<td>11.6</td>
</tr>
<tr>
<td>$p =$</td>
<td>0.10</td>
<td>0.03</td>
<td>0.02</td>
</tr>
</tbody>
</table>

**Results.** The results are shown in Figure 7.6.2. With the aiding interface, students performed more correct data modifications (not significant; $F(1,18)= 2.95, p=0.10$), formulated more correct conclusions ($F(1,18)= 5.36, p<0.05$), and had acquired more task knowledge ($F(1,18)= 7.07, p<0.05$). The mean number of errors of the minimal interface and aiding interface were, respectively, 2.1 and 2.7 ($F(1,18)= 0.15, p=0.7$). The correlation between knowledge acquired and overall performance (the sum of correct modifications and correct conclusions) was 0.54.

Further, it appeared that users of the aiding interface had a more efficient strategy (the strategy incorporated in the expert model). They did less often need to look back at statistics already seen and “forgot” less often to perform the required sub tasks.

**Discussion.** The results of the second experiment are promising; it provides empirical evidence for the effectiveness of this type of aiding. It was not possible to get more than twenty participants in the experiment and it was not possible to use a very fine-grained task measure. For this relatively small number of participants — potential HOMALS-users— and tasks, two performance measures showed significant effects while the other performance measure showed a trend in the predicted direction. The aiding improved the overall performance with 48% and learning with 78%.

## 7.7 Discussion

The overall aim of this research is to design computer systems that fit better with the human. This is difficult, because all choices and decisions in the design process have an effect on the usability of the final result. This research tries a model-based approach to system design: instead of jumping to a complete system, intermediate models are used that are less complex and easier to amend. In the construction of these models, human requirements are taken into account. Models can be tested at an early stage using prototypes generated with model-based user interface design techniques. In the above we have elaborated this model-based approach into a complete design method for classical master-slave interfaces and for a cognitive support function called aiding.

In the preceding a number of requirements were proposed for cognitive support in general (see Figure 7.1), and for aiding functions in particular (see Figure 7.6). A framework for design of cognitive support was drafted (see Figure 7.5), and this was refined into a method for design of aiding functions (see Figure 7.7). This method was applied to design aiding for a statistical analysis program called HOMALS (section 7.5). Finally, the effect of the HOMALS aiding was evaluated in two experiments.

In the evaluation of this work, we consider two aspects: the method for design of aiding and the effect of aiding on human-computer performance.
Human-computer interaction research has provided theory and principles regarding the harmonisation of technology to its users (e.g., Norman, 1986), but it is not explicit how the principles can be given shape in system design. Substantial improvements of the usability of systems may be accomplished by bridging the gaps between practice and theory, and by bridging the gaps between disciplines, such as HCl, human factors, experimental psychology, artificial intelligence and software engineering. Adopting this as a strategy we have developed a design method that operationalizes psychological principles in the system design. The method integrates and extends software engineering, user psychology and empirical techniques. Empirical techniques for knowledge acquisition (i.e., the experimental dialogue techniques) and for usability testing (i.e., experiments), complement the modelling approach. Principles and techniques are combined that in themselves are simple, but that together provide a good result. The result is a comprehensive integrated design method.

Being integrated is a desirable characteristic for a method, but other criteria are also important: completeness, usability, and generality. To assess the design method on these criteria, it was applied to design aiding for HOMALS.

The design method comprises a complete procedure using an evolutionary (generate and test) view on the Life Cycle Model, with separate cycles for base system and aiding function. Within each cycle, user interface prototypes are used for early testing. The base system prototype (i.e., the plain interface), is also used to determine whether future users have a need for aiding, and, in the design of the aiding function, to collect the expert knowledge for the users’ task.

The model-based approach supports the efficient generation of prototypes. The analysis model provides the functional requirements and model-based user interface design techniques support the transformation to a user interface design. Recently, similar approaches have been explored by Janssen, Weisbecker & Zieger, (1993), Vanderdonckt & Bodart (1993) and Kim & Foley (1993). These approaches enable support and automation of parts of the design process which will be more and more available in the future.

To collect the expert knowledge for the aiding function, the plain interface is used in the experimental dialogue technique. In the HOMALS study, this technique showed to be superior to interviewing: it provided sufficient task information at the proper abstraction level of expert goals to allow construction of a complete expert model for the users’ task. At this point our approach deviates from the GOMS-approach to system design. The plain interface is mainly based on the functional model which identifies goals and sub goals in the users’ task, but which is far from a complete expert model and does not provide a complete prescription of expert behaviour. The GOMS-approach would recommend a detailed analysis of the users’ task even for the design of the plain interface (e.g., Gray et al., 1993). In practice, we conclude that it is often too difficult to obtain accurate and complete behavioural information without a prototype of the base system. This may explain why GOMS is mainly used to evaluate and compare systems or system designs rather than designing them.

Often, controlled experiments are thought to be time consuming and expensive. Karat (1988) even maintains that they are held in a higher regard than their results would support. In contrast with this statement, our results show that experiments are valuable for the evaluation of (parts of) existing systems and for the evaluation of...
of human-computer interaction theory. To test the usability of software, one should analyze users performing tasks with the interface (cf., Desurvire et al., 1992; Karat et al., 1992). Summative evaluation focussed on performance and learning proves to be a good method to assess the utility of help; it identifies beneficial and harmful effects. Compared to the total effort of system design, it is not expensive, especially with the use of prototypes or mock-ups as in the HOMALS-experiment.

The HOMALS case study showed the method to be fairly complete, efficient and usable. The generality of the method may be limited, as not all types of tasks (e.g., writing) can be explicitly understood and modelled. The HOMALS expert model can be based on procedures. The question is whether procedures are suitable for complex, dynamic tasks with interruptions of tasks and parallelism such as process control. For such tasks, we may need different modelling languages and different aiding functions. In follow-up research we will test the method in another domain: the control of railway traffic. This domain is rather dynamic —many things going on at the same time, events happening— and one might expect difficulties with the repertoire of modelling techniques, as it is based on a single thread of sequential processing. However, if one assumes that the model can be embedded in a "main loop", the modelling techniques may be used for analysis/design of the expert model for the users' task.

HOMALS analysis is a difficult, but small task. The method would also be applicable for larger tasks, provided that it is possible to have a decomposition of small expert models that do not interact. If not, for example due to long and haphazard feedback relations, the behavioural modelling would become unwieldy. For such tasks aiding might not be possible and one might look for other forms of support (e.g., information support, Raaijmakers & Verduyn, 1993).

Another generality issue is whether the method could be used for real problem-solving tasks, such as planning and diagnosis. Modern models for such tasks are based on abstract procedures and can, in principle, be modelled with the same techniques (e.g., the KADS task and inference structure, Schreiber et al., 1993). A recent investigation of support of diagnosis showed that the method is helpful in generating and evaluating alternative allocations of functions to human and machine, (Post, 1996), but more research is needed.

In summary, the method appears to be rather complete and usable. Regarding the generality of the method, more research is needed to test whether it can be applied. With this respect, a follow-up investigation of aiding for railway traffic control is promising.

7.7.2 The Effect of Aiding

The empirical evidence regarding conditions for effective use of help information is meagre (Elkerton, 1988). This study shows a beneficial effect of a specific type of support, called aiding. The aiding facility of the HOMALS interface is integrated into the task execution of the user: the computer takes initiative to provide the user with complete context-specific task knowledge about how to do the task. The conclusion of the experiments is that this type of aiding is beneficial for users with a basic knowledge of the analysis task.

The HOMALS-interface is a result of the application of a design method in which the design of an aiding function is integrated into system design. Consequently, the
communication with the aiding facility is consistent with the rest of the communication and the overall communication with the WIMP-interface is easy for novices. Users who are lacking experience with this interface can almost immediately operate on it effectively.

With regard to the first experiment, conclusions can hardly be drawn. However, in comparison with the second experiment, it suggests that this type of aiding is not beneficial for completely naive users. For users with some knowledge about the HOMALS-analysis task, the aiding is effective. It may be that such users recognize the relevant information; without aiding they cannot recall it. This suggests that the aiding is especially suitable for casual users who cannot retrieve all relevant information from their memory.

In the second experiment, learning of the students correlated positively with their performance. This is in line with the research of Barnard et al. (1989) and Neerincx & de Greef (1993a). A knowledge test seems to be a good tool to assess the usability of software.

GOMS (Goals, Operators, Methods and Selection rules; Card et al., 1983) is a hierarchical model of expert computer use for simple tasks. In recent years, empirical research has sought to model complex user tasks in a GOMS-isomorph structure for system design (Elkerton & Palmier, 1991; John & Vera, 1992; Gray et al., 1992; Johnson, 1992). Such a model seems to capture real user characteristics. In the study of Gugerty et al. (1991), subjects recalled medical procedures in a GOMS-isomorph structure and remembered more in one condition if the procedures were presented in such a structure. The knowledge tests in the evaluation of the SPSS interface (Neerincx & de Greef, 1993a) and the second HOMALS experiment are in line with this finding: learning from the “unstructured” SPSS help information proves to be hard, whereas learning from the “structured” HOMALS information seems to be relatively easy.

The use of procedural knowledge results in efficient user behaviour. In the second experiment, users needed less often to look back at statistics already seen, that is, they did less actions. Furthermore, the cognitive actions may be relatively efficient; it may be mainly rule-based actions (Rasmussen, 1983; Reason, 1990). In other domains, in which task load is a major performance constraint, this type of aiding may be fruitful. Future research will centre on the design and evaluation of an aiding interface in such a domain: railway traffic control.

### 7.7.3 A Comprehensive Design Method

It is widely known that usability is an important success factor of software systems. Software companies have separate usability laboratories and methods for usability engineering are being developed. Current usability methods — heuristics, formal and empirical — center mainly around the evaluation of interface designs and are not well integrated into methods for system design. Consequently, usability issues are often addressed in a late design stage in which important decisions about the human-computer interaction have been taken already.

Our research started from current model-based design methods and extended both the analysis and design stage, thus incorporating HCI in the early stage. In this way, a comprehensive design method was developed which corresponds with current model-based design techniques for user interfaces. In the first step of this method, a plain
interface is built which is easy to use and learn. In the second step, possible knowledge deficiencies of future users are detected. In the third step, an aiding function which compensates these deficiencies is constructed and added to the plain interface. This function is based on a procedural expert model of the user's task ensuring completeness and allowing system initiative for presentation of context-specific task knowledge.

Investments in usability analyses can be very profitable. Nielsen & Phillips (1993), for example, found that the profits of choosing the best of two interface designs with one of the current heuristic, formal or empirical methods for usability testing are 1000 times greater or more than the costs of applying the method. A second example is the study of Gray et al. (1993) which conveyed that a new, intuitively attractive workstation would cause 3% slower task performance than the current workstation, which translates into an additional cost of almost $2 million a year to the telephone company. The design method for aiding can add on to current usability analyses: the costs of applying this method are relatively small due to the integration into system design, while the profits as shown in the second HOMALS-experiment may be as large as the two examples.
Acknowledgement. We thank Bernhard Slaap, Marjolein van Hooff, and Klaas van Aarsen for their contribution to the research. Our thanks are also due to the Homals-experts for their provision of knowledge about HOMALS.