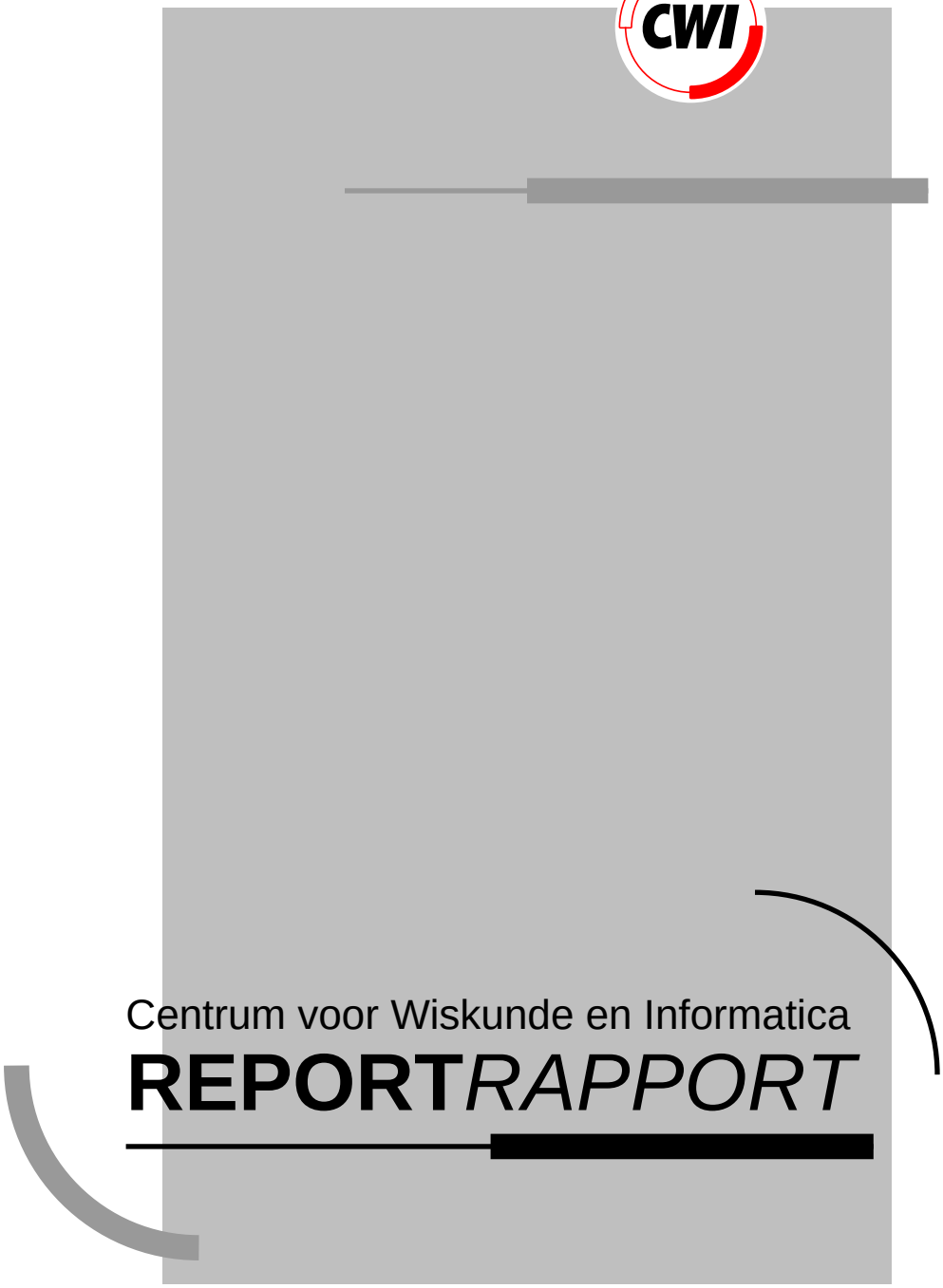

Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

The ergonomics of computer interfaces. Designing a system for human use

L.G.L.T. Meertens, S. Pemberton

Computer Science/Department of Algorithmics and Architecture

CS-R9258 1992

The Ergonomics of Computer Interfaces

Designing a System for Human Use

Lambert Meertens and Steven Pemberton

CWI
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands
Email: Lambert.Meertens@cwi.nl, Steven.Pemberton@cwi.nl

Abstract

This document discusses the ergonomic problems with currently available software products, and what in general is necessary in order to make an application pleasant to work with. The application of these principles to a new open-architecture user interface system, Views, is then described.

The
VIEWS
System

1991 Mathematics Subject Classification: 68U99.

1991 CR Categories: H.1.2, H.5.2, D39, H.5.0, I.7.2.

Keywords and Phrases: user interfaces, ergonomics of computer software, application architectures, information interfaces and presentation, document preparation.

1 Introduction

Current software products and applications are typically ‘non-cooperating’, in the sense that each has its own usage conventions and data-formats, even in computer environments (such as the Apple Macintosh) where interface standards are clearly defined. This lack of uniformity puts a high cognitive load on the user, increasing errors, reducing productivity, and limiting the potential users of a system.

This document discusses what makes a system pleasant to use in general, largely abstracting from the specifics of any particular application context, and presents the ideas behind a new open-architecture user interface system, Views. Ideally, the Views system will bring the users in control of their environment, and offer them a completely unified user interface for all applications running under it.

1.1 The changing environment

Within this century workstations that by today’s standards are extremely powerful will come to occupy a place in virtually every employee’s working environment.

This massive ‘invasion’ will be caused, of course, by the continuing rapid drop in prices. With current prices, equipping every office worker with their own powerful workstation would be economically entirely unfeasible, whereas providing them all with now affordable microcomputers would not result in productive use of the new technology. The time will inevitably come when affordable systems become available that can increase the productivity of almost every employee. Once this point has been reached, it is to be expected that the ‘take over’ will take place with unprecedented speed, even amongst those who have sworn never to touch a computer.

The key issue here is the productive use of the relatively new computer technology. Currently, computers are used productively by direct users (in contrast to being embedded in a larger system) only if at least one of the following four criteria is met:

- the user happens to be an able programmer;
- the task involves heavy computation, usually of a scientific nature;
- the task is of an entirely routine character, usually but not necessarily carried out by several ‘interchangeable’ workers collaterally;
- the task is one of a small list of ubiquitous ‘standard applications’, like word processing and spreadsheets.

In due time software products will appear with which a majority of potential computer-users can make productive use of affordable systems. This is clearly inevitable, since in ten years from now there will be a mass demand for such products. The availability of such products is, conversely, instrumental to the above-mentioned ‘take over’, and so they will help to create their own market, in the same way that spread-sheets and word-processing software have already advanced the sales of personal computers. With that ‘take over’, a change in the software market that already has started to take place will become irreversible. The software market will then become and stay predominantly a mass market, on which the quality of products will be assessed more by their ability to meet the manifold needs of an individual user than by the extent to which they conform to specific needs of organisational entities. Although there is no reason to assume that the turn-over of custom-made applications will decrease in absolute terms, its relative significance on the overall market will diminish.

Yet another change in the shape of the market is predictable. Whereas now commissioned software is mainly produced by the larger software houses, and many if not most mass software products are — at least initially — produced by very small groups, or sometimes even a single person, these roles will tend to be reversed. The kind of software

that will meet the requirements of the indicated future market segment is so complex, and the competition will be so intense, that only the stronger, better prepared producers will be able to acquire a share of the newly opened market.

1.2 Objectives of Views

The Views project [1] aims at creating a framework for the user interface of applications such that the system (given sufficiently powerful microcomputers or workstations) can be put to productive use by end users without much training in using computers, for varying tasks that are relatively complicated and not well structured. This is to be achieved by reaching a large and hitherto unparalleled degree of ease for the end user in controlling the environment, through the consistent uniformity and extreme conceptual simplicity of the interface and the high level of integration between functions of the environment.

The objective of Views, then, is the development of a conceptual framework and a methodology for designing and implementing integrated applications in the form of a 'framework' user-controlled operating environment that allows easy addition of functions and tools (including end-user applications) in such a way that the desired cooperation and uniformity is guaranteed, and the design of a single uniform interface both for built-in functions and for functions added later.

The architectural framework of Views can be described as an open system architecture, with interface standards both for intra-system data exchange and for user-system communication, in which it is very easy to add applications conforming to these standards. This ensures that it is possible to maintain a very high level of integration between various applications, including standard services.

Views is equipped with a number of standard services and document types, making it a flexible and valuable operating environment even if no special-purpose applications are added.

2 Ergonomic principles

Although software ergonomics as a science is still in its infancy, the active research in this area has already produced many valuable insights concerning factors that influence the productivity in the use of software. In particular, the research has concentrated on the cognitive ergonomic aspects of the user interface (or human-computer interface). One of the lessons that has been learned is that human beings are quite varied (for example, in planning: ad-hoc *vs*. planning ahead; and in cognitive abilities or skills, like the skill to use formal systems or the ability to use spatial cues to their advantage) [2]. Most competent, intelligent and successful people score low on one or more of these many skills. In coping with life, they often have *bypass* strategies: they use other skills to overcome deficiencies. Many user interfaces are unsatisfactory because they are *too narrow*: they require a certain fixed combination of skills and allow no substitutes. A good interface must cater for a variety of users.

In the use of computers for solving tasks we can distinguish between more human-oriented and more computer-oriented approaches in human-computer interfaces. A human-oriented *task analysis* is a key issue in the design of good user interfaces. A substantial part of this analysis can be made relatively domain-independent, and thus it is possible to obtain insights of a general validity that have a bearing on the design of user interfaces, such as: evaluation is an integral part of human cognitive behaviour; humans are in general constitutionally unable to perform a complex task according to a fixed 'programme' (stack-wise, or breadth-first); switching task contexts in the middle of a task is expensive and error-prone.

From such ergonomic principles, it is not hard to understand why it is that current software cannot be put to sufficiently productive use in a context of varied and ill-structured tasks. Looking at typical present-day products, we find the following kinds of shortcomings.

No integration

To obtain the desired results, end users often have to combine the workings of several applications. Each application has its own requirements on the format of the data. This requires more often than not a manual intervention of the user, for example by means of a text editor, to massage the data output by one application into some required input format for another. Another problem is that once the data has been imported, its structure is often lost as a consequence, so that to make a change the data has to be reimported, rather than changing it *in situ*.

Inconsistency and mode confusion

Each application and the operating system itself have their own mode of addressing, and some applications have several such modes, exacting from the user not only knowledge about all these modes and the way of expressing commands and functions under these modes, but also at all times awareness of the current application and mode, which is a high cognitive load, and thus a source of errors [3].

Loss of context

It is the rule rather than the exception that in performing a task a user finds somewhere on the way that certain data needed to proceed is not immediately available and must be obtained somehow outside the context of the current task. Many systems allow such task switching only if the pending task is first brought into some well-defined, quiescent state: rather than being suspended, it must be 'closed'. This means then that the context of the unfinished task is lost and has to be explicitly restored by the user at the time it is resumed.

Inflexibility

The phenomenon of inflexible software, especially as regards the input format, but also concerning the order in which the user can take certain steps, is so well known that it need not be laboured here.

Arcaneness

The exact forms or formats required for commands or functions that are not regularly used are often such that it is impossible to remember them when they are needed, so that they have to be dug up from huge and usually hard-to-interpret manuals.

The 'Swiss army knife' syndrome

Many software applications 'package' functions, like, for email, finding a letter (retrieval), archiving it (storing), composing a letter (editing), etc. Such functions are in fact emasculated versions of much more general functions that happen to be applied to one specific context. This can lead to a tremendous duplication of functionality, and increased inconsistency, entailing the constant danger of mode confusion and adding to the overall complexity of the user interface.

It is almost as if these shortcomings have been purposely designed with one single objective in mind: to thwart users in their attempts to make productive use of technology. For together they conspire to put such a cognitive load on the user, distracting from the real task to be performed, that a large rate of user mistakes becomes unavoidable. It is

especially at routine tasks that computers easily outperform human beings, who start making errors due to waning attention just in repetitive tasks, and yet a substantial part of the effort in using computers as an intellectual tool consists of clerical tasks without intellectual content, like transforming data formats. The current situation forces users to spend a substantial part of their attention continually on low-level aspects of the communication, even down to the motoric level, rather than on the conceptual level of the task to be performed. While computers are far better than humans in keeping track of the precise status of a task in operation, it is the user who is forced to commit the details of the context to memory, not the computer.

The mistakes made are most of the time just ‘silly’ mistakes, causing annoyance and loss of time but no great damage. Every now and then, however, an entirely understandable user mistake is costly, causing wrong results, or the loss of much work, or even irretrievable loss of information, causing great anguish. Even without such mishaps, the total time users spend on guessing, being puzzled, or trying to find out something quite trivial, is staggering.

However, these ‘unavoidable’ mistakes are unavoidable only because of the failure to apply the most basic ergonomic principles to the design of the user interface [4]. An important prerequisite for a good interface is that the system appear to the user as an integrated system with a single interface, rather than a collection of disparate interfaces, one for each different function of the system, however well designed each individual interface may be. This, then, requires in its turn that one single conceptual framework underlie the whole system, a framework in which diverse applications can be integrated in a natural way.

3 New trends in user interfaces

The new trends in user interfaces are the result of the attempts of software designers to overcome some of the problems mentioned above.

WYSIWYG

The term ‘WYSIWYG’, What You See Is What You Get, is usually applied to text formatting systems in which a document being edited appears on the screen as it will appear when printed in hardcopy form. The popularity of WYSIWYG systems is due to the immediate feedback given to the user. Although it may be argued that the amount of user control in more traditional systems is equally large, this argument denies the psychological reality of the immediacy of control. Since silly mistakes are directly apparent, they can be corrected instantly in the WYSIWYG approach, whereas the users of the traditional systems have to re-create their intention within the context under the indirect edit-print-proofread-correct loop. Also, the possibility of repeating a particular mistake consistently throughout the preparation of a document becomes remote.

The same principle can be applied in a much broader sense to all kinds of systems in which the effect, in logical terms, of a user action is made instantly visible, providing feedback. In that sense all arcade-type of computer games are examples of WYSIWYG systems, but also spreadsheet applications that continually update the derived information. A further extension of the principle could be stated as ‘TAXATA’: Things Are eXactly As They Appear. It may seem that this is a consequence of WYSIWYG, but in practice the equivalence is effective in one direction only: the effect of a user action is made immediately visible to that user, but not the effect of changes from other causes. In fact, it is usually the case that the user edits a *copy* of the document, rather than the document itself, so that such changes could not become visible. If the operating environment permits a task switch without leaving the edit application, the co-existence of two versions may then lead to user confusion.

Direct manipulation

Rather than giving commands in some arcane command language to a slave who, if not complaining about syntax, scurries off to execute the command in the hidden dungeons of the computer, the user ‘directly executes’, for example, the printing of a document by dragging the icon of the document to the icon of a printer. Although it is still the computer that executes the command according to the directions given by the user, psychologically it is the same as the real physical execution of the action by the user. To use a metaphor, the traditional dialogue mode is like taking a taxi ride with a driver who has to be told to turn left or right all the way while Direct Manipulation is like driving the car yourself. An important advantage of Direct Manipulation is that there is much less chance of making an error in phrasing the command, there is a notable increase in the uniformity of the external form of certain functions that can be modelled on the same or similar metaphors, and the user gets explicit visual clues on how to express the action.

The experience has been indeed that novice users can learn to use systems based on Direct Manipulation productively in a fraction of the time needed for more conventional systems. As with WYSIWYG, the notion of Direct Manipulation is still used in a narrower sense than warranted: restricted to spatial metaphors such as with icons on a desktop and shoving them around. A much more common form of user control that can also be viewed as being a form of Direct Manipulation is that of entering text while editing a document: each key hit appears on the screen, and to the user this is psychologically no less direct than when using a mechanical typewriter. Yet another form is that in which modifying the name of a document as it appears in a caption while it is displayed also renames the document name as it is stored in the archiving system.[1]

The common aspect here is that the psychologically physical action by the user (moving an icon, hitting a key, changing a name) has a direct effect in the universe of the computer, which obeys certain logical, as it were pseudo-physical, laws. Indeed, in the real world physically changing the name of a document, say the title of a book on a shelf, causes that book to carry a different title. This last example can also be seen as an application of the WYSIWYG principle: what you *see* as title in the caption is what you *get* as title for the document. Carried to their logical extremes, it can be argued that the WYSIWYG and Direct Manipulation coincide, and in arcade computer games they do indeed.

Integrated information systems

The term ‘integrated’ is increasingly used as a sales argument for systems that do not merit this epithet. Yet it is not a buzzword, but has a clear and definite meaning, albeit hard to formalise. An information system is integrated if it presents itself to the end user as a collection of functions and tools that are able to cooperate smoothly and that can be handled by the user in a uniform way. It is a well-known and not surprising fact that software packages usually do not cooperate smoothly. Each time, a separate substantial effort is needed to get them to cooperate. This sometimes requires access to the source code, which may be unavailable, and generally requires skills that cannot be expected from end users. In view of the consequences of lack of integration for user productivity, it is easy to understand that truly integrated systems are among the best-selling software products. Available integrated systems consist of some cooperating functions and tools (e.g. a text editor, a spreadsheet part and a database part), but are not extensible with new functions and tools. Typically, each package has either a ‘friendly’ user interface, but

1. In some well-known and quite successful commercially available systems based on Direct Manipulation, this specific user action is not possible. To novice users of such systems this comes as a surprise.

then each package has its own conventions, or a rigid interface that cannot easily be adopted to another style. Although modern interfaces have a WYSIWYG approach, this is usually only partially pursued and not fully taken advantage of.

‘Intelligent’ systems

In contrast to the preceding terms mentioned, the properties that warrant calling a system ‘intelligent’ are not well delineated, and it may be argued that even the least dumb systems are not worthy of being ascribed intelligence. But, at least, a trend is discernible from exceedingly dumb towards somewhat less exceedingly dumb, and therefore more intelligent, systems.

To make this more precise, the first step is to make clear that the ‘intelligence’ is to be understood in a very narrow sense, namely that of providing assistance to the user in performing a specific intellectual task, consisting of the (usually repeated) identification, evaluation and selection of alternatives, usually with the additional requirement that no decisions are definitive but can be changed or undone at any time.

There are many examples of intellectual tasks that fit this description: all kinds of design (not only industrial, but also system design, or the design of, say, a marketing tactic, a departmental reorganisation, or the floor plan of a stand on an exhibition), all kinds of planning and scheduling (work charts, rosters, assignments, routing), packing problems, and tasks involving combinations of such problems. (It would not be unreasonable to call systems for providing support to such tasks generically ‘Decision Support Systems’, but currently DSS’s are generally understood to address specifically management decision tasks, and also include kinds of systems that do not fit the above characterisation.)

Still few in number, such ‘intelligent’ systems will become increasingly important in the whole software market. The diversity is too large to handle this category in a uniform way, especially since successful programs in this category concentrate on a single type of task, but the common aspect and the key to their success is that these programs combine domain-specific expertise (often not only in the form of clever algorithms or an embedded expert system, but also in the form of brute force, made feasible by the dramatic increase in computing power) with a highly visual and interactive user interface. The user has the freedom to try certain decisions and is presented (ideally instantaneously) with the effects of that decision in combination with the earlier decisions taken and with an overview of the remaining design freedom given the constraints of the particular problem. This corresponds to what is known in the DSS world as a ‘What If’ facility.

Thus, this form of user-system interaction, which is crucial to the productive use of such systems, is a combination of the generalisation described above of the WYSIWYG and Direct Manipulation principles, where the laws referred to are not given once and for all, but are specific to a given problem setting. It is also worth remarking that for simple problems of this general kind a spreadsheet approach may be quite satisfactory.

Fourth-generation languages/systems

Although there is a reasonably common understanding what is meant by ‘Fourth-Generation Languages’ (or ‘Systems’) [5], there is also a surprising agreement that most systems marketed as such fall short of their promise. Still, whether such systems meet the agreed definition or not, they often succeed in what is their primary aim: to effect a dramatic increase in programmer productivity.

It should be pointed out that they have a limited domain of applicability, which, however, is important by virtue of the number of applications fitting that domain. Fourth-Generation Languages/Systems provide a convenient way to describe certain applications (which, if that is what is emphasised, gives rise to calling it a ‘Language’), and these

descriptions can be used to generate programs realising the application (whence ‘Systems’). The success is also due to a form of integration, albeit of a different form than described above. Here, the integration is achieved, not by the uniformity of data exchange between the parts of the final application (such as a data-entry part, a database, and a report generator), but by the fact that the program generator also generates the programs for the necessary adaptations of data formats between these parts, thus taking away this drudgery from the application programmer.

The kind of application generated is usually highly conventional and not substantially different from applications as they were written fifteen years ago; what has changed is the time needed to produce them. The obvious next step is to use similar techniques in a more flexible way to generate more diverse and modern applications, and a few software products that have taken large strides in this respect have already appeared. A further obvious step that will inevitably be taken sooner or later is the integration of intelligent tools for system design with such sophisticated application generators.

Rapid prototyping

The term ‘Rapid Prototyping’ is used in two related but distinguishable senses. The first is that in which a fast technique is used to realise a fully functional but possibly inefficient prototype of an information system. This makes it possible to recognise and correct certain shortcomings, for example in the user interface, before tremendous investments have been spent on the coding phase. Increasingly, it is the case that the prototype that is deemed satisfactory (from a functional point of view) after a number of iterations is not thrown away, but is used as starting point for a production version by identifying bottlenecks and crucially time-efficient parts of the system and recoding only those. The second meaning is that in which it is possible to create a mock-up version that is not fully functional but models, for example, only the user interface. In such a case, shortcomings in the user interface can be identified in a timely fashion. This is important since experience has shown both that a large part of the major deficiencies tends to be in the user interface, and that coding the user interface is relatively costly [6][7]. The distinction is not sharp, and it is increasingly possible to ‘grow’ prototypes by piecemeal addition of functionality. In a sense, Rapid Prototyping of the first type is what Fourth Generation Systems try to achieve, with the addition that the latter already aim (not always quite successfully) at production-quality code for the generated result. If Rapid Prototyping is seen as distinct, this is because it usually has less strict limitations on the application domain, and because, as a result, the techniques are less complete.

End-user programming

The wider group of end users and the attendant larger variety in tasks has spurred research into methods by which end users can build their own relatively simple applications from available building blocks. This is basically a form of programming, and although programming is intrinsically hard, this task can be eased in several ways, to an extent that it comes within reach of many end users.

Open system architectures

The trend towards open system architectures has a significant industrial basis and economic advantage as driving force. It gives the vendor a bigger potential market, while offering the user more choice and less vendor dependence. It holds the promise that the kind of integration that is needed can be achieved not only within single systems, but also without much additional effort for whole networks of systems.

Object-centred programming

It is not accidental that the technique of object-oriented programming was developed by the same research group that gave the impetus for current WYSIWYG and Direct-Manipulation environments. The coupling of operations relating to objects to the objects themselves has made it far easier to specify and build flexible environments. For example, there is then no need to provide a print capability that is able, once and for all, to print all conceivable kinds of documents, which would effectively limit the set of available types of documents and thus impair flexibility. An application designer introducing a new kind of document has, instead, to specify what it means for such documents to be ‘printed’. In combination with an open-architecture approach, this means a large increase in the flexibility of systems. A step beyond this is the new paradigm of object-centred programming, in which objects are no longer necessarily passive most of the time, responding only to messages sent to them, but can actively monitor the system state. The functionality of the system results from the interplay of the objects.

From office automation to personal computing

Of the new trends discussed here, this is the one that has as of yet come least to expression. The prevailing point of view is yet one in which Office Automation is seen as being an information system that is logically and even physically separate from Personal Computing environments used in the office. Those (still few) managers who have started to make use of information systems usually do not find that existing Office Automation systems meet their needs and use an environment designed for Personal Computing instead.

Information plans for new information systems to be designed for an enterprise increasingly cater for interfacing between a central or departmental information system and decentralised personal information systems. Also, to meet user demand, more and more facilities that are commonly viewed as being personal-computing facilities, are being built into information systems designed for office automation. The trend here is that eventually the whole distinction will disappear, at least from the point of view of the individual user. The only difference that will remain, compared to a collection of purely personal and unrelated information systems, is that certain information is shared and that workers may have different rights with regard to accessing or modifying the data and responsibilities beyond their own immediate needs for keeping the information up to date.

3.1 The relevance of the new trends for Views

To result in a framework that can serve as the basis for a viable product, the research carried out has to take all the trends sketched in the previous section into account. A single significant shortcoming in any one of these aspects may be sufficient to doom products to eventual failure on the future market. It is therefore mandatory that the architectural framework be based on a unifying conceptual framework that is so versatile and powerful and nevertheless conceptually so simple that it can be established a priori that its embodiment will allow to meet the most demanding requirements. Such a framework is described in the next section. A separate discussion of its relationship to the new trends mentioned here can be found later.

4 The conceptual framework

The best way to achieve conceptual simplicity in software systems design is to use the power of mathematical abstraction in the design and description of the underlying concepts. The approach that will be followed here has been chosen by virtue of its extreme conceptual simplicity and of its power to directly model the key aspects of most of

the ‘new trends’ identified. The others, insofar as they are not properties that are meaningful only on the level of the system architecture, appear as what is known in phenomenological terms as ‘emergent properties’ of the system.

An informal description of the conceptual framework will be given first, and then it will be briefly indicated why this is a powerful basis indeed for modelling the crucial aspects.

Before the conceptual framework is described (informally) in mathematical terms, two remarks are in order. The first is that the framework described here is emphatically not the architectural framework of the envisaged system. It gives a way to reason about the behaviour of the system, neither more, nor less. In particular, it has no implications whatsoever for the way an embodiment displaying this behaviour is to be obtained. The second remark is that end users need not be aware of, let alone understand, the mathematics behind the framework, any more than they need understand the physics of semiconductors. In spite of this last remark, it is claimed however that the conceptual framework corresponds to a natural way of describing in mathematical terms the kind of models end users develop about software that follows the WYSIWYG and Direct-Manipulation paradigms, so that reasoning about high-level ergonomic aspects of software is facilitated by the conceptual framework.

4.1 Objects with maintained relationships

For the purpose of this exposition and in particular of the subsequent discussion, the terms ‘document’, ‘object’ and ‘form’ will be used interchangeably. The difference is one in emphasis. The term ‘document’ is most appropriate if we keep the user’s logical view of certain data kept in the system in mind. Documents can be conventional textual documents, but also tables, forms, and pictures (e.g., a ‘desk-top’ with icons), and arbitrary mixtures of these. The term ‘form’ is most appropriate if we want to stress the external appearance of a certain kind of documents. The term ‘object’ is neutral as to the user’s view, but emphasises a prolonged identity in time during which the value (contents) of an object may change. Note that the distinction between documents and forms, although comparable to the distinction between ‘conceptual schema’ and ‘external schema’ in the ISO/OSI reference model [8], does not necessarily coincide with it; for the present conceptual framework the distinction is in fact irrelevant.

The conceptual framework is now simply that the system consists of a collection of objects that are linked by certain ‘logical’ relationships that hold between their values. This may be described in mathematical terms as a graph or network whose nodes are objects and whose edges are the links connecting objects. Objects may be atomic, but also structured (composed of other objects). In the course of time, new objects and relationships may be created or existing objects may be annihilated, but always following a certain discipline that ensures that the logical consistency of the system can be maintained. There are ‘autonomous agents’ that may ‘spontaneously’ cause objects to change. The user, editing a form, is such an agent, but other agents may, for example, be the ticking of a clock, or the mail delivery subsystem depositing mail (documents) in the user’s mailbox (which is itself an object).

Such a change may, for a fleeting instant, invalidate some of the relationships linking this object to other objects. However, there are ‘daemons’ guarding these relationships, and whenever the validity of one is impaired, they intercede and restore it by also changing the object at the other end of the link. This may, in turn, invalidate other relationships, which then will also be restored. Thus, a single change may in principle trigger a cascade of changes throughout the network of linked objects, so that the ‘events’ generated by the autonomous agents drive the semantics of the system.

In preliminary research conducted at CWI, it has already been shown that the overall maintainability (a new consistent system state can always be reached) can be guaranteed if certain simple conditions are met by each modification to the network. With the

additional natural constraint that the network contains no other cycles than imposed by the algebra of functional composition, it appears that these conditions can even be checked locally and incrementally. This research has also identified certain ‘normalising transformations’ that facilitate reasoning about a network.

We stress once more that the relationship between the system as implemented and the conceptual framework is primarily at the level of system functionality. For example, it is entirely possible that certain objects that are present in the conceptual view of the system have no independent physical existence in its realisation, but exist merely as ‘implied’ by virtue of their logical relationship to other objects.

4.2 The user interface

It is desirable that the user interface is based on a unifying conceptual framework that is versatile and powerful (and nevertheless conceptually simple). Direct manipulation can provide an easy-to-learn interface, but not necessarily an easy-to-use one. The approach described here assumes the existence of a general structure editor. The basic idea is that functions exerted by users are modelled as direct manipulation on some structured document. The manipulation is most comfortably thought of as ‘editing’ the document. Since modern editors also permit viewing and browsing, these functions are then subsumed. The user action of starting an edit session on a document, perhaps only with the purpose of viewing it, will more neutrally be called ‘visiting’ that document.

4.3 Some examples

The best way of introducing the ideas is to describe some typical applications.

Electronic mail

Consider an email facility, in particular for handling incoming mail. The user’s mailbox can be modelled as a structured document, consisting of a sequence of letters. Letters are again structured: they have an administration part identifying the sender, subject, time received etc., a default title derived from this information, and a body. The user can visit the mailbox, which is represented on the screen not as a sequence of full letters, but as an index (a sequence of identifications, or headers). Using standard editing commands, the user can walk through the index, or move to the last item, or search by some text pattern. If the user wants to view a particular letter, this is done by focusing (with the edit focus) on the corresponding index entry and giving the edit command VISIT. This would then open a sub-window to display the letter. To end the visit, there is a command EXIT.

Mail archives can have the same structure as mailboxes, and to archive a letter the user has only to copy (using generic editor commands) the index entry from the main mailbox to the archive. To delete a letter from a mailbox or archive, it suffices to delete the index entry.

This use of an editor for reading mail obviously puts some ergonomic requirements on the editor itself. Assuming that this can be solved in a satisfactory manner, the obvious advantage is that the user does not have to learn any new commands (next to the editor commands which have to be learned anyway) to read mail.

Document maintenance

The same model can be used for document maintenance in general. In traditional operating systems, documents are called files. These are usually organised in ‘directories’, which term is used both for a ‘workspace’ of documents and for the index to this workspace. To see the index, some specific command is used, for example ‘LS’, or ‘DIR’, or ‘CATALOG’, depending on the brand of operating system. To find a document if it is known that its name contains ‘dav’, the user can give a command like ‘LS *dav*’. The

system responds, e.g., with ‘letter.from.david’. To see this document then, another command is needed like ‘TYPE letter.from.david’ or (rather arcane) ‘CAT letter.from.david’, to rename it something like ‘MV letter.from.david letter.from.dave’, or ‘RN SRC=letter.from.david TRG=letter.from.dave’ or whatever, and to delete it ‘DEL letter.from.david’ or ‘RM letter.from.david’.

Now many of these commands result in a change in the index. This change is not displayed, and so a common thing is for users (and not only novices) immediately to issue another ‘DIR’ or ‘LS’ to obtain (hopefully) positive feedback. Instead, we can simply allow the user to edit the index directly. To find a document whose name contains ‘dav’, the user uses the edit command FIND (possibly a single key) with the search pattern ‘dav’. To rename the document, the user can simply position on the letters ‘id’ and replace them with ‘e’. To delete it, the user can simply delete the entry from the index document using the editor DELETE command. It is then reasonable that the editor’s COPY capability can be used also to make a copy of a document.

For viewing the index in the first place, the user does not have to know a separate command like ‘CATALOG’ or ‘LS’, since the index can be ‘visited’ using the editor like any other document. Similarly, a ‘TYPE’ command is superfluous, since here the editor can likewise be used to view the ‘file’.

To model this using the conceptual framework, all that is needed is the invariance of the relation: contents(index) = collection of document names in workspace. This invariant also ensures that if the index is being displayed and a new document gets created by some process, the index is updated to show the presence of this new document. It can thus be seen that a plethora of file maintenance commands can be replaced by edit commands that the user has to know anyway, and the desirable feedback is guaranteed.

(This is already the case for modern operating environments based on Direct Manipulation like that for the Macintosh. Still, the user has to learn and use sometimes quite different mechanisms for document maintenance than for text editing. For example, although the DELETE key will delete a selected piece of text in a document, the same key cannot be used to delete a selected entry in a workspace index, but instead it has perhaps to be ‘dragged’ by means of the mouse to a wastebasket; similarly, the methods for copying and moving objects are quite different)

Printing documents

In traditional systems, there are commands to print documents, like ‘LPR letter’ which usually spool the document to a printer queue, a command to show the state of the printer queue, a command to delete items from the queue, such as ‘LPRM 1221’ where the identifying number has to be obtained from the display of the queue, as well as system-administration commands. In the Views approach, the queue can be maintained as an index of entries, to which the user can append items by pasting, delete them using the DELETE operation, and where allowed, reorder entries with cut and paste. Additionally, keeping the index on display lets the user monitor the progress of print jobs.

Job control

In operating systems that allow multi-tasking, jobs have some ‘status’. To change a job status (for example, to ‘kill’ a running job), a user has first to find out the job’s process number, for example by issuing a command ‘PROCS’ (which then displays the current jobs with their statuses and numbers), and then to give some command like ‘KILL 786623’. Here, just as in the preceding example, the job statuses can be maintained in an index document that the user can edit, and to kill a job all that is necessary is to select and delete its entry. Similarly, a user could by editing change a job status from Suspended to Running, or change the priority of a job.

Thus, the user has no need to learn new commands here. And, just as above, by keeping the document on display, the user can simply monitor the progress of jobs without having to repeat some command.

A comparison with modern operating environments is not possible, since none offer such multi-tasking facilities yet except perhaps in a clumsy fashion. However, an even simpler approach to multi-tasking is possible than the one sketched above, which is still close to the user-hostile approach that prevails in traditional operating-system design.

'Jobs' in Views are nothing but tasks deriving from the maintenance of invariants that have not yet been completed. Both in traditional and in modern operating designs, a user has normally to await the completion of a command before anything else can be done (including giving a 'PROCS' command!). In some operating systems it is, however, possible to suspend the currently executing 'foreground' job (with yet some other mechanism), after which it is possible to issue new commands, for example to let the suspended job run again in the 'background'.

In an operating environment based on the Views conceptual model, there is no reason to prohibit the user from visiting another document while the currently active document is not quiescent (the restoration of invariants is not finished). (It may however be the case that the contents or even the existence of the document to be visited crucially depends on the task initiated from the prior document, and this can entail logical restrictions on what the user can do.) Tasks resulting from an active document would correspond to 'foreground processes', and other pending tasks related to visible documents would have higher priority than those involving only invisible objects. Thus, the switching from 'foreground' to suspension to 'background', or the administration of priorities, is unneeded.

Likewise, to 'kill' a job it is sufficient to edit the non-quiescent document to some quiescent state. Indeed, any other (non-equivalent) way of job killing available to the user would mean a command to the system to give up trying to ensure the consistency of the system state, which is clearly undesirable.

It is worth noting that this form of multi-tasking, although to the user functionally equivalent to what is usually offered on the basis of a multi-processing system, can actually be implemented on a single-process system.

Thus, one by one the facilities offered by current systems can be 'explained' away: reduced to direct manipulation, using a standard editor, of documents representing relevant aspects of the system state. The resulting system is both easier to learn to use in terms of its operation and as far as conceptual understanding is required, since the interface is uniform, everything that is relevant can be made visible, and the user can see the immediate effect of all actions. If designed properly, it is also easier to use. Strict adherence in the design to the conceptual framework induces finding conceptually simpler approaches, as illustrated by the example of dealing with job control. As with any new paradigm, it will require some time to learn to cut the ties to the traditional operating designs and use the new possibilities to their utmost.

5 The editor

As all communication between the user and the system takes place by means of an inbuilt editor, the design of this editor requires more care even than user interface design does in general. This editor operates on 'forms', that is, the external appearance of documents, and so it is called a 'forms editor' here. The relationship between the forms editor and interface design in specific applications is treated in more detail later. Here, the focus is on general requirements for the editor.

The most important functional requirements for the editor are as follows:

- There is one *single* editor for *all* kinds of documents (that is, it is a so-called 'generic' editor). It is not envisaged that all commands of the full command set will be meaningful for all document kinds. For example, a rotate command for pictures is not applicable in a purely textual document. The rule is, however, that if a specific command is meaningful in a given context, it is also allowed there and has the expected effect.
- It should go without saying that the forms editor is fully visual. This implies also that the editor has no memory of its own; the effect of a command depends only on the state of the form and not on the way that state was reached (with the exception of the UNDO command mentioned below). However, for some commands that take a parameter (for example, FIND <pattern>), omission of the parameter may mean the last parameter specified.
- The forms editor supports both structured and unstructured editing operations, fully intermixed; that is, these are not separate editor modes. More precisely, usually the form visible on the screen is linked to a more structured internal document. Insofar as applicable, edit operations are interpreted as operations on the internal document, rather than on the unstructured external form. For example, it is possible to copy a numeric value from a field where it is not represented in textual form, but, for example, in the form of the position of a slider on a scale, to a numeric field whose external representation is textual. However, operations that are meaningless in terms of the internal document but meaningful in terms of the visible form are also permitted, insofar as not prohibited by some constraint implied by the linkage. Thus, it is also possible to copy a text from a purely textual document without internal structure to a numeric field, but only if that text has a number format. Also, the external form may contain the equivalent of 'protected fields' in data-entry systems. These are added by the function that converts the internal document to a screen representation, and can therefore not be changed, since that would result in an irreparable violation of the link invariant. They can, however, be copied just like any other text.
- There is a general unlimited UNDO command, unlimited in the sense that not only the last but an arbitrary number of edit operations can be undone.[1] The effects that these operations had on the system state are undone as well — nothing less would be consistent with the conceptual model. Thus, if the user accidentally deletes a document, a sufficient number of UNDO's will bring it back. Some actions, like printing a document, or sending mail, can in general not be undone. This limitation applies only insofar as physical effects external to the 'metasystem' consisting of user and system are caused by edit operations; otherwise, all operations can be undone. Not only is the presence of an unlimited UNDO facility one of the tenets of ergonomic research, it is also vital in an environment in which all user actions take immediate effect.
- The number of editor commands is limited. A good way to reduce the number of commands is the introduction of a so-called 'edit focus' for forms (a highlighted area, which generalises the notion of 'cursor position'), and the strict separation of commands to move the focus around and of commands that perform an operation

---

1. *In an actual implementation there is some limitation implied by the finiteness of memory, but the limit is much higher than the one or two UNDO's that are usually permitted, more in the order of a few hundred UNDO's. Preliminary research at CWI has shown that, in contrast to what is widely believed, it is possible to implement such a facility with very little overhead [10]; in fact, it can be naturally combined with facilities that make the distinction between foreground and background memory fully transparent to the user, and that at the same time provide a sophisticated back-up protection in case of mishaps like power failure (but not against disk crashes, of course) [11].

on the contents of the focus. This has also advantages from the point of view of cognitive ergonomics, such as that it is visible in advance to the user what exactly will be deleted by a DELETE command. The last focus is remembered with each form when it is not being edited, and thus revisiting a form brings the user back to the prior context.

- The forms editor is capable of constraint checking on the input, where the constraints can be specified by the application. Thus, it is possible to ensure consistency and integrity constraints at all times, rather than in an edit-check loop.
- On ‘rigid’ forms (with a fixed syntactic skeleton) the forms editor behaves like a data-entry facility. This is not the result of a special provision in the editor, but the effect of its general behaviour when applied to such forms.
- The forms editor understands the notion of ‘index’ to a document, and allows visiting indexed subdocuments with one keystroke if the focus is on an index entry. For hierarchical documents, the index can mirror the hierarchy. If the user ‘exits’ a document and later ‘revisits’ it, the context will be exactly as when it was last active — in particular, the document focus, and the mode of representation if there are several options.

As has been mentioned already, the editor has a central role in the user interface of Views. Flaws in the design of the editor will inevitably impair not only the usability of the whole system — which for a prototype system can be acceptable — but also the possibility of judging its merits. A very careful design of the editor is therefore called for. The initial design of the forms editor takes into account extensive existing experience with several quite diverse editors developed by the authors, which in their variety share already most of the required functionality [9]. Nevertheless, it is foreseen that at least one major design iteration will be needed to come up with a version of the editor that is satisfactory. Fortunately, the editor is a module of Views that is relatively easily singled out, and re-design of its user interface aspects need hardly involve its interface with the rest of the system.

6 Application design

A new application is added to the system by adding a description of one or more new document types, giving the relationship between the external representation and the logical view and logical inter- or intra-document relationships. These descriptions, being documents themselves, can be constructed largely using standard tools of Views, and for many simple applications creating them should be within the reach of a sophisticated user.

The exchange of data between documents is standardised and invokes appropriate user-transparent data transformations whenever necessary.

For example, to create an application in which data is entered, modified and retrieved through an application-customised form, using standard editor commands for entering and modification and standard ‘database’ retrieval commands for retrieval, it suffices to provide an appropriate definition of the form and its relationship (‘logical view’) to the data described. It is equally possible to define relationships between fields or sub-forms of a form, extending the concept of a spreadsheet to arbitrary forms, by providing appropriate annotations to the logical-view definition. The user can transfer data between documents with different external representations but compatible logical views, but also between documents with incompatible logical views if the standard external representation is the same. These two facilities correspond, respectively, to structured and unstructured edit operations.

6.1 Methodology

Although Views should greatly facilitate the design of applications that are naturally integrated with the whole collection of facilities offered by it, there is nothing that can stop a determined application developer from designing and implementing an application that violates the spirit of Views in every conceivable respect. A more serious problem is that it is not guaranteed that an application designer who wants to conform to the spirit of Views and achieve integration will naturally choose in all cases the most appropriate solution. As in any sufficiently powerful and flexible system, there are usually more roads that lead to the same place. The development of a methodology for application design, in parallel with the development of the system itself, aiming at a certain uniformity in design decisions, is therefore in order. Among the aspects that should be covered are: how to model actions in terms of relationships; how to decompose complex relationships into primitive ones; how to validate the consistency of the composition of relationships; how to describe new document kinds, and how to base them on standard, predefined kinds of document; when to write program code for effecting a change of data representation and when to resort to standard representations; how to design external forms; when and how to design and implement editor extensions for new kinds of documents. As currently envisaged, a document describing the methodology would take a practical rather than a theoretical form, giving various illustrative worked-out examples, applicable rules-of-thumb and ‘cookbook’ instructions.

It is certainly possible to provide the application designer with a sophisticated workbench, including tools for systematic development starting from a more abstract initial design, for logical analysis, and for version control. To a large extent, it should then be possible to support the methodology with the tools made available. From the point of view of Views, such a workbench is just another application. It is currently not foreseen that the prototype version will be equipped with such a state-of-the-art workbench. This may be attractive for a future production version. The standard tools provided by Views will already surpass, in many respects, the facilities offered by many if not most environments under which applications are currently developed.

6.2 The role of the interface

A possible point of contention is the reasonable objection that an optimally designed user interface for a given specific application is unlikely to concur with the default interface offered by Views. The application designer should therefore have full control of the user interface (which would defeat a major aspect of Views).

The thesis implied by the first part of this argument is undoubtedly true. It applies, however, to the still prevalent situation in which applications are stand-alone programs, of which the user uses only a few, and one at a time: a user ‘enters’ an application, stays there for a considerable time, and then ‘quits’ the application. In a system based on the Views framework, a user does not ‘enter an application’, but ‘visits’ a document. Although at any time at most one document will be the active one, several documents can be on display simultaneously, and the user can arbitrarily switch among them, copy parts from one document to another, browse through yet another document not yet on display and next resume the original task, and so on. This is indeed what is needed for the user at which the design of Views is aimed. In such a situation, not only is it mandatory that there be a high degree of uniformity in the operation of the editor on various documents, but also in the resulting semantic effect of comparable operations.

6.3 Some examples

In this section a few examples are given of what has to be done by an application designer to add an application to the framework system. The examples have been chosen to be illustrative of the possibilities, and are not at all typical of likely mainstream applications. It

is assumed in some of the examples that a rather sophisticated ‘picture’ document kind is standardly supported, comparable to some of the best current commercially available picture editors. In actuality (and in view of the foreseen kind of applications) the prototype system will probably not go as far as assumed below.

Musical-score entry

To make it possible to enter a musical score, an external form has to be chosen, and also some internal format. In view of the graphical nature of the standard conventions for representing music, the external form is based on the built-in picture form, which already provides a default, hierarchically structured, internal format. Pictures built for notes and various signs will serve as elementary ingredients for this application. A grammar has to be given for scores which describes a sub-‘language’ of the general language for pictures. The restrictions imposed by the grammar automatically induce certain constraints on the edit operations allowed. Further constraints cannot be naturally expressed syntactically, in particular, those relating the duration of notes in various voices to their horizontal relationship.

The simplest approach is to use a further internal form, e.g. a table (another standard type) indexed by voice and ordinal position whose entries give the duration of the notes. A two-way transformation between the pictorial and the tabular form has to be provided, in which a few simple numerical relationships can now express the further constraints. The effect is the automatic vertical alignment of notes that should be aligned according to musical convention. Many of the standard editor operations will apply to editing scores, such as selecting a group of notes, deleting them, moving (for example by dragging) them to another vertical position, or copying them to another place. There are, however, no standard ways for entering, say, a G-clef or a semiquaver, so new editor operations have to be provided here. There are various methods for doing this, and the methodology to be developed (as described in the previous section) has to suggest a specific way. One possibility is to have a fixed window of musical icons at the side, from which the user can drag copies to designated positions. To implement this, the score grammar has to contain a pre-composed, immutable sub-picture for the side window.

Another possibility is to re-assign existing keys, if possible with some mnemonic value, to stand for the various musical symbols. This is realised by providing a few new editor entries for score forms, consisting of a mapping of the re-assigned keys to simple sequences of edit operations. These two possibilities can be combined, and the side window can display the appropriate key next to each icon.

Inventory control

A user maintains a small database containing the amounts of certain goods that have to be replenished, and therefore ordered, in time. The database also contains fields for keeping track of what is currently on order. The database itself can be entirely maintained using standard tools.

Another table contains, for each article, data giving the rate of use and the time needed for an order to be fulfilled. These data may initially have to be estimated, but can also be derived from the database by suitable statistical formulae, using an appropriate amount of pessimism depending on the vitality of timely replenishment.

Setting up the necessary relationships is within the reach of a user who can master the necessary mathematics; no programming is needed here. Likewise, it is possible to create another table giving the expected stock shortage for each article at the time ahead of ‘now’ by the order-fulfilment time if no order is placed (basically, shortage = rate $\times$ time - stock; for a realistic application the formula should take into account the last time the stock data in the database was updated). An invariant that has to be maintained is then that for each article the amount on order is at least the projected shortage. To restore the

invariant if violated, a new order has to be placed. Now the ‘on order’ field of the database is linked to a set of order forms, with the invariant that the data in the ‘on order’ field corresponds to the data in the form. So, to restore the invariant, the system has to create a new order form.

Data such as the address of the supplier, etcetera, can be filled in automatically by linking the field to an address field in the articles table. It should be possible to take into account customary ordering quantities here (again using some formulae). The form contains an ‘OK’ field that can only be set by the user, and a further constraint is that this field has to be checked to validate the form (if so desired); alternatively, the forms could be printed without explicit OK-ing. In the latter case, if a printed order form cannot be sent out as is, the user can still edit it and have it printed again. The OK-ing possibility is needed if the forms are not printed in hardcopy form for further manual processing, but are forwarded electronically to another department, or possibly directly to the supplier. The OK field can be linked to an invisible ‘has-been-printed-out field’, and the maintenance of an invariant for the latter field causes the form to be printed (or forwarded). Various refinements are possible, such as combining orders destined for one supplier, only producing order forms on set dates, etc.

Job applicants processing

In a certain environment, say a young and fast-growing software house, job applicants may come in in response to advertisements or by applying spontaneously, but can also directly come in view through personal contacts with employees, either at their own initiative or by being approached. They can qualify as formal applicants or rather as informal ‘prospects’. The selection procedure is highly distributed, various managers having their own habits of approaching applicants by telephone for an interview etc. Also, it may be the case that a given applicant is of potential interest to more than one group.

As the volume involved is considerable, it is not quite possible for a single person to be fully aware of the situation without some form of support. Occasional embarrassments happen, such as that an applicant is not written off when this should have happened, or even that an applicant receives a letter of rejection when some manager is still expressing interest. The person in the Personnel Department charged with overviewing this needs a better tool to keep track of the status in which the ‘processing’ of an applicant in the organisation is.

Without going into too many details, it is possible to set up a database with fields for the relevant information. A number of rules can be formulated in terms of the data, such as a maximum time span that should not be exceeded between some form of contact with an applicant whose status is not yet finalised (either definitely written off or definitely hired), the conditions under which the status can be finalised, whether a formal letter will be needed or not, who are the persons that will have to see (a draft of) letters going out, what are the steps still to be taken to reach a final decision, conditions and maximum times under which managers have to make some decision, and so on. As in the previous scenario, violations of the invariants expressed by these rules can result in a document identifying currently problematic situations and giving an overview of the steps to be taken first now, and in particular in the automated production of reminder memos addressed to managers.

6.4 Adapting existing applications to Views

An issue of interest is to what extent it is possible to take an existing application and make it conform to the rules of the Views framework without entirely reprogramming it. An essential assumption is of course that source code is available. In view of the prevalent, somewhat unstructured, approaches to software production, there is no reason to be very optimistic here. In any case, the user interface will in general have to be recreated entirely.

The effort required here will possibly be much less than the original effort, in view of the fact that Views provides excellent tools for doing just this. If the core of the application is substantial compared to its user interface, one can only hope that the program has been appropriately modularised, separating the user interface from the processing parts. In that case, adaptation should be relatively straightforward. Otherwise, the application has to be modularised first, which, if it consists of rather unstructured code, can be an unrewarding to hopeless exercise.

Once the user interface has been adapted, the application will run (in principle — there may be still all sorts of usual problems involved in porting an application to a different environment) in Views. The handling of constraints will in general not have been achieved by means of the facilities provided by Views framework, but is dealt with by the existing application code. To the user, however, this difference in internal organisation compared to 'true' native Views applications may remain transparent. In many cases, the integration with other Views tools will result automatically through the use of appropriate intermediate internal representations in the new user interface. It may, however, be desirable to create a direct linkage between internal data formats used in the application core and standard Views internal formats, for example because of efficiency reasons. This will entail a certain amount of programming. Whether this is appropriate or not (it may also create a processing bottle-neck), and whether it will be possible to make this linkage two-way (usually not), depends very much on the specifics of the application and can hardly be discussed in general.

7 Standards

The product to be developed is a framework that is intended, after proper product development, to lead to a marketable product. In fact, an important step after a successful demonstration of the prototype should be a standardisation effort for the interface between applications and the framework system. Different manufacturers could then produce and supply their own implementations of Views, possibly optimised for specific hardware configurations and/or areas of use.

At the abstraction level at which the research leading to a prototype Views system will be conducted, current standards will not play a dominant role. Still, it is important to produce the design in such a way that it does not unnecessarily stand in the way of adherence to important standards in future production versions. Although in most cases the issues that have to be solved are independent of existing standards, there are also issues that overlap. It is furthermore conceivable that the research will provide input to the ongoing discussions with respect to the as of yet undefined standard for the applications layer of the ISO/OSI model.

It is also important to recognise the existence of coexisting incompatible standards, and informal *de facto* standards, as a fact of life. The multi-layer nature of the ISO/OSI standard can be mirrored in a multi-layer open system architecture that permits interfaces to other standards to be developed at appropriate levels.

8 Discussion

In prior presentations of these ideas we have repeatedly encountered two objections. Some say: It exists already. Others say: It is impossible. As these objections together are inconsistent, they cannot be both right. Obviously, we feel that both are wrong, but quite understandable, since none of the ideas underlying Views are in themselves novel, but are somewhat revolutionary in the degree in which they are taken seriously and carried to their logical extreme. Thus, it is understandable that it is difficult to appraise a purely verbal description of Views.

As to the first objection: there exists an increasing body of *specific* applications that fit more or less in the model described here. Each of these applications has been produced by a large, sometimes tremendous, effort, and in many cases the same problems have been solved and programmed again and again. Most of these applications have shortcomings in their user interface that correspond to an incompleteness if formulated in terms of the conceptual model described here. Also, each application is self-centred, and bringing it to cooperation with another complementary application is each time a new major effort. The purpose of Views is emphatically not to create 'Lotus 1-2-3-4-5', but to create a system in which such a tool is readily configured. There exist a few *framework* systems for realising applications of which one or two come rather close to the spirit of Views. They are, however, either more limited in scope, or less flexible, or offer no support to the application designer, or have a closed architecture, or suffer from all of the preceding.

As to the objection of impossibility: it is certainly the case that this project is not a mere application of or integration of known state-of-the-art techniques. There are still many open research problems, some of which are central to the whole undertaking. On the other hand, the project members avail between themselves of a solid amount of expertise concerning these issues, both for the practical and for the theoretical aspects. The existence of an increasing amount of applications, and even a few framework systems, that approach the ideas presented here, is an indication of the viability of the ideas. In designing the framework system, it will inevitably be necessary to cut many knots to meet the deadlines, and sometimes to let practical requirements outweigh theoretical considerations. However, our collective experience is that a solid theoretical basis is an asset rather than an impediment, both in the design and in the realisation phases of a large project, and that premature consideration of pragmatic issues may in fact stand in the way of the simpler solution. All in all, the development of a methodology oriented towards application designers may well prove to be among the more difficult aspects of the whole undertaking.

8.1 Views and 'ergonomic principles'

The 'conspiring' shortcomings of the traditional approach identified were: *No integration; Inconsistency and Mode confusion; Loss of context; Inflexibility; and Arcaneness.*

After the preceding expositions, may it suffice to touch only briefly on those. Integration is at the core of Views. As there are no modes in the system and all tasks are performed using a uniform interface, mode confusion is out of the question. If the user has to switch tasks, it is not necessary to destroy the current context. The user can at all times switch tasks and continue work on a partially completed task later, in a manner that offers the functionality of a multi-tasking capability. When the user comes back to a previous task, the original context is fully restored. Remember also that there is no notion of entering or quitting an application. In fact, several independently developed applications can be simultaneously 'active' by virtue of the fact that they entertain a direct or indirect link with the currently active document. As to the traditional arcaneness, there are no operating commands that have to be memorised, or syntax, as all communication takes place through the uniform interface provided by the forms editor. It has already been shown that modern operating environments based on WYSIWYG and Direct Manipulation are far easier to master, and this must hold *a fortiori* for a design based on a single underlying conceptual framework. The ergonomic importance of direct confirmation is well recognised, and is automatically catered for. A further strong point is the presence of an unlimited UNDO facility. Next to the obvious advantages, it is also helpful for mastering new applications, since it invites experimentation — a sort of 'what if' facility in the extreme — and it takes away the otherwise constantly present cognitive load of verifying that some operation will not accidentally destroy vital information.

8.2 Views and the ‘new trends in software’

Views allows modelling of a large variety of increasingly important software trends in a natural way.

WYSIWYG

The essence of WYSIWYG is that the user, while modifying a document that is to be processed by a certain application, can view the external form as it will appear as a result of that processing. In other words, the object that is visible on the screen is linked to an object corresponding to the user’s logical view by a function determined by the given application and thus is continually updated with the modifications to the logical document. A particular very popular WYSIWYG application is formed by so-called spreadsheets. It is obvious how spreadsheets can be modelled in terms of the conceptual framework. An equally valid view is, in fact, to say that any system conforming to the conceptual framework is one, possibly gigantic, spreadsheet. The main differences with traditional spreadsheets are that the organisation of the whole need not be two- or three-dimensional, but can be an arbitrary hierarchy or network, that the relationships can be much more complex than most spreadsheet applications can possibly handle, and that the user interface for defining the relations is accordingly more powerful.

Direct manipulation

In terms of the car driver’s metaphor for Direct Manipulation, the angle of the wheels with respect to the car frame is linked to the amount the steering wheel is turned. It is possible that there is a servomechanism interposed between the steering wheel and the car wheels, but this does not detract from the driver’s having control. In general, in Direct Manipulation the user directly manipulates some visual representation on the screen of relevant aspects of the system state, and thereby effects actions or system state changes. Thus, a document representing (parts of) the system state is linked to the system state itself, so that changes to the representing document entail changes to the system state. Not only is it apparent that this is easily modelled in the conceptual framework, but this way of viewing Direct Manipulation also reveals the deep connection with the WYSIWYG principle.

Integrated information Systems

In a system built upon the conceptual framework, a new application can be added to it by giving the relationship between the external representation and the logical view of the documents that are relevant to the application, and their logical inter- or intra-document relationships. It may also be necessary to provide descriptions for one or more new document types. These descriptions are documents themselves. To achieve true integration across applications, it is necessary that the exchange of data between documents follows a standard method, and that appropriate user-transparent data transformations are applied whenever necessary. This can be achieved by supplying for each document type a number of alternate representations in terms of a few standard object types (for example, text, number, date, table). A document is thereby (conceptually) linked by default to objects of these types, and data interchange between different documents is possible provided that both are linked to at least one object of the same type. An important special case is that a document type may be created as a specialisation (subtype) of an existing type, and has then that other type as alternate representation and, moreover, inherits by default its alternate representations.

‘Intelligent’ systems

There are two aspects to ‘Intelligent’ Systems. One is the algorithmic component: domain-specific knowledge, special algorithms for solving problems, etc. This aspect is not directly addressed by Views, although it is expected that the techniques made available to application designers by Views can sometimes be profitably used here, for example to link an internal expert system to a problem-solving component. The other aspect is the user interface, on which high demands are placed for this type of system. The realisation of a good user interface for an ‘Intelligent’ System in a traditional operating system is, if not always the larger, then usually still a very substantial part of the whole effort. It is here that the facilities provided by Views are useful. Given what has been said already about WYSIWYG, Direct Manipulation and Integrated Information Systems, it should be clear that designers of ‘Intelligent’ Systems to be realised in the Views framework can concentrate their effort on the algorithmic component.

Fourth-generation languages/systems

A Fourth-Generation System usually comprises all or most of: a facility for generating data entry subprograms from a high-level description of the data and the screen layout; a facility for generating database accessing subprograms from a high-level data description; a facility for generating report-generator subprograms; and a language for describing (once) the information needed by these generators and (usually already in program form) the core of the application. From the latter, a driver program is then generated calling the subprograms as needed.

To create an equivalent application with the tools provided by Views, basically the same information has to be provided. A difference, however, is that this information is (conceptually) not used to *generate* the application, but *is* the application. Also, the flexibility is arguably much higher. A data-entry component is simply realised by specifying the screen representation of the data to be entered, and a report generator likewise by specifying the hardcopy representation of the data to be output. A database is nothing but an object with a specific structure residing in the system.

There are, in fact, several, equally valid and yet quite different views on where ‘the’ database resides in Views in an application that would traditionally be viewed as a ‘database application’. One extreme approach is to see the whole of the information kept by the system as one huge, although probably structured, database. The difference with traditional databases schemas, if any, is then mainly the greater flexibility with which constraints and exceptional cases can be handled. At the other extreme, one may point at a specific document kept within the system and call it ‘the database’. Which view is preferred may depend on the application, the kind of data handled, and the background of the person involved. It should be emphasised, however, that from the conceptual point of view there is no special characteristic of databases that distinguishes them from other documents in the system. Although an application designer may choose to use an existing database management system to provide an internal representation of, and operations on, a document, for example for reasons of efficiency or compatibility, this choice remains transparent to the end user. For other reasons, an application designer may choose to provide forms for viewing and modifying a document that have the same ‘feel’ as current methods for query and update of databases, and yet use an entirely other semantic substrate than any of the existing approaches to databases.

Rapid prototyping

In the sense of Rapid Prototyping as rapid testing of the user interface, the main contribution is the same as has already been mentioned several times (e.g., under ‘Intelligent’ Systems), namely that creating the user interface is almost no effort in the given framework. For Rapid Prototyping in the sense that a fully functional prototype is

made, this applies as well, and, moreover, everything that has been stated above under Fourth-Generation Languages/Systems. There will always be classes of applications in which speed is so essential, or the amount of data to be processed so massive, that a separate second coding phase aiming at increased efficiency is warranted. However, with processor power becoming ever cheaper, it is increasingly the case that techniques that formerly were only reasonable for prototyping are directly used to create production versions, and the same development is foreseen to apply to the outcome of Views. It is evident that the kind of system aimed at is eminently suited for ‘growing’ a system by adding more and more functionality to it.

End-user programming

Many aspects conspire to make programming hard. Some are not intrinsic, like the unforgiving syntax and obscure semantics of some programming languages, and the unfriendly user interface of some programming systems. Some are. Foremost among these is the requirement that a program specify actions to be taken upon not yet actualised circumstances, covering all contingencies.

In general, improvements that make the task of the professional application designer easier can also be used to alleviate the task of programming end users. In particular, if the components to be put together share a common model of the data to be transferred, as is required in a truly integrated system, if the system provides a built-in user interface for all kinds of data, and if a programming system, having itself a good user interface, is built on top of this, the task left to the programmer is dramatically reduced. This has a clear relationship to techniques used in advanced Fourth-Generation Systems and Rapid Prototyping. In particular, the end-user-programming subsystem should allow the user to combine in an easy way existing functionality into new functionality. Conversely, improvements aiming specifically at end users can conceivably be useful for professional programmers as well.

Approaches to end-user programming that consist of attempts to somehow bypass the intrinsically hard part of programming will eventually be unsuccessful, since the effect is inevitably that the flexibility sought — the rationale for providing a form of end-user programming at all — is sacrificed. A better approach is to allow the user to concentrate directly on the essential aspect of the task, with the exclusion of all other aspects.

It has been recognised for some time that the traditional approach to programming distracts from the essential constructive task in that it forces or invites the programmer to think in terms of the total system state and of dynamically arising conditions. This observation has led to different programming paradigms, collectively known as declarative programming, of which the essence is that the programmer’s task is to specify relationships that serve to construct the result or intermediate results from intermediate results and the input data. This allows the programmer to consider during the construction process only the immediately relevant aspects of the system state, and the dynamic reasoning is replaced here by static reasoning.

In a successful approach such as JSP, programmers still have to deliver procedural programs, but they are trained to first perform a static task analysis much like that of declarative programming (in particular homomorphic programming), and are supplied with techniques to translate the result as it were mechanically into a program. Using the conceptual framework, it should be possible to create a programming system in which the programmer’s task is reduced to specifying, in a purely statical way, relationships between data. The mechanical translation to procedural steps can and should be delegated to the system. Instead of specifying dynamic *iteration* (‘do each time’), the same functionality can be obtained by specifying what is called in homomorphic programming a *map* (‘apply function to each element of a table’).

Sometimes it is conceptually easier to use a dynamic specification in which a certain action is triggered by a given event. For example: ON there are five or less working days left to the end of the month PERFORM check that office equipment requests are in. Although this can be expressed statically in terms of an invariant to be maintained (such as: month(request) >= month(today + five workdays)) and thus can be made to fit the conceptual framework, this is conceptually not the most natural way, and an end-user-programming facility should therefore also support the direct expression of triggered actions.

Open system architectures

An open architecture is a collection of fixed and public interfaces by which different suppliers of equipment or software can independently manufacture components or modules that are guaranteed to work in an environment composed of parts that also follow the standard set by these interfaces. Although an open architecture could be defined for most systems, the term usually implies that the kind of system it is applied to is itself open.

An open system is a system that does not have a predefined fixed functionality, which could be defined once and for all in a user manual, but can be composed at will from loosely coupled components. In hardware the trend towards open systems started some time ago with the standardisation of buses. Closer to home we have the example of hi-fi equipment.

The first steps towards open software systems are taking longer. That Views is an open-architecture system does not follow by itself from the conceptual framework, but is nevertheless an essential aspect of the undertaking. Rather, an implementation of a system based on the conceptual framework is feasible only by virtue of a set of fixed communication interfaces, as sketched above under ‘Integrated Information Systems’.

The domain at which Views operates corresponds to the top layer (the so-called ‘application layer’) of the ISO/OSI reference model, but is of a generally higher abstraction level than the issues that have thus far been addressed there. Yet, existing open-system standards (for example ODA/ODIF [12], or PCTE [12][14]) can be valuable both in the design phase and in achieving further openness. Here the conceptual model can be helpful again, in that representational document types based on existing standards can be included, insofar as appropriate, among the set of default linkages.

As has been mentioned already, the architecture of Views is itself an open system architecture, and this crucial aspect is possibly only because of the achievements that have already been reached here. This objective implies also that independence of particular hardware is aimed at, possibly not in all implementation aspects of the prototype demonstration system, but certainly and vigorously in its design.k

Object-centred programming

It would be misleading to state that the conceptual framework ‘allows modelling’ of object-centred programming. Instead, a more appropriate characterisation of the relationship to the object-centred programming paradigm is that the framework of Views is a further extension of object-centred programming that still bears a strong kinship to the latter. In fact, it is felt that even a prototype system of the scope and complexity of Views could not be conceived, let alone realised, without the prior advances in object-based programming. Inasmuch as these advances have facilitated the creation of highly flexible systems with substantially improved user interfaces, they are subsumed by the programming paradigm supported by the conceptual framework.

From office automation top Personal computing

As has become apparent from the preceding points, an emergent property of the framework system to be designed is that all kinds of data are integrated in the system, and that the user interface for these is uniform. In a system built around this framework, containing as subsystems for example a manager's Personal Information System (agenda with reminder service, small personal data bases, memos, annotations with half-baked ideas, spreadsheets, etc.) an Office Automation System and a Management Information System, the boundaries between these subsystems are not real, and in fact they are all integrated, so that, e.g., a memo can be dropped in the secretary's mailbox, or the data for a spreadsheet calculation can be read in from a database kept in the Management Information System. Only one control interface has to be learned for all, and data can be transferred without further ado. The Object-Centred paradigm provides an easy and natural way to specify the constraints of the Office Automation System and the links with the Management Information System and the Personal Information Systems (for example, by automatically generating overviews at set times, or reminders).

8.3 Open research problems

In this section an (incomplete) list is given of research problems that have to be addressed, mainly to give an indication of the nature of these problems. In most cases the problem is not so much to find a solution, but to select the most satisfactory solution between several that present themselves, or to try to identify theoretical rather than pragmatic-heuristic approaches, which may be satisfactory from a functional point of view but lead to unforeseen problems if applied in combination.

- Priorities for task scheduling. It is clear that tasks involving the active document should have priority, but should this be an absolute or a relative priority? Should incomplete tasks not involving visible documents have no priority and steal cycles only if the processor would otherwise be idle, or still have some low but real priority? Must the algorithms for tasks scheduling take the cost of process switches into account, or can this be abstracted from? Is the task scheduling data driven, or demand driven, or some mixture? Does this depend on the topology of the network formed by the links, and if so, how?
- Is it possible to support a form of 'Visual Programming' for end users? Or of 'Programming by Example'? A combination perhaps? The state-of-the-art of these approaches is still rather unsatisfactory, but can profitable use be made here from the simplicity offered by the conceptual framework?
- Sometimes the maintenance of an invariant should be as continual as possible, and sometimes it is preferable to restore invariants only upon some closure of an operation. Should this be decided upon and specified on a case-by-case basis by an application designer, or are there general principles by which the decision can be left to the framework system?
- Displaying of an object on the screen can be done via an invariant 'The representation on the screen must match the contents of the object'. To what extent can editing be reduced to (reverse) application of this invariant through the normal invariant management part of the Views system?
- Although the prototype system will primarily be designed as a single-user system, the ultimate intention is that the system may serve many users, either on a single processor or in distributed form. What ramifications does this have for the design? What forms of concurrency control are compatible with the conceptual model? How would security issues in multi-user environments be best addressed?

9 References

[1] Steven Pemberton, *Views: An Open-Architecture User-Interface System*. In Proceedings “Interacting with Computers: Preparing for the Nineties”, Noordwijkerhout, December 1990.

[2] Jakob Nielsen, *The Matters that Really Matter for Hypertext Usability.* In Proceedings of Hypertext ‘89, Second ACM Conference on Hypertext, Pittsburgh, Pennsylvania, pages 239—248, November 1989.

[3] A.J. Sellen, G.P. Kurtenbach and W.A.S. Buxton, *The Role of Visual and Kinesthetic Feedback in the Prevention of Mode Errors*. In Proceedings Interact ‘90, Cambridge, U.K., pages 667—673, August 1990.

[4] D. Norman, *The Psychology of Everyday Things*. Basic Books, NY, 1988.

[5] K.M. Misra and P.J. Jalics, *Third-Generation versus Fourth-Generation Software Development.*IEEE Software, July 1988.

[6] M. Flecchia and D. Bergeron, *Specifying Complex Dialogs in ALGEA.* In Proceedings Human Factors in Computing Systems and Graphics Interface (CHI+GI ‘87), Toronto, Canada, pages 229—234, April 1987.

[7] J.A. Sutton and R.H. Sprague Jr., *A study of Display Generation and Management in Interactive Business Applications.* IBM Research Report RJ2392(31804), Yorktown Heights, N.Y.

[8] J.J. van Griethuysen, editor, *Concepts and Terminology for the Conceptual Schema and the Information Base*. ISO TC97 / SC5 / WG3, American National Standards Institute, New York, March 1982.

[9] Lambert Meertens, Steven Pemberton, Guido van Rossum, *The ABC Structure Editor*. CWI Report CS-R9256, CWI, Amsterdam, December 1992.

[10] L.G.L.T Meertens, S. Pemberton. *An Implementation of the B Programming Language.* In Proceedings USENIX Conference Washington, January 1984, Also Note CS-N8406, CWI, Amsterdam, June 1984.

[11] Tim Budd, *The Cleaning Person Algorithm.* CWI Report CS-R8610, CWI, Amsterdam, 1982.

[12] J. Rosenberg, M. Sherman, A. Marks and J. Akkerhuis, *Multi-media Document Translation — ODA and the EXPRESS Project.* ISBN 0-387-97397-4, Springer-Verlag New York Berlin Heidelberg, 1991.

[13] T.G.L. Lyons and M.D. Tedd, *Recent Developments in Tool Support Interfaces: CAIS and PCTE.* Ada User Volume 8 Supplement, pages 565—572, 1987.

[14] T.G.L. Lyons and M.D. Tedd, *Technical Overview of PCTE and CAIS*. Ada User Volume 8 Supplement, pages 573—578, 1987.