The significance of user-interfaces for historical software

Bert Bos and George Welling

1. Introduction

There are good scientists who are prima donna's in their laboratories, but who lack the ability to communicate their findings in the classroom. They tell boring stories or are too shy to speak out. Either way they cannot keep the attention of an audience. Sometimes the quality of their written words saves them, but there are good studies that have never reached the audience they were meant for. Not because the content was poor or uninteresting, but simply because they were written in a bad style. The great emphasis that is put on developing writing capacities in the training of historians is justified. If historians cannot write books that will be read, the whole role of the profession in society can be questioned. It is difficult to single out exactly what makes a good style, but the importance of style is beyond discussion. If we may put it in computer jargon, style is the user-interface of the written word. It is good scientific writing style to make intelligent use of metaphors, to structure the narrative in a consistent framework, and to use plain and unambiguous language. It is a good habit to provide indexes, which help the reader find passages of specific interest. Footnotes can provide further explanations and references, to verify the content. We will show that several of these aspects can be found in building user-interfaces to computer programs.

'Historical computing can only be defined in the terms of the distinctive contribution it can make to historical research. As a subject it exists on the methodological plane. It must be viewed at a high level, not in terms of grubby practicalities. It is the design of research procedures that counts, not hardware or software,' Charles Harvey said in 1988 (Harvey 1990, 207). Since then things have changed. Boonstra in 1992 saw a development in three phases in history and computing: first, historians simply tried to use information technology in their research. In the second phase a reevaluation of the traditional skills of the historian brought about a more responsible use of the computer in historical research. The third phase will witness a greater specialisation: next to the group of users of information technology there will be a group of people trained in history and information science, that will make an independent contribution to the development of real historical software(Breure 1992, 24-6). We agree with Boonstra and we think the third phase has indeed started (Boonstra 1993). It is fitting to make the distinction between *History and Computing* and *Historical computing*. If Charles Harvey would agree with interpreting his *historical computing* as *history and computing*, his words may still be true. *Historical computing* however, will have to deal with all the ‘grubby practicalities’ of hardware and software. It will have to deal more with applying what information science has taught us. If we want to make software for historians, we must stop producing programs that require attending several summer schools before you can work with them.

We cannot pretend that there is a special corner in the market for historical software that has no connections with the rest of the market. Historians have seen the benefits of good user-interfaces in the wordprocessors, database management systems and statistical packages they use. They will not change their attitude towards software just because a product is meant for historians. They will prefer working with easy to use software that may not completely solve their problems, to struggling with almost incomprehensible, but potentially powerful software. They will rather use ingenious combinations of off-the-shelf software than go through the frustrating experience of having to work with software with poorly designed user-interfaces.

Jan Oldervoll has explained why the historical-software that has been developed during the last twenty-five years is not in general use by historians. ‘The programs are either too restricted or too difficult to learn.[.......] The reason for this is that to develop a high standard piece of software with all the features needed by a historian is a very time consuming and expensive task, much more expensive than any institution within the historical community can be expected to finance.'(Oldervoll 1992, 8) He suggests that we try to develop software-parts that can be made to work together. In the same volume, Manfred Thaller suggests the Daemon-concept as a solution. But the daemon will only make pieces of historical software work together (Thaller 1992, 53-67). The user[1] will still be confronted with the user-interface of one of those programs.

We suggest that it is time to broaden our horizon. Quite a number of the problems we face are not restricted to the historical community. 'The outline of a possible paradigm for software development', that Nancy Ide and Jean Veronis have suggested for text software could be applied to historical software also (Ide 1993, 1-12). Both academic communities face the same problem: it is highly unlikely that the software industry will produce exactly what academics in the humanities need. We will have to write the programs that can handle the specific problems of our disciplines. In this situation, trying to create large integrated systems is waste of time. There will always remain problems that cannot be solved by one program. On the other hand, writing new programs from scratch for every new problem is very unproductive. What we need are libraries of useful algorithms, which may be implemented in any language. The algorithms should be well documented, if possible illustrated with implementations, and made available for the whole community. Developers could use them, expand them and build their own userinterfaces around them.

There are more general rules for software-development. It is time to listen to what computer scientists have to say about the emerging new paradigms for Human Computer Interaction (HCI).[2] In interdisciplinary groups, like Alfa-Informatica, where linguists, computer scientists, text experts, art-historians and historians work together, all participants have a lot to gain. We cannot continue to disregard the importance of user-interfaces in the development of historical software, but we do not have to reiterate a discussion that has been held before. In this paper a trained computer scientist and a historian give some guidelines for the development of user-interfaces.

2. The triumph of GUI's

The success of the PC on the market can be explained by pointing at the ease of use of these machines compared to the mainframes of the old days. Ever since the introduction of the PC there have been attempts to make it simpler to work with these machines. The duller part of the family, the IBM compatible machines, based on the Intel 80x86 series, have been holding on to a command line interface for a long time. The more frivolous Apples, Ataris, and Amigas, based on Motorola's 68000 series, and the RISC based Archimedes have all chosen for a mouse-driven graphic user interface (**W**indow **I**con **M**ouse **P**ointer)[3]. Graphics and windows are not the same thing, you can have one without the other, but the combination - with a mouse thrown in - is clearly the best of both worlds.

A number of half-hearted attempts have been made to improve the interface of the Intel-family. All sorts of program-managers and shells have been on the market. Digital Research copied some ideas of the MacIntosh-interface in their GEM-interface for the PC, but the graphic capacities of the machine were still too limited. MicroSoft hoped to change all that with their MS Windows interface, while IBM gambled on the graphic Program Manager (PM) in OS/2. But the software industry has been slow to react to these developments. Still, the success of the windows-approach on other hardware platforms, like the Xwindows interface developed in project Athena by MIT, determined that the future would be window-based.

The arrival of faster DOS machines and video-cards with higher resolutions (like VGA and SVGA) have prepared the way for MS Windows 3.1. Not only the idea of the graphic user interface has been adopted, but also the integration philosophy that all programs should present themselves to the user in a similar fashion. Still, MS Windows 3.1 runs on top of MS DOS and that hinders its performance. Windows NT (New Technology) will be the next step: a new operating system for the PC, combined with a graphic user interface and the integration philosophy.

Why did the graphic user interface win the battle? The answer is fairly simple: most people do not use computers because they want to use computers, but because they hope to get a task done quicker and better. They want to invest as little time as possible to learn to handle these machines. They are like car-drivers: willing to spend a limited amount of time to get a drivers license. After that, they want to be able to drive in any car regardless of make. They do not want to take drivers' lessons every time they buy a car. So the automobile industry has responded to that. You have left handed steering and right handed steering, but for the rest all cars are the same. That's what end-users would like to expect from software, too: if you know how to start the engine and where the switches and pedals are, a program should get you where you want to go.

The almost religious following the Apple Macintosh has gathered bases its choice on the intuitive quality of the user-interface of this computer. Within an afternoon a novice user can learn to work with this machine. The amazing popularity of Borland's Reflex in the historical community and the fact that is still used in some History and Computing courses even after Borland stopped supporting it, can only be explained by the quality of its user-interface. The functionality of the program itself is fairly limited, but the intuitive user-interface lets you use up to 80 per cent of the full power of the program without having to consult the manual. Larger DBMS's demand a much larger time-investment before you can really use them. Exploiting the full power of these programs demands taking classes.

The choice for **G**raphical **U**ser **I**nterfaces in educational environments is not surprising.(Ree 1993) Computers are a means to an end, using computers is not a goal in itself. In the historical discipline it is the same: historians are not interested in computers (let us be honest, most of them hate the machine), but in the results they can obtain with computers. It is depressing to see that promising historical projects, like MEMDB or the prototype of the Chronos-interface of the NHDA (Doorn 1992), come up with absolutely horrible user-interfaces. This cannot be what Bob Morris dreamed of for the year 2001.(Morris 1992)

Historical computing should take advantage of its position as a late-comer in the field of applied computer science. We do not have to make the mistakes that have been made before. Our programs do not have to look like early 1970's mainframe applications. We can start with state of the art software. If we, who are interested in historical-computing, want our products to be used, we should not underestimate the importance of good user-interfaces. Manfred Thaller has suggested that there is a trade-off (Thaller 1990, 239); if limited resources are available, one should concentrate on the functionality and not on the user-interface. Of course this is absolutely wrong: what use is a Rolls Royce without a steering-wheel; what is beauty if we cannot see it. How can a program be *functionally significant*, if that functionality cannot be accessed? As long as the user-interface to Kleio is as it is, the program will never serve its purpose. How many historians are willing to spend part of their summer holidays for instruction in one single program? I suppose Manfred Thaller will have to agree with us, because in the same article he argues that it must be possible to use historical databases 'without an introductory course how to do so'.(Thaller 1990, 243) And here he is right, of course, and the same goes for all programs for historians.

In the demonstrations at the AHC-Bologna conference in 1992 the majority of the programs were Windows-applications and we think that may be a step in the right direction. But simply adopting the industry's standards is taking the easy way out. We should try to explicitly state the requisites of good user interfaces. In historical-computing we should focus on lowering the amount of learning needed to be able to work with a program. Programmers and designers, the developers of historical software, have the skills and hence a moral obligation to do so.

3. The goals of user interfaces

The user-interface is that part of a software system that interacts with the user. This interaction has a physical level and a cognitive level. We will not discuss the physical level of input and output devices such as the keyboard, the mouse, and the display. These hardware aspects define the framework within which the system interacts on the cognitive level with the user. We will accept the most commonly used devices - keyboard, mouse, screen and internal speaker - as parameters for the discussion of cognitive aspects of the user-interface. This consists of three aspects: the underlying model for the presentation of information, the following interpretations by the user, and the intentions that the user then formulates.

The ultimate goal of a user-interface is complete transparency: the interface should recede from the users consciousness to allow him to fully concentrate on the task he wants to get done. We do not have a roadmap to this Shangrilah. We will focus on aspects of cognitively compatible user-interfaces, trying for the highest attainable degree to which the model of the task presented by the interface conforms to the expectation of the user.(Parsaye 1989, 30-1)

Many designers only describe the goal of their work as creating a ‘user friendly’ system and follow a subconscious idea of what that means for their project. However, the main objective of a system is not ‘friendliness’, but helping the user to perform a task.(Willemse 1988, 48) But still, the majority of user-interfaces are designed to meet one or more of the following goals, even if often the goals are not made explicit.

3.1 The user perspective

The goals of a user-interface can be evaluated from the point of view of the user or of the developer. In the real world a good user-interface is the best possible compromise between these points of view. Let us have a look at what a user would expect first.

3.1.1 Ease of use without prior training and after training

In the most extreme case, someone who knows nothing about the system, but knows enough about the subject domain, should be able to come up to the terminal and start using it. This would be the ideal situation, but reality is still far from that. The integrated approach, where all programs share the same user-interface, brings this goal as close as possible. If programs have a common user-interface, you only have to get used to it once.

After the user has been taught the principles behind the program (or rather behind the interface) and has had some practise, he should feel confident with it. He should have the feeling that it helps, rather than hinders him in doing his work. If the user suspects that he could achieve more with another program or completely without computer-assistance, it is obvious that there is something wrong.

3.1.2 Ease of first-learning and of re-learning after a period of absence The ease of learning can be measured in various ways. One measure could be the time required to master a certain task, another could be the degree to which users feel they understand the system after some period of time.

It is said you never unlearn to ride a bicycle. But many other skills and types of knowledge are easily forgotten. If a system is used intermittently, it should be easy to get the hang of it again quickly.

3.1.3 Ease of documentation, maintenance and support Documentation is always important, but some systems rely on it more than others. If programs are to be installed by untrained people, or if people have to learn the system from the books, good documentation is especially important. The design of the system could take this into account and avoid things happening on the screen that are more easily done than 'said'.

Many programs are created for extended use, which nearly always means that they have to be updated a few times. If the users themselves or another non-programmer has to do this, the interface has to provide some form of support.

If there is support for a program in the form of a help-desk or a telephone number, the system must be such that it is easy to describe.

3.1.4 Speed, error-prevention and flexibility By speed is usually meant the number of keystrokes (mouse clicks, mouse travel distance) required for performing some task. To activate the most frequently needed functions, the user should have to input as little as possible.

The earlier errors are reported, the easier it is for the user. The earliest possible error check is as soon as the user strikes a key. Of course, for this to work, the user should also be guided towards the correct input, by providing suitable prompts.

Some systems are meant to be used by a wide variety of users in many different situations. They can all have different preferences and every user would like to customize the program to his own needs.

Many of these goals may seem contradictory. It is difficult to create an interface that is usable without training and that also pleases the user who has to enter a few thousand records into a database with it. A beginner will not object to working his way through a number of menus, but when you get used to a system, you expect to find short-cuts to where you want to go in the system. The user is not a static entity. We must differentiate between the beginning user and the experienced user. Both have different expectations of a program and both needs must be met. If we design a layered interface, where each layer corresponds to a level of skill of the user, we may be able to supply a solution for this problem.

3.2 The developers perspective

But interfaces have other goals, too. The list above presents the view from the user's perspective, but seen from the developers side, the interface has other tasks as well.

3.2.1 Security By limiting user-input and validating it, the interface can help in maintaining the system's integrity. The worst thing that can happen is a user causing a system hang-up. The interface should provide the core of the program with some defence against malicious or inept or inexperienced users. The user will always blame a system failure on bad programming and in most cases this view is correct. It is the programmer's task to prevent these events from happening.

3.2.2 Off-loading tasks This is most obvious if the interface runs on a different computer from the application. Tasks that the interface can take over are, for example, tokenizing and syntax checking, displaying static information (which can be cached by the interface) and providing help and elaborated error messages.

3.2.3 Portability and 'internationalization' Since the application and the interface often use very different computer resources, porting a system to another computer may very well require changes to the one and not the other.

‘Internationalization’ means translating commands and messages to other languages, changing some icons, sorting order, time and date display, fonts, etcetera. Most if not all of these changes can in many cases be accommodated by changes in the interface. The use of icons can simplify this task, but we should not make the mistake of thinking that all icons are self-evident and that they have the same meaning in all cultures.

4. How to achieve these goals

The designer of an interface has many means at his disposal in trying to achieve the best compromise between his own goals and those of the user. We will not discuss the algorithms needed, but we will focus on the concepts of building intelligent user-interfaces.

4.1 Direct Manipulation

DM is a characteristic of certain interfaces that strive to give the user the impression that he is handling data objects with his own hands, instead of giving the computer commands to do it for him. Objects are the central items in DM: the user always starts by selecting an object and then applies an action to it, often with the mouse. An example is the operation of moving a file from one directory to another: with the mouse that can be done by ‘dragging’ the file’s icon from one area of the screen to another. Contrast this with the traditional command languages, where the command is central and the data is only an argument to the command. Other goals of DM are, among others: to make actions reversible, preferrably by applying the same action in reverse; prevent errors by making illegal actions impossible; provide immediate visual feedback of any change to a data object.

4.2 Robustness

The interface must be able to reassure the user by appearing fool-proof and incapable of doing harm. The user should not have the fear that the system can crash any moment if he makes a minor mistake. And if the system crashes, there should be provisions that make sure that the user has not lost all his valuable data.

4.3 Metaphors

The interface tries to mimic something that the user supposedly knows well. The calculator in a number of programs looks like a pocket calculator, the relational database model uses the table-concept, though it may be represented internally in a completely different way. The interface tries to communicate the model of the task by disguising itself in familiar shapes to facilitate acceptance.

Spreadsheet programs like VisiCalc and later Lotus 1-2-3 closely resembled the multi column paper that accountants and bookkeepers were used to. The choice of metaphor may very well be the key to successful software. If we want to write programs for historians we will have to study they way traditional historians do their job, to find the right metaphor. The popularity of HyperCard among historians can be explained by its resemblance to the traditional stack of index-cards. The complexness and rigidity of the relational database model do not resemble the structured chaos methods of most historians: free-text systems come much closer.

4.4 On-line help.

Help-systems should provide good explanations everywhere in the program, preferably context-sensitively. This is a mixed-initiative situation: sometimes it may be necessary for the program to provide information, sometimes the user may want information. The information should be tailored to the needs of the user and should alow for broad or narrow views of the subject by allowing out-

and in-zooming. Associative searching of related topics can put information in its context.

Writing good help-systems is extremely time-consuming, but pays off. Since context-sensitive help should be available everywhere, even within the help-system, hypertext-approaches are very suitable.

4.5 Menus, pick-lists, buttons and radio-boxes

They help the user because they list the available options. Just as reading a foreign language is easier than speaking it, chosing from a list is easier than formulating your own choice. A limited list of possibilities to choose from helps avoiding mistakes caused by unintended but strictly correct input. Allowing the user to key in commands can have undesired results: on a qwerty-keyboard S and D are located next to each other, so typing DELE instead of SELE can easily happen. DELE and SELE[4] are both grammaticly correct input for dBase, but their meaning is dramatically different.

4.6 Feedback

For feedback we have options other than text alone. Even in a restricted hardware environment we can use sounds and pictures too. Feedback should not be limited to negative feedback, like error messages and beeps. To stimulate confidence of the user the system should try to be as transparent as possible. Progress meters, echoing of the user’s input, scroll bars, showing the actual posistion in a text, system messages, etc. all work together to give the user the impression of understanding the working of the system. The user needs to feel that he is in charge.

4.7 Clever defaults

An intelligent interface should reason to interpret what the user really wants. It must try to deduce information that cannot be directly obtained. Using rule-based reasoning the system could dynamically generate defaults. This stimulates confidence and increases the speed of interaction.(Welling 1991,1992,1993)

4.8 Early error checking

Error checking should take place during input instead of at the end. Good examples are syntax directed editors, that check the whole input during the edit process. On the other hand, compilers by their nature save all error messages for the end of the process. Having to deal with a long list of error messages can be rather discouraging. That is why inexperienced programmers often prefer interpreters to compilers.

4.9 Modeless interfaces and consistency

Interface should be modeless. This means that a command always means the same thing. There are no (or as few as possible) different contexts, in which the same user actions have different effects. In a windowing environment, this can often be achieved by keeping the different contexts in different windows which are both on screen at the same time. (Context that changes with location is not considered as harmful as context that changes with time.)

By consistency we mean that the same action should always follow the same procedure everywhere in the program. Inconsistency leads to disorientation of the user. A good example how to do things wrong can be found in WordPerfect. Sometimes you have to press F7 to finish an action, sometimes the Spacebar, sometimes the zero, sometimes the Escape key. A programmer may see some logic in that, since the different keys indicate a return to a different level. For a user so many keys for apparently the same action is quite confusing. This shows, however, how problematic it is to discern which actions are alike.

4.10 Simple screen layout and fixed-screens

It is desirable to keep the layout of the screen simple, if the system must be easy to describe, but there are also ergonomic reasons. Flashy screens with loads of windows scattered all over are not really nice to work with for longer periods. They will cause the user to feel disoriented.

Fixed screens are not the same as simple screens, although they may serve the same purpose. A fixed screen is a screen in which everything has a fixed location. A part of the success of the Windows interface is based on the fact that buttons, scroll bars and menus are always in the same place and menus always have the same organisation.

4.11 Conspicuous clues and memory aids.

Icons can help the user to associate particular actions and goals with each other. A good use of colour can have the same effect. However, colours and icons are not unambiguous and may often need further explanation.

A good example is the ‘hour-glas icon’, which is quite often used to indicate that some internal action is in progress and patience is required. The ‘trash-can’ is easily associated with things that should be thrown away. Flashing text suggests that acute action is required. etc.

4.12 Configurable interfaces and separate resource files

Many things can be made customizable: colours, screen layout, fonts, mouse speed, etcetera. Choosing his own preferences creates a more personal bond between the user and the program.

Resource files can give people a way to change aspects of the interface without programming and re-compilation.

4.13 Good ergonomics

Ergonomics is not a very exact science, but some of its concepts can be of help in the process of user-interface development. In the design of screen layout, in the use of colour and sound effects, we must always keep in mind the fatigue that they may cause. Likewise actions that require great hand-movement should be restricted to a minimum. If possible, constant switching between mouse and keyboard should be avoided.

A good example of how to do things wrong can be found in the early versions of MacWrite, in which you had to move the cursor with the mouse during text-input because there were no keyboard equivalents.

4.14 Context-independent keystrokes, speed keys and composite commands

Context-independent keystrokes are a less stringent requirement than modelessness. If there are keystrokes that mean the same in every context, the user can always get back on track if he gets lost. A (trivial) example is the ‘quit’ command, which could be made to abort the program from every context, thus returning the user to a known state.

One-letter commands are useful for speeding up the interaction. However, they are rather error-sensitive. Choosing keys that have some mnemonic relation to the action can be a solution, but the location on the keyboard of speed-keys can be equally important. Research on the Emacs-editor has shown that deviations from strict mnemonics can actually be more effective.

Composite commands such as macros, can help the experienced user accomplish more with a single command. Travelling through a number of menu’s may be fine for a novice user, but will become increasingly irritating.

4.15 User differentiation - environment differentiation

User profiles, in which the skills of the users of the program are recorded, make it possible to automatically customize the user-interface to the level of skill and to the preferences of the user. Some users may prefer a command-line interface, some may like restricted help etc. But, as we have said before, the user is not static; he evolves while using the program. This poses some serious problems. Should the user decide his level of expertise or the program? If the program builds profiles of all users, how should it treat an experienced user, who has not used the program for a longer period. Will his skill still be the same?

The environment in which a program is used, has great influence on the appreciation of the user-interface. There is a great difference between using a program in the quiet of your study, or in a noisy office with twenty other people. The potential level of concentration will be different in both situations. The ever growing use of portable computers creates other problems. The more commonly used ‘notebooks’ do not have the same hardware capacities as the desk-top models have. Sometimes a mouse is lacking and affordable (monochrome) LCD-screens cannot present the desired quality of images. The same user may prefer a rather complicated command-line interface that provides great performance in a quiet situation, but not in the office. An interface that relies heavily on colours and mouse-support can be very unpractical for use on a notebook-computer.

4.16 Natural language

Using natural language can give the user the impression of communicating at his own level. Speech-recognition will bring the interaction between man and computer on even higher ground. However, natural language processing in dialogue systems and voice- and speech-recognition are some of the more complex topics in applied informatics. Solving the ambiguities in natural language is no simple matter. In this respect, we will take a conservative approach and advice against the use of these methods, for the time being.

5. Can it be done

Yes, it can be done, but we will have to realise that ‘design’ is not a luxury, but a necessity. If you have something to offer, you must reach your market. If you cannot do it yourself, let someone else help you. If you can provide the ideas, a professional may find the right form to present them.

Making a good user-interface is almost an artistic process. To acquire the skill to do it, we advise following the classical approach of painter’s apprentices: learn by copying. Study how your favourite programs have dealt with the user-interface and try to copy it. The industry has developed some ‘de facto’ standards for user-interfaces, study them. Read the ‘style guidelines’ of a number of computer manufacturers and published ‘interface guidelines’.

For most computer languages there are libraries with routines for building user-interfaces: use them. There are programs such as Object Vision, and Matrix Layout a.o. that will help you create very professional user-interfaces with very little programming. There are a number of programs to help you structure your programs in the MS-Windows environment (Visual Basic, OPAL, Paradox, FoxPro 2.5, Windows Libraries for Clipper). GIST can help you in the X-windows environment.

If you are not into programming yourself or if the project is too large for your own capabilities, do not hesitate to hire external expertise; it will pay off.
Whatever you do, do not underestimate the value of a good user-interface.

6. Conclusion

We have made an inventory of the wishes of users and designers and we have shown possible areas of conflicting interests of these groups. We have tried to provide insights in what makes a good user-interface. Not all these ideas can be realised in every program, but we do not have to make the mistakes that have been made by others before us. This list of concepts we have discussed may not be exhaustive, but we do think that the important aspects of user-interfaces are covered.

Hurting your knee a little every day will in the end damage the whole joint (Safire 1988, 171). With this in mind one can hardly overrate the importance of high quality user interfaces. Potentially powerful historical programs will never reach the audience they were intended for because of bad user-interfaces. We do not have to repeat the examples, they are well known.

Notes

1. We have consistently used "user" in the male form. This was not intended to be offensive.

2. It is rather confusing that the abreviation HCI is also used for *Historisch Culturele Informatiekunde* (Computing in the Humanities) in the Netherlands

3. We do not want to imply that the choice of CPU is important for the user-interface: it is not.

4. delete and select

References

Boonstra, O. (1993), ’Wat is historische informatiekunde. Een repliek.’ In : Cahiers voor Geschiedenis en Informatica 6. (Hilversum)

Bos, B. (1992),’Human-computer interaction and the changing role of text.’, In: D.Gilbers and S. Looyenga (eds), Language and Cognition 2.(Groningen),43-50

Breure, L. (1992), ’Tools for the tower of Babel. Some reflections on historical software engeneering.’ in: Oldervoll, J. (ed.), Eden or Babylon? On future software for highly structured historical sources., (St. Katharinen).

Doorn,P.K.(1992),’Data are sacred, opinion is free: the Netherlands Historical Data Archive.’ In: Cahier Vereniging voor Geschiedenis en Informatica 5,20-42.

Harvey, C. (1990), ’The nature and future of historical computing.’, in: Mawdsley, E., Morgan, N., Richmond, L. and Trainor, R. (eds.), History and Computing III. Historians, Computers and Data. , (Manchester).

Ide, N. and Veronis,J. (1993),’What next, after the Text Encoding Intiative? The need for text software.’ In: ACH Newsletter, winter 1993, 1-12

Morris, R.J. (1992),’The historian at Belshazzar’s feast: a data archive for the year 2001.’, In: Cahier van de Vereniging voor Geschiedenis en Informatica 5,42-52.

Oldervoll, J. (1992), ’Introduction’, in: Oldervoll, J. (ed.), Eden or Babylon? On future software for highly structured data., (St.Katharinen).

Parsaye, K., Chignell, M., Khoshafian, S. and Wong, H. (1989), ’Intelligent databases. Object-Oriented, deductive hypermedia technologies.’ (New York)

Ree, B.v.d. (1993),’ Het comenius project voor het primair onderwijs.’ In: **Cahiers van de Vereniging voor Geschiedenis en Informatica 6**

Safire, H. (1988), ’How to avoid the bruised-knee effect.’ In : The Databased Advisor, November 1988

Thaller, M. (1990),’Databases and expert systems as complementary tools for historical research.’, In: Tijdschrift voor Geschiedenis 103

Thaller, M. (1992), ’On the conception, training and employment of historical data and knowledge daemons.’ in: Oldervoll, J. (ed.), Eden or Babylon? On future software for highly structured data., (St.Katharinen).

Welling, G.M. (1991),’The Paalgeld Project. Methods and first results.’, Paper presented to the VI AHC Conference, (Odense) .

Welling, G.M. (1992),’Intelligent large-scale historical direct-data-entry programming. In: J. Smets (ed.) History and Computing V., Montpellier.

Welling, G.M. (1993), ’A strategy for intelligent input programs for structured data’, in: History and Computing Vol 5 no 1, (Edinburgh).

Willemse, H. and Lindijer, G. (1988),’Software ergonomie.’ (Schoonhoven)

Bert Bos, (bert@let.rug.nl), Department of Alfa-Informatica, University of Groningen, The Netherlands. Telephone : + 31 50 635936, Fax : + 31 50 634900

George Welling, (welling@let.rug.nl), Department of Alfa-Informatica, University of Groningen, The Netherlands. Telephone : + 31 50 635474, Fax : + 31 50 634900